AT Block 1 - Task 3
Squad AI Controller

Matteo Bolognesi
19025911

University of the West of England

July 19, 2023

The chosen task for block 1 was "Squad AI Controller". It was required to research and develop a system that allows the player to control a squad of agents. It was required to have simple interactions with objects when clicked and possibly be context sensitive, avoiding as much as possible sub-menus to minimize inputs. As a stretch goal, the task required deploying the system into a simple game to demonstrate its abilities.

1 Introduction

As requested in the specifics for this task, the aim of the system was to minimize the input for the player to control the squad, focusing on context sensitivity. The player should have been able to interact with the environment directly, avoiding menus where possible. Considering that in the specifics there was no particular genre required, the system was developed on the model of classic Real Time Strategy games (RTS). To conclude, the system had to be deployed into a simple game. For this purpose, the manager system developed for block 3 was used and adapted where necessary to fit the game's needs.

2 Related Work

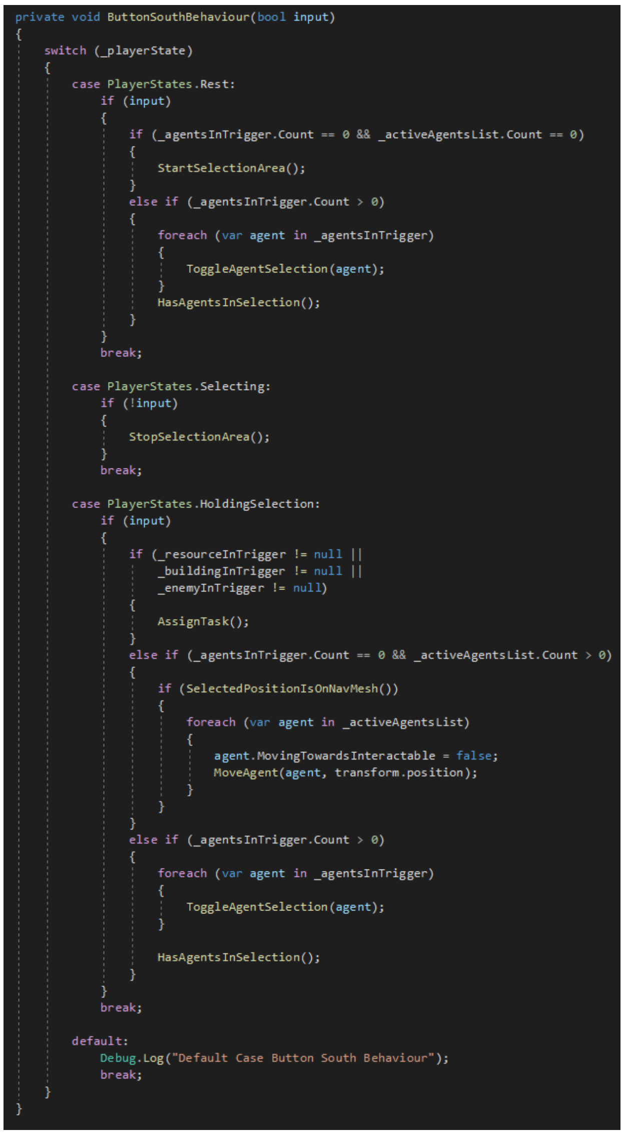
Two of the most famous series of the RTS genre are **Warcraft** (Fig.1) and **Age of Empires** (Fig.2). These two series, developed respectively by **Blizard Entertainment** and **Ensemble Studios**, represent perfectly what the task requires. They allowed the player to interact with single or groups of units at the same time and assign them tasks by just clicking on the environment. A small number of sub-menus were present, easily interactable with simple mouse clicks and properly implemented in the UI.

