Computer Games Development
CW208
Software Functional Specification
Year IV

Ruslan Gavrilov

C00273521

13/05/2024

Functional Specification

World Generation / Structure

Overview

A key aspect of this project is the layout and generation of the game world. I had to organise the system of the map and the generation of the world in such a way that would work well together and give me the freedom to explore procedural generation in this project.

World Structure

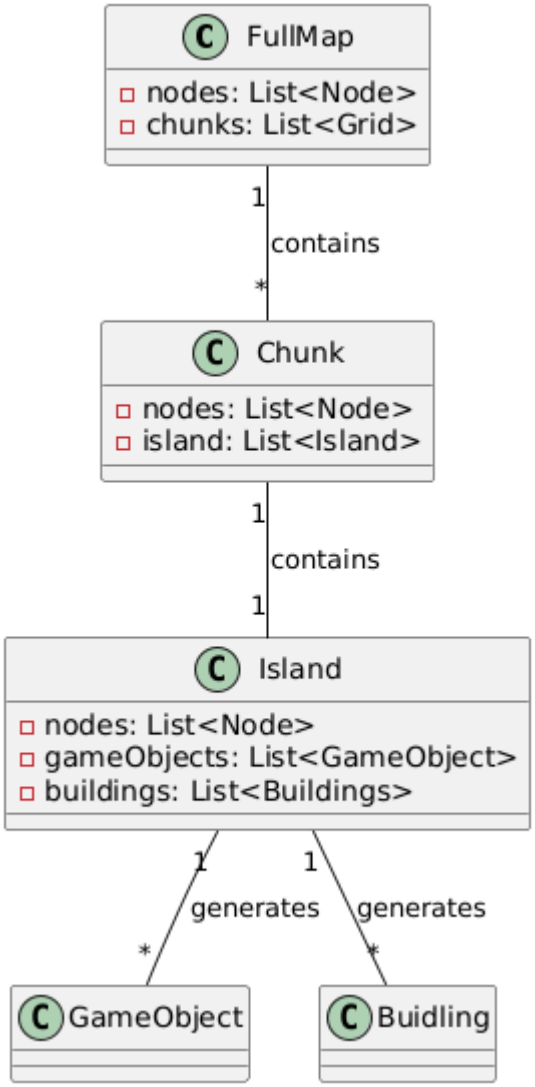
The world is designed with a 2D tile-based system in mind. It is split into individual nodes that are interconnected, each holding a list of their own neighbours so that in-game traversal, organisation and future path finding algorithms are easier to implement with said structure.

Once adding in the initial node map, I wanted to add more of a hierarchy in the world. After the Tile map is generated, it is split into individual chunks, each holding their own nodes so that pinpointing locations, updates and world generation is split up into individual interconnected chunks rather than the entire world at once. This allows me to lessen search time on search algorithms as I would only have to search a certain chunk rather than the entire map.

To generate the map, I have an input of how big the map is, ex. 1x1, 2x2, 3x3, these symbolise the chunks. Each chunk would have 32x32 nodes, therefore on a map that is set to be 3x3 chunks, the entire map is 96x96 nodes long. With such a large number of nodes to possibly search, I believe it was crucial to split my map in such a way that would lessen the total searched nodes.

Each chunk would then generate its own Island/Islands and would be responsible for keeping track of those islands. This would lessen possible searches even more, as it is now limited to individual islands, rather than the entire chunk.

Those islands now can be populated with possible game objects. This relation between world generation makes it easier and more concise to control generation and interactions with other game elements.



Procedural Generation

A key technical feature of this project is the procedural generation of the individual Islands. I had to experiment with many different algorithms in order to satisfy accurate and unique island shapes within my game. I then had to search through more algorithms to generate objects such as trees, buildings and barrels on those islands in an intelligent manner so that the game world felt unique but also the generation looked correct.

Island Generation

I had a look at 2 general shape algorithms for the generation of the shape of the Island. Those being Cellular Automata and the Diamond Square Algorithm.

Diamond Square Algorithm

This algorithm works to create believable terrain using height maps. A roughness is defined in this algorithm to define how bumpy our terrain ends up generating. This algorithm follows two steps ; Diamond Step, Square Step.

The Square step would average the value of corners in said square and place it as the midpoint. It would also add a random value to that. Then you would move onto the square step, which would generate a diamond around the midpoint of the square step, ultimately splitting your grid into 4 squares from the initial one. This would count as one iteration of the diamond square algorithm, and you would continue this process until you reach the smallest possible square of nodes being 3x3. After the run through of the algorithm, each node would have been assigned a height value, and it is from this height value that you would determine which texture I would give the tile (would it be water or land depending on a threshold I would set), This would hopefully generate a random centralised shape in my chunk that I would then appropriately texture to form an island.

