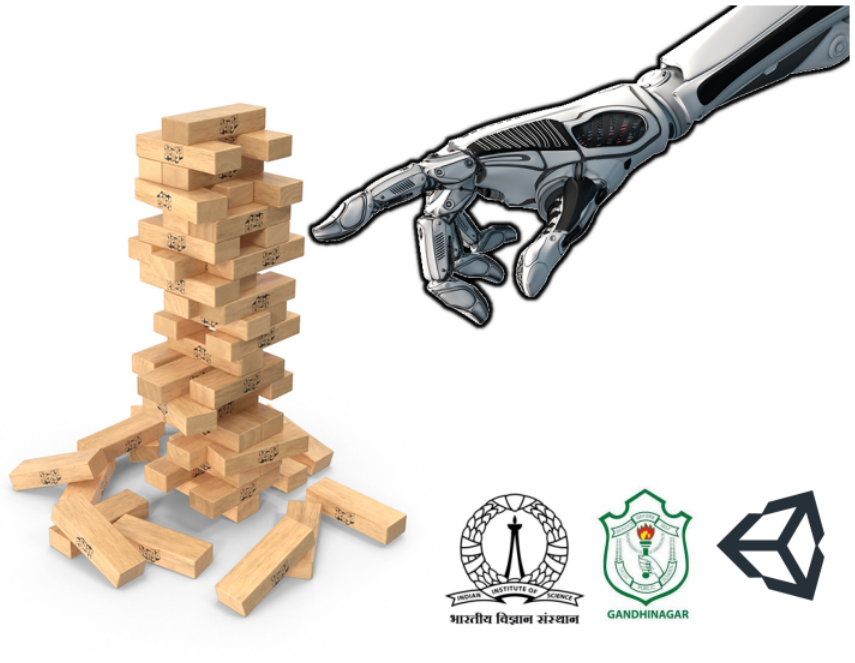
Learning to play Jenga

JengaZero: Manual GamePlay



LEARNING TO PLAY *Jenga*

JengaZero: MANUAL GAMEPLAY

REPORT

Bhavya Sharma
Delhi Public School(DPS)
Gandhinagar, Gujarat,
India -
sarvbhavya@gmail.com

Jaya Kumar Alageshan
Department of Physics,
Indian Institute of Science,
Bangalore, India - 560012
jayaka@iisc.ac.in

Animesh Kuley
Department of Physics,
Indian Institute of Science,
Bangalore, India - 560012
akuley@iisc.ac.in

September 28, 2022

ABSTRACT

We develop *JengaZero*, a simplified version of Jenga game in Unity3D with manual gameplay. We use Unity game engine to build the Jenga blocks, and incorporate collision and static-, kinetic-friction between the blocks, in the presence of gravity. *JengaZero* forms the base for our studies on *JengaAlpha* where we intent to use machine learning (in particular reinforcement learning) methods to play *JengaZero* and extract the optimal strategies.

Keywords Jenga · JengaZero · Reinforcement learning

1 Introduction

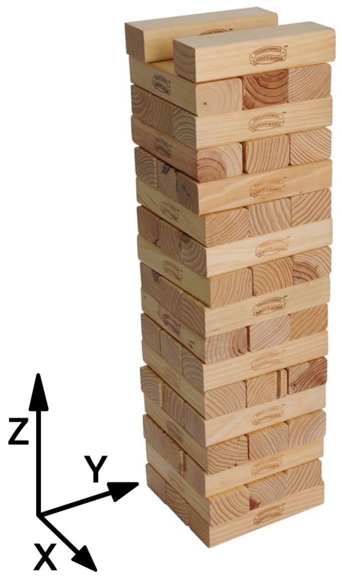


Figure 1: Initial organization of the Jenga blocks.

Jenga is a board game developed by designer and author Leslie Scott, where players take turns removing one block at a time from a tower constructed of 54 blocks. The removed block is then placed on top of the tower, in the process reducing the mechanical stability of the structure. A typical Jenga game has 54 blocks, with each block of dimension $L \times L/3 \times L/5$. The game is initialized by arranging the blocks into a solid rectangular tower of 18 layers, with three blocks per layer. The blocks within each layer are oriented in the same direction, with their long sides touching, and are perpendicular to the ones in the layer immediately below.

The players take turns removing one block from any level below the highest completed one and placing it horizontally atop the tower, perpendicular to any blocks on which it is to rest. Once a level contains three blocks, it is complete and may not have any more blocks added to it. A turn ends when the next player in sequence touches the tower or when 10 seconds have elapsed since the placement of a block, whichever occurs first. The game ends when any portion of the tower collapses, caused by either the removal of a block or its new placement. The last player to complete a turn before the collapse is the winner.

2 *JengaZero*

JengaZero is a simplified version of the Jenga game, where only one player is allowed to play at a time without replacing the block that is removed on the top of the stack. We award points for each block removed and the game ends when the tower collapses. The player who manages to accumulate the most points by the end of the game wins.

Our goal in development of *JengaAlpha* is to implement an algorithm that can find the optimal sequence of blocks to remove and maximize the total points.

3 Game development in Unity

The Unity game engine was selected as a base for the physics and collision calculation for the Jenga tower and rendering of the graphics as it provides real-time rendering and simulation.

3.1 Version 0.1

V0.1 establishes a base structure for the game to be built upon with elements such as a moving camera; for the base structure, the layout of the play area and Jenga tower was built from scratch. To keep the Jenga tower as authentic as possible, we used 54 blocks in a ratio of 1.5 units × 2.5 units × 7.5 units per block.

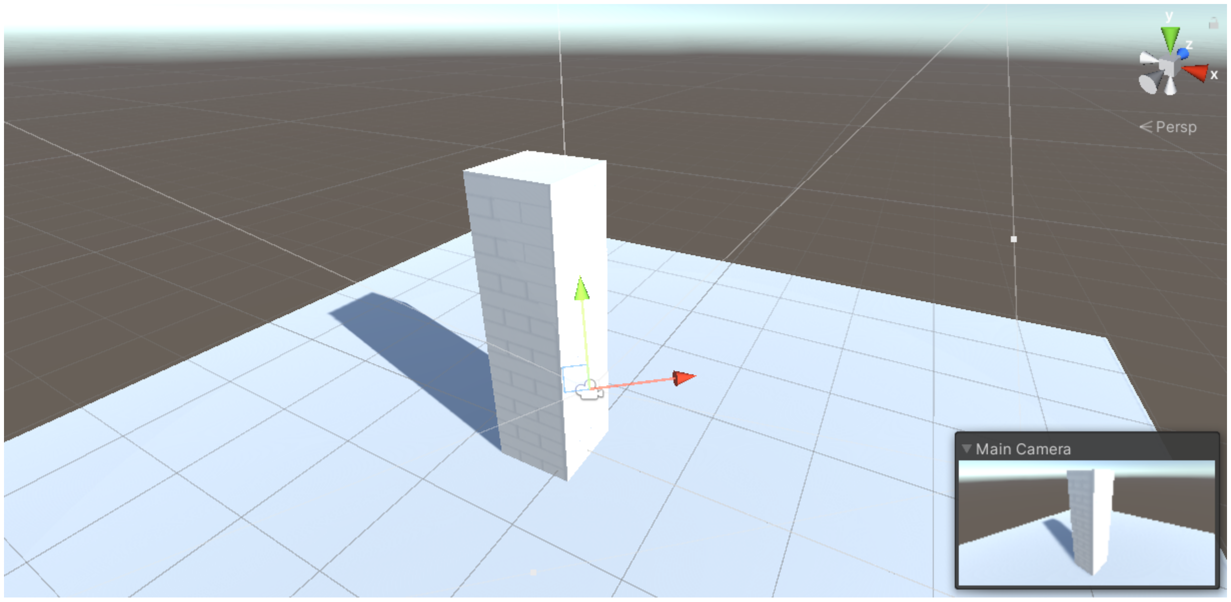
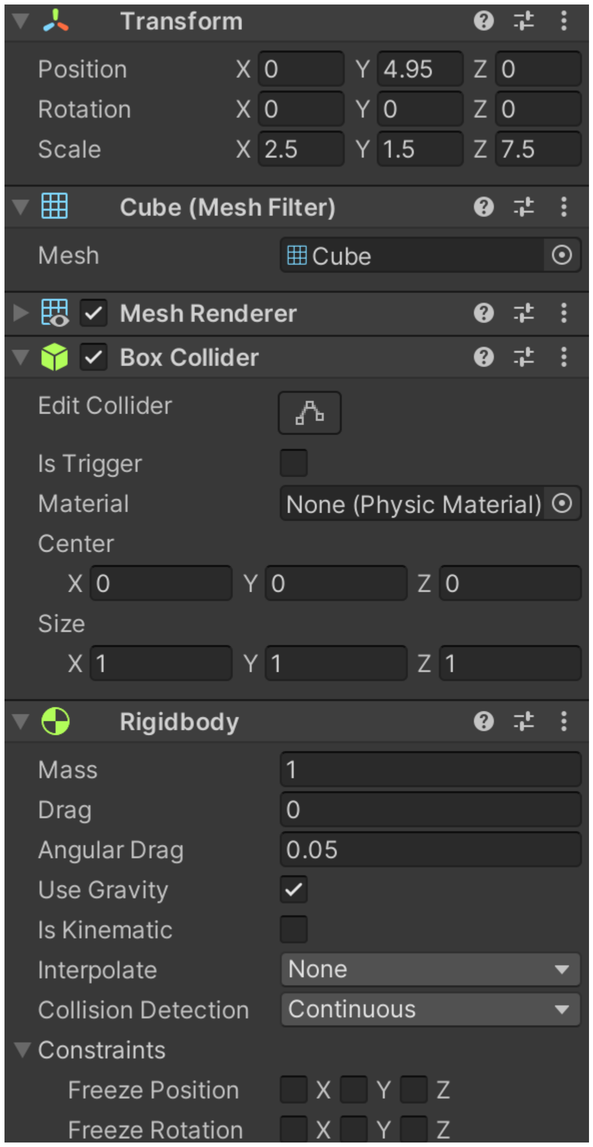


Figure 2: Base Game V0.1

3.1.1 Rigidbody

The Rigidbody function was applied to all blocks to let them interact with each other and their surroundings. Each block was given a variable value of mass which allows for fine-tuning of gravity and continuous collision detection that prevents blocks from phasing into one another.



Transform

Position	X 0	Y 4.95	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 2.5	Y 1.5	Z 7.5

Cube (Mesh Filter)

Mesh: Cube

✓ Mesh Renderer

✓ Box Collider

Edit Collider	
Is Trigger	☐
Material	None (Physic Material)
Center	X 0 Y 0 Z 0
Size	X 1 Y 1 Z 1

Rigidbody

Mass	1
Drag	0
Angular Drag	0.05
Use Gravity	✓
Is Kinematic	☐
Interpolate	None
Collision Detection	Continuous
Constraints	
Freeze Position	☐ X ☐ Y ☐ Z
Freeze Rotation	☐ X ☐ Y ☐ Z

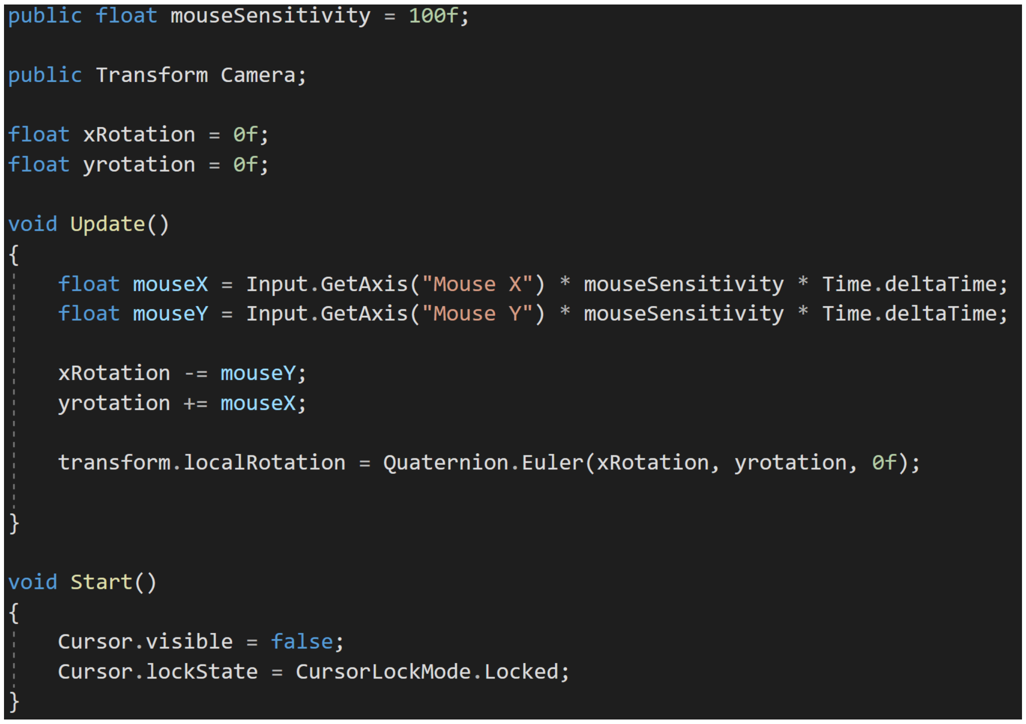
Figure 3: Properties of an individual block in the tower

3.1.2 Camera

The camera acts as the main medium through which the player interacts with the game. Using the in-built camera from Unity, a simple script in *C#* was made that allowed allowed the user to move the camera using the mouse input. The idea behind the camera-based input system is that the camera references center of the screen (Denoted by a circle known as the crosshair). When the mouse is hovered over a block by altering the X and Y axis of the mouse input (moving the mouse on a surface), the user can hold the left click to pick up and move the mouse to drag the block outside the tower to score points.

Cameramovement.cs

The main script responsible for camera movement is called *Cameramovement.cs* (refer to figure 4).
Using the in-built mouse movement float values of *mouseX* and *mouseY*, the program is able to calculate and adjust the positioning of the camera with respect to its *'Transform'* function (The Transform function allows the script to reference the camera's positional values once assigned in Unity).
The mouse movement is calculated on a frame-by-frame basis in the *'Update'* function of the script, and thus the speed of the mouse varies relative to the processing speed of the hardware being used. To prevent this dependency of mouse speed with respect to game framerate, the output float values of mouseX and mouseY were multiplied by *Time.deltaTime* (it is the interval in seconds from the last frame to the current one). This allows the speed of the game to be independent of the framerate.
This script also hides and locks the cursor, preventing the user from accidentally clicking outside the game window.
Lastly, the mouse sensitivity can be adjusted in-game by altering the value of the *'float mouseSensitivity'* higher or lower than 100 (default value).



```
public float mouseSensitivity = 100f;

public Transform Camera;

float xRotation = 0f;
float yrotation = 0f;

void Update()
{
    float mouseX = Input.GetAxis("Mouse X") * mouseSensitivity * Time.deltaTime;
    float mouseY = Input.GetAxis("Mouse Y") * mouseSensitivity * Time.deltaTime;

    xRotation -= mouseY;
    yrotation += mouseX;

    transform.localRotation = Quaternion.Euler(xRotation, yrotation, 0f);

}

void Start()
{
    Cursor.visible = false;
    Cursor.lockState = CursorLockMode.Locked;
}
```

Figure 4: Camera Movement script

3.2 Version 0.3

V0.3 introduces basic User Interface (UI) elements, initial graphical improvements, and highlighting of selected blocks via the camera.

3.2.1 Graphical Changes and Highlighting Blocks

We created two separate materials (textures or colors) for the Jenga tower that allow for better game-play visibility. Alternating the block materials by having the central block on each floor be a different color made it easier to identify the individual blocks throughout the tower.
The integration of materials allowed for an additional material to be created, which would be used as a selection material. A new script was created responsible for managing the highlighting of blocks called 'SelectionManager.cs'

SelectionManager.cs

Using the center of the screen as a reference point, this script replaces the material of the block in front of this point with the selection material. The selection material would be vibrant relative to the rest of the scene, allowing for the player to easily identify which block was going to be selected.

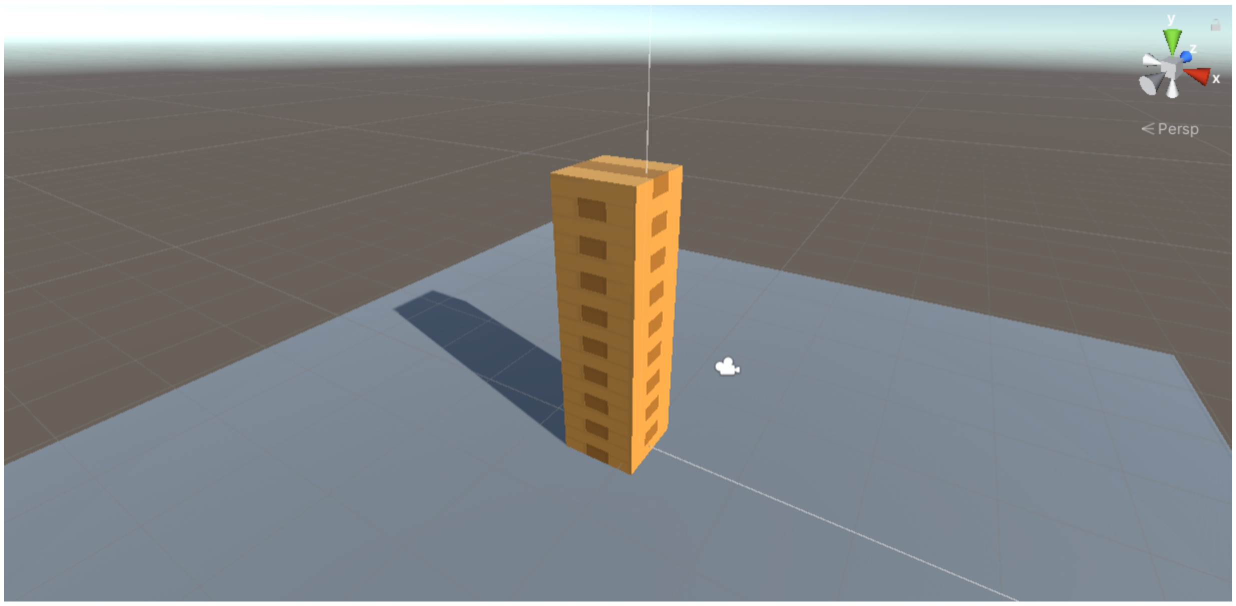


Figure 5: Game after Graphical Improvements

All functions of the SelectionManager script take place within the *'Update'* function, and thus it constantly checks per frame whether the specified condition is valid or invalid; this allows for real-time highlighting for the player to visualize which block can be picked up.
The script casts a raycast (ray) from the center of the camera and checks if the ray has hit a rigidbody; when the rigidbody (in this case, a Jenga block) is hit, the script changes the block's present material to the selection material termed as *'highlightMaterial'*. Once the ray stops hitting the current rigidbody, it replaces the *'highlightMaterial'* with *'defaultMaterial'*.
The assignment of materials and interchanging of tags is handled by an empty object in the game scene called *'SelectionManager'*.

3.2.2 Implementing basic User Interface components

All of the user interface components were added using the Unity UI toolkit.
This version adds a point of reference in the center of the screen represented by a white dot known as the cross-hair; it allows the user to determine where to move the mouse in order to select a desired block.
Any block in front of this point changes color with reference to the highlight script.

3.2.3 Limitation of Selection Script

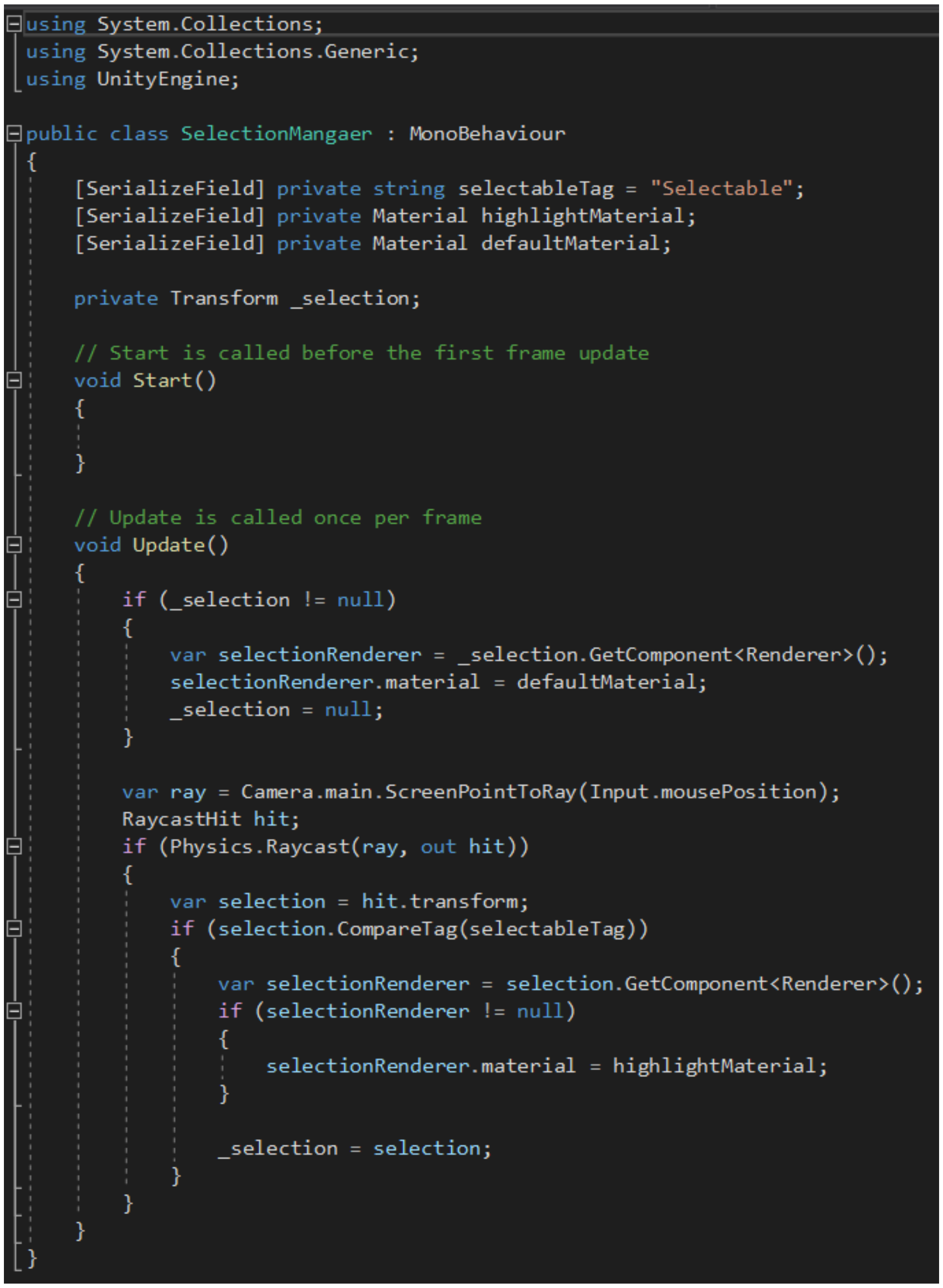
In order to make the game look better, two separate materials were used for the Jenga tower (see 3.2.1); however, the selection manager script is only able to allocate one material as the replacement material.
This limitation of the script results in both shades of the tower being replaced by one, reversing the allocation of distinct colors to better the in-game visibility. To overcome this issue, the next version introduced a new shading and rendering method (see 3.3.1).

3.3 Version 0.5

V0.5 changes the shading and rendering system of the game. A system of picking up rigidbodies (blocks) from the tower was also implemented.
Shading and rendering the game in different ways allows for more flexibility, giving a unique and interesting look to the overall image.

3.3.1 Major Graphical Improvements

Due to the limitations of the block selection highlighting system, a custom rendering method was used in order to simulate the game.



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

public class SelectionMangaer : MonoBehaviour
{
    [SerializeField] private string selectableTag = "Selectable";
    [SerializeField] private Material highlightMaterial;
    [SerializeField] private Material defaultMaterial;

    private Transform _selection;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        if (_selection != null)
        {
            var selectionRenderer = _selection.GetComponent<Renderer>();
            selectionRenderer.material = defaultMaterial;
            _selection = null;
        }

        var ray = Camera.main.ScreenPointToRay(Input.mousePosition);
        RaycastHit hit;
        if (Physics.Raycast(ray, out hit))
        {
            var selection = hit.transform;
            if (selection.CompareTag(selectableTag))
            {
                var selectionRenderer = selection.GetComponent<Renderer>();
                if (selectionRenderer != null)
                {
                    selectionRenderer.material = highlightMaterial;
                }

                _selection = selection;
            }
        }
    }
}
```

Figure 6: Selection Manager script

This new system allowed for each object to have its own outline around the edges. The shade of the object color was dependent upon the angle of observation, allowing for unique blends and different shades to appear as an object rotated. Blocks that could not be selected had their own separate look, indicating that they were not interactable.
In order to achieve these results, multiple scripts work in tandem with some custom imported objects, graphical renderers, and node maps. *DepthNormalsFeature.cs*, *DecodeDepthNormals.hlsl*, and *OutlinesInclude.hlsl* are the scripts alongside the subgraphs of *'Outlines'*, *'SobelFineTuning'*, and the shader graphs *'SimpleOutlines'* and *'TransparentOutlinesPass'* are responsible in working together and produce new graphical outputs for the revamped rendering system.

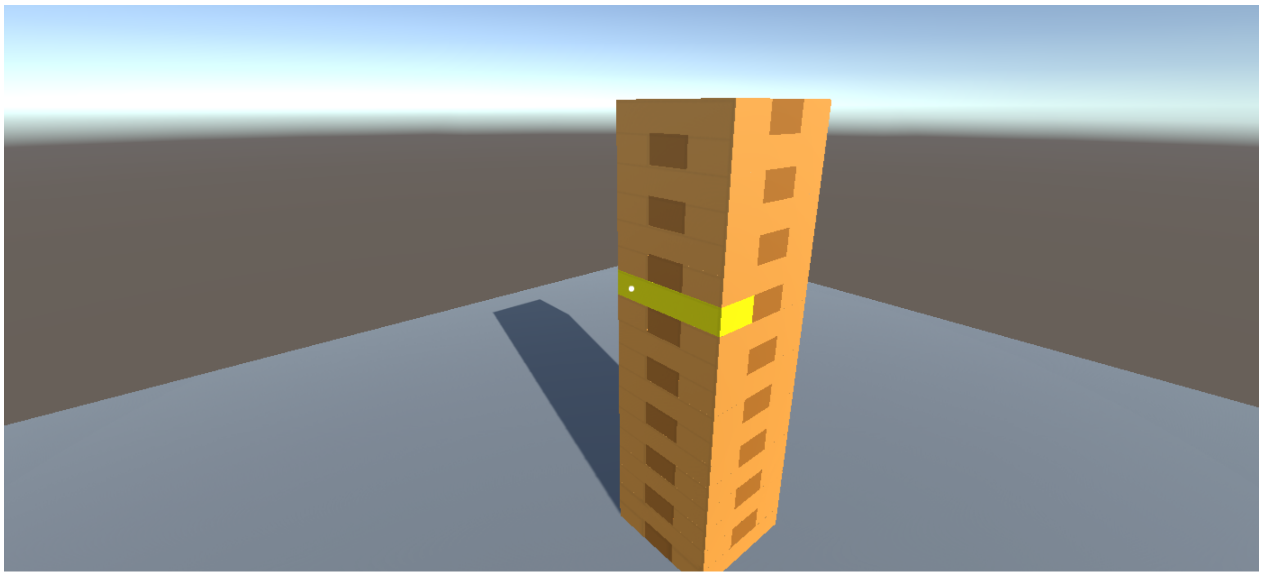


Figure 7: A block being highlighted

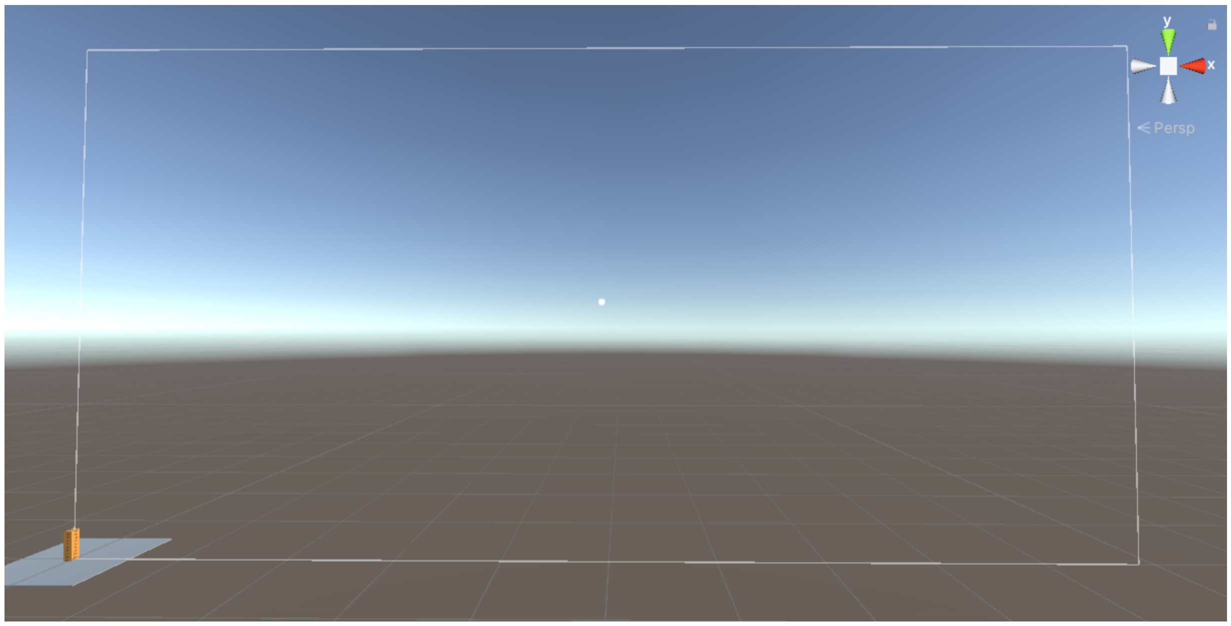


Figure 8: Introduction of UI elements

3.3.2 Dragging Rigidbodies

After implementing camera movement, a script that would allow the user to use an input (in this case, the left mouse button for convenience) was created to emulate the action of picking up a Jenga block. The player can then move their mouse input which causes the selected block to move; the user can then drag the block outside the tower and attempt to do so without it toppling.
SC_DragRigidbody.cs handles all of the rigidbody (jenga block) movement operations.

SC_DragRigidbody.cs

This script samples the primary camera from the game and, by casting a raycast from the central point of the screen, checks to see if an active rigidbody is obstructing the raycast. If the player presses the left-click button from the mouse input, the desired rigidbody is locked to the crosshair (center of the screen). With the help of the camera movement

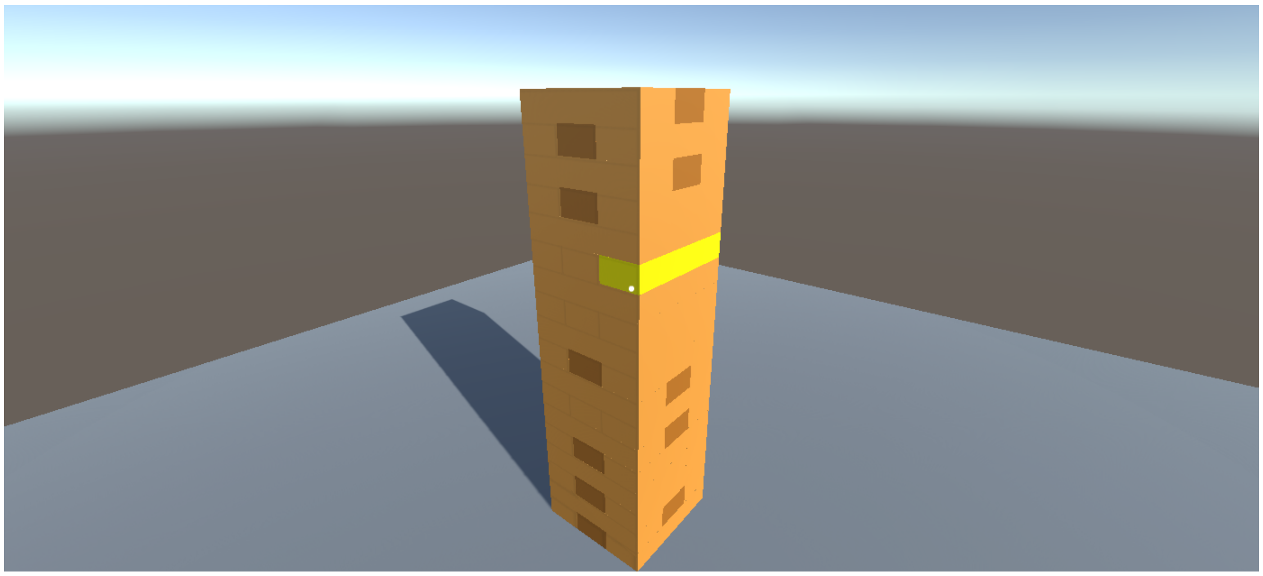


Figure 9: Limitation of Block Highliting Script

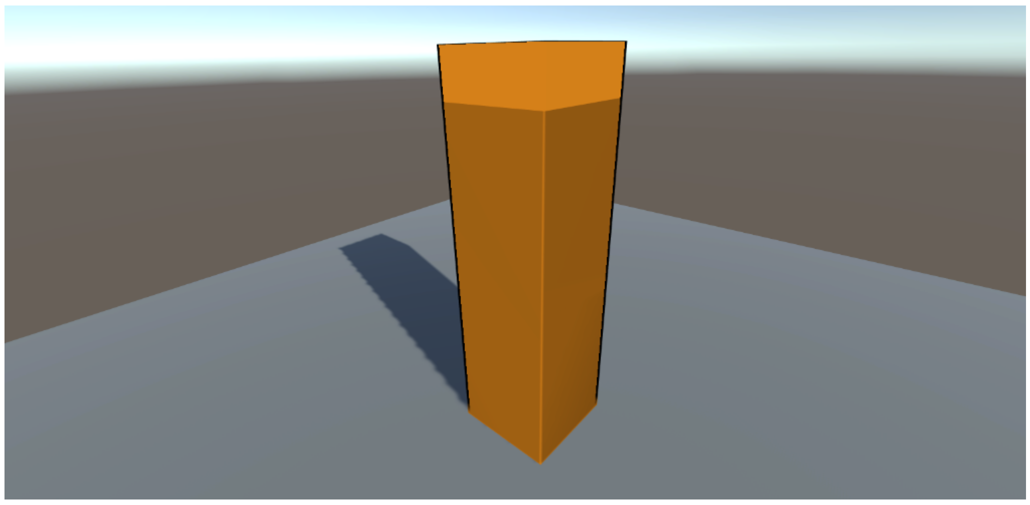


Figure 10: Game after Rendering Changes

script previously created, once the player changes the rational values of the camera, the highlighted block is also moved accordingly by actively keeping it locked to the crosshair. The player can then release the block in a different location by releasing the left-click button.

3.4 Version 0.7

This version includes the addition of a game failstate and a method to count the number of blocks that have been removed from the tower.
The block counting method can be further developed into a score system.

3.4.1 Game Failstate

This version focuses on implementing a way that would allow the game to detect if the player had toppled the tower over, or in other words, had lost the game.
The script called *"CollisionDetection.cs"* handles and executes the necessary actions to check if the layout for the current game qualifies for the necessary requirements for an active failstate.

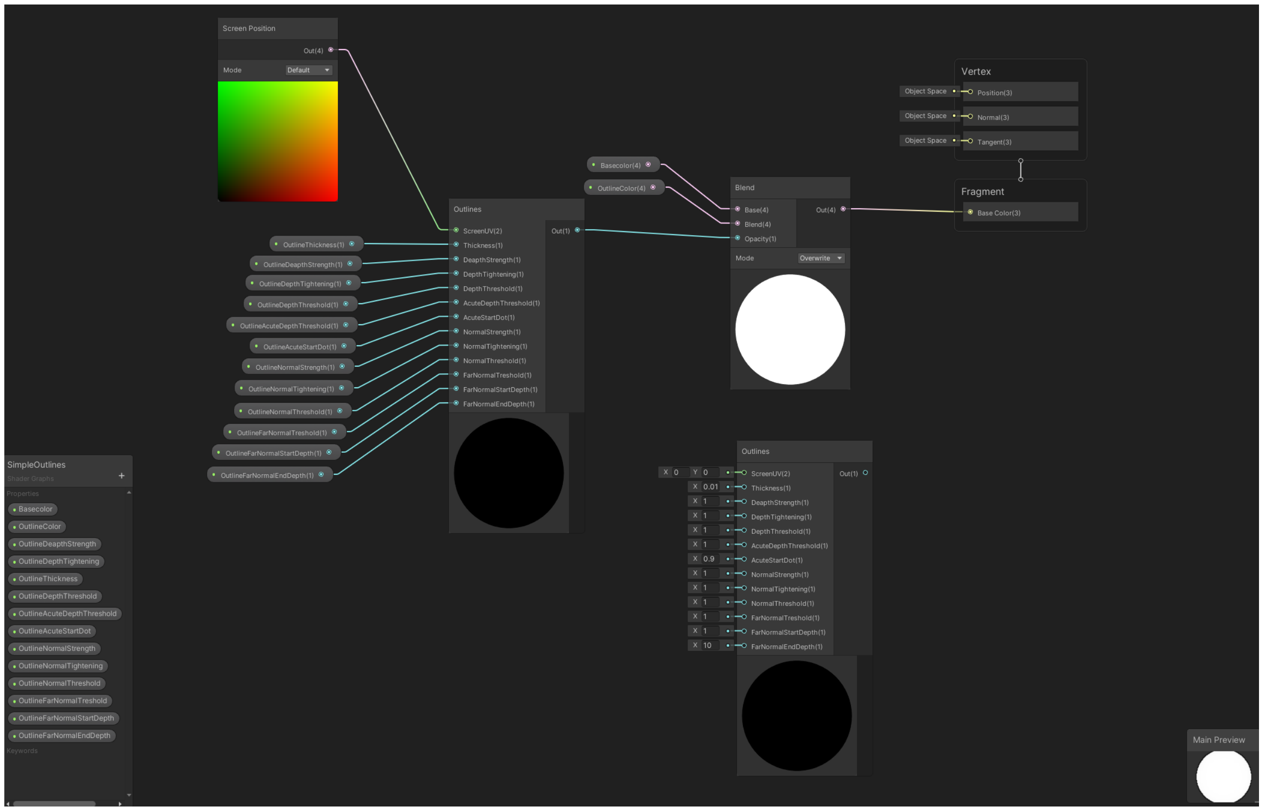


Figure 11: SimpleOutlines' node map

CollisionDetection.cs

In order to check for if the tower has toppled over, a special algorithm was created which would check the number of blocks falling within a short burst of time on loop while the game was running.

CollisionDetection.cs is able to achieve this with the help of 4 invisible rectangular blocks surrounding the tower acting as walls. Each wall is not rendered so the player can not see them. All detectors lack the rigidbody function so they do not collide with the falling blocks. Each detector also has the unity box collider function running which allows *CollisionDetection.cs* to check if a rigidbody (block) has touched or entered a collider via the function *"OnTriggerEnter"*.

OnTriggerEnter - Upon calling the function *OnTriggerEnte*r, the variable *"blocksFallen"* increases per block that falls and the "Logic" function is called after exactly 0.2 seconds of adding the blocks.

Logic - When the logic function is called with a delay of 0.2 seconds, the *blocksFallen* variable is reset to 1

Update - While this script is running, the Update function constantly checks if the variable *blocksFallen* is greater than 2 and ends the game if the required condition is met.

In order to explain this algorithm in simpler terms, *CollisionDetection.cs* checks whether more than 2 blocks have fallen from the tower in a 0.2 second span on loop.
When a player is removing blocks from the tower one at a time, under normal circumstances it would take more than 0.2 seconds to remove each block, hence the *blocksFallen* variable would reset each time a block is removed. However, in the scenario of the when tower falls, multiple blocks would trigger the script within a time period smaller than 0.2 seconds and the algorithm would declare that the tower has fallen and the player has failed the game.

It is also worth mentioning that each of the detection walls have been offset from the edges of the tower from a few units to prevent accidental failstate triggers when the tower leans in one particular direction but does not fall. The unit offset has been optimised after several playthroughs as to not be too far or close to the tower in order to minimize the limitations and errors of the *CollisionDetection.cs* script.

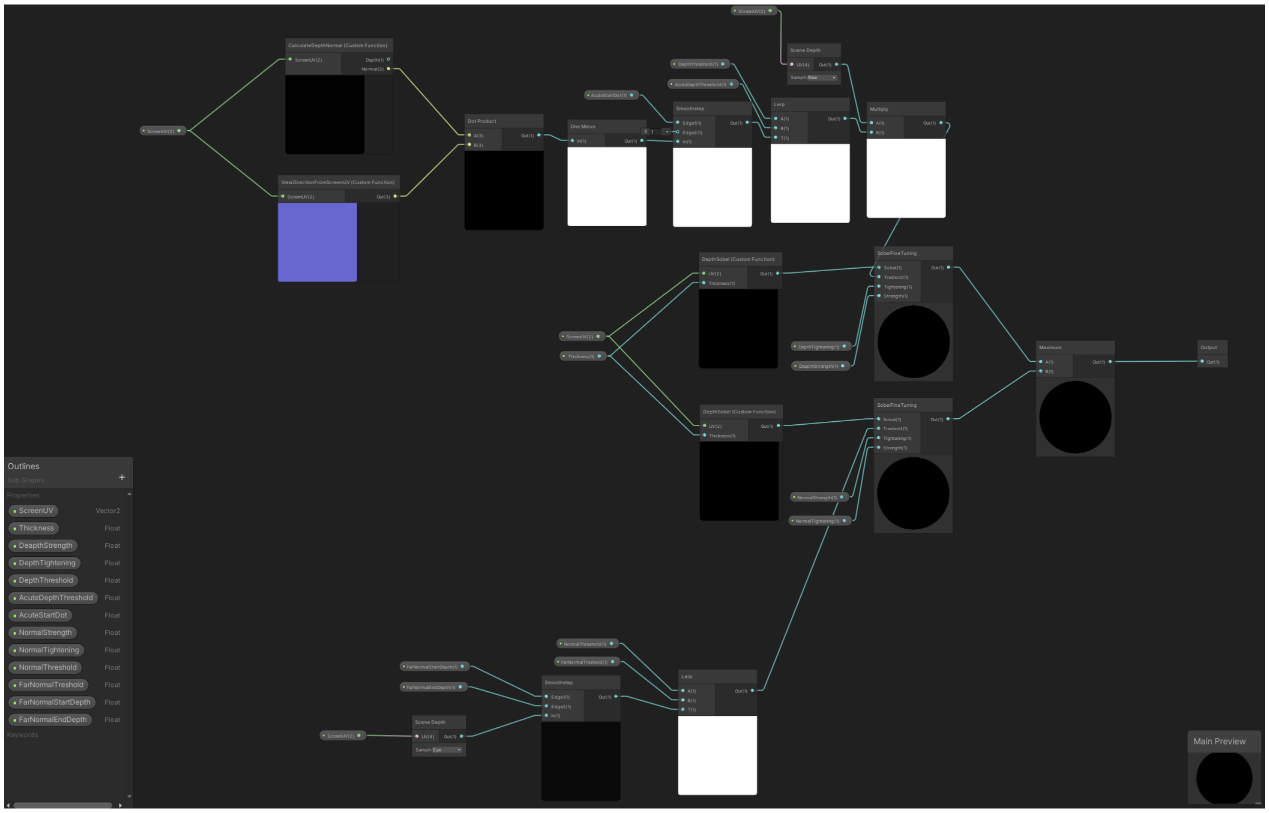


Figure 12: Outlines' node map

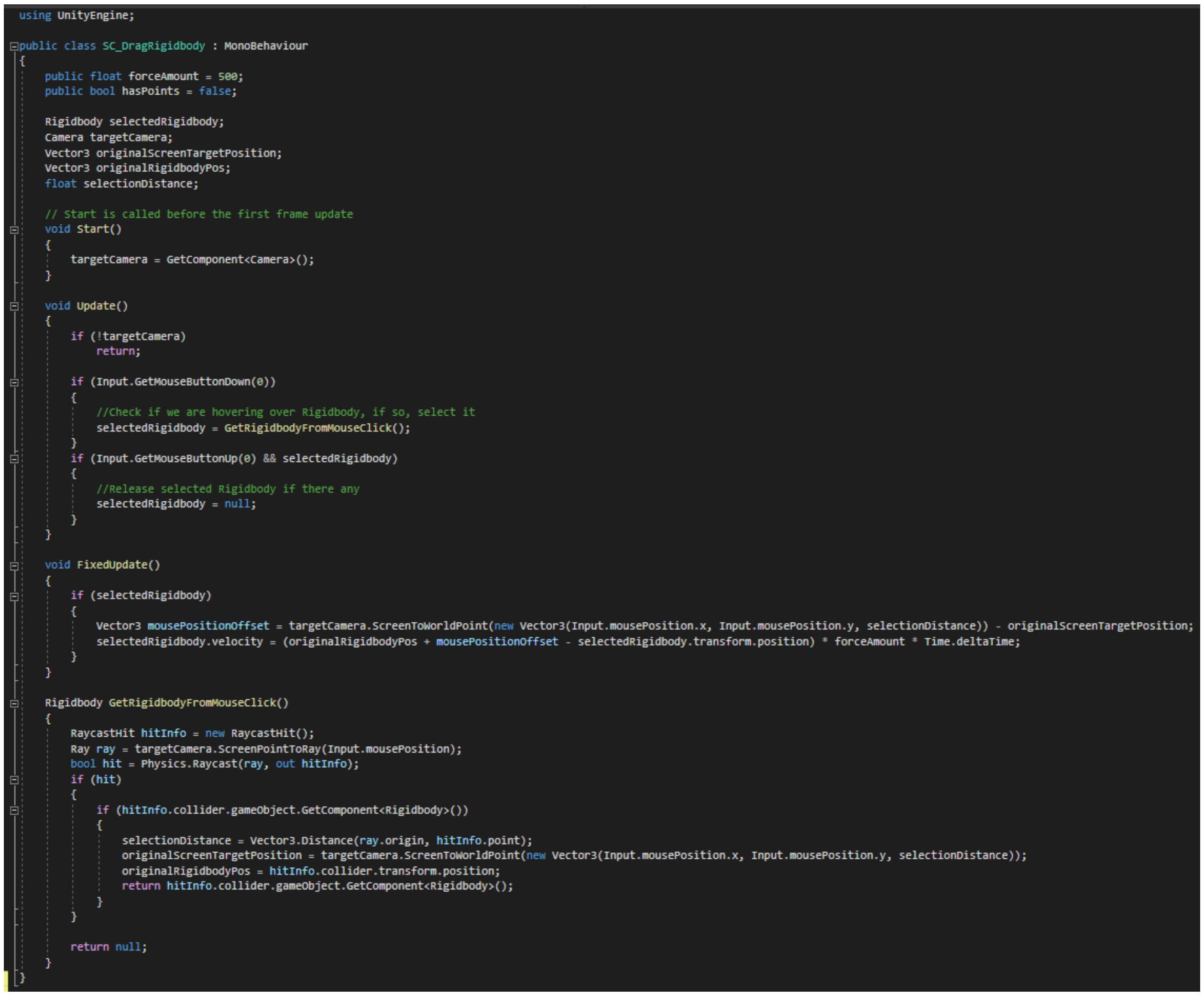
3.4.2 Point System

A point system was implemented so that the player could keep track of how many blocks they had removed and provide a gameplay element of letting the user beat their previous score.
More importantly, the point system allows the implementation of artificial intelligence mode in a simpler manner. The central concept was, with the help of the score, the AI would be rewarded when it received a higher score without the tower toppling (which would automatically set the score to 0 as a punishment).
This point based reward system allows for implementation of evolutionary based learning algorithms that run several iterations of the same AI and choose the one that got the highest score. This process can be repeated multiple times, replacing the new AI with the best one from the previous generation.

ScoreSustem.cs

The initial implementation of this system was to use the same cuboid detectors that were responsible for checking for a valid gamestate, and re-purpose them in a manner where a fixed value would be added to a set variable dedicated to keeping track of the score.
This method of trying to keep track of the score did not work because each cuboid added to the score assigned variable separately, which resulted in 4 different outputs each time a block was detected by the script. This also meant that there was no concrete value to the score variable as each detector only added the amount of points relative to the number of blocks that enter it.

The process of counting the score was then changed to a system where the entire tower would reside within one big invisible cuboid that would be slightly bigger than the 4 detectors used to check for a game failstate. This giant block would provide data when a block was removed from its faces, thus allowing for a more concrete way of adding to the score.
ScoreSustem.cs uses a function similar to *"OnTriggerEnter"* used by the *CollisionDetection.cs* script, called *"OnTriggerExit"*. This function allows the script to add 10 points to the scoreboard every time a rigidbody (in this scenario a Jenga block) leaves the detection cube tower.



```
using UnityEngine;

public class SC_DragRigidbody : MonoBehaviour
{
    public float forceAmount = 500;
    public bool hasPoints = false;

    Rigidbody selectedRigidbody;
    Camera targetCamera;
    Vector3 originalScreenTargetPosition;
    Vector3 originalRigidbodyPos;
    float selectionDistance;

    // Start is called before the first frame update
    void Start()
    {
        targetCamera = GetComponent<Camera>();
    }

    void Update()
    {
        if (!targetCamera)
            return;

        if (Input.GetMouseButtonDown(0))
        {
            //Check if we are hovering over Rigidbody, if so, select it
            selectedRigidbody = GetRigidbodyFromMouseClick();
        }
        if (Input.GetMouseButtonUp(0) && selectedRigidbody)
        {
            //Release selected Rigidbody if there any
            selectedRigidbody = null;
        }
    }

    void FixedUpdate()
    {
        if (selectedRigidbody)
        {
            Vector3 mousePositionOffset = targetCamera.ScreenToWorldPoint(new Vector3(Input.mousePosition.x, Input.mousePosition.y, selectionDistance)) - originalScreenTargetPosition;
            selectedRigidbody.velocity = (originalRigidbodyPos + mousePositionOffset - selectedRigidbody.transform.position) * forceAmount * Time.deltaTime;
        }
    }

    Rigidbody GetRigidbodyFromMouseClick()
    {
        RaycastHit hitInfo = new RaycastHit();
        Ray ray = targetCamera.ScreenPointToRay(Input.mousePosition);
        bool hit = Physics.Raycast(ray, out hitInfo);
        if (hit)
        {
            if (hitInfo.collider.gameObject.GetComponent<Rigidbody>())
            {
                selectionDistance = Vector3.Distance(ray.origin, hitInfo.point);
                originalScreenTargetPosition = targetCamera.ScreenToWorldPoint(new Vector3(Input.mousePosition.x, Input.mousePosition.y, selectionDistance));
                originalRigidbodyPos = hitInfo.collider.transform.position;
                return hitInfo.collider.gameObject.GetComponent<Rigidbody>();
            }
        }

        return null;
    }
}
```

Figure 13: Picking Up Blocks Script

In addition, if the script detects the game has ended, the points automatically revert to 0. This feature has been implemented in order to train artificial intelligence, since the AI's goal is to reach the highest possible score within a given time, this addition severely punishes the AI.
Lastly, the script makes all Jenga blocks non-interactable once the game has ended.

3.4.3 Bug Fixes for Score System

In terms of accuracy for detecting if blocks had been removed from the tower, the current score detection system worked very well. However, there was one major issue plaguing the current implementation; a block could be picked up, moved back and forth through the detector and the script would constantly add more points to the scoreboard even though additional blocks were not being removed from the tower, hence acting as a game breaking exploit in the point detection system.
To fix this issue, a separate script was created in order to tag blocks that had already moved through the detection system and prevent the same blocks from being able to further change the score. This script responsible for this action is *"ScorePrevent.cs"*.

ScorePrevent.cs

This script essentially tags all blocks that leave the tower using a similar single block detector and marks them as *"GoneThrough"* via a positive boolean value. In addition, the box collider of the removed block is disabled, which prevents it from being detected by the *scoredetector*.

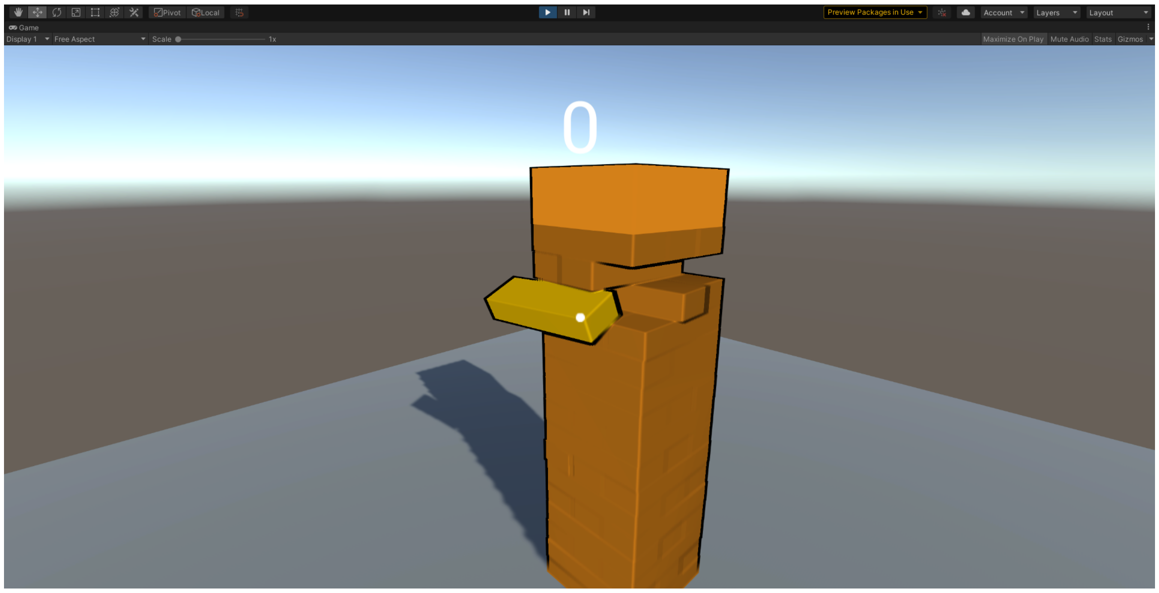


Figure 14: A block being picked up and removed from the tower

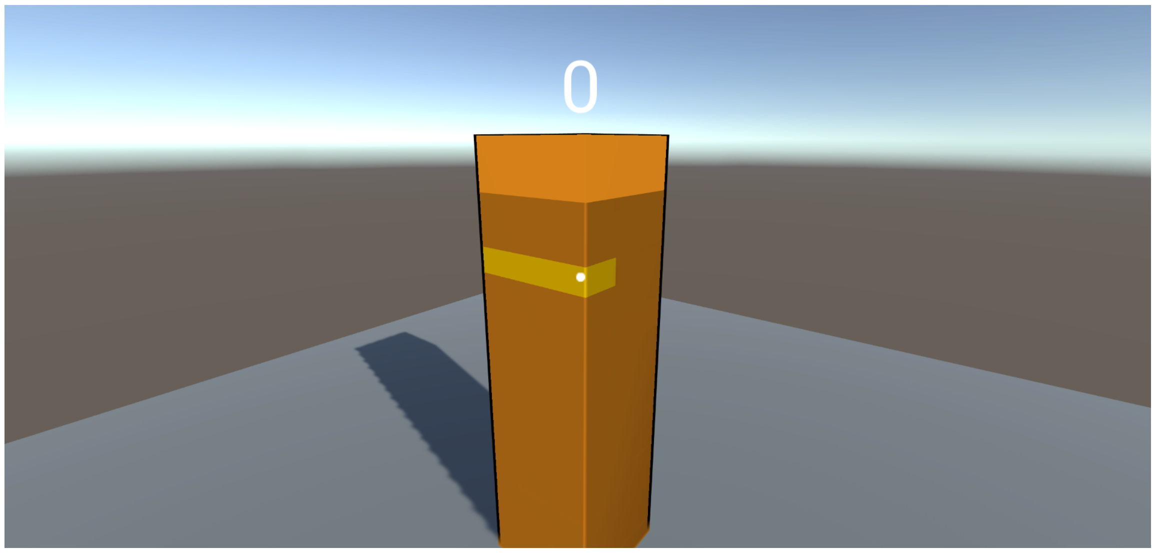


Figure 15: Tower with failstate

3.5 Version 0.8

This version focuses on improving upon the previously added user-interface elements and interlinking of previously implemented features via a GameManager object.

3.5.1 GameManager

In order to interlink core game components such as conveying or referencing whether the game has ended, an empty object called the game manager was created as a placeholder that contained all essential information that could be passed on to it and then further referenced publicly in the game by other objects and scripts.

GamemanagerScript.cs

This game manager is also directly linked to a script named *"GameManagerScript"* that uses information passed on from the *"CollisionDetection.cs"* script and checks if the game has entered an active failstate.

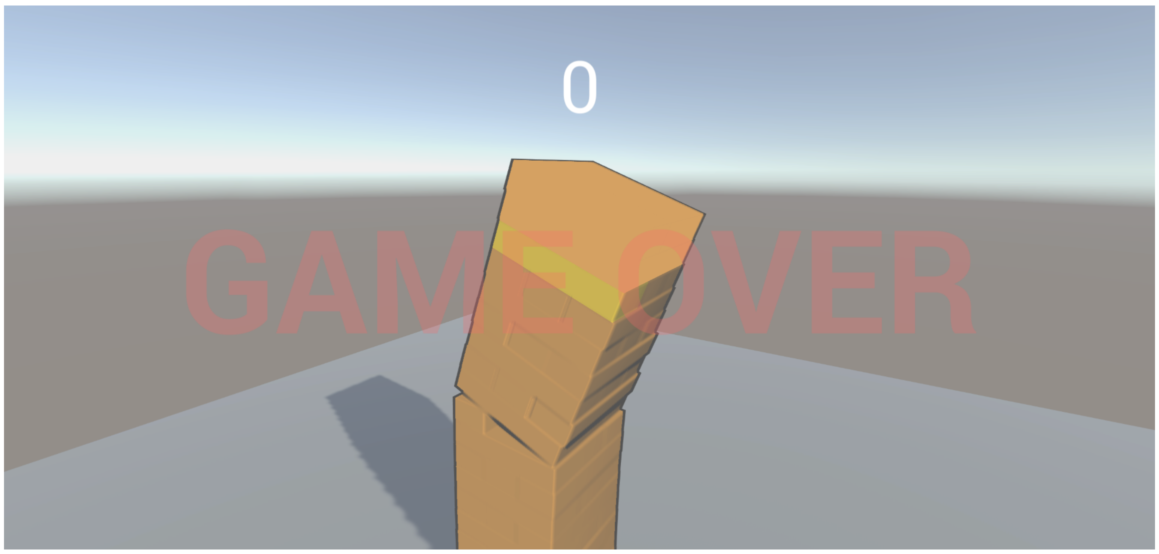


Figure 16: Tower in valid failstate

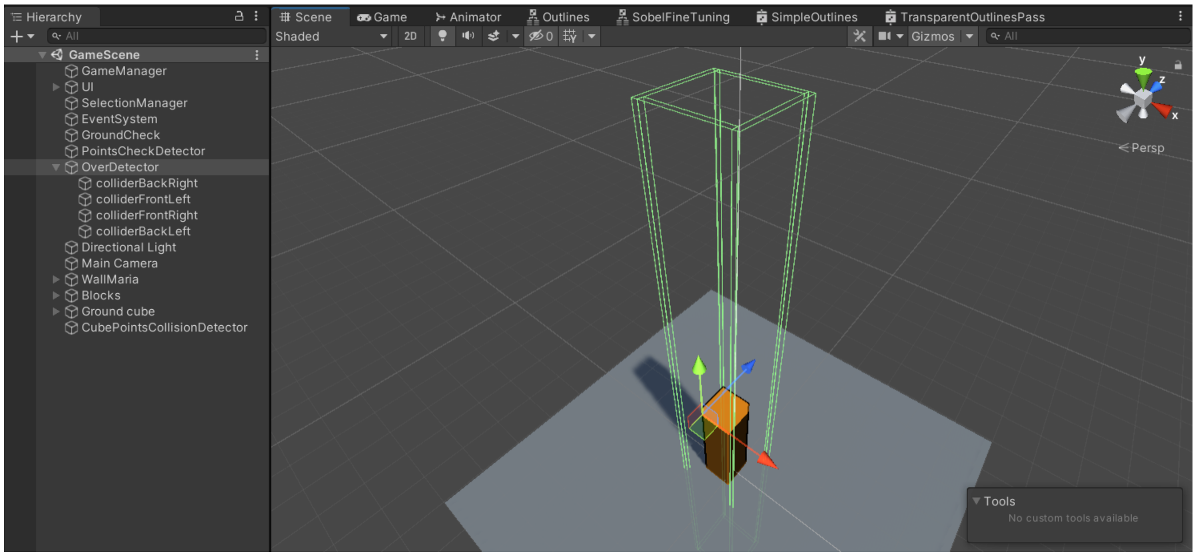


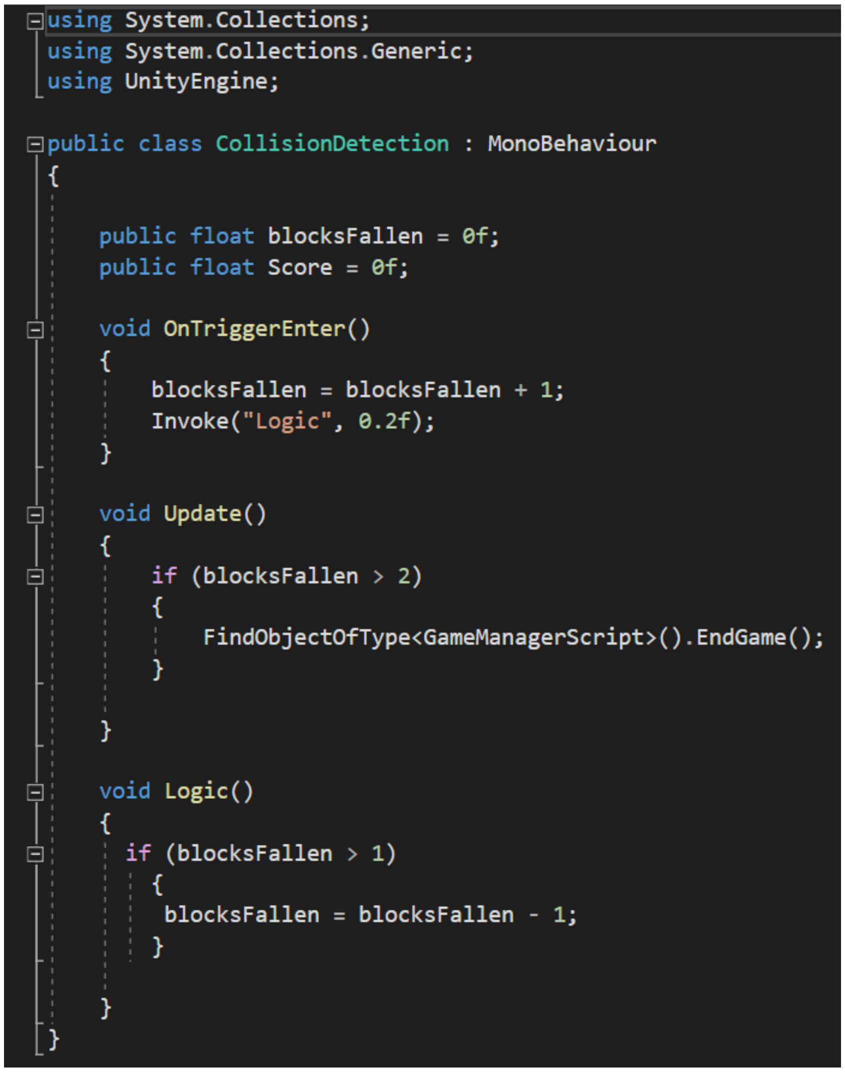
Figure 17: Invisible wall detectors required for block detection

Once the presence of a failstate is detected, this script is able to restart the game by loading the game scene from the start where the score is reset to zero and the tower is returned to its original state.
The text "GAME OVER" is displayed when the tower topples over.
Lastly, the script restarts the game after 5 seconds of the tower toppling.

3.5.2 Score Based User Interface Improvements

A real time score counter was added to user interface. Other elements such as a "GAME OVER" screen were also added. The script that updates the score in real time is called *"ScoreTextScript.cs"*.

ScoreTextScript.cs is executed in real time to reference the score being output from the points detector via the *scoresustem.cs* script and pass it on as a string variable that can be read and changed to the score being currently displayed.
Lastly, a few animations and other minor features were added to make the UI feel cohesive and fluid.



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

public class CollisionDetection : MonoBehaviour
{

    public float blocksFallen = 0f;
    public float Score = 0f;

    void OnTriggerEnter()
    {
        blocksFallen = blocksFallen + 1;
        Invoke("Logic", 0.2f);
    }

    void Update()
    {
        if (blocksFallen > 2)
        {
            FindObjectOfType<GameManagerScript>().EndGame();
        }

    }

    void Logic()
    {
      if (blocksFallen > 1)
        {
         blocksFallen = blocksFallen - 1;
        }

    }
}
```

Figure 18: CollisionDetection.cs

3.6 Version 0.9

The second last iteration of the base game version includes two essential features, one being a main menu that opens when the game is run and allows the user to edit some game preferences, select game modes and close the game. Secondly, pause menu which pauses the game and allows the user to travel across sub menus to implement changes into the game such as some settings options, closing the game or moving back to the main menu was also added.

3.6.1 Main Menu

The main menu is displayed when the user executes the game's main ".exe" file. It provides options of choosing which game mode to enter, applying changes in the user settings and closing the game.
A separate scene was created that contained user interface elements in order to have a functioning main menu.
The main menu is a GUI driven and contains buttons and sub menus in order to traverse and find the desired function within the menu.
All of the functions of the main menu are operated by *"MainMenu.cs"*.

MainMenu.cs

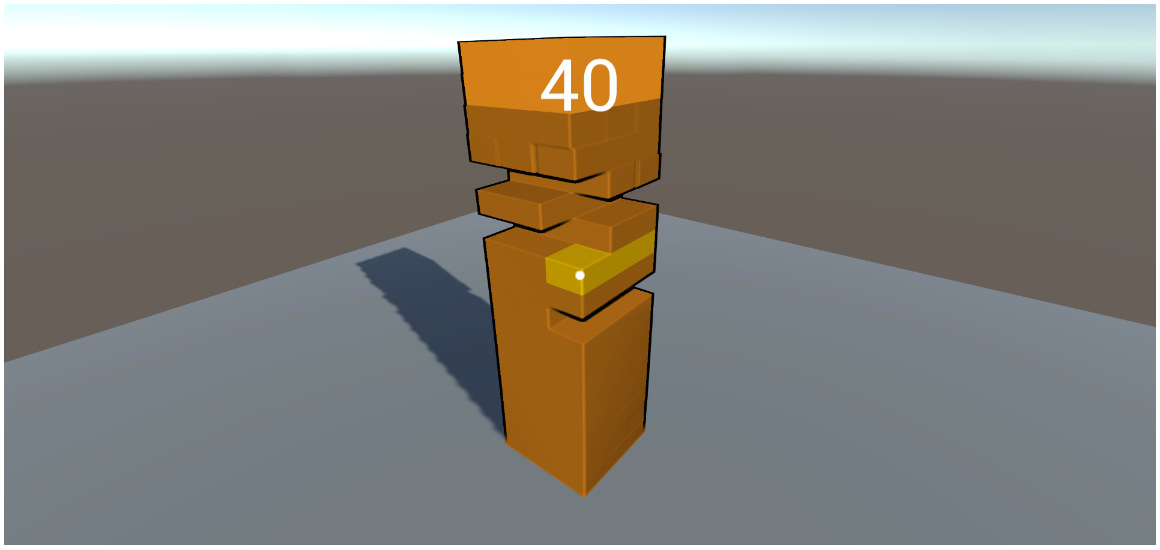


Figure 19: Points being calculated based on number of removed blocks

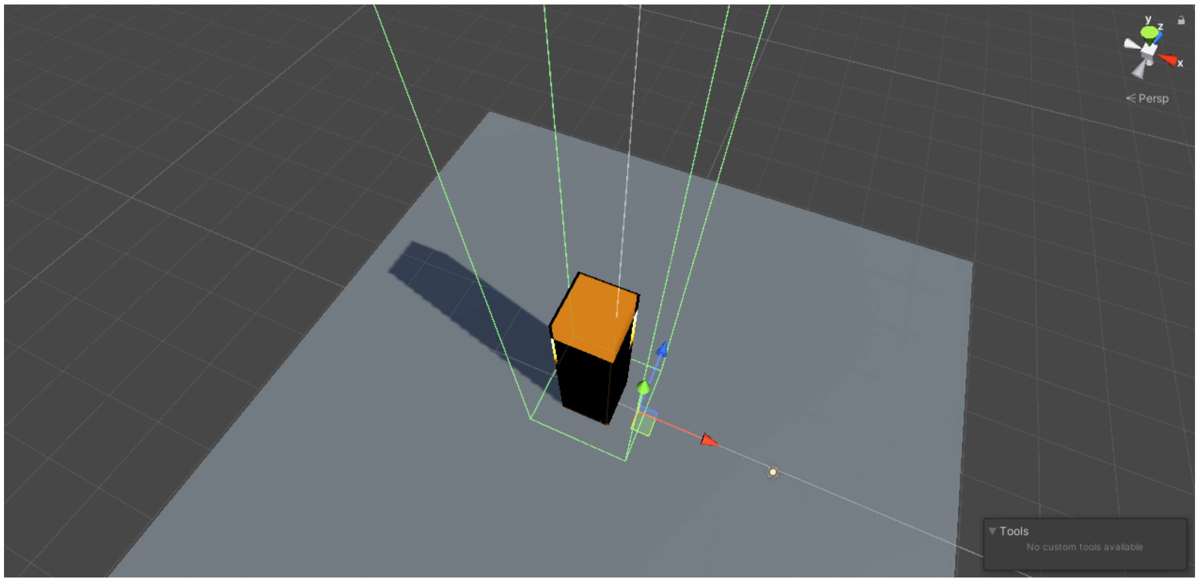


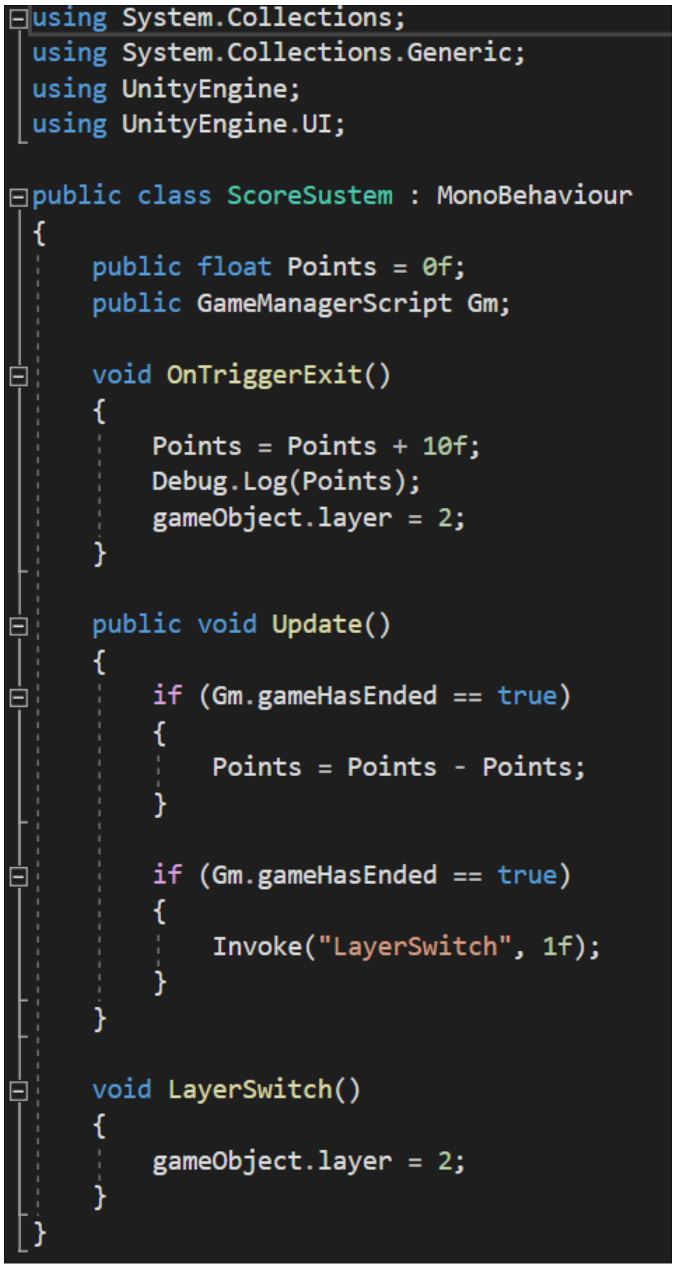
Figure 20: Points Detector

This script runs when the game is launched. Once the game has loaded in the first scene known as the main menu scene which contains all the user interface elements required for the main menu, it allows the player to move to the next scene which is the game scene that contains all the game elements.

3.6.2 Pause Menu

The pause menu allows the user to pause and resume the game to increase the convenience of the game. It also provides access to the main menu for directly closing the game.
The pause menu is activated when the user presses the esape key and no changes can occur within the game scene while the pause menu is activated.
The pausemenu exists in the main game scene within the UI canvas. It is a GUI based menu that uses submenus and buttons in order to allow for user navigation.
All the components and fucntions of the pause menu are present within the *PauseMenu.cs* script.

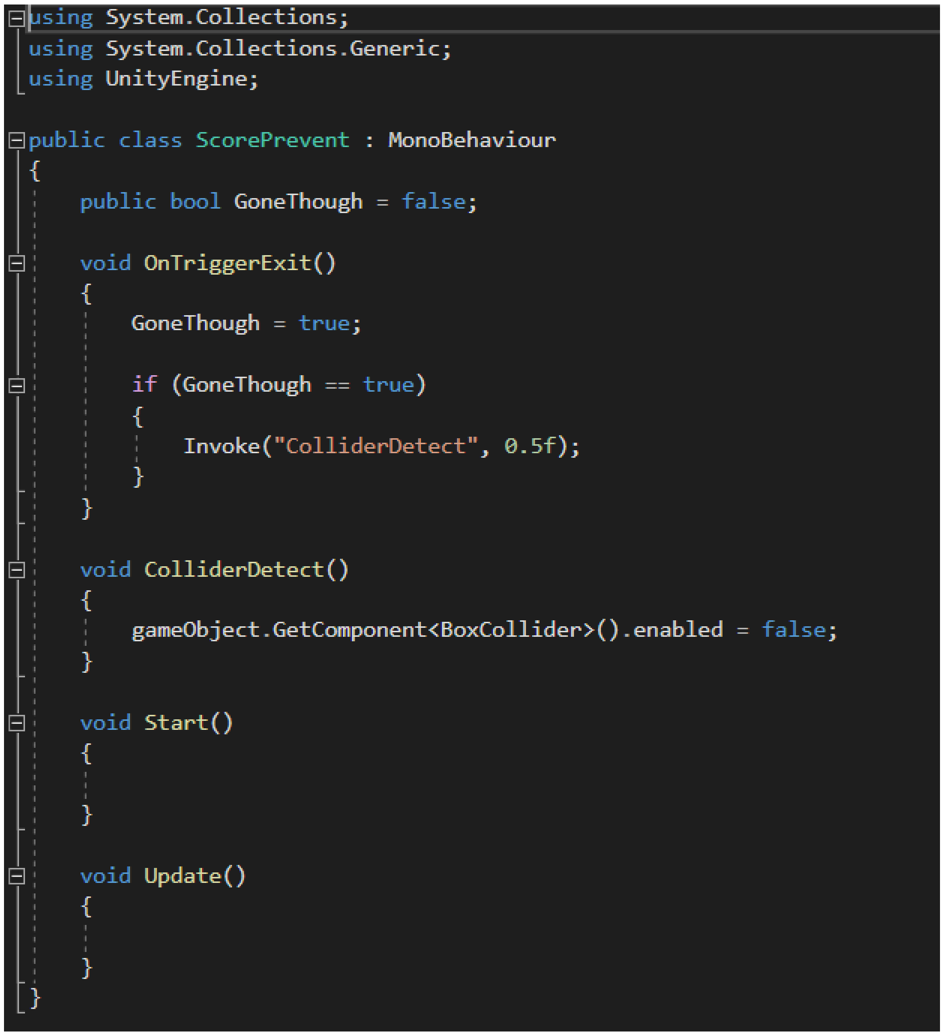
PauseMenu.cs



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

public class ScoreSustem : MonoBehaviour
{
    public float Points = 0f;
    public GameManagerScript Gm;

    void OnTriggerExit()
    {
        Points = Points + 10f;
        Debug.Log(Points);
        gameObject.layer = 2;
    }

    public void Update()
    {
        if (Gm.gameHasEnded == true)
        {
            Points = Points - Points;
        }

        if (Gm.gameHasEnded == true)
        {
            Invoke("LayerSwitch", 1f);
        }
    }

    void LayerSwitch()
    {
        gameObject.layer = 2;
    }
}
```

Figure 21: ScoreSustem.cs

When the escape key is pressed while the game is running, this script is activated. There are 4 main functions that are present within this script, being Resume, Pause, LoadMenu and QuitGame.
Firstly, the Pause function makes the cursor visible to the user and unlocks it to allow for the user to easily select their desired option from the menu. It also changes the UI to the pause menu interface and freezes the game by setting the timescale to zero, which essentially stops time within the gamescene.
The resume function reverses all changes made by the Pause function such as locking and hiding the cursor, resuming time within the game and removing the pause menu UI.
LoadMenu loads the Main Menu scene.
QuitGame quits the game.



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

public class ScorePrevent : MonoBehaviour
{
    public bool GoneThough = false;

    void OnTriggerExit()
    {
        GoneThough = true;

        if (GoneThough == true)
        {
            Invoke("ColliderDetect", 0.5f);
        }
    }

    void ColliderDetect()
    {
        gameObject.GetComponent<BoxCollider>().enabled = false;
    }

    void Start()
    {

    }

    void Update()
    {

    }
}
```

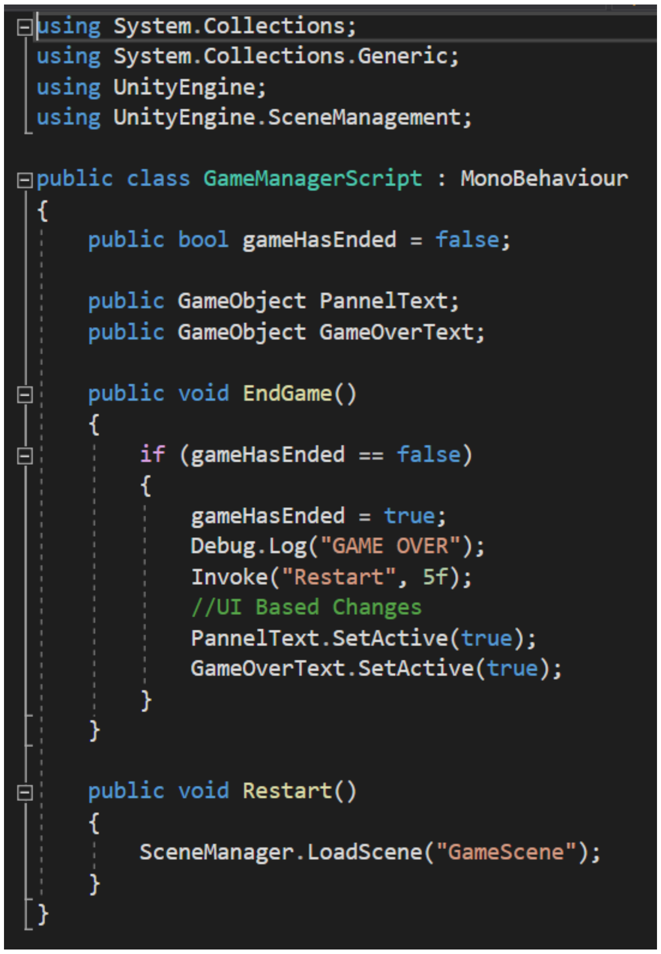
Figure 22: Bug fixing script

3.7 Version 1.0

Version 1.0 finalizes all components and includes minor bug fixes such as preventing the player from being able to interact with two layers at the top of the tower.
This feature prevents the AI or player from disassembling the tower piece by piece from top to bottom while maintaining its stability due to the decreasing height.

This is the final base game without the implementation of artificial intelligence. In terms of functionality, the player is able to individually play and try to reach a high score, or play in turns with another player with them using the same set of inputs. A future multiplayer option may be implemented to allow for jenga matches over the web as well.

V1.0 is a manual play mode, wherein the physics is enabled, but the moves are performed by the human player. The game starts with the standard initial configuration, and the person playing can select a block and perform a move of pulling out the block in the X- or Y- direction, in the coordinate system specified in Fig. 1. The game ends when a block other than the one removed from the stack hits the ground, which is detected using a collision module.



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

public class GameManagerScript : MonoBehaviour
{
    public bool gameHasEnded = false;

    public GameObject PannelText;
    public GameObject GameOverText;

    public void EndGame()
    {
        if (gameHasEnded == false)
        {
            gameHasEnded = true;
            Debug.Log("GAME OVER");
            Invoke("Restart", 5f);
            //UI Based Changes
            PannelText.SetActive(true);
            GameOverText.SetActive(true);
        }
    }

    public void Restart()
    {
        SceneManager.LoadScene("GameScene");
    }
}
```

Figure 23: GameManagerScript.cs

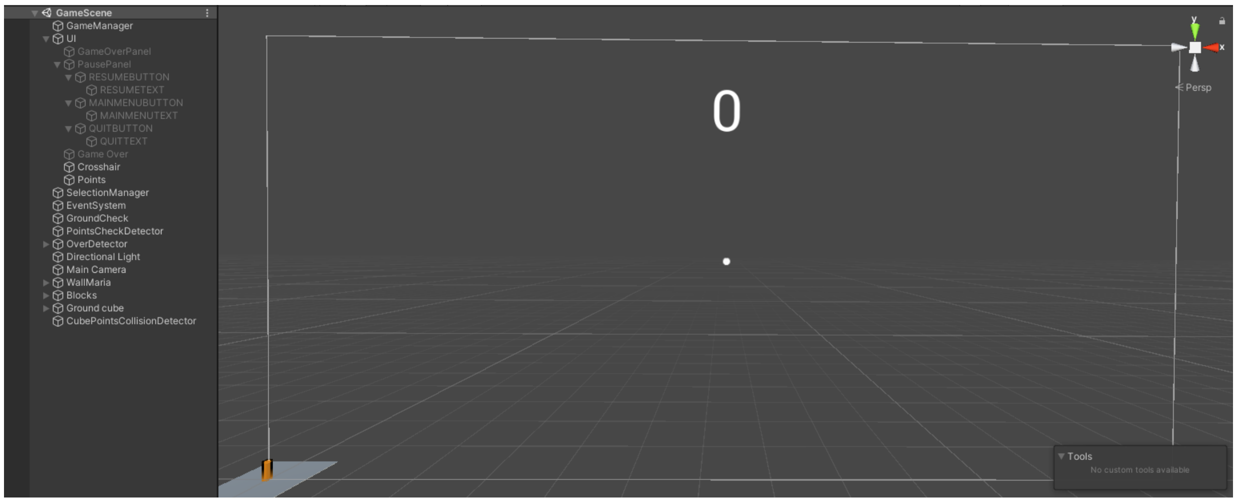
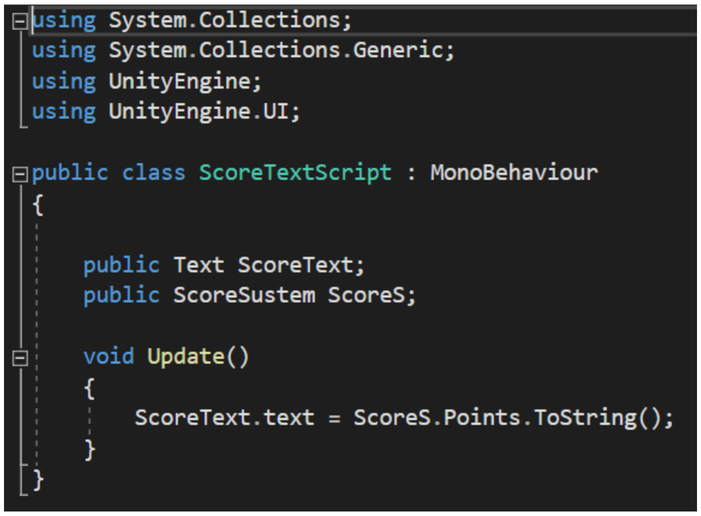


Figure 24: Improvements to user interface



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

public class ScoreTextScript : MonoBehaviour
{

    public Text ScoreText;
    public ScoreSustem ScoreS;

    void Update()
    {
        ScoreText.text = ScoreS.Points.ToString();
    }
}
```

Figure 25: Scrpit for updating score

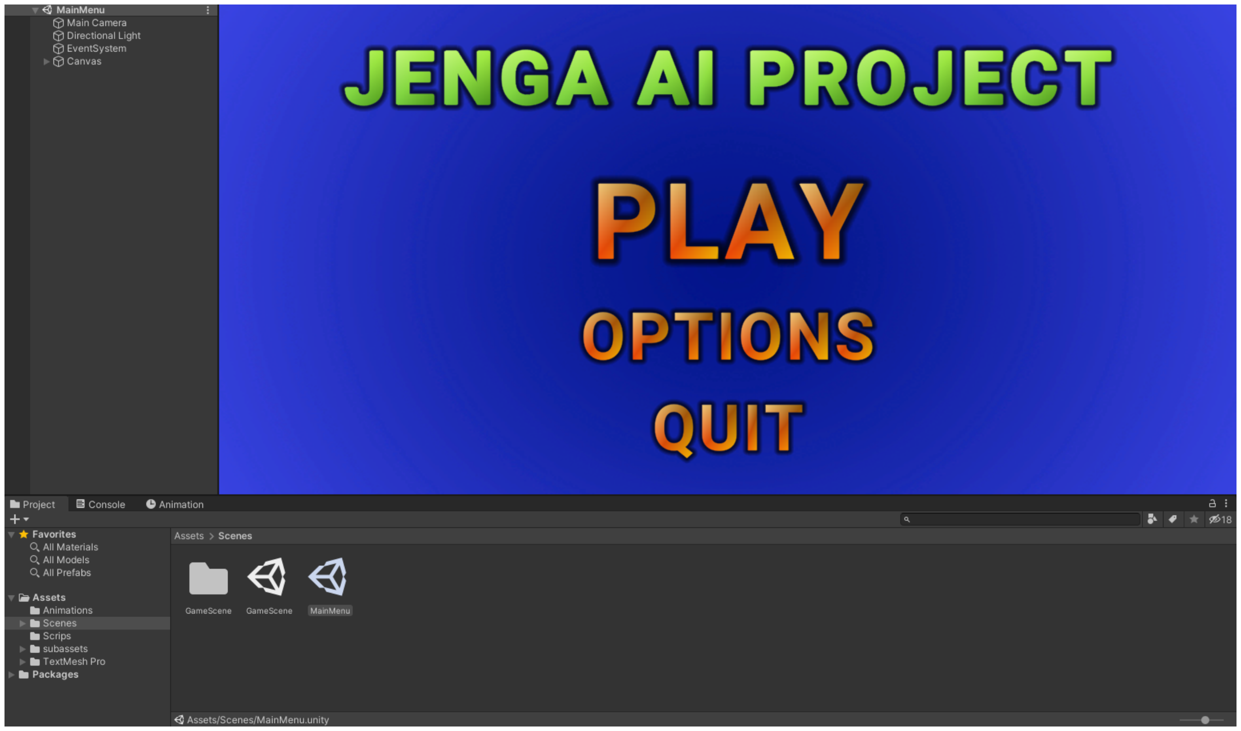


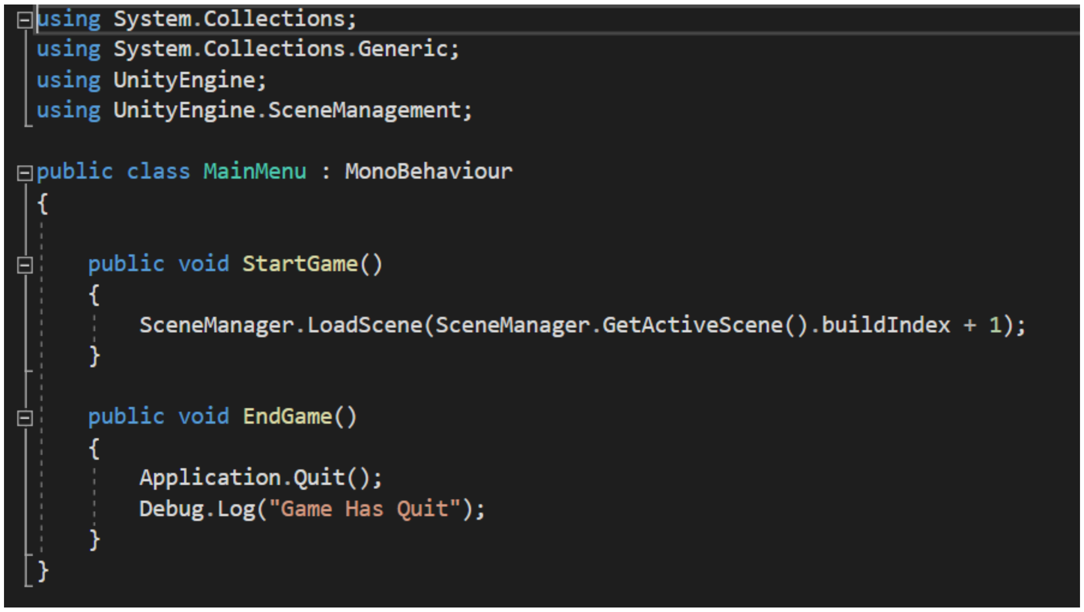
Figure 26: Main Menu Interface

4 *JengaAlpha*

In *JengaAlpha*, we incorporate the Q-learning, so that the machine can learn to perform the moves using the exploration and exploitation. Here we define the states $\vec{S}$ such that the sequence of the moves are taken into account.

We explore the possibility that only the current state is relevant and not the sequence. So the number of states are reduced to 54. To make the state invariant w.r.t. the sequence we define a new state as $\vec{S}' := [sort(\mathcal{S}_i)]$.

We can include variations in the mass and friction in different blocks and check if the Q-learning is able to devise schemes that learn these variations and use them in the play.



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

public class MainMenu : MonoBehaviour
{

    public void StartGame()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
    }

    public void EndGame()
    {
        Application.Quit();
        Debug.Log("Game Has Quit");
    }
}
```

Figure 27: Main Menu Script

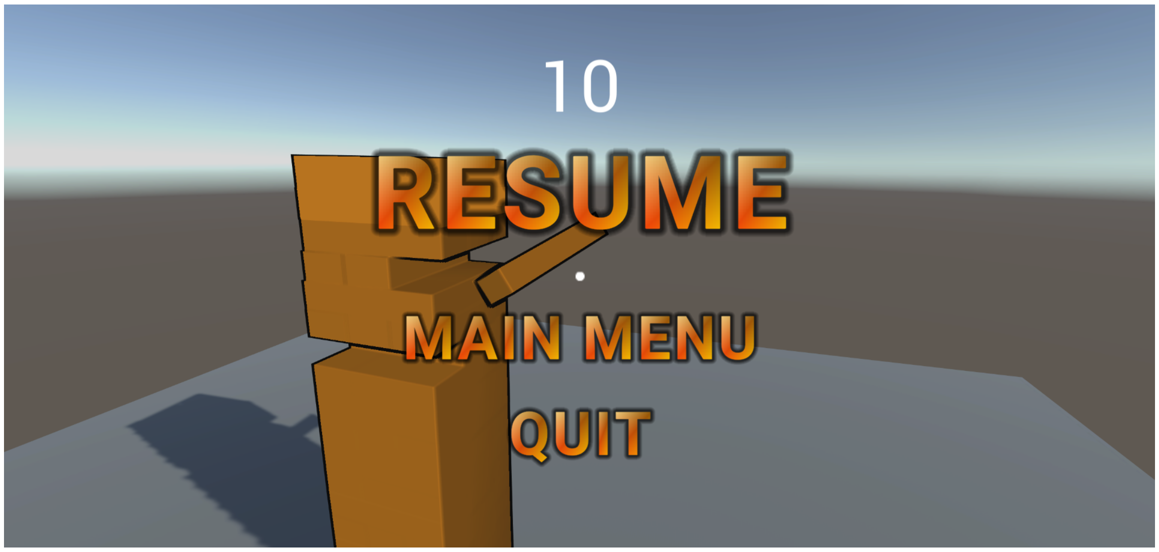


Figure 28: Pause Menu

4.1 Q-Learning

4.1.1 States

We define the state as a 54-bit number, such that each bit represents the presence ($= 1$) or absence ($= 1$) of the block. Hence, the total number of states is $2^{54} \approx 1.8 \times 10^{16}$. Now, if $b_i \in \{0, 1\}$ is the i^{th} bit, with $i \in \{1, 2, ...54\}$, then

$$\mathcal{S} := \sum_{i=1}^{54} b_i \; 2^i \tag{1}$$

By definition, the states are memory-less, i.e., the states don't have any information about the sequence in which the blocks were removed. The presence or absence of the i^{th} block, we can be determined by extracting the state of b_i as

$$\tilde{b}_i = modulo\left[\left(\frac{\mathcal{S}}{2^{i-1}}\right), \; 2\right] \tag{2}$$

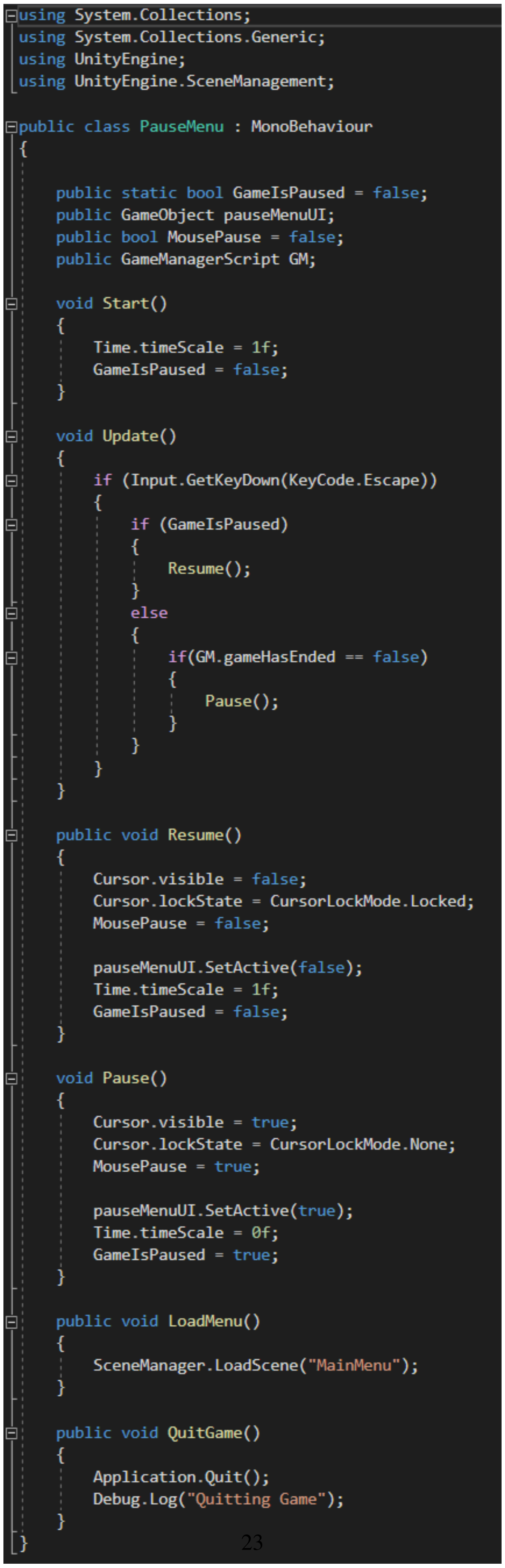
4.1.2 Action

We constrain the blocks to be removed by only sliding it along its long axis. So the action needed to be performed is reduced to a selection of one of the 54 blocks, which is still part of the stack, i.e.,

$$\mathcal{A} := \{i \mid b_i = 1\} \tag{3}$$

5 Conclusion

We have successfully implemented *JengaZero* that enables manual gameplay of Jenga. We have enabled mouse-based interaction to play the game and the points system has been tested. Now, we plan to incorporate the Q-learning module and work on *JengaAlpha*.



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

public class PauseMenu : MonoBehaviour
{

    public static bool GameIsPaused = false;
    public GameObject pauseMenuUI;
    public bool MousePause = false;
    public GameManagerScript GM;

    void Start()
    {
        Time.timeScale = 1f;
        GameIsPaused = false;
    }

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Escape))
        {
            if (GameIsPaused)
            {
                Resume();
            }
            else
            {
                if(GM.gameHasEnded == false)
                {
                    Pause();
                }
            }
        }
    }

    public void Resume()
    {
        Cursor.visible = false;
        Cursor.lockState = CursorLockMode.Locked;
        MousePause = false;

        pauseMenuUI.SetActive(false);
        Time.timeScale = 1f;
        GameIsPaused = false;
    }

    void Pause()
    {
        Cursor.visible = true;
        Cursor.lockState = CursorLockMode.None;
        MousePause = true;

        pauseMenuUI.SetActive(true);
        Time.timeScale = 0f;
        GameIsPaused = true;
    }

    public void LoadMenu()
    {
        SceneManager.LoadScene("MainMenu");
    }

    public void QuitGame()
    {
        Application.Quit();
        Debug.Log("Quitting Game");
    }
}
```

Figure 29: Pause Menu Script