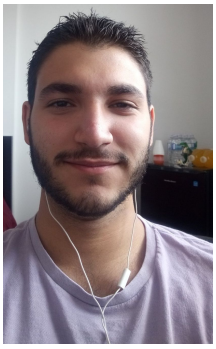
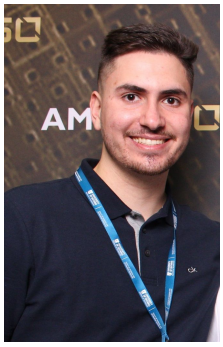
INFR 3110U – Game Engine Design & Implementation
Assignment 2 – Profiling, Tool Development

Giulia Santin
100657351



Noah Colangelo
100659538

Regan Simmonds
100651217



Dimitris Stefanakos
100621470

Adam Smith
100577594

Max Kemp
100671306

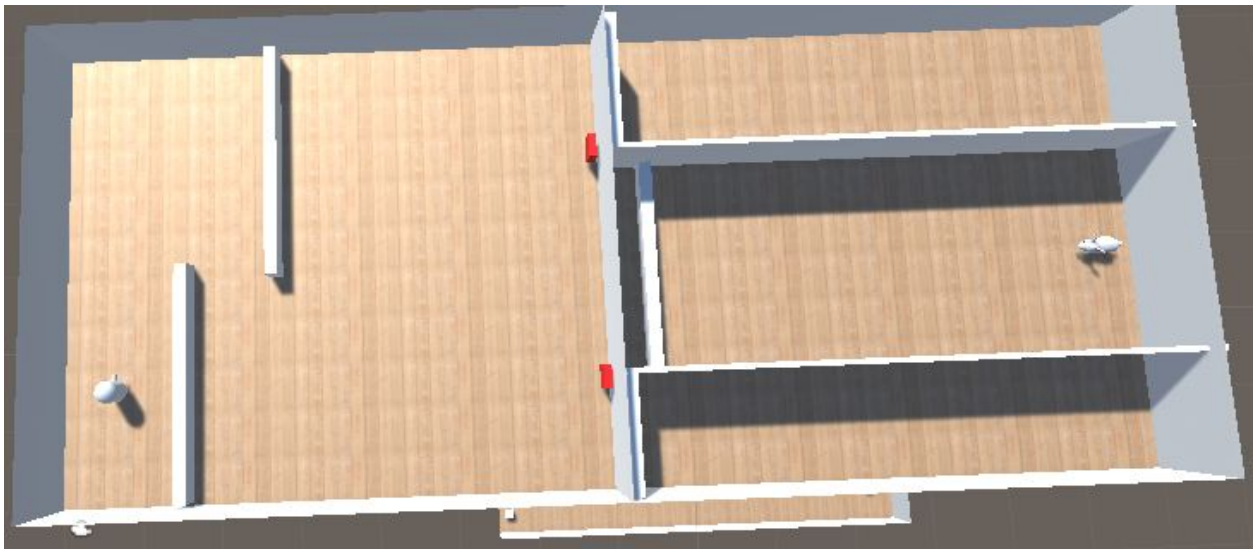
Kevin Lounsbury
100654226

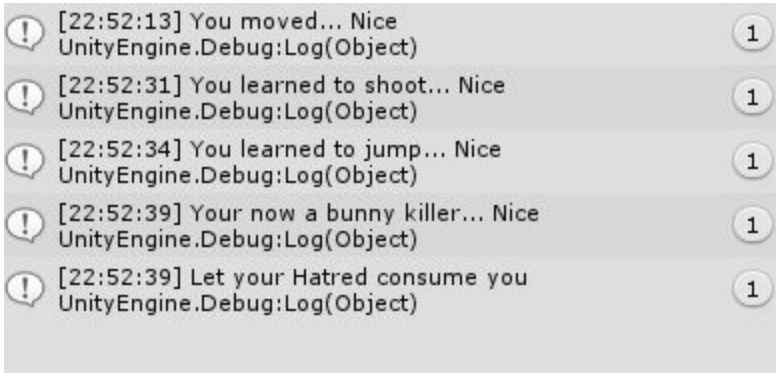
TABLE OF CONTENTS

BASE:

Designing the tutorial:

For this assignment, we decided to create a tutorial level, in which the player has to navigate through the walls, which will fall over, and they will have to jump over the obstacle in order to shoot the bunny in the scene. Once it's shot and eliminated, the game will begin.





Controls:

Walk - WASD

Jump - Spacebar

Shoot - Left mouse click

PART 1: DESIGN PATTERNS

Object Pooling:

This game design pattern has been implemented in the project for both the bullet pool and enemy pool. We used an object pool to avoid creating instances of the class by reusing them instead. To do so, a client will access the object pool for an object that has already been instantiated. An observer is used to observe certain objects and classes, to notify classes when certain events have happened. This allows classes to interact with one another, without being coupled to each other.

Below are screenshots of the enemy pool. The enemy pool notifies the achievement manager when an enemy has been killed, that way the number of enemies killed can be tracked and will activate the adrenaline rush after a set number of enemies have been killed.

```
8 references
public class EnemyPool : Subject
{
    [SerializeField]
    int _poolSize = 40;
    Queue<GameObject> _enemyPool;
    Factory _factory;
    static EnemyPool _instance;


    2 references
    public static EnemyPool Instance
    {
        get
        {
            if (_instance == null)
                _instance = new EnemyPool();

            return _instance;
        }
    }


    1 reference
    private EnemyPool()
    {
        _enemyPool = new Queue<GameObject>(_poolSize);
        _factory = Factory.Instance;

        GameObject enemy;

        for (int i = 0; i < _poolSize; ++i)
        {
            _factory.CreateGameObject(ObjectTypes.Bunny, out enemy);
            enemy.SetActive(false);
            _enemyPool.Enqueue(enemy);
        }

        addObserver(AchievementManager.Instance);
    }
```

```
1 reference
public GameObject getEnemy()
{
    GameObject obj = null;

    if (_enemyPool.Count > 0)
    {
        obj = _enemyPool.Dequeue();
        obj.SetActive(true);
    }

    return obj;
}

1 reference
public void recycleEnemy(GameObject enemy)
{
    if (enemy && (_enemyPool.Count < _poolSize))
    {
        enemy.SetActive(false);

        _enemyPool.Enqueue(enemy);

        notify(enemy, ObsEvent.ADRENALINE_RUSH);
    }
}
```

Next are the screenshots for the bullet pool.

```
9 references
public class BulletPool : MonoBehaviour
{
    [SerializeField]
    private GameObject _bullet;
    private List<GameObject> _inactivePool;
    private List<GameObject> _activePool;
    [SerializeField]
    private int _size = 20;
    private Transform _bulletSpawn;
    private static BulletPool _instance;

    private static int shotsFired;

    3 references
    public static BulletPool Instance
    {
        get
        {
            if (!_instance)
            {
                GameObject obj = new GameObject("BulletPool");
                _instance = obj.AddComponent<BulletPool>();
            }

            return _instance;
        }
    }

    0 references
    private void Awake()
    {
        if (!_instance && (!_instance == this))
            _instance = this;

        _bullet = Resources.Load("Prefabs/Bullet") as GameObject;
        _bulletSpawn = transform.Find("BulletSpawn");

        // Initiate bullet pools
        _inactivePool = new List<GameObject>();
        _activePool = new List<GameObject>();
        GameObject obj;

        for (int i = 0; i < _size; ++i)
        {
            obj = Instantiate(_bullet, _bulletSpawn.transform.position, Quaternion.identity);
            _inactivePool.Add(obj);

            obj.hideFlags = HideFlags.HideInHierarchy;
```

```
39          // Initiate bullet pools
40          _inactivePool = new List<GameObject>();
41          _activePool = new List<GameObject>();
42          GameObject obj;
43
44          for (int i = 0; i < _size; ++i)
45          {
46              obj = Instantiate(_bullet, _bulletSpawn.transform.position, Quaternion.identity);
47              _inactivePool.Add(obj);
48
49              obj.hideFlags = HideFlags.HideInHierarchy;
50              obj.SetActive(false);
51          }
52
53          shotsFired = 0;
54      }
55
56      2 references
57      public static int ShotsFired
58      {
59          get
60          {
61              return shotsFired;
62          }
63          set
64          {
65              shotsFired = value;
66          }
67      }
68      1 reference
69      public GameObject getBullet()
70      {
71          GameObject bullet;
72          shotsFired++;
73
74          // Use from the inactive pool
75          if (_inactivePool.Count > 0)
76          {
77              bullet = _inactivePool[_inactivePool.Count - 1];
78              _inactivePool.RemoveAt(_inactivePool.Count - 1);
79              _activePool.Add(bullet);
80          }
81          // Use from the active pool
82          else
83          {
84              bullet = _activePool[0];
85              GameObject temp = _activePool[_activePool.Count - 1];
86              _activePool[0] = temp;
87              _activePool[_activePool.Count - 1] = bullet;
```

```
        // Reset the current bullet
        bullet.transform.position = _bulletSpawn.transform.position;
        bullet.transform.rotation = _bulletSpawn.transform.rotation;
        bullet.GetComponent<Rigidbody>().velocity = Vector3.zero;

        return bullet;
    }

    2 references
    public void reclaim(GameObject bullet)
    {
        // Reclaim the current bullet.
        _inactivePool.Add(bullet);

        if (_activePool.Count > 1)
        {
            GameObject temp = _activePool[_activePool.Count - 1];
            _activePool[_activePool.IndexOf(bullet)] = temp;
        }
        _activePool.RemoveAt(_activePool.Count - 1);

        bullet.SetActive(false);
        bullet.GetComponent<Bullet>().Active = false;
        bullet.GetComponent<Bullet>().CurrTime = 0.0f;
    }
}
```

Observer Pattern:

This design pattern has been implemented in the project in order to trigger events.

```
public void onNotify(GameObject obj, ObsEvent _event)
{
    switch (_event)
    {
        case ObsEvent.PLAYER_MOVED:
            Debug.Log("You moved... Nice");
            PlayerMoved();
            break;
        case ObsEvent.TARGETS_DOWN:
            Debug.Log("You learned to shoot... Nice");
            TargetsDown();
            break;
        case ObsEvent.Player_JUMPED:
            Debug.Log("You learned to jump... Nice");
            PlayerJumped();
            break;
        case ObsEvent.TUTORIAL_DONE:
            Debug.Log("Your now a bunny killer... Nice");
            TutorialDone();
            break;
        case ObsEvent.GAME_BEGINS:
            Debug.Log("Let your Hatred consume you");
            GameBegins();
            break;
        case ObsEvent.ADRENALINE_RUSH:
            Debug.Log("Adrenaline Rush Activate, Your a monster!");
            AdrenalineRush();
            break;
        default:
            break;
    }
```

As seen in the screenshot above, we call the *Notify* function, where once called, it goes into the corresponding case below it and runs the function. Not every function triggers an event, as some don't do anything, and others will either activate an ability or start the game. The screenshot below shows the functions where something is done as soon as the event has been triggered.

```
void TutorialDone()
{
    player = GameObject.FindGameObjectWithTag("Player");
    player.transform.position = new Vector3(0, 1, 0);
}

1 reference
void GameBegins()
{
    EnemySpawner.spawnersActive(true);
}

1 reference
void AdrenalineRush()
{
    _playerMovement = GameObject.FindGameObjectWithTag("Player").GetComponent<Movement>();

    ++EnemySpawner.Kills;
    CommandHub.LogMetrics();
    Debug.Log("Kills: " + EnemySpawner.Kills);
    Debug.Log("Shots Fired: " + BulletPool.ShotsFired);

    if ((EnemySpawner.Kills % _killStreak) == 0)
    {
        _playerMovement.setAdrenalineRush(true);
        adrenCounter++;
    }
}
```

In addition, we have created 2 classes for notifying. Below is the screenshot for one of the classes.

```
0 references
public class TBunny : MonoBehaviour
{
    tutorialDone obj;
    GameObject Bunny;
    // Start is called before the first frame update
    0 references
    void Start()
    {
        obj = new tutorialDone();
        Bunny = GameObject.FindGameObjectWithTag("TBunny");
    }

    // Update is called once per frame
    0 references
    void Update()
    {
        if(!Bunny.activeInHierarchy)
        {
            gameObject.SetActive(false);
        }
        obj.Update();
    }
}
```

The class seen above is attached to a game object in order for the non *MonoBehaviour* class to be able to be initialized and called for the update.

```
public class tutorialDone : Subject
{
    GameObject TBun;
    // Start is called before the first frame update
    1 reference
    public tutorialDone()
    {
        addObserver(AchievementManager.Instance);
        TBun = GameObject.FindGameObjectWithTag("TBun");
    }

    // Update is called once per frame
    1 reference
    public void Update()
    {
        if (TBun.activeInHierarchy == false)
        {
            notify(null, ObsEvent.TUTORIAL_DONE);
            notify(null, ObsEvent.GAME_BEGINS);
        }
    }
}
```

In the screenshot above, we can see the observer pattern in use, where when the bunny in the tutorial level is shot, it will notify that the tutorial has been completed and can move on to starting the game for the player.

State Pattern:

Similarly to the previous assignment, the state design pattern was exceptionally useful in this code for us to be able to easily switch states when in-game (walking, jumping). These two different states each have their own class. Each of these classes will implement a certain state for the game and will allow specific things exclusively to that state. Our states are the walking state and the jumping state.

```
0 references
public enum PlayerStates
{
    Walk,
    Jump
}

11 references
public interface IPlayerState
{
    4 references
    void entry(Movement movement);
    3 references
    IPlayerState input();
    3 references
    void update();

    3 references
    void fixedUpdate();
}
```

The walking state is any time that the player is on the ground, it is helpful to know this information so as to not allow players to multi-jump or have air control. Only when the state is in walking state will it allow for the jump button to be accessed.

```
2 references
public class PlayerWalkState : Subject, IPlayerState
{
    Movement _movement;
    Transform _playerTrans;
    Transform _camTrans;
    Rigidbody _playerRB;
    float _angle = 0.0f;
    float _movementSpeed;
    float _adrenalineBoost;
    bool firstMove = false;

    1 reference
    public PlayerWalkState()
    {
        addObserver(AchievementManager.Instance);
    }

    4 references
    public void entry(Movement movement)
    {
        _movement = movement;
        _camTrans = Camera.main.transform;
        _playerTrans = _movement.gameObject.GetComponent<Transform>();
        _playerRB = _movement.GetComponent<Rigidbody>();
        _movementSpeed = movement.MovementSpeed;
        _adrenalineBoost = movement.AdrenalineBoost;
    }

    3 references
    public IPlayerState input()
    {
        if (Input.GetKeyDown(KeyCode.Space) && _movement.OnGround)
            return Movement.PlayerJumpState;

        return null;
    }
```

```
3 references
public void update()
{
    // Camera follow player
    float horizontal = Input.GetAxis("Mouse X");
    //float vertical = Input.GetAxis("Mouse Y");


    _playerTrans.transform.Rotate(new Vector3(0.0f, horizontal * 2.0f, 0.0f));


    //_cam.transform.RotateAround(_playerTrans.position, Vector3.up, horizontal);
    _angle += horizontal;


    //_camTrans.transform.position = _playerTrans.position + Quaternion.Euler(0.0f, _angle, 0.0f) * camControl.CamOffset;
    //_camTrans.transform.LookAt(_playerTrans);


    if (EnemySpawner.AllEnemiesDead)
    {
        //camControl.stop();
#if UNITY_EDITOR
        EditorApplication.isPlaying = false;
        Debug.Log("Game Over");
#else
        Application.Quit(0);
#endif
    }
}

3 references
public void fixedUpdate()
{
    if (Input.GetKey(KeyCode.W))
    {
        _playerRB.AddForce(_playerTrans.forward * _movementSpeed * _adrenalineBoost);
        if (!firstMove)
        {
            notify(_movement.gameObject, ObsEvent.PLAYER_MOVED);
            firstMove = true;
        }
    }
    if (Input.GetKey(KeyCode.S))
        _playerRB.AddForce(-_playerTrans.forward * _movementSpeed * _adrenalineBoost);
    if (Input.GetKey(KeyCode.A))
        _playerRB.AddForce(-_playerTrans.right * _movementSpeed * _adrenalineBoost);
    if (Input.GetKey(KeyCode.D))
        _playerRB.AddForce(_playerTrans.right * _movementSpeed * _adrenalineBoost);

}
```

The jump state is any time the player is in the air, although you are still allowed to shoot midair it does stop you from having movement control and will stop you from spamming the jump button.

```
2 references
public class PlayerJumpState : Subject, IPlayerState
{
    Movement _movement;
    Transform _playerTrans;
    Transform _camTrans;
    Rigidbody _playerRB;
    float _angle = 0.0f;
    float _movementSpeed;
    float _adrenalineBoost;
    bool firstJump = false;


    1 reference
    public PlayerJumpState()
    {
        addObserver(AchievementManager.Instance);
    }

    float _jumpWalkDelay = 0.0f;


    4 references
    public void entry(Movement movement)
    {
        _movement = movement;
        _camTrans = Camera.main.transform;
        _playerTrans = _movement.gameObject.GetComponent<Transform>();
        _playerRB = _movement.GetComponent<Rigidbody>();
        _movementSpeed = movement.MovementSpeed;
        _adrenalineBoost = movement.AdrenalineBoost;

        _playerRB.AddForce(_playerTrans.up * _movementSpeed * _adrenalineBoost * 40.0f);

        if (!firstJump)
        {
            notify(_movement.gameObject, ObsEvent.Player_JUMPED);
            firstJump = true;
        }

    }
```

```
3 references
public IPlayerState input()
{
    if (_movement.OnGround)
    {

        return Movement.PlayerWalkState;
    }

    return null;
}

3 references
public void update()
{
    // Camera follow player
    float horizontal = Input.GetAxis("Mouse X");
    //float vertical = Input.GetAxis("Mouse Y");


    _playerTrans.transform.Rotate(new Vector3(0.0f, horizontal * 2.0f, 0.0f));


    _angle += horizontal;


    if (EnemySpawner.AllEnemiesDead)
    {
        //camControl.stop();
if UNITY_EDITOR
        EditorApplication.isPlaying = false;
        Debug.Log("Game Over");
else
        Application.Quit(0);
endif
    }
}

3 references
public void fixedUpdate()
{
}
```

The movement class holds all the states and its updates are just going through all the states.

```
// Update is called once per frame
0 references
void Update()
{
    IPlayerState state = _state.input();

    if (state == null)
    {
        _state.update();
    }
    else
    {
        _state = state;
        _state.entry(this);
    }
}

// Update is called once per frame
0 references
void FixedUpdate()
{
    _state.fixedUpdate();
}
```

```
9 references
public class Movement : MonoBehaviour
{
    [SerializeField]
    private float _movementSpeed = 1.0f;
    private float adrenalineBoost = 1.0f;
    private bool adrenaline = false;
    float adrenalineTimer = 0.0f;
    private Rigidbody _rb;
    private Material _material;
    private IEnumerator coroutine;
    static IPlayerState _playerWalkState = new PlayerWalkState();
    static IPlayerState _playerJumpState = new PlayerJumpState();
    private IPlayerState _state = null;
    private bool _onGround = true;
```

PART 2: MANAGEMENT SYSTEMS

Quest Management System:

This is very much alike the tutorial system as it is the same design, the achievement system is intertwined with this. The basics of it is when something important happens the achievement manager which is attached to this system is notified. An example of this is when the red targets appear and they are shot at this system is notified and will send the notification to drop the wall down so the player can continue. This is the same as when you shoot the trial bunny and then you spawn into the game area and the game begins, both of these instances will notify the this achievement system to keep everything running smoothly and having the code know when to

enable access to certain things so the gameplay works. The following is the list of events that are notifiable when something happens. These are all the events from the tutorial

```
public void onNotify(GameObject obj, ObsEvent _event)
{
    switch (_event)
    {
        case ObsEvent.PLAYER_MOVED:
            Debug.Log("You moved... Nice");
            PlayerMoved();
            break;
        case ObsEvent.TARGETS_DOWN:
            Debug.Log("You learned to shoot... Nice");
            TargetsDown();
            break;
        case ObsEvent.Player_JUMPED:
            Debug.Log("You learned to jump... Nice");
            PlayerJumped();
            break;
        case ObsEvent.TUTORIAL_DONE:
            Debug.Log("Your now a bunny killer... Nice");
            TutorialDone();
            break;
        case ObsEvent.GAME_BEGINS:
            Debug.Log("Let your Hatred consume you");
            GameBegins();
            break;
        case ObsEvent.ADRENALINE_RUSH:
            Debug.Log("Adrenaline Rush Activate, Your a monster!");
            AdrenalineRush();
            break;
        default:
            break;
    }
```

This bit of code happens when the tutorial is ended, the observer event gets notified and the game begins, as seen here.

```
if (TBun.activeInHierarchy == false)
{
    notify(null, ObsEvent.TUTORIAL_DONE);
    notify(null, ObsEvent.GAME_BEGINS);
}
```

User Metrics Logger:

We called an external DLL function called logMetrics, first we passed in the name of the text file, then we access the value that records the number of kills the player has achieved along with the accuracy (dividing the number of kills by the number of shots fired divided by the number of shots it takes to kill the enemies), and the number of times that the ability is used and pass both these values in as well.

```
public static void LogMetrics()
{
    logMetrics("Metrics.txt", EnemySpawner.Kills, (float)EnemySpawner.Kills / ((float)BulletPool.ShotsFired / 4.0f), AchievementManager.adrenCounter);
    //Debug.Log("Accuracy: " + (float)EnemySpawner.Kills / ((float)BulletPool.ShotsFired / 4.0f));
}
```

This DLL function takes in these variables, opens a text file and deletes its contents, then prints out each metric and closes the file.

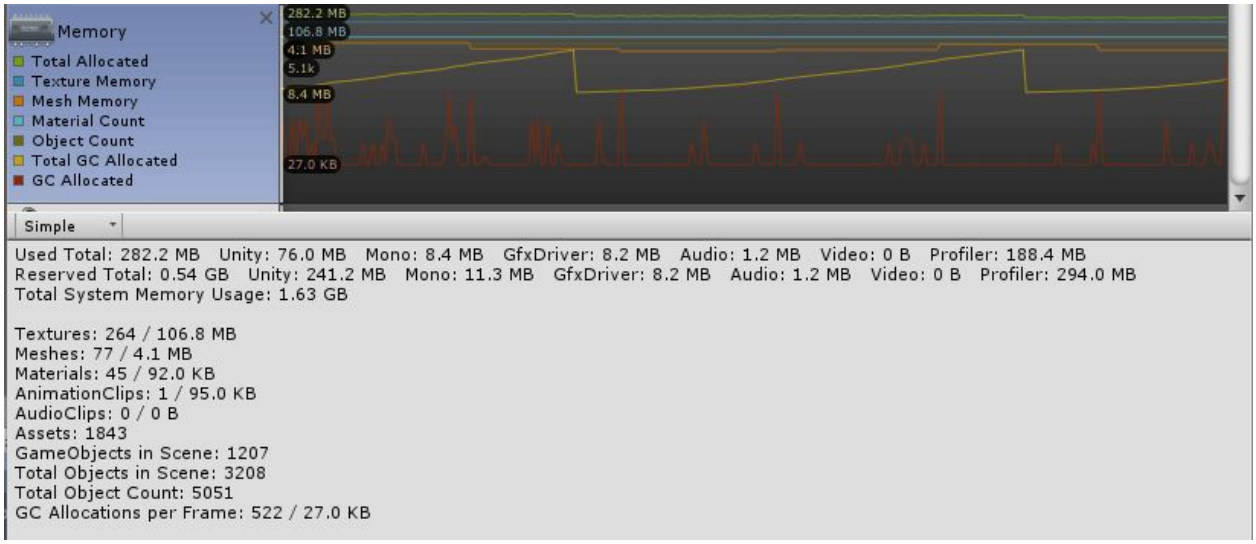
```cpp
void FileManager::logMetrics(char* filePath, int kills, float accuracy, int adrenCounter)
{
    ofstream file(filePath, ios::out);

    if (file.is_open())
    {
        file << "Kills: " << kills << std::endl;
        file << "Accuracy: " << accuracy << std::endl;
        file << "Number of times Adrenaline Rush was used: " << adrenCounter << std::endl;

        file.close();
    }
}
```

This is what the text file output looks like.

```
Kills: 20
Accuracy: 0.816327
Number of times Adrenaline Rush was used: 2
```

PART 3: MEMORY OPTIMIZATION

The biggest optimization from assignment 1 to assignment 2 is that this time the code is using object pooling, whereas initially it was simply just instantiating for each bullet and enemy. Two pools were used this time, one for the bullets and one for the enemies. This was discussed earlier in the report under PART 1: DESIGN PATTERNS. This is the resulting memory improvement statistics as provided by Unity. As you can see these are reduced numbers and the object pooling can definitely make a difference.



CONTRIBUTIONS

Giulia

- Writing/scene design; helping write the report and tutorial

Noah

- Programming; specifically tutorial design code and the quest management system

Regan

- Writing/cleaning up the code; helped write the report and cleaned up the code

Dimitris

- Writing/scene design; helping write the report and set up the scene

Adam

- Writing/scene design; helping write the report and design the tutorial

Max

- Programming; specifically helping in implementing all the game design patterns

Kevin

- Programming; specifically user metrics logger

REFERENCES

Game Programming Patterns by Robert Nystrom
https://gameprogrammingpatterns.com/contents.html
3D Character - White Rabbit - Unity Asset Store
https://assetstore.unity.com/packages/3d/characters/animals/white-rabbit-138709
NavMeshComponents - Navigation for Enemies - Unity
https://github.com/Unity-Technologies/NavMeshComponents