Code

```
void Grid::DiamondSquare() {

nodeGrid[Index(0, 0)]->height = std::rand() % 4 +1; //random height to each corner to start nodeGrid[Index(0, SCREEN_HEIGHT / gridNodeSize - 1)]->height = std::rand() % 4 + 1; nodeGrid[Index(SCREEN_HEIGHT / gridNodeSize - 1, 0)]->height = std::rand() % 4 + 1; nodeGrid[Index(SCREEN_HEIGHT / gridNodeSize - 1, SCREEN_HEIGHT / gridNodeSize - 1)]->height = std::rand() % 4 + 1; int step = SCREEN_HEIGHT / gridNodeSize - 1; float roughness = 6; while (step > 1) { int halfStep = step / 2;

  //diamond step
    for (int y = halfStep; y < SCREEN_HEIGHT / gridNodeSize; y += step) {
        for (int x = halfStep; x < SCREEN_HEIGHT / gridNodeSize; x += step) {
            float avg = (nodeGrid[Index(x - halfStep, y - halfStep)]->height +
                nodeGrid[Index(x - halfStep, y + halfStep)]->height +
                nodeGrid[Index(x + halfStep, y - halfStep)]->height +
                nodeGrid[Index(x + halfStep, y + halfStep)]->height) * 0.25f;
            nodeGrid[Index(x, y)]->height = avg + randomFloat(-roughness,roughness);
        }
    }

    //square step
    for (int y = 0; y < SCREEN_HEIGHT / gridNodeSize; y += halfStep) {
        for (int x = (y + halfStep) % step; x < SCREEN_HEIGHT / gridNodeSize; x += step) {
            float avg = 0.0f;
            int count = 0;

            if (x >= halfStep) { avg += nodeGrid[Index(x - halfStep, y)]->height; count++;
}
            if (x + halfStep < SCREEN_HEIGHT / gridNodeSize) { avg += nodeGrid[Index(x +
halfStep, y)]->height; count++; }
            if (y >= halfStep) { avg += nodeGrid[Index(x, y - halfStep)]->height; count++;
}
            if (y + halfStep < SCREEN_HEIGHT / gridNodeSize) { avg += nodeGrid[Index(x, y +
halfStep)]->height; count++; }

            nodeGrid[Index(x, y)]->height = avg / count + randomFloat(-roughness,
roughness);
        }
    }
    step /= 2;
    roughness *= 0.5f; //reduce roughness for finer details
}
```

```
for(auto node : nodeGrid)
{
    if (node->height > 3)
    {
        node->drawableNode.setTexture(textureManager.getTexture("landTile"));
        node->setTileType(TileType::LAND);
        node->setLand(true);
    }
    else
    {
        node->drawableNode.setTexture(textureManager.getTexture("waterTile"));
        node->setTileType(TileType::WATER);
        node->setLand(false);
    }
}


}
```

```
int Grid::Index(int x, int y) { return ((y) * (SCREEN_HEIGHT / gridNodeSize) + x); }

float Grid::randomFloat(float min, float max) { static std::random_device rd;

static std::default_random_engine generator(rd());

std::uniform_real_distribution distribution(min, max); return distribution(generator); }
```

Outcome

When testing out this algorithm, I noticed that the shapes it generated with my implementation of it didn't have the desired shape I imagined Islands to look like, they would come across as very circular and central to the chunk and ultimately only generate one large central island. I thought that for my vision of many islands small or large I would have to find another algorithm to suit my purposes.

Cellular Automata

Cellular Automata is a procedural generation algorithm that I saw used for generating cave-like structures for dungeons. I noticed that if I was to flip what is a wall and open area, it would give off possible Island shapes.

It works by first populating a grid with noise based on an input density, for example a density of 70% means that 70% of the nodes would be set as land for my example while the others would be considered as water. It would then copy the grid and determine what to change each node to depending on the neighbours. If a current node had 4 or more water neighbours it would change that node to a water node, if it was land then it would be changed to land. It would follow this step for each node until the end where it would then reassign the node grid and update it to the newly generated noise map. It could follow this step for multiple iteration with each iteration making a more and more concise shape where random noise would begin to form more uniform and tighter shapes depending on how many times you iterated and how dense the intimal random noise map was.

Code

```cpp
void Grid::ApplyCellular(int _interations)
{
    for(int i =0;i<_interations;i++)
    {
        // Deep copy the grid (create new nodes, not just copy pointers)
        std::vector<Node*> tempGrid;
        for (auto node : nodeGrid)
        {
            Node* newNode = new Node(*node);  // Create a copy of the node
            tempGrid.push_back(newNode);
        }

        for(auto* node : tempGrid)
        {
            int landCount = 0;

            for(auto& neighbour : node->getNeighbours())
            {
                if(neighbour.first->getIsLand())
                {
                    landCount++;
                }
            }

            if(landCount > 4)
            {
                node->drawableNode->setTexture(TextureManager::getInstance().getTexture("landTile"));
                node->setParentTileType(LAND);
                node->setLand(true);
            }else
            {
                node->drawableNode->setTexture(TextureManager::getInstance().getTexture("waterTile"));
                node->setParentTileType(WATER);
                node->setTileType(DEFAULT_WATER);
                node->setLand(false);
            }
        }
        //apply changes back to the original nodeGrid
        for (size_t j = 0; j < nodeGrid.size(); j++)
        {
            nodeGrid[j]->setLand(tempGrid[j]->getIsLand());
            nodeGrid[j]->drawableNode->setTexture(*tempGrid[j]->drawableNode->getTexture());
            nodeGrid[j]->setTileType(tempGrid[j]->getTileType());
        }
        // delete temp deep copied nodes
        for (Node* node : tempGrid)
        {
            delete node;
        }
    }
```

Outcome

With this generation I noticed that the shapes that cellular automata provided were more random and less circular than those done by The diamond Square algorithm. It also had the possibility of creating multiple islands rather than one concise island, making it a lot more customizable by allowing multiple small islands or one large island, all based on the input of iterations and randomness of the initial noise map. I was happy to continue with this algorithm and move onto texturing.

Once the Island shape was generated, I had to find an algorithm to texture the appropriate tiles now that I had a concise shape. For this I looked at the Wave Function Collapse Algorithm and applied my own tile-based rules onto it.