Figure 1: *Warcraft: Orcs Humans*

3 Method

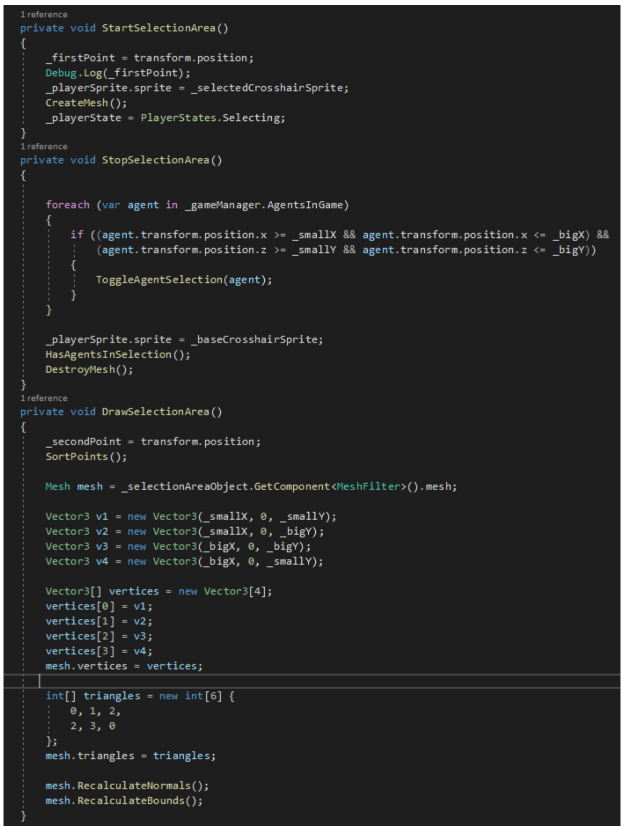
The system requires one click to perform each action (Fig.3). based on what object the player is hovering, on button press, the object's specific action is enabled. This allows the player to select and deselect agents through direct click or selection area, and assign tasks to selected agents by clicking on the environment. The system can handle multiple agents at the same time and, on click, the chosen task is assigned to each agent if the task's conditions are met. The system can be easily expanded to incorporate more tasks and increase its complexity.

Figure 2: *Age of Empires 1*



```
private void ButtonSouthBehaviour(bool input)
{
    switch (_playerState)
    {
        case PlayerStates.Rest:
            if (input)
            {
                if (_agentsInTrigger.Count == 0 && _activeAgentsList.Count == 0)
                {
                    StartSelectionArea();
                }
                else if (_agentsInTrigger.Count > 0)
                {
                    foreach (var agent in _agentsInTrigger)
                    {
                        ToggleAgentSelection(agent);
                    }
                    HasAgentsInSelection();
                }
            }
            break;

        case PlayerStates.Selecting:
            if (!input)
            {
                StopSelectionArea();
            }
            break;

        case PlayerStates.HoldingSelection:
            if (input)
            {
                if (_resourceInTrigger != null ||
                    _buildingInTrigger != null ||
                    _enemyInTrigger != null)
                {
                    AssignTask();
                }
                else if (_agentsInTrigger.Count == 0 && _activeAgentsList.Count > 0)
                {
                    if (SelectedPositionIsOnNavMesh())
                    {
                        foreach (var agent in _activeAgentsList)
                        {
                            agent.MovingTowardsInteractable = false;
                            MoveAgent(agent, transform.position);
                        }
                    }
                }
                else if (_agentsInTrigger.Count > 0)
                {
                    foreach (var agent in _agentsInTrigger)
                    {
                        ToggleAgentSelection(agent);
                    }

                    HasAgentsInSelection();
                }
            }
            break;

        default:
            Debug.Log("Default Case Button South Behaviour");
            break;
    }
}
```

Figure 3: *Input Button Function*



```
1 reference
private void StartSelectionArea()
{
    _firstPoint = transform.position;
    Debug.Log(_firstPoint);
    _playerSprite.sprite = _selectedCrosshairSprite;
    CreateMesh();
    _playerState = PlayerStates.Selecting;
}
1 reference
private void StopSelectionArea()
{

    foreach (var agent in _gameManager.AgentsInGame)
    {
        if ((agent.transform.position.x >= _smallX && agent.transform.position.x <= _bigX) &&
            (agent.transform.position.z >= _smallY && agent.transform.position.z <= _bigY))
        {
            ToggleAgentSelection(agent);
        }
    }

    _playerSprite.sprite = _baseCrosshairSprite;
    HasAgentsInSelection();
    DestroyMesh();
}
1 reference
private void DrawSelectionArea()
{
    _secondPoint = transform.position;
    SortPoints();

    Mesh mesh = _selectionAreaObject.GetComponent<MeshFilter>().mesh;

    Vector3 v1 = new Vector3(_smallX, 0, _smallY);
    Vector3 v2 = new Vector3(_smallX, 0, _bigY);
    Vector3 v3 = new Vector3(_bigX, 0, _bigY);
    Vector3 v4 = new Vector3(_bigX, 0, _smallY);

    Vector3[] vertices = new Vector3[4];
    vertices[0] = v1;
    vertices[1] = v2;
    vertices[2] = v3;
    vertices[3] = v4;
    mesh.vertices = vertices;

    int[] triangles = new int[6] {
        0, 1, 2,
        2, 3, 0
    };
    mesh.triangles = triangles;

    mesh.RecalculateNormals();
    mesh.RecalculateBounds();
}
```

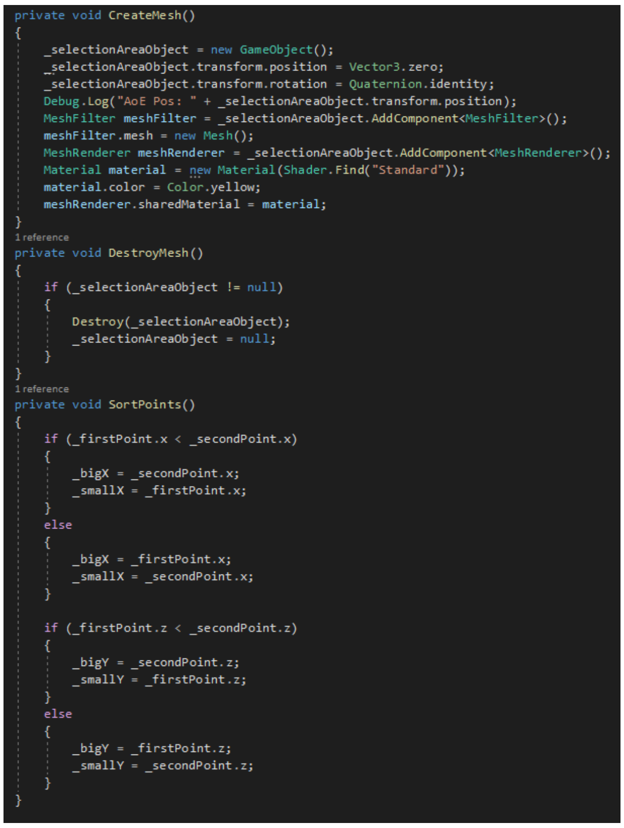
Figure 4: *Functions to draw the selection area 1/2*

3.1 Player

The player object is responsible for moving the player's cursor and checking the player's target. It is used to assign tasks to agents based on the cursor's position on the map. The player script is also responsible for handling some of the UI elements and drawing the selection area. The area is generated by taking the coordinates of the point where the player clicked (if not hovering any interactable object and if no agents are selected) and, at every frame, drawing a mesh based on the first point and the current position of the player's cursor (Fig.4) . This mesh is attached to a game object that is created when the selection starts and is destroyed when the selection ends (Fig.5). When the player releases the button, the selection ends and the second point is registered. The script then checks the position of all the agents stored in the game manager's agent list (which collects all the agents in the game) and compares their position to the area formed using the points' coordinates combined together (AABB check). If they are inside, the agent/s become selected, otherwise, nothing happens.

3.2 Agents/Resources/Buildings

Each agent can perform a series of actions independently (Fig.6). If not selected and not assigned to any task, they will go into an idle state and start walking around their position. Once selected, they can move



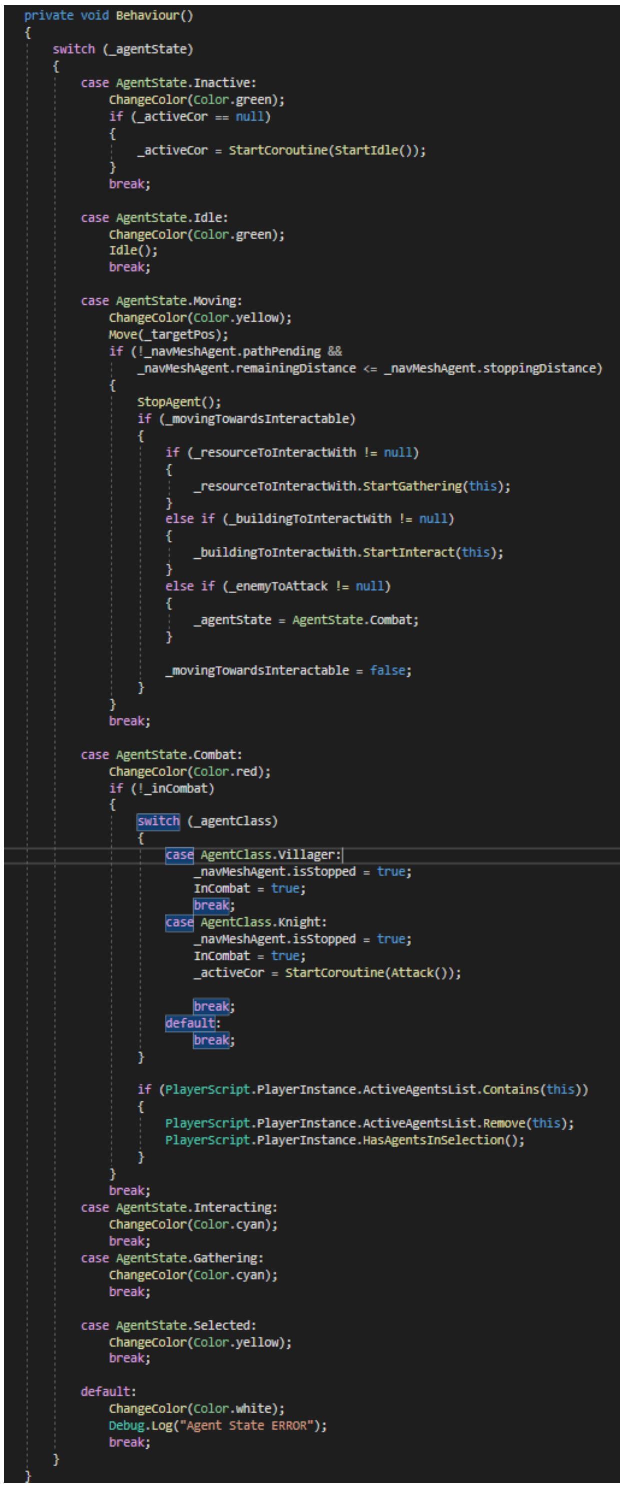
```
private void CreateMesh()
{
    _selectionAreaObject = new GameObject();
    _selectionAreaObject.transform.position = Vector3.zero;
    _selectionAreaObject.transform.rotation = Quaternion.identity;
    Debug.Log("AoE Pos: " + _selectionAreaObject.transform.position);
    MeshFilter meshFilter = _selectionAreaObject.AddComponent<MeshFilter>();
    meshFilter.mesh = new Mesh();
    MeshRenderer meshRenderer = _selectionAreaObject.AddComponent<MeshRenderer>();
    Material material = new Material(Shader.Find("Standard"));
    material.color = Color.yellow;
    meshRenderer.sharedMaterial = material;
}
1 reference
private void DestroyMesh()
{
    if (_selectionAreaObject != null)
    {
        Destroy(_selectionAreaObject);
        _selectionAreaObject = null;
    }
}
1 reference
private void SortPoints()
{
    if (_firstPoint.x < _secondPoint.x)
    {
        _bigX = _secondPoint.x;
        _smallX = _firstPoint.x;
    }
    else
    {
        _bigX = _firstPoint.x;
        _smallX = _secondPoint.x;
    }

    if (_firstPoint.z < _secondPoint.z)
    {
        _bigY = _secondPoint.z;
        _smallY = _firstPoint.z;
    }
    else
    {
        _bigY = _firstPoint.z;
        _smallY = _secondPoint.z;
    }
}
```

Figure 5: *Functions to draw the selection area 2/2*

around the map using as their destination the button input of the player. If the player's cursor is placed on one of the resources present in the game upon interaction, the agent/s will start walking towards the resource and once reached will start a coroutine to gather it(Fig.7). Upon certain conditions, the agent/s will stop the coroutine and go back to idle. The coroutine can also be interrupted by selecting the agent. Instead, if the player's cursor is on a building upon input, the agent will move towards it and start a different coroutine based on the type of building. The agent can deposit the currently carried resources if interacting with a "deposit" or become a guard if interacting with an "armoury"(Fig.8). Once turned into a guard, the agent will gain the ability to attack the enemies but will lose the ability to gather resources. If a guard is assigned to a gathering task, it will reach the resource location and will go back to idle. This was a gameplay choice but could be easily changed to not move completely or any other intended reaction.

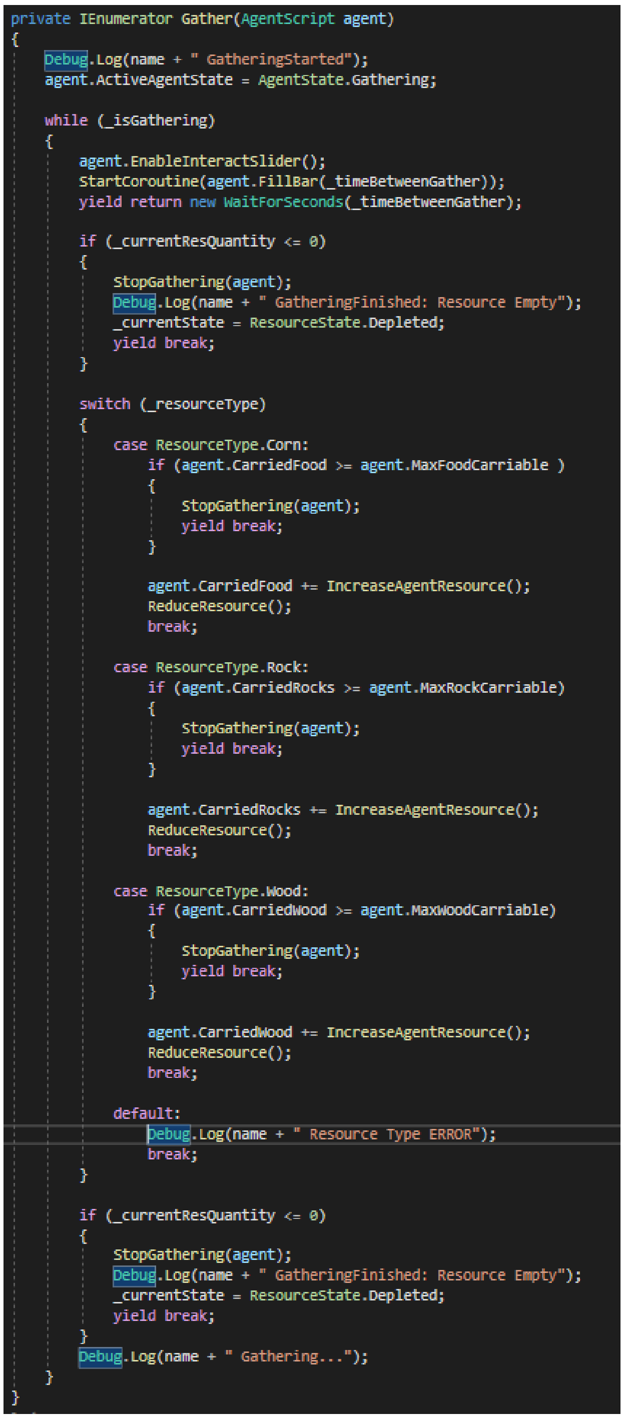
3.3 Enemies

The game present just one type of enemy that, once instantiated, checks which agent is the closest and starts moving towards it. If another agent becomes the closest, the enemy will switch target, chasing it instead. Once reached, both the target and the enemy are set in the combat state. When in the combat state, the enemy will deal periodic damage to the targeted agent



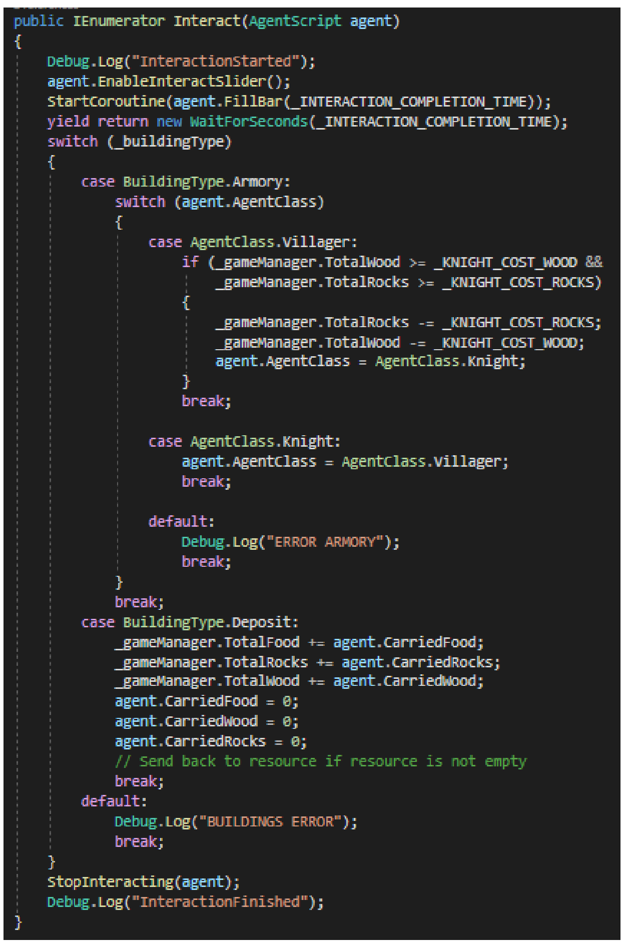
```
private void Behaviour()
{
    switch (_agentState)
    {
        case AgentState.Inactive:
            ChangeColor(Color.green);
            if (_activeCor == null)
            {
                _activeCor = StartCoroutine(StartIdle());
            }
            break;

        case AgentState.Idle:
            ChangeColor(Color.green);
            Idle();
            break;

        case AgentState.Moving:
            ChangeColor(Color.yellow);
            Move(_targetPos);
            if (!_navMeshAgent.pathPending &&
                _navMeshAgent.remainingDistance <= _navMeshAgent.stoppingDistance)
            {
                StopAgent();
                if (_movingTowardsInteractable)
                {
                    if (_resourceToInteractWith != null)
                    {
                        _resourceToInteractWith.StartGathering(this);
                    }
                    else if (_buildingToInteractWith != null)
                    {
                        _buildingToInteractWith.StartInteract(this);
                    }
                    else if (_enemyToAttack != null)
                    {
                        _agentState = AgentState.Combat;
                    }

                    _movingTowardsInteractable = false;
                }
            }
            break;

        case AgentState.Combat:
            ChangeColor(Color.red);
            if (!_inCombat)
            {
                switch (_agentClass)
                {
                    case AgentClass.Villager:
                        _navMeshAgent.isStopped = true;
                        InCombat = true;
                        break;
                    case AgentClass.Knight:
                        _navMeshAgent.isStopped = true;
                        InCombat = true;
                        _activeCor = StartCoroutine(Attack());

                        break;
                    default:
                        break;
                }

                if (PlayerScript.PlayerInstance.ActiveAgentsList.Contains(this))
                {
                    PlayerScript.PlayerInstance.ActiveAgentsList.Remove(this);
                    PlayerScript.PlayerInstance.HasAgentsInSelection();
                }
            }
            break;
        case AgentState.Interacting:
            ChangeColor(Color.cyan);
            break;
        case AgentState.Gathering:
            ChangeColor(Color.cyan);
            break;

        case AgentState.Selected:
            ChangeColor(Color.yellow);
            break;

        default:
            ChangeColor(Color.white);
            Debug.Log("Agent State ERROR");
            break;
    }
}
```

Figure 6: *Agent Behaviour Function*



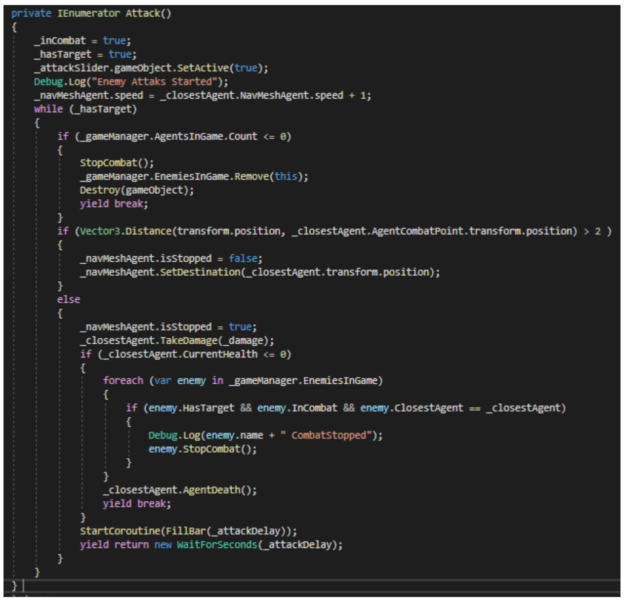
```
private IEnumerator Gather(AgentScript agent)
{
    Debug.Log(name + " GatheringStarted");
    agent.ActiveAgentState = AgentState.Gathering;

    while (_isGathering)
    {
        agent.EnableInteractSlider();
        StartCoroutine(agent.FillBar(_timeBetweenGather));
        yield return new WaitForSeconds(_timeBetweenGather);

        if (_currentResQuantity <= 0)
        {
            StopGathering(agent);
            Debug.Log(name + " GatheringFinished: Resource Empty");
            _currentState = ResourceState.Depleted;
            yield break;
        }

        switch (_resourceType)
        {
            case ResourceType.Corn:
                if (agent.CarriedFood >= agent.MaxFoodCarriable )
                {
                    StopGathering(agent);
                    yield break;
                }

                agent.CarriedFood += IncreaseAgentResource();
                ReduceResource();
                break;

            case ResourceType.Rock:
                if (agent.CarriedRocks >= agent.MaxRockCarriable)
                {
                    StopGathering(agent);
                    yield break;
                }

                agent.CarriedRocks += IncreaseAgentResource();
                ReduceResource();
                break;

            case ResourceType.Wood:
                if (agent.CarriedWood >= agent.MaxWoodCarriable)
                {
                    StopGathering(agent);
                    yield break;
                }

                agent.CarriedWood += IncreaseAgentResource();
                ReduceResource();
                break;

            default:
                Debug.Log(name + " Resource Type ERROR");
                break;
        }

        if (_currentResQuantity <= 0)
        {
            StopGathering(agent);
            Debug.Log(name + " GatheringFinished: Resource Empty");
            _currentState = ResourceState.Depleted;
            yield break;
        }
        Debug.Log(name + " Gathering...");
    }
}
```

Figure 7: *Resource Gather Coroutine*



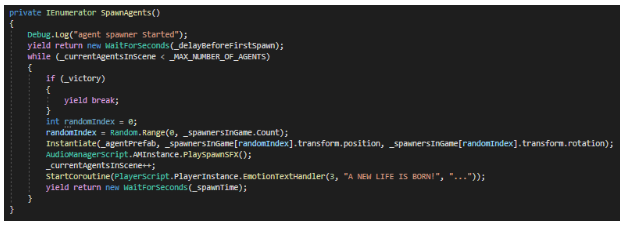
```
public IEnumerator Interact(AgentScript agent)
{
    Debug.Log("InteractionStarted");
    agent.EnableInteractSlider();
    StartCoroutine(agent.FillBar(_INTERACTION_COMPLETION_TIME));
    yield return new WaitForSeconds(_INTERACTION_COMPLETION_TIME);
    switch (_buildingType)
    {
        case BuildingType.Armory:
            switch (agent.AgentClass)
            {
                case AgentClass.Villager:
                    if (_gameManager.TotalWood >= _KNIGHT_COST_WOOD &&
                        _gameManager.TotalRocks >= _KNIGHT_COST_ROCKS)
                    {
                        _gameManager.TotalRocks -= _KNIGHT_COST_ROCKS;
                        _gameManager.TotalWood -= _KNIGHT_COST_WOOD;
                        agent.AgentClass = AgentClass.Knight;
                    }
                    break;

                case AgentClass.Knight:
                    agent.AgentClass = AgentClass.Villager;
                    break;

                default:
                    Debug.Log("ERROR ARMORY");
                    break;
            }
            break;
        case BuildingType.Deposit:
            _gameManager.TotalFood += agent.CarriedFood;
            _gameManager.TotalRocks += agent.CarriedRocks;
            _gameManager.TotalWood += agent.CarriedWood;
            agent.CarriedFood = 0;
            agent.CarriedWood = 0;
            agent.CarriedRocks = 0;
            // Send back to resource if resource is not empty
            break;
        default:
            Debug.Log("BUILDINGS ERROR");
            break;
    }
    StopInteracting(agent);
    Debug.Log("InteractionFinished");
}
```

Figure 8: *Building Interact Coroutine*



```
private IEnumerator Attack()
{
    _inCombat = true;
    _hasTarget = true;
    _attackSlider.gameObject.SetActive(true);
    Debug.Log("Enemy Attaks Started");
    _navMeshAgent.speed = _closestAgent.NavMeshAgent.speed + 1;
    while (_hasTarget)
    {
        if (_gameManager.AgentsInGame.Count <= 0)
        {
            StopCombat();
            _gameManager.EnemiesInGame.Remove(this);
            Destroy(gameObject);
            yield break;
        }
        if (Vector3.Distance(transform.position, _closestAgent.AgentCombatPoint.transform.position) > 2 )
        {
            _navMeshAgent.isStopped = false;
            _navMeshAgent.SetDestination(_closestAgent.transform.position);
        }
        else
        {
            _navMeshAgent.isStopped = true;
            _closestAgent.TakeDamage(_damage);
            if (_closestAgent.CurrentHealth <= 0)
            {
                foreach (var enemy in _gameManager.EnemiesInGame)
                {
                    if (enemy.HasTarget && enemy.InCombat && enemy.ClosestAgent == _closestAgent)
                    {
                        Debug.Log(enemy.name + " CombatStopped");
                        enemy.StopCombat();
                    }
                }
                _closestAgent.AgentDeath();
                yield break;
            }
            StartCoroutine(FillBar(_attackDelay));
            yield return new WaitForSeconds(_attackDelay);
        }
    }
}
```

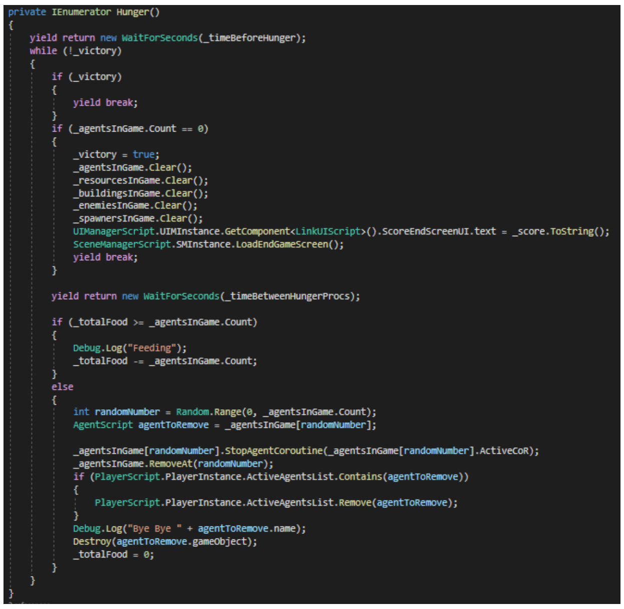
Figure 9: *Enemy Attack Coroutine*



```
private IEnumerator SpawnAgents()
{
    Debug.Log("agent spawner Started");
    yield return new WaitForSeconds(_delayBeforeFirstSpawn);
    while (_currentAgentsInScene < _MAX_NUMBER_OF_AGENTS)
    {
        if (_victory)
        {
            yield break;
        }
        int randomIndex = 0;
        randomIndex = Random.Range(0, _spawnersInGame.Count);
        Instantiate(_agentPrefab, _spawnersInGame[randomIndex].transform.position, _spawnersInGame[randomIndex].transform.rotation);
        AudioManagerScript.AMInstance.PlaySpawnSFX();
        _currentAgentsInScene++;
        StartCoroutine(PlayerScript.PlayerInstance.EmotionTextHandler(3, "A NEW LIFE IS BORN!", "..."));
        yield return new WaitForSeconds(_spawnTime);
    }
}
```

Figure 10: *Agent Spawner Coroutine*

using a coroutine(Fig.9). If the agent is a knight, it will deal damage back, otherwise it will just stand still, without triggering the idle animation. On death, the enemy awards the player with some points and then is removed from the list of enemies in the game present on the game manager. In the end, the enemy object is destroyed and the game continues.

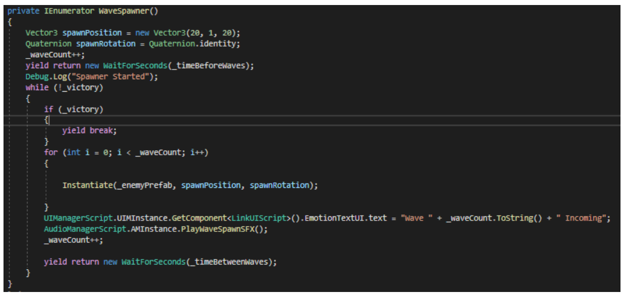


```
private IEnumerator Hunger()
{
    yield return new WaitForSeconds(_timeBeforeHunger);
    while (!_victory)
    {
        if (_victory)
        {
            yield break;
        }
        if (_agentsInGame.Count == 0)
        {
            _victory = true;
            _agentsInGame.Clear();
            _resourcesInGame.Clear();
            _buildingsInGame.Clear();
            _enemiesInGame.Clear();
            _spawnersInGame.Clear();
            UIManagerScript.UIMInstance.GetComponent<LinkUIScript>().ScoreEndScreenUI.text = _score.ToString();
            SceneManagerScript.SMInstance.LoadEndGameScreen();
            yield break;
        }

        yield return new WaitForSeconds(_timeBetweenHungerProcs);

        if (_totalFood >= _agentsInGame.Count)
        {
            Debug.Log("Feeding");
            _totalFood -= _agentsInGame.Count;
        }
        else
        {
            int randomNumber = Random.Range(0, _agentsInGame.Count);
            AgentScript agentToRemove = _agentsInGame[randomNumber];

            _agentsInGame[randomNumber].StopAgentCoroutine(_agentsInGame[randomNumber].ActiveCoR);
            _agentsInGame.RemoveAt(randomNumber);
            if (PlayerScript.PlayerInstance.ActiveAgentsList.Contains(agentToRemove))
            {
                PlayerScript.PlayerInstance.ActiveAgentsList.Remove(agentToRemove);
            }
            Debug.Log("Bye Bye " + agentToRemove.name);
            Destroy(agentToRemove.gameObject);
            _totalFood = 0;
        }
    }
}
```

Figure 11: *Hunger Coroutine*

3.4 The Game

The game revolves around a few basic mechanics. The player needs to hoard resources to be able to transform villagers into guards to protect them and try to not starve at the same time. Villagers are spawned periodically on the map until a maximum number is reached (Fig.10). If a villager dies, the game will start spawning again. If all the villagers and the knights are killed, the game ends. Over time, a number of food resources are removed (from the total food hoarded) equal to the number of agents currently in the game (Fig.11). If the player does not have enough food, a random agent is destroyed and the total food count becomes zero. Waves of enemies are spawned periodically through the course of the game and the number of enemies during each wave is equal to the wave number(Fig.12).



```
private IEnumerator WaveSpawner()
{
    Vector3 spawnPosition = new Vector3(20, 1, 20);
    Quaternion spawnRotation = Quaternion.identity;
    _waveCount++;
    yield return new WaitForSeconds(_timeBeforeWaves);
    Debug.Log("Spawner Started");
    while (!_victory)
    {
        if (_victory)
        {
            yield break;
        }
        for (int i = 0; i < _waveCount; i++)
        {

            Instantiate(_enemyPrefab, spawnPosition, spawnRotation);

        }
        UIManagerScript.UIMInstance.GetComponent<LinkUIScript>().EmotionTextUI.text = "Wave " + _waveCount.ToString() + " Incoming";
        AudioManagerScript.AMInstance.PlayWaveSpawnSFX();
        _waveCount++;

        yield return new WaitForSeconds(_timeBetweenWaves);
    }
}
```

Figure 12: *Enemy Waves Coroutine*

4 Evaluation

Overall the game works in its entirety and the playtest sessions did not report any major bug. Being based on the framework developed for block 3, the problems related to it are explained in that report. Although, while implementing this game, some refactoring happened and some problems have been fixed. The game covers all the points required for the task, proposing a system that requires just one button press to interact with the environment with one or more agents. It also adjusts the current action performed by the agent based on the agent's current state and the player's choices.

5 Conclusion

The task required creating a system to control one or more agents at the same time, allowing them to perform tasks based on where the player's input happened. The game is fully functioning although maybe a little bit unbalanced. Despite its simplicity and lack of proper coding structure, the game conveys the feeling of the old RTS taken as an example.