CJ Karinen

Eric Blackburn, Jr.

CS-420

WHACK A ROBOT: Technical Report

Github Repository

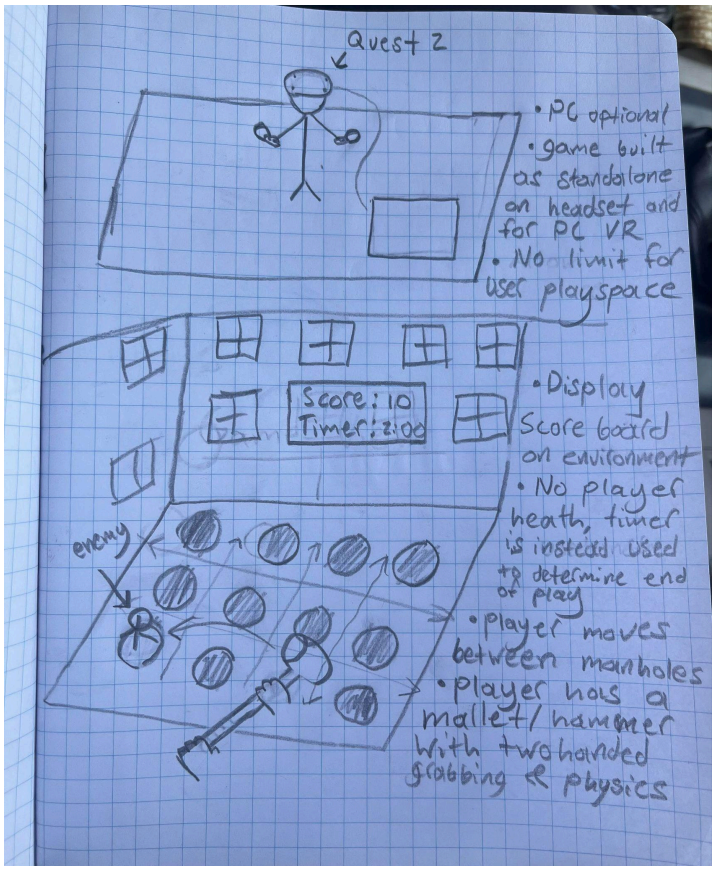
– BASE CONCEPT –

Take the base concept of the “Whack-A-Mole” arcade game and expand it into a larger playspace. Instead of having the playfield being a tabletop in front of the player like in traditional Whack-a-mole, the player inhabits the entire place space. The player will have to navigate the gamespace using direct locomotion and snap-turning, in order to reach each “mole”. Instead of moles, our game will take place in an urban environment with robots taking the place of moles and emerging from manhole covers. Furthermore, the player will utilize a large two-handed mallet which will have proper weight and physics and will be required to physically hit each robot with said mallet. The gameplay duration will be controlled by a two minute timer which will be displayed on the walls of the game environment alongside the score. The player will gain points for hitting bots but will be deducted points for hitting other environmental hazards.

– TECHNICAL REQUIREMENTS –

Our game is to have "Boneworks" style physics in which the player has a full-body IK to where they are able to see their full body and have it match their movements. In addition to having it match their movements, the players hands and limbs will have full collision with the game world and won't "clip" through non-interactive objects as seen in other VR games. Additionally, our game will also utilize controller-based locomotion in which you move around using the controller's left analog stick, while the right analog stick will be used for snap-turning. The hammer will have two handed physics meaning that the hammer will have too much weight to be held with one hand, requiring the second hand to balance it as you would with a real hammer.

– INITIAL DESIGN SKETCHES –



Pictured above, the first sketch (in the upper half of the page) illustrates a potential playspace of our game. There are no specific requirements, and therefore there is no limit as to where and how the user can interact with the game. Due to the type of locomotion (controller-based), players will ideally physically be in the same area throughout. It will be optional to play our game through a PC, as it will also be built as a standalone and can be run either from the headset or through the PC.

In the second sketch, an ideal in-game view is pictured, in which everything described in the base concept is illustrated, including a heads-up-display for the score and timer, as well as the manholes and the hammer the player will be wielding.

– ROMOLES AND GAME MECHANICS –

RoMoles

RoMoles, or **RoMos** are essentially what the 'moles' of our game are. Using the "Robot Sphere" asset by *Razgrizz Demon*, they each have several animations packaged with the asset– Closed, Closed with Eyes, Open, and Standing.

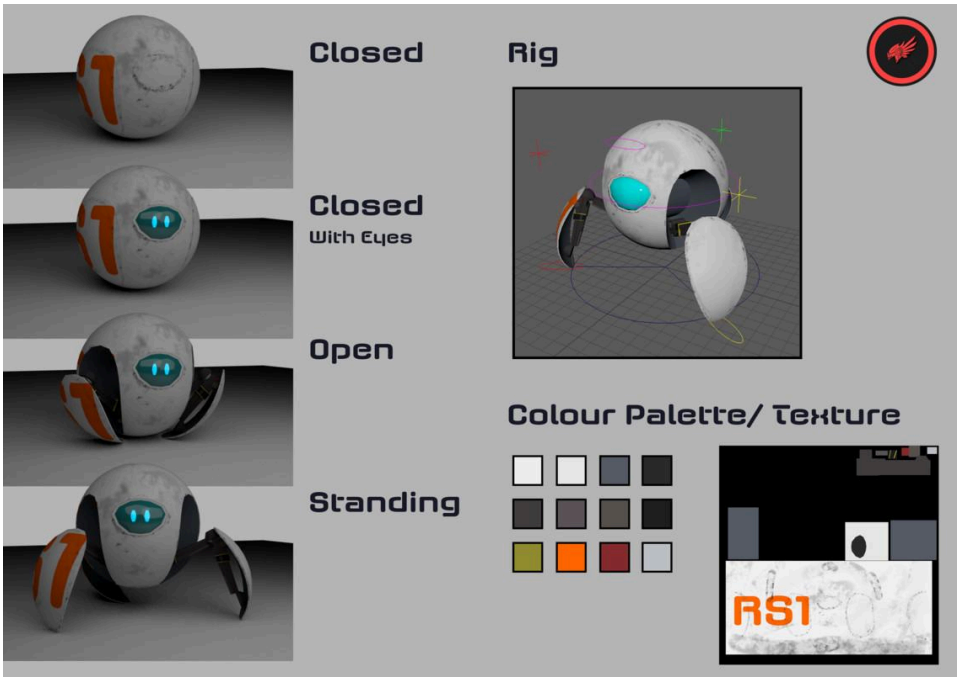


Figure 1: A screenshot showing the "Robot Sphere" asset (from the Unity Asset Store Page)

The "Robot Sphere" asset also comes with two animations: *standing* and *rolling*. For both animations, all states shown in Figure 1 are utilized.

The previous logic for RoMoles is as follows:

```
1   public class RoMo : MonoBehaviour
2   {
3       // PUBLIC VARIABLES
4       public float visibleHeight = 0.2f;
5       public float hiddenHeight = -0.3f;
6       public string difficulty;
7       public float hideSpeed;
8
9       // PRIVATE VARIABLES
10      bool isHit = false;
11      bool vulnerable = false;
12      float hideTimer = 0f;
13      Vector3 targetPosition;
14
15
16      // Start is called before the first frame update
17      void Start()
18      {
19              targetPosition = new Vector3(transform.localPosition.x, hiddenHeight,
transform.localPosition.z);
20              transform.localPosition = targetPosition;
21      }
22
23      // Update is called once per frame
24      void Update()
25      {
26              hideTimer -= Time.deltaTime;
27              if (hideTimer <= 0f && !isHit)
28              {
29                      HideMole();
30              }
31      }
```

```
32
33      public void RiseMole()
34      {
35              // TODO: implement animations here
36              isHit = false;
37              targetPosition = new Vector3(transform.localPosition.x, visibleHeight,
transform.localPosition.z);
38              vulnerable = true;
39              hideTimer = hideSpeed;
40      }
41
42      void HideMole()
43      {
44              // TODO: implement animations here
45              isHit = false;
46              vulnerable = false;
47      }
48
49      void onHit()
50      {
51              isHit = true;
52      }
```

This class starts with the public variables **visibleHeight**, **hiddenHeight**, **difficulty**, and **hideSpeed**. Currently, as the class is not used by the game in its current state, however these represent visible parameters the user will be able to configure from within the editor. **visibleHeight** refers to the area in space where the RoMole will be 'visible' (the area above its respective manhole), and will be considered 'vulnerable'. **hiddenHeight** is vice versa– it refers to the area in space where the RoMole will not be 'visible', and therefore will be considered not 'vulnerable'. **hideSpeed** is a variable that will be used in tandem with **difficulty**, as it is essentially the timer in which the RoMole will shift between the 'hidden' and 'visible' states. The **difficulty** variable itself was intended to be a string (debated to change this to an array of values that the game manages, for example, index 0

for easy, 1 for medium, etc.) which would set the general difficulty of the game. The **hideSpeed** would ideally scale depending on the difficulty selected. For example, if the difficulty is set to an easier level, **hideSpeed** will decrease. In contrast, if the difficulty is set to a harder level, **hideSpeed** would increase. In game, the player was intended to be able to interact with a physical panel (UI) in the virtual world, allowing them to toggle between different difficulty levels and start the timer for the round.

A notable private variable declared on line 10 is the boolean **isHit**. **isHit** will be managed by a work-in-progress collider script for the RoMole. If it is hit by the hammer, the method onHit() on line 49 will set it to true and the RoMole will disappear. In the Update() method, Unity will check for this every frame, and if this is false and the hide timer has been exhausted, the RoMole will, in the future, shift animation states (as seen via comment on line 44) and become hidden. Another else statement will be added to Update(), which will instead call the RiseMole method on line 33. Overall this class was refined to include less clutter (hideSpeed is pointless due to hideTimer, and the logic in Start() could moved to RiseMole), more completeness (HideMole did not set the RoMole's y coordinates to the **hiddenHeight** variable) and more comments.

In the current iteration of the game, the RoMo class was reworked. The current version is shown below:

```
0  using System.Collections;
1  using System.Collections.Generic;
2  using TMPro;
```

```
3  using Unity.Mathematics;
4  using UnityEngine;
5  using UnityEngine.Experimental.GlobalIllumination;
6  using UnityEngine.Rendering;
7  using UnityEngine.XR.Interaction.Toolkit.AffordanceSystem.Receiver.Primitives;
8
9  public class RoMo : MonoBehaviour
10 {
11      // PUBLIC VARIABLES
12
13      public float visibleHeight = 0f;
14      public float hiddenHeight = -0.8f;
15      public float hideTimer = 50f;
16      public float speed = 1f;
17      public bool isHit = false;
18      public RoStates state;
19      public ParticleSystem hitEffect;
20
21      public AudioClip hitSound;
22      public AudioClip[] hitSounds;
23
24
25      // PRIVATE VARIABLES
26
27      Vector3 targetPosition;
28      Animator anim;
29      AudioSource audioSource;
30
31
32      private void Awake()
33      {
34      anim = gameObject.GetComponent<Animator>();
35      anim.speed = 4;
36      }
37
38      // Start is called before the first frame update
39      void Start()
40      {
41
42      audioSource = GetComponent<AudioSource>();
43
44      }
45
```

```
46      // Update is called once per frame
47      void Update()
48      {
49
50      if (this.state == RoStates.VISIBLE)
51
52      {
53      anim.SetBool("Roll_Anim", false);
54      anim.SetBool("Open_Anim", true);
55
56      targetPosition = new Vector3(transform.localPosition.x, visibleHeight,
transform.localPosition.z);
57      transform.localPosition = Vector3.MoveTowards(transform.localPosition,
targetPosition, speed);
58      }
59
60      if (state == RoStates.HIDING)
61
62      {
63      anim.SetBool("Open_Anim", false);
64      anim.SetBool("Roll_Anim", true);
65
66      targetPosition = new Vector3(transform.localPosition.x, hiddenHeight,
transform.localPosition.z);
67      transform.localPosition = Vector3.MoveTowards(transform.localPosition,
targetPosition, speed);
68
69      }
70
71
72
73
74      }
75
76
77      public enum RoStates
78      {
79      HIDING,
80      VISIBLE,
81      }
82
83
84      private void OnTriggerEnter(UnityEngine.Collider other)
```

```
85      {
86      if (other.gameObject.CompareTag("HammerHead"))
87      {
88
89
90      if (state == RoStates.VISIBLE)
91
92      {
93              onHit();
94      }
95
96
97      }
98
99
100
101
102     }
103
104
105
106     public void RiseMole()
107     {
108
109
110     this.state = RoStates.VISIBLE;
111     }
112
113     public void HideMole()
114     {
115     this.state = RoStates.HIDING;
116
117     }
118
119     void onHit()
120     {
121     Debug.Log("boop");
122
123     audioSource.mute = false;
124
125     AudioClip randomHitSound = hitSounds[UnityEngine.Random.Range(0,
hitSounds.Length)];
126
```

```
127    audioSource.PlayOneShot(randomHitSound);
128
129    hitEffect.Play();
130
131    }
132
133
134
135    // Coroutine for testing
136    IEnumerator Peek()
137    {
138    Debug.Log("Started Coroutine at timestamp : " + Time.time);
139    while (true)
140    {
141            yield return new WaitForSeconds(6);
142            RiseMole();
143            Debug.Log("Mole visible");
144            yield return new WaitForSeconds(6);
145            HideMole();
146            Debug.Log("Mole hidden");
147    }
148    }
149}
150
151
```

As previously discussed, the RoMo class was reworked for the current build of the game. It makes use of new states , **HIDING** and **VISIBLE** (starting at Line 50) to determine logic for what a RoMo should do in each state. RoMos also now animate upon shifting between these two states, and also can detect a collision from the hammer (asset shown in Figure 2 below), and upon doing so will play a random sound effect from a random index in the new variable hitSounds. Useless or obsolete variables were removed, efficiency was added by moving core game logic (such as difficulty) to the Game Manager, and RoMos are now in a

near-complete state. RoMos are notably lacking the ability to destroy upon collision and respawn (a planned feature that grew outside the scope of our time constraints).

Figure 2: A screenshot showing the "Hammer PBR" asset that is being used. By BasicCore (from the Unity Asset Store Page)

Game Manager

The script for the Game Manager is shown below:

```
1 using System;
2 using System.Collections;
3 using System.Collections.Generic;
4 using TMPro;
```

```
5  using Unity.VisualScripting;
6  using UnityEditor.ShaderGraph.Internal;
7  using UnityEngine;
8  using UnityEngine.UI;
9
10 public class GameManager : MonoBehaviour
11 {
12      //public MainMenuController mmc;
13
14
15      // public Canvas mainCanvas;
16
17      public RoMo[] romos;
18
19      public float minWait = 0.5f;
20      public float maxWait = 1.5f;
21      public float chanceToRise = 1f; // 100% chance to rise
22      public float chanceToHide = 0.5f; // 50% chance to hide
23
24      public float gameTime;
25      public bool started = false;
26
27
28
29      private int score = 0;
30      private int pointsOnHit = 20;
31
32
33      // Start is called before the first frame update
34      void Start()
35      {
36
37      if (!started) return;
38
39      else
40      {
41      StartCoroutine(RomoLoop());
42      // StartCoroutine(GameTime());
43      }
44
45      //scoreLabel.text = score.ToSafeString();
46
47      }
```

```
48
49      // Update is called once per frame
50
51      private void Update()
52      {
53
54      // LINQ for iterating through the array of RoMos
55
56      foreach (RoMo romo in romos)
57      {
58
59      if (romo.isHit)
60      {
61              onMoleHit();
62
63      }
64
65      }
66      }
67
68
69      IEnumerator GameTime()
70      {
71
72
73      /*
74      if (mmc.difficulty == 0)
75      {
76              // Easy - 3 Minutes
77
78              gameTime = 240f;
79      }
80      else if (mmc.difficulty == 1)
81      {
82              // Medium - 2 Minutes
83
84              gameTime = 120f;
85      }
86      else
87      {
88              // Hard - 1 Minute
89
90              gameTime = 60f;
```

```
91      }
92
93      */
94
95      gameTime = 240f;
96
97      while (gameTime > 0)
98      {
99      yield return new WaitForSeconds(1);
100             gameTime--;
101     }
102
103
104}
105
106     IEnumerator RomoLoop()
107     {
108     while (true)
109     {
110
111             foreach (RoMo romo in romos)
112             {
113
114             if (UnityEngine.Random.value < chanceToRise)
115             {
116             romo.RiseMole();
117             }
118
119             if (UnityEngine.Random.value < chanceToHide)
120             {
121             romo.HideMole();
122             }
123
124             }
125             yield return new WaitForSeconds(UnityEngine.Random.Range(minWait,
maxWait));
126     }
127
128     }
129     private void onMoleHit()
130
131     {
132     AddScore(pointsOnHit);
```

```
133     }
134
135     public void AddScore(int points)
136     {
137     score += points;
138
139     }
140}
```

In this class, central game mechanics are handled. Previously, the movement of RoMos was handled in its own respective class, however, nuanced randomizing was originally not considered. This was resolved by using an array of RoMos and using a LINQ (Language Integrated Query) to iterate through them and using two new variables (chanceToRise and chanceToRise). This allows for RoMos to randomly rise and hide, irrespective of the order in which they appear.

Game Manager makes use of two gameplay loops: RomoLoop and GameTime. RomoLoop provides the functionality as mentioned before, and GameTime decrements the time for a round. The currently unused variable mmc, or Main Menu Controller, was intended to be a menu in which the player would select a difficulty for a round. The Game Manager would use this to grab information about the difficulty and set it accordingly. The difficulty menu was omitted due to time constraints.

Further Plans

Some more ambitious goals for the RoMoles that could introduce a unique feature of the gameplay would be for there to be RoMoles of different types. For example, the basic RoMole works as intended, with no twists. A different type of RoMole (also shown with a different material/color) could add double points to the score when hit. Also, there could be another type of RoMole with deducts from the time and/or score when it is hit. Adding these different types of RoMoles will add complexity to the project, and in turn, could take more time to develop, but ideally adding these could add more interesting twists to the game.

– VR ENVIRONMENT AND NAVIGATION –

Initially, the XR Rig was configured and ready to use but upon testing, it was discovered that the hands were not tracking. Eventually this step was resolved through an indistinct series of steps mainly involving resolving errors in the XR tab of the project settings menu and by ensuring that all of the XR interaction components for the Controllers and Headset were configured properly. This is demonstrated below and showcases all of the progress that we've made when compared to our current build.

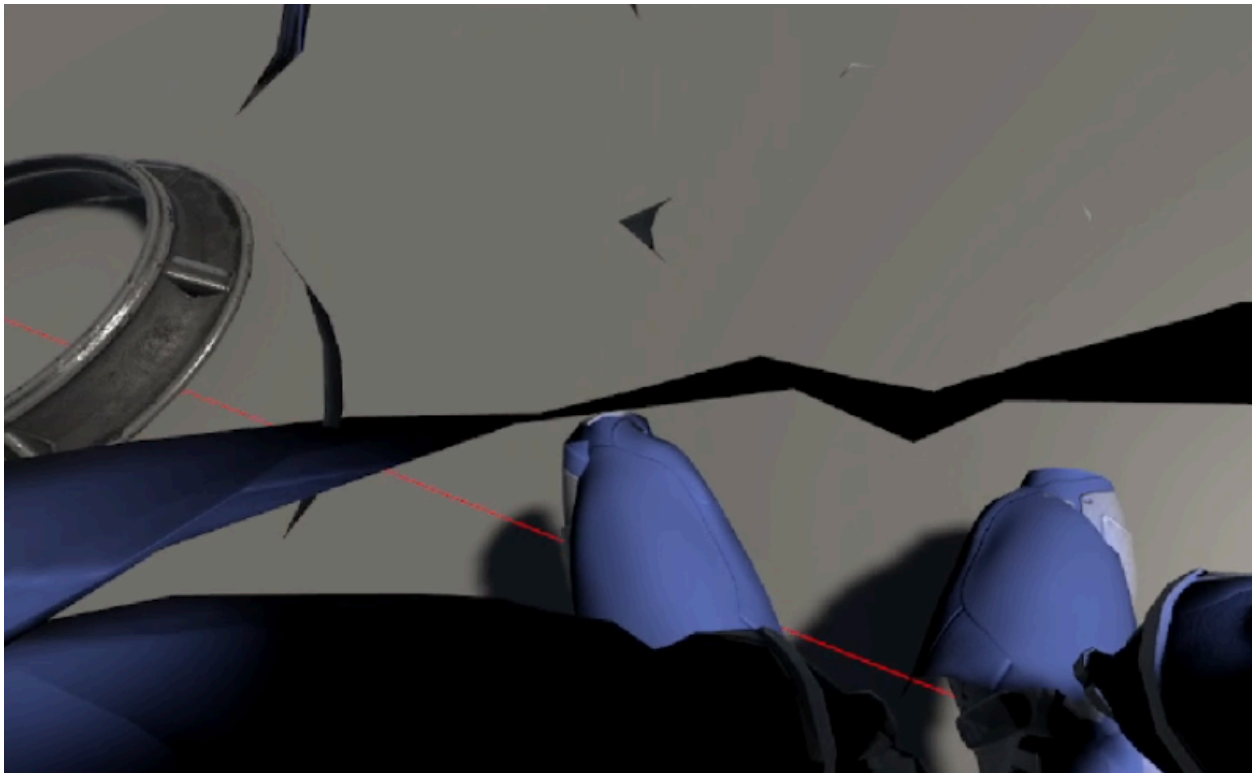


Figure 3: A view of the player in first-person perspective. The body is shown, however, the camera is colliding with the model. This is how it looked in submission 2.

To explain the VR rig, it is actually a parent object which houses three interconnected subsystems. The first rig is the IK rig, which houses the IK target and hint objects for each limb, allowing for us to have connected arm and leg joints that respond to the rest of the body's position. The second system is the XR rig which houses the tracking information for our controllers and headset which is utilized in a script on the IK rig to tether the movement of the IK rig to the controllers and headset so that the connected joints of the IK skeleton move realistically according to the position and rotation of these three factors. The last system is the physics rig which houses rigidbodies, colliders and connected joints for the head, body, and controllers. To pair with the physics system, I added colliders and kinematic rigidbodies to all of the reachable walls in the environment. The controller objects in the XR rig contain XR Direct Interactor objects which have custom attach transforms added to work best with tracking our hammer grabbing. On that note, the hammer also contains rigidbodies which are used for collision against the environment, player and moles.

The hammer contains an XR direct interactor which uses dynamic attach to allow grabbing anywhere on the handle, as well as two attach transforms on the top and bottom of the handle to which the players hands will snap into place upon grabbing. The player is controlled via direct movement in a custom script in order to utilize the rigidbody.move function in order to maintain collision. It was learned early on that by attempting to use physics colliders such as ours with the included XR Direct Locomotion component you will bypass the collision system and will not have collisions. This custom system uses

rigidbody.move to deliver direct movement while also allowing for continuous turning. Originally, the plan was to allow for the option of with snap-turn or continuous but the XR toolkit's snap-turn component was not usable due to the same physics issue mentioned above. Furthermore, I was unable to find a way to custom write a (simple) snap-turn method using our rigidbody collision system so I decided to leave it to only continuous movement for this submission. Below I have included the code from the direct movement script that was created:

VRMovementController

```
1   public float speed = 1;
2   public float continuousTurnSpeed = 60;
3
4   public ActionBasedSnapTurnProvider snapTurnProvider;
5
6   public InputActionProperty moveInputSource;
7   public InputActionProperty turnInputSource;
8
9   public Rigidbody rb;
10
11   public LayerMask groundLayer;
12
13   public Transform directionSource;
14   public Transform turnSource;
15
16   public CapsuleCollider bodyCollider;
17
18   private Vector2 inputMoveAxis;
19   private float inputTurnAxis;
```

```
20
21   void Update()
22   {
23      inputMoveAxis = moveInputSource.action.ReadValue<Vector2>();
24      inputTurnAxis = turnInputSource.action.ReadValue<Vector2>().x;
25   }
26
27   private void FixedUpdate()
28   {
29      bool isGrounded = CheckIfGrounded();
30
31      if(isGrounded)
32      {
33         Quaternion yaw = Quaternion.Euler(0, directionSource.eulerAngles.y, 0);
34         Vector3 direction = yaw * new Vector3(inputMoveAxis.x, 0, inputMoveAxis.y);
35
36         Vector3 targetMovePosition = rb.position + direction * Time.fixedDeltaTime *
speed;
37
38         Vector3 axis = Vector3.up;
39
40            float angle = continuousTurnSpeed * Time.fixedDeltaTime * inputTurnAxis;
41
42            Quaternion q = Quaternion.AngleAxis(angle, axis);
43
44            rb.MoveRotation(rb.rotation * q);
45
46            Vector3 newPosition = q * (targetMovePosition - turnSource.position) +
turnSource.position;
47
48            rb.MovePosition(newPosition);
49      }
50   }
51   public bool CheckIfGrounded()
52   {
53      Vector3 start = bodyCollider.transform.TransformPoint(bodyCollider.center);
54      float rayLength = bodyCollider.height / 2 - bodyCollider.radius + 0.05f;
55
56      bool hasHit = Physics.SphereCast(start, bodyCollider.radius, Vector3.down, out
RaycastHit hitInfo, rayLength, groundLayer);
57      return hasHit;
```

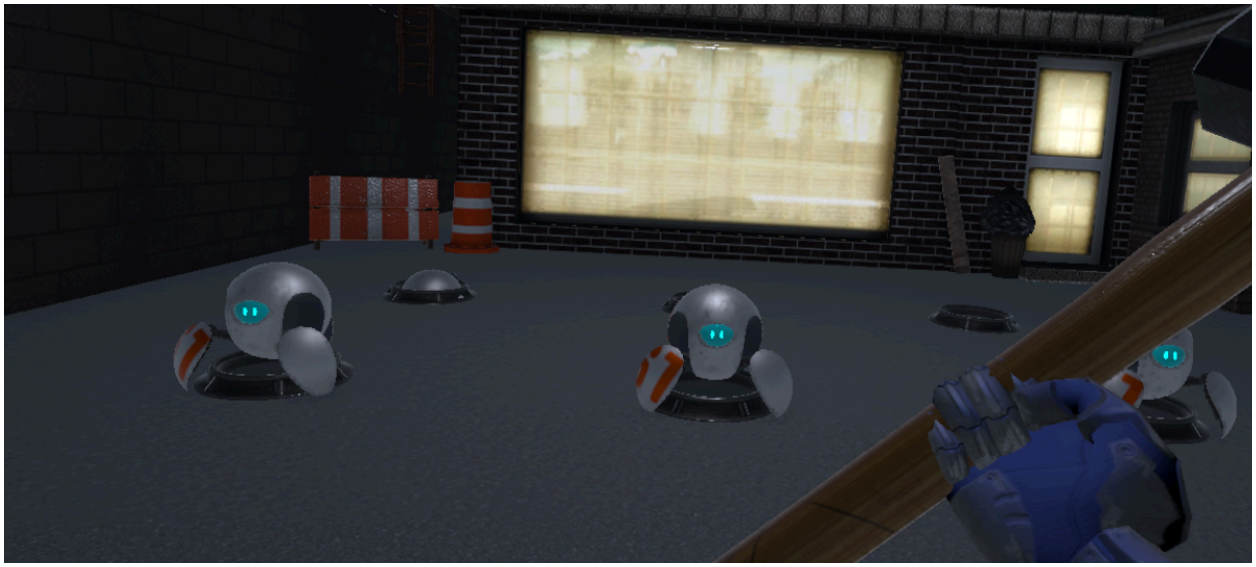


Figure 4: First person view of the current XR Rig.

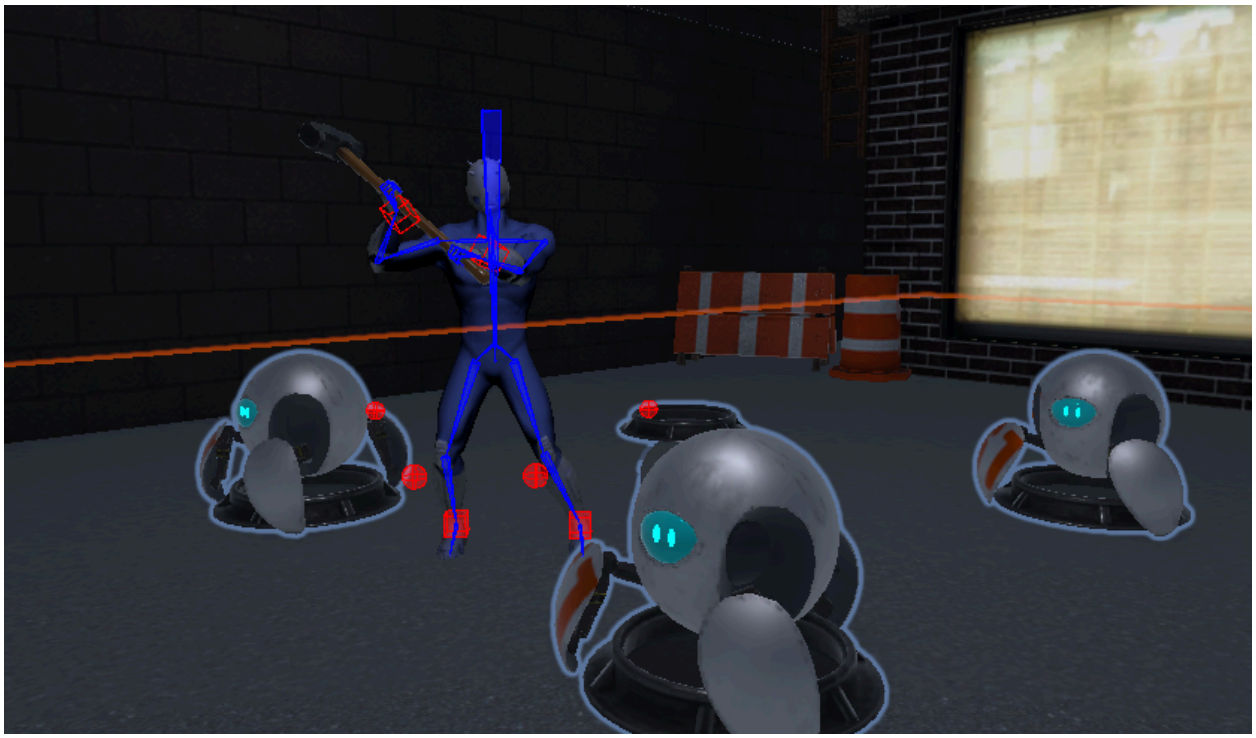


Figure 5: Third person view of the current XR Rig.

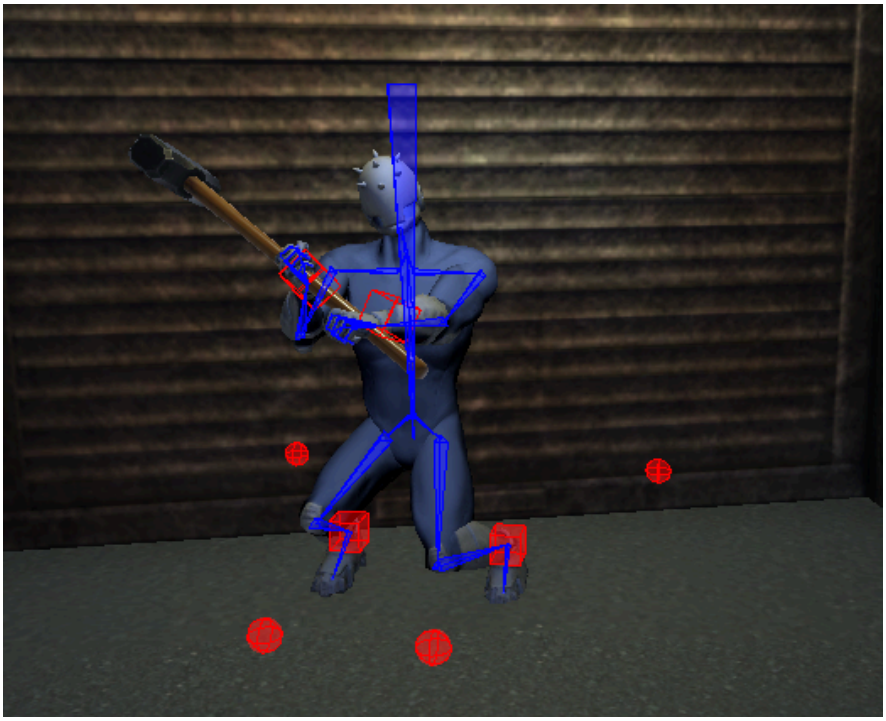


Figure 6: Demonstrating a crouching position of the XR Rig.

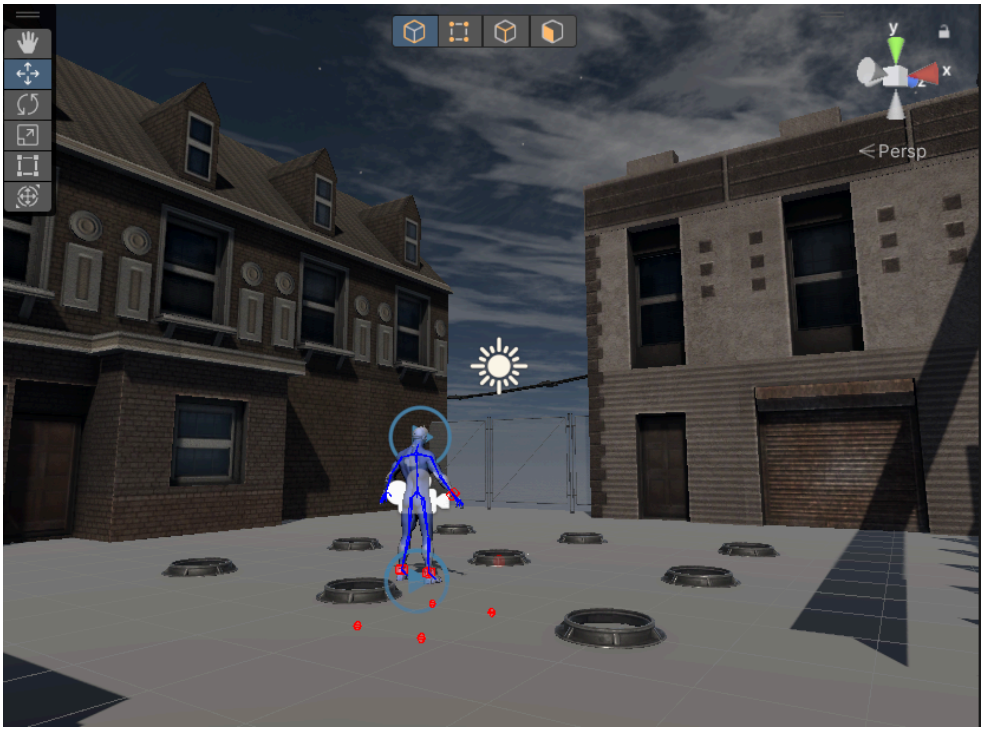


Figure 7: A look at the old virtual environment, from Angle A.

Figure 8: A look at the old virtual environment, from Angle B.

Our virtual environment makes major use of the “Town Constructor Pack” asset by *Purple Jump*. With our avatar starting out in the center (clearly pictured in Figure 5), the environment will take place in an urban area, as shown in the initial sketch for the game. Note that the manholes (pictured more clearly in Figure 5) are arranged in a 3x3 grid. These manholes are where the RoMoles will reside. The asset used for the manholes is “Mini Pack: Manholes” by *Warren Marshall Biz*. This means that there are 9 RoMoles in total that disappear and reappear at different slots in random intervals (shown in the Game Manager).

Currently UI elements are in development, though there will essentially be two panels– one for difficulty and round starting, and another for tracking the time and the score. Previously, a HUD was considered for the score and the time, however, for VR this may not be optimal. Instead, a diegetic alternative (a viewable UI element that can be seen in the game world) will be mounted on a building of the environment, where the player can loop up to see the score and time as they play the game. Similarly, the UI for starting a round was planned to be added near the manholes, where the player will be able to walk up to it and interact with it.

The entire play area was designed to emulate the feeling of a city alleyway at night, with the entire scene being lit mostly by moonlight, illuminated windows of buildings and other ambient light sources.

The environment has been updated significantly (Shown in Figures 9 and 10). While not making the most logical sense in real city planning, the game area being a square alleyway is in order to incorporate the required playspace for whack-a-mole into the game's urban aesthetic. The bounds of the map are the edges of the buildings, while the two entrances of the alley are blocked off by barriers and fencing. Beyond these physical barriers, the environment continues for all the visible areas not reachable to the player. This was done to further the illusion that the game map takes place in a tangible world and isn't just a floating plane (which it actually is). Assets were strategically placed out of bounds to give the illusion of distant buildings just beyond the alley. In addition to this, a 3D spatial sound source was placed in our street facade that resides out of the playable area and emits sounds of street ambience (people talking, walking, cars honking).

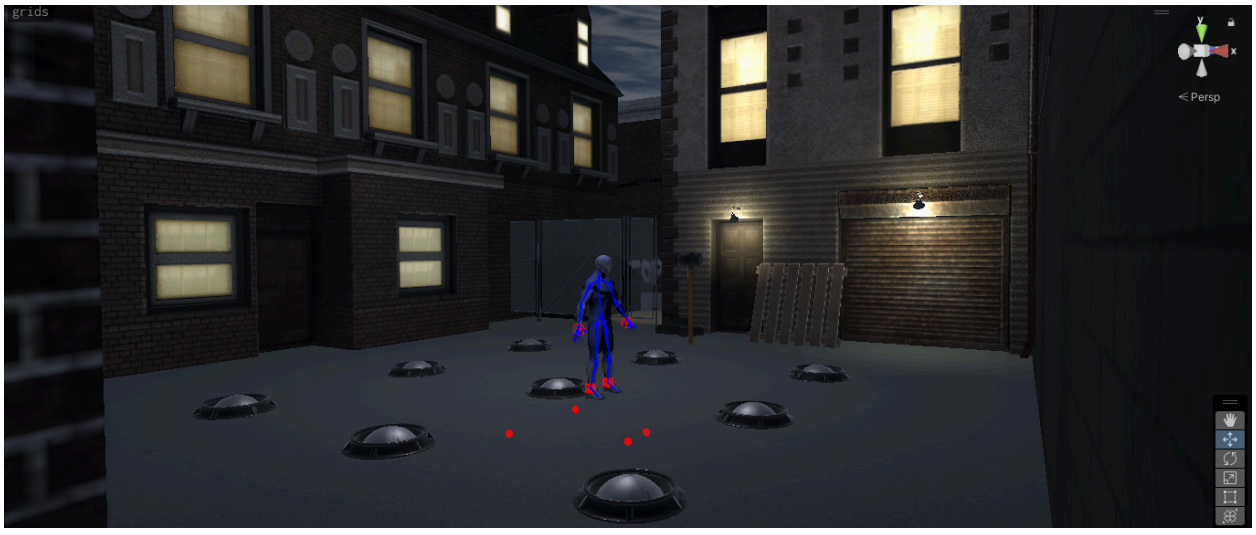


Figure 9: The current virtual environment, from Angle A.

Figure 10: The current virtual environment, from Angle B.

A new start scene (shown in Figure 11) was also implemented to start the game. It makes use of an interactable menu which allows the user to start the game, options (which was intended to have more functionality. The Start button will transition into the main gameplay scene, which was previously seen in Figures 9 and 10. Quit is set up to quit the game to the desktop when it is pressed.

Figure 11: The Start Scene

App Modules – The table below shows the app modules for this project.

Key:

Added

Omitted/Not present

Present but not fully implemented

Module/Asset	Purpose, Source (if any)
Probuilder	Level builder included in Unity Professional used for prototyping level designs and simple geometry.
ProGrids	Complementary to Probuilder, adds grid based object snapping.
XR Interaction Toolkit	For user VR interaction/controller input.
Character Controller	For the player avatar.
UI Elements	For a panel that gives players information about the score and the time, as well as setting the difficulty and starting a round.
Background Music/Ambience	May be optional, but to give more flair to the game and to make gameplay more enjoyable.
Audio Cues	To indicate when a RoMole has hidden and when it has appeared. Also, when a RoMole has been hit with the hammer.
Town Constructor Pack	For building the map. *Purple Jump*

Mini Pack: Manholes	For building the map. *Warren Marshall Biz*
Hammer PBR	For interacting with the RoMoles ('whacking' them). *BasicCore*
Robot Sphere	The model used for the RoMoles. *Razgrizz Demon*