Wave Function Collapse

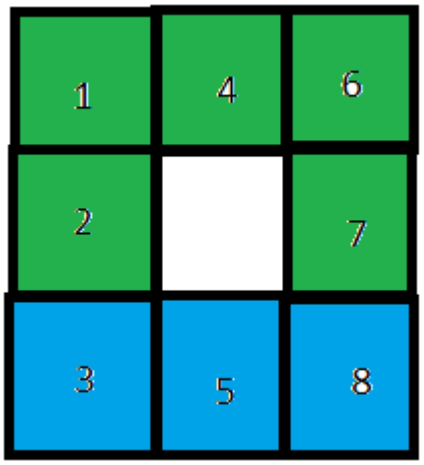
How this algorithm would work is it would start at a random part of the island and then determine the texture based on a predefined set of possible textures, choose one at random and then move onto the next neighbour and repeat. With this, each node would be textured based on how many of its neighbours were land against how many were water.

I altered this algorithm so that I would first search through my entire chunk until I found a land node rather than selecting a random node. Once it was selected I used predefined rules and checks based on neighbours to see which texture I should change this node to, for example, if I had 3 water neighbours under a node and the rest land, I would want to make this tile a texture that was a sandy edge facing downwards.

The Tile rules were set as follows, where the number represents the neighbour and the Boolean represents if its land:

{{ 1, true}, {2, true} ,{3, false}, {4, true }, {5, false }, {6, true},{7, true}, {8, false }}

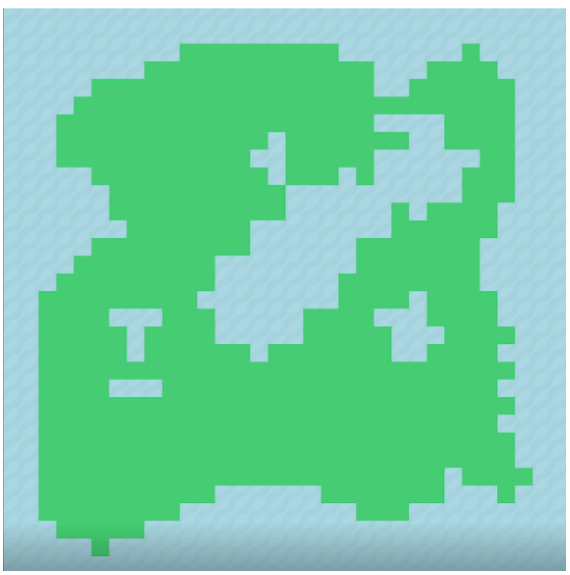
This shape would look as so, with the centre being the one determined.

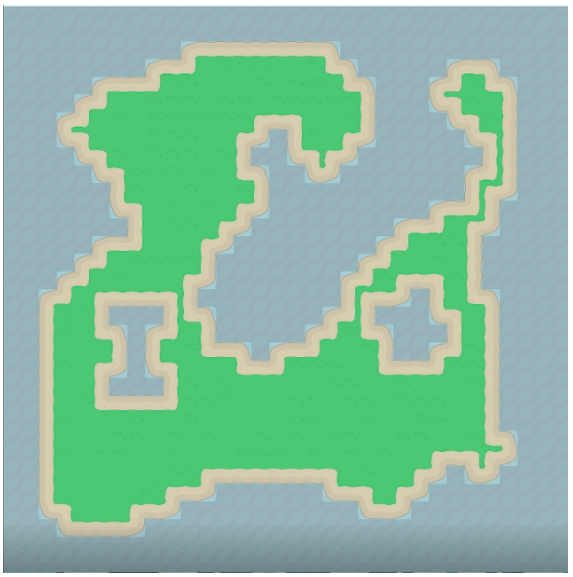


This function would then Check the tile based on all possible rules.

```
bool Grid::CheckPattern(const std::shared_ptr<Node>& _currentNode, const std::vector<std::pair<int, bool>>& _pattern) const
{
    int patternCount = 0;
    for (int i = 0; i < _pattern.size(); i++)
    {
        if (_currentNode->getNodeData().neighbourIDs.size() > 5) {
            if (_currentNode->getNeighbours()[_pattern[i].first].first->getIsLand() == _pattern[i].second)
            {
                patternCount++;
                if (patternCount == _pattern.size())
                {
                    return true;
                }
            }
            else {
                return false;
            }
        }
        else
        {
            return false;
        }
    }
    return false;
}
```

I would then traverse the entire land mass from that first initial node, only texturing the land nodes and check the rule set for each of them until the island was fully textured. With these rule sets in place, I was free to also set rules for undesired shapes, such as jutting out nodes and removing them or changing them to a water node and removing the land texture altogether. This would give me valid texturing to the entire land mass.





Challenges

With each island being vastly different from each other, the generation had a few areas that could not be textured. For example in the screenshot above:

This area would have to be accounted for as the generation got confused with these undesired tiles. Areas would be too close to each other or lakes would be generated too close to each other, making the process a lot more challenging. To overcome this as part of my algorithm rules, I specifically filter for undesired tiles, once those tiles are spotted, that tile and sometimes the area around it are set to water tiles. This eliminates broken generation by cutting down on the total amount of land nodes on the island and breaks it up further in order for each texture to connect seamlessly.

Island Population

Once the Island was successfully generated and stored, I then had to populate it with random game objects like trees, barrels and buildings. In order to do this I created an algorithm inspired by Disk Sampling that would place my objects onto the Island

Trees

First I would dictate the amount of desired trees based on the size of an island. Trees would populate 7-10% of an island. The total amount of trees would then be distributed into clumps between 3-5 trees per clump to then be placed onto the island. This would space out my trees.

With the clumps set, I would select a random node on the island, perform a breadth first search from the selected node to a depth of 1-3 depending on the size of the clump. If the majority of those nodes were not occupied by other trees, I would be able to place trees onto the area that had been searched randomly. Each node that is occupied is then removed from the possible search area that will be re-used to place the remainder of the trees.

This way I am able to control how many trees I want placed, in what density, and the proximity of how close they are to each other.

```
void Island::GenerateTrees(std::vector<std::shared_ptr<Node>>& _nodes)
{
    int totalTrees = CalculateTreeCount(_nodes.size());

    std::vector<int> clumps;

    totalTrees <= 1 ? clumps = { 1 } : clumps = DistributeTreesIntoClumps(totalTrees); //more than one clump

    for(int clump : clumps)
    {
        std::vector<std::shared_ptr<Node>> PossibleNodes = ObjectPlacement::placeTrees(_nodes, clump); //get nodes to place trees

        for(auto &node: PossibleNodes)
        {
            gameObjects.push_back(std::make_shared<Tree>());
            node->updateOccupied(true);
            node->debugShape->setFillColor(sf::Color::Red);
            gameObjects.back()->setNodeId(node->getID());
            gameObjects.back()->setPosition(node->getMidPoint());
            node->AddPresentObject(gameObjects.back());
        }
    }

    ClearIslandNodeConditions();
    MarkAreaAroundTrees(); //marks area aound trees as occupied
}
```

Barrels

Barrel generation follows a scaled down version of the algorithm used to generate trees, it still searches through any remaining unoccupied land nodes and decides where to place the barrels based on if a search area is open enough.

```
std::shared_ptr<Node> findBarrelArea(std::vector<std::shared_ptr<Node>>& availableNodes)
{
    while (!availableNodes.empty()) {
        int randomIndex = std::rand() % availableNodes.size();
        std::shared_ptr<Node> startNode = availableNodes[randomIndex]; // random start node

        std::vector<std::shared_ptr<Node>> area = PathFindingFunctions<Node>::BreathSearchEuclydianIslands(startNode, 2);

        if (area.size() >= 6 && isAreaValid(area)) {
            for (auto& node : area) {
                node->UpdateIsBuildingArea(true); //mark area as in use
            }
            return startNode; //return node
        }

        availableNodes.erase(availableNodes.begin() + randomIndex); //remove node from search area (invalid)
    }

    return nullptr;
}
```

Buildings

Buildings are generated in the same way, but buildings take up a node space of 3x3, so if an island is too densely populated by trees, generating a building may not happen. I give 250 possible attempts to look for a node with an area large enough to place a building before moving on. This way, not every island will have a building.

```
int attempts = 0;
const int maxAttempts = 250;

while (!_nodes.empty() && attempts < maxAttempts)
{
    int randomIndex = rand() % _nodes.size();
    startNode = _nodes[randomIndex];

    if (startNode->getParentTileType() == LAND && !startNode->IsInBuildingArea()) {
        TownArea = PathFindingFunctions<Node>::BreathSearchEuclydianIslands(startNode, 3);

        if (TownArea.size() >= 17) {
            break;
        }
    }

    _nodes.erase(_nodes.begin() + randomIndex);
    attempts++;
}

if (TownArea.size() < 17) {
    std::cout << "Failed to generate town area within 250 attempts.\n";
    TownArea.clear();
    continue;
}
```

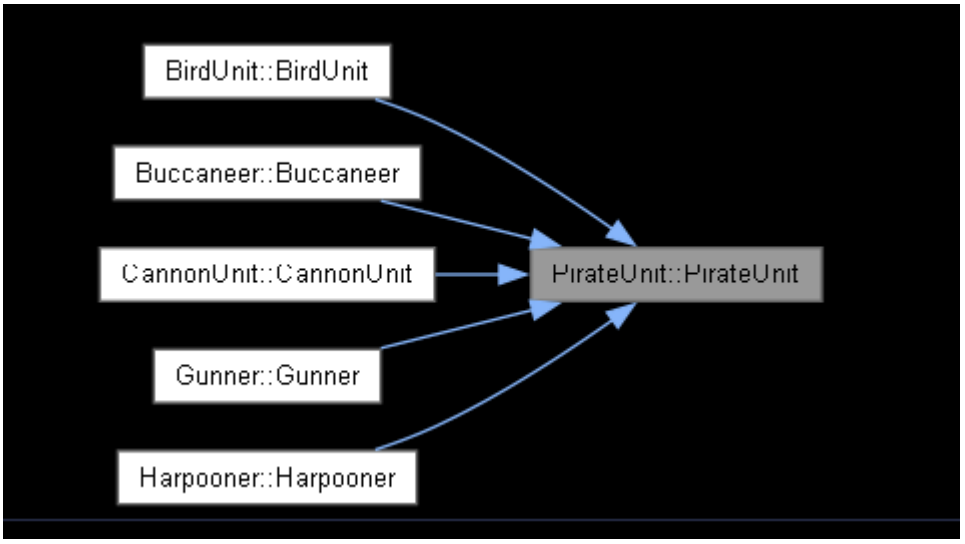
The generation would combine these 3 game objects to generate a desirable Island as such.

BattleSystem

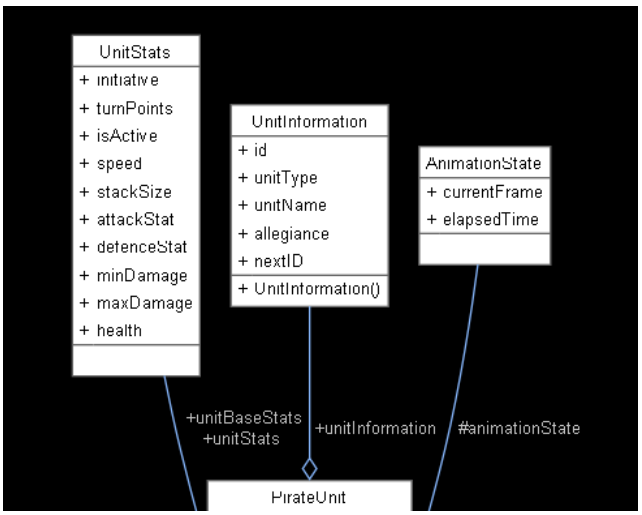
I wanted to incorporate a turn based battle system inspired by games like Heroes of Might and Magic. The aim was to create a dynamic battle system that would incorporate an initiative system for units to move in a sequential order between the player and the in-game enemies, as well as create a way for the enemies to control their armies in order to intelligently attack the players Units. To split up this task I would have to create Units, The mechanics of pathfinding and attacking in the turn based system, and create a way for the enemies to play against the player.

Army System

The army system follows a simple hierarchy of each unique unit inheriting from a PirateUnit base class where they can then be organised into separate individual armies. I created many individual units to follow this order.



Each unit would be responsible for themselves, as well as what their current state is, who they belong to and what stats they provide.



This way it's easy to expand upon the Pirate Unit system when you can easily create new units by simply altering stats and using different sprite sheets.

To store these units, the player and enemies are given an Army object which stores a vector of these pirate units. The army class is able to Add/Remove and combine units inside of the army in context. It is then simple to control the army by calling those functions, allowing the army to be dynamically adjusted inside the game, which would be important as the army needs to grow/diminish.

These functions control the adding and combination of units inside of an army:

```
void Army::addUnit(std::shared_ptr<PirateUnit> _unit)
{
    if (!combineUnits(_unit)) {
        army.push_back(_unit);
    }
}

void Army::addUnitNoCombine(std::shared_ptr<PirateUnit> _unit)
{
    army.push_back(_unit);
}

bool Army::combineUnits(std::shared_ptr<PirateUnit> _unit)
{
    auto it = std::ranges::find_if(army, [&](const std::shared_ptr<PirateUnit>& unit) {
        return unit->unitInformation.unitName == _unit->unitInformation.unitName;
        });

    if (it != army.end()) {
        (*it)->addToCurrentStack(_unit->getStackSize());

        _unit.reset();

        return true;
    }

    return false; // No matching unit found
}
```

Using the values from the inherited structs of each pirate unit, I am able to filter the existing vector to find If the army already contains a unit of the same type that is trying to be added. I am then able to add onto the existing units stack, increasing their size and then discarding the duplicated unit that is trying to be added.
This way there is a dynamic army that can be adjusted, as well as individual distinct units that can be used.

Battle Scene

I had to create a separate battle scene that would be incharge of handling this part of the game. The scene contains its own tile map that is used as the area your pirate units can move across. How a unit moves is determined by their predefined stats, in this case their speed. The speed value is used in a **breadth search** to create a valid area for the player to move their unit. The scene explicitly checks that if you are trying to move a unit from one node to the other, that node has to be within the nodes that have been traversed and added to your unit's search path. The unit can then utilize the Astar pathfinding algorithm to move towards their intended node.

```
void BattleScene::moveUnit()
{
    if (path.empty()) {
        finalizeMoveUnit();
        return;
    }

    //distance to next node in path
    sf::Vector2f distance = path[currentNodeInPath]->getMidPoint() - currentSelectedUnit->getPosition();
    float magnitude = Utility::magnitude(distance.x, distance.y);

    if (magnitude < 2.0f) {
        currentNodeInPath++;

        if (currentNodeInPath == path.size()) {
            //final node reached
            finalizeMoveUnit();
            return;
        }

        //updated distance for next node
        distance = path[currentNodeInPath]->getMidPoint() - currentSelectedUnit->getPosition();
    }

    currentSelectedUnit->moveUnit(Utility::unitVector2D(distance));
}
```

This function handles the path that has been made by using Astar and moves the currently selected unit to its proper position before finalising its turn.

A challenge then came of how to set up a turn order that updates every time a unit finished their action in order for the system to accurately work as a turn based system. For this I created an Initiative system that took both of the armies that are involved in the battle and filtering which units turn it was based on the initiative stat each unit had. The initiative system orders units based on their stat and returns the unit on top of that list, while also lowering their turnpoints so when their turn is over, lowering it so that they are sorted to the back of the turn order.

```
std::shared_ptr<PirateUnit> getNextUnit() {
    for (auto& unit : units) {
        if (unit->unitStats.isActive) {
            unit->unitStats.turnPoints += unit->unitStats.initiative;
        }
    }
    auto it = std::max_element(units.begin(), units.end(),
        [](const std::shared_ptr<PirateUnit>& a, const std::shared_ptr<PirateUnit>& b) {
            return a->unitStats.turnPoints < b->unitStats.turnPoints;
        });

    if (it != units.end()) {
        (*it)->unitStats.turnPoints -= 100;
        return *it;
    }

    return nullptr;
}
```

By calling this function at the end of a unit's turn, you successfully organise the new initiative order while lowering the turn points of the first unit so that the next time the function is called, that unit is added to the back of the line.
Implementing a timer then between turns so that there is a wait period allows the Initiative system to work reliably, always maintaining the same unit order as their values are adjusted accordingly. The currently selected unit can then be updated, and it is this unit that determines whether it's the players vs the computers turn.

```
currentSelectedUnit = tacticsArmyUI->initiativeSystem.getNextUnit();
```

Computer Decision

The next challenge was to find a way for the enemy to move their units and attack the player accordingly. The system would need to know if the current unit in the turn order was ranged, if there are any units around them, and then to decide which unit to attack if there are multiple units around them, or to move closer to a unit if there was none.

```cpp
void BattleScene::EnemyTurn()
{
    //Ranged Unit Attack
    if (currentSelectedUnit->unitInformation.unitType == RANGED)
    {
        clearArea(walkableNodesIDs);

        currentDefendingUnit = PickUnitToAttack(playerRef->getArmy()->getArmy());

        TriggerAttack();
    }
    else
        //Melee Attack
    {
        int selectedNode = -1;
        auto closeEnemies = ScanNearByUnits();
        if (closeEnemies.empty()) //No nearby units
        {
            selectedNode = SelectNodeToWalkTo();
            //add ability to attack if you on edge node
            scanForAttack = true;
        }
        else
        {
            selectedNode = SelectAttackNodeToWalkTo(closeEnemies);
            attackNode = selectedNode;
        }
        aStarPathFind(battleGrid[currentSelectedUnit->getCurrentNodeId()], battleGrid[selectedNode]);
        move = true;
    }
    startEnemyTurnTimer = false;
}
```

Here I filter between ranged and Melee units. Ranged units are able to skip directly to picking a unit to attack while the Melee unit has to scan their initial area. Nearby units are calculated by using the units speed stats

```cpp
std::vector <std::shared_ptr<PirateUnit>> BattleScene::ScanNearByUnits()
{
    std::vector<std::shared_ptr<PirateUnit>> possibleUnits;

    for (auto& unit : playerRef->getArmy()->getArmy()) {
        if (unit->unitStats.isActive) {
            if (EnemyMoveConditions::distanceToEnemy(currentSelectedUnit->getPosition(), unit->getPosition()) <= currentSelectedUnit->unitStats.speed) //unit close
            {
                possibleUnits.push_back(unit);
            }
        }
    }

    return possibleUnits;
}
```

Here the function can return any unit that is within range of the walking area that is set for each unit every time it's a new turn.
We can then determine if we want to walk towards a unit to attack them or walk closer to a unit while not triggering an attack. For this example, I'll use the function to move towards attacking a unit,

```cpp
int BattleScene::SelectAttackNodeToWalkTo(const std::vector<std::shared_ptr<PirateUnit>>& _possibleUnits)
{
    int shortestDistance = 1000;
    int selectedID = -1;

    currentDefendingUnit = PickUnitToAttack(_possibleUnits);
    auto enemyNode = battleGrid[currentDefendingUnit->getCurrentNodeId()];

    auto possibleAttackNodes = PathFindingFunctions<BattleGridNode>::BreathSearchNodes(battleGrid, enemyNode, 1); //nodes neighbouring enemy position

    for (auto& ID : possibleAttackNodes)
    {
        if (EnemyMoveConditions::distanceToEnemy(battleGrid[ID]->getMidPoint(), currentSelectedUnit->getPosition()) < shortestDistance)
        {
            shortestDistance = EnemyMoveConditions::distanceToEnemy(battleGrid[ID]->getMidPoint(), currentSelectedUnit->getPosition());
            selectedID = ID;
        }
    }

    return selectedID;
}
```

Here a unit is picked based on how much damage the currentlySelectedUnit will do to each unit in the opposing army:

```
std::shared_ptr<PirateUnit> BattleScene::PickUnitToAttack(const std::vector<std::shared_ptr<PirateUnit>>& _possibleUnits) const
{
    int highestDamage = 0;

    std::shared_ptr<PirateUnit> selectedUnit;
    for(auto & unit : _possibleUnits)
    {
        if (unit->unitStats.isActive) {
            int predictedDamage = DamageCalculations::calculateHitPointsLost(currentSelectedUnit, unit);
            if (predictedDamage > highestDamage) //Damage will always prioritise the closest units
            {
                selectedUnit = unit;
                highestDamage = DamageCalculations::calculateHitPointsLost(currentSelectedUnit, unit);
            }
            //Unit will die so prioritise
            if(!DamageCalculations::calculateDamage(unit->unitStats, unit->unitBaseStats, predictedDamage).isActive)
            {
                return unit;
            }
        }
    }
    return selectedUnit;
}
```

Depending on if the unit will die or the largest amount of damage is, the defending unit is set, where their currently occupied node is searched. This is done so that the unit can move closer while not being on top of the defending unit. A node is then selected that is not currently occupied, where the unit will move towards (see moveUnit function) to then trigger their attack. The turn is then ended as soon as the attack is triggered and a one second wait time has passed, for the initiative system to update and have the next unit take their turn.

End Screen

Finally, I needed to register the result at the end of each battle to make sure that each army is properly edited with the units that were lost in the battle, whether to completely remove a unit from a players army, and whether or not the system could detect in the players army was empty or if the enemies army was empty.

To do this I kept track of any dead units between the armies by using their own deadArmy vector. Each time a unit was to die I would mark the unit as ‘isActive = false’. This stopped any consideration of that unit in the computers decision and the initiative system while still keeping their body on the battle screen. The unit would then be added to the deadArmy, where a check would be triggered at the end of every turn to see if the currently active army has any surviving units remaining. It was important for me to store the dead army as It was needed to display the units lost at the end of a battle, and the ‘isActive flag would then enable me to remove those units from the army of the player if he survived the battle.

```
bool BattleScene::CheckBattleOver(const std::unique_ptr<Army>& _army)
{
    if (_army->isEmpty())
    {
        endBattleUI->updateUnitsDestroyed(enemyArmyDead);
        endBattleUI->updateUnitsLost(playerArmyDead);

        if (_army->getArmy()[0]->unitInformation.allegiance == HUMAN_PLAYER)
        {
            UIInterface->updateModeString("Enemy Wins");
            RemoveDeadUnits();
            endBattleUI->Lose();
        }
        else
        {
            UIInterface->updateModeString("Player Wins");
            RemoveDeadUnits();
            endBattleUI->Win();
            AddCoinsToPlayer();

        }
        currentState = END;
        endGameTimer.restart();
        return true;
    }
    return false;
}
```

Challenges

A major challenge in setting up this process was combining the initiative system with the turn order of the units. I came across a problem early on where if units in the same army would move after one another in the initiative bar, the unit they were attacking would not change and it wouldn't register the need to move towards a unit. So if a melee unit was still far away from their target, they would still trigger an attack. I had to carefully overlook how the system is set up ensuring there is no confusion with which unit is currently attacking/defending, which unit is the one taking damage, and when can the enemy army start to update their thinking process.

Managing dead units and transitioning back to the over world scene was also difficult as outright removing the would interrupt the initiative system that was currently on going, it would remove enemies from the field where their bodies were used as obstacles and accurately removing the correct amount of units to then change scenes proved difficult until I began storing enemies in a separate dead army that would then be strictly used for UI, This was I could keep the units as part of the players/enemies army while not removing them unit the battle was fully over.

Enemies

Individual enemies had to be created in order for the battle system to be triggered. These enemies would have to be in charge of controlling themselves though the environment of the generated island, as well as interact with their ability to dock their boats at those islands, be able to re board their boat and be able to interact with the player

Enemy Set-Up

For ease of editing and managing enemies, I used file loading with JSON. I wanted to create an easy system for initialising any enemies I would need without directly managing the enemies and their army through the code base. I set up my initial JSON structure :

```
 "Allegiance": "Blue",
      "PlayerReputation":100,
      "Enemies": [
         {
            "EnemyId": 0,
            "name" : "Alex",
            "army": [
               { "unittype": "Buccaneer", "amount": 14 },
               { "unittype": "Gunner", "amount": 10 }
            ]
         }
         ]
```

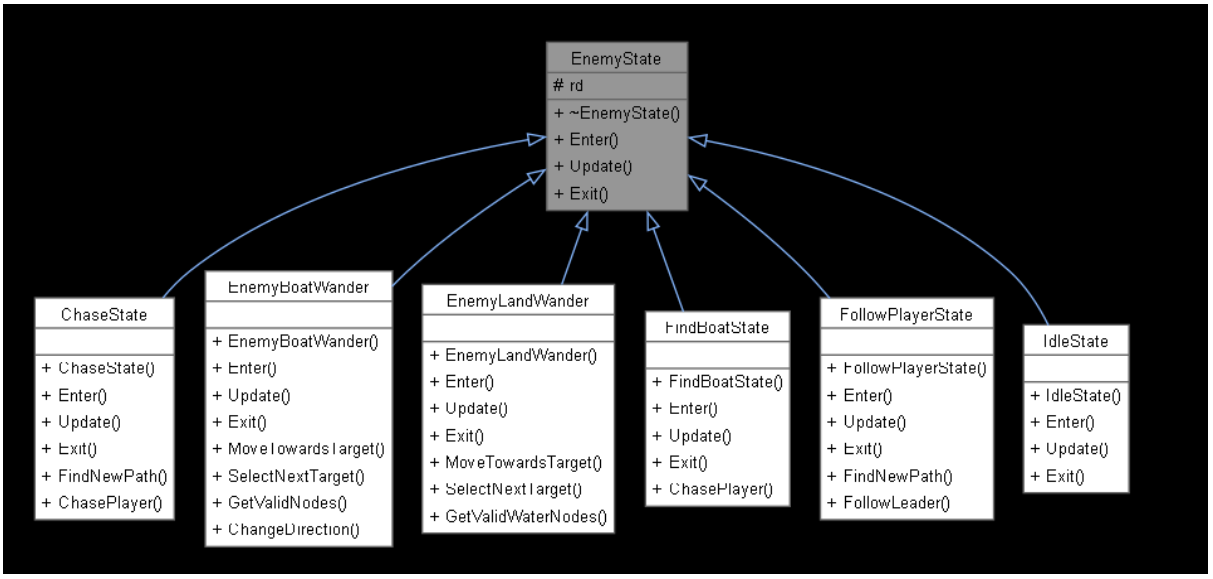
Using file reading, Enemies are effectively loaded in with their armies responding to the json structure. This function shows how the individual units and their quantities are assigned to the specific enemy that is being spawned into the world:

```
void GamePlayScene::InitialiseEnemyArmy(const std::shared_ptr<Enemy>& _enemyRef, nlohmann::json& jsonData)
{
    for (const auto& allegianceData : jsonData) {
        if (allegianceData["Allegiance"] == _enemyRef->GetEnemyAllegiance()) {
            // Search for the correct EnemyId within the allegiance
            for (const auto& enemyData : allegianceData["Enemies"]) {
                if (enemyData["EnemyId"] == _enemyRef->GetEnemyID()) {
                    _enemyRef->setPirateName(enemyData["name"]);
                    _enemyRef->setAllegiance(allegianceData["PlayerReputation"]);
                    for (const auto& unitData : enemyData["army"]) {

                        if (unitData["unittype"] == "Buccaneer") {
                            _enemyRef->getArmy()->addUnitNoCombine(std::make_shared<Buccaneer>(unitData["amount"], _enemyRef->GetEnemyTeam()));
                        }
                        else if (unitData["unittype"] == "Gunner") {
                            _enemyRef->getArmy()->addUnitNoCombine(std::make_shared<Gunner>(unitData["amount"], _enemyRef->GetEnemyTeam()));
                        }
                        else if (unitData["unittype"] == "Harpooner") {
                            _enemyRef->getArmy()->addUnitNoCombine(std::make_shared<Harpooner>(unitData["amount"], _enemyRef->GetEnemyTeam()));
                        }
                        else if (unitData["unittype"] == "Cannon") {
                            _enemyRef->getArmy()->addUnitNoCombine(std::make_shared<CannonUnit>(unitData["amount"], _enemyRef->GetEnemyTeam()));
                        }
                        else if (unitData["unittype"] == "Bird") {
                            _enemyRef->getArmy()->addUnitNoCombine(std::make_shared<BirdUnit>(unitData["amount"], _enemyRef->GetEnemyTeam()));
                        }
                        else {
                            std::cerr << "Unknown unit type: " << unitData["unittype"] << "\n";
                        }
                    }
                }
            }
        }
    }
}
```

Enemy States

In order for the enemy to be animated and work with the world around it, I decided to use a class based State machine in order to dictate how the enemy moves. Each enemy is equipped with a current state that can be managed, and each state has the capabilities to be

entered, updated and left.



If the enemy wants to enter a new State all it has to do is call their change state function which will automatically update their behaviour to the new state.

```
void Enemy::ChangeState(EnemyState* newState)
{
    if (currentState)
        currentActionState->Exit(*this);

    currentActionState = newState;

    if (currentActionState)
        currentActionState->Enter(*this);
}
```

Chase State

Responsible for changing the player. Determines whether or not the enemy is currently on a boat or if the player is currently on a boat. This distinction allows it to call different AStar pathfinding algorithms, which are updated every second rather than 60 times a seconds, and path find towards them accordingly. The chase state works similarly to how Units used pathfinding to move in the battle scene, except it factors in avoidance to nearby enemies:

```
distance = Utility::unitVector2D(distance);

sf::Vector2f avoidance(0.f, 0.f);
for (const auto& other : enemy.GetSurroundingEnemies())
{
    sf::Vector2f toOther = enemy.GetPosition() - other->GetPosition();
    float distance = Utility::magnitude(toOther.x, toOther.y);
    if (distance < 30.0f && distance > 0.01f)
    {
        toOther = Utility::unitVector2D(toOther);
        avoidance += toOther * (30.0f - distance);
    }
}

sf::Vector2f finalMove = distance + avoidance * 0.1f;
finalMove = Utility::unitVector2D(finalMove);

enemy.FaceDirection(distance);

enemy.SetPosition(enemy.GetPosition() + finalMove);
```

This way it will chance the player while avoiding other ships that may or may not be chasing the player.

Idle State

The idle state is responsible for putting the enemy on a wait timer while they are on land or if a player interacts with them. It also transitions enemies into other states after they have idled for a few seconds:

```
if (enemy.isOnBoat()) {
    enemy.ChangeState(new EnemyBoatWander(playerRef));
}else
{
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> disFindShip(1, 10); //10% to go back to boat
    if(disFindShip(gen) == 1)
    {
        enemy.ChangeState(new FindBoatState(playerRef));
    }
    else {
        enemy.ChangeState(new EnemyLandWander(playerRef));
    }
}
```

The idle state is used as more of a transition state between the enemies actions.

BoatWander

This is used as an intelligent wander behaviour while on a boat. In order to avoid the boat randomly selecting a water node that is behind them and travelling to it, I incorporated a directional system inside of the enemy that updates depending on which direction they are going. This way if the enemy is heading north, they will ignore any nodes that are behind their current position as I wanted to avoid the complete 180.

```
for (auto& node : enemy.getUpdateableArea()->getUpdateableNodes())
{
    if (!node->getIsLand())
    {
        sf::Vector2f nodePos = node->getPosition();

        switch (enemy.GetDirection())
        {
        case NORTH:
            if (nodePos.y < enemy.getCurrentNode()->getPosition().y) waterNodes.push_back(node);
            break;
        case SOUTH:
            if (nodePos.y > enemy.getCurrentNode()->getPosition().y) waterNodes.push_back(node);
            break;
        case EAST:
            if (nodePos.x > enemy.getCurrentNode()->getPosition().x) waterNodes.push_back(node);
            break;
        case WEST:
            if (nodePos.x < enemy.getCurrentNode()->getPosition().x) waterNodes.push_back(node);
            break;
        }
```

A random node is then selected from the available nodes and the player proceeds to use A Star to path find towards their goal node.

LandWander

This state shuffles from possible land nodes inside of the player's range when they are on an island. It selects one of the nodes to walk to and Begins to path find towards it, while avoiding occupied nodes. This state is able to transition to the Chase and Idle state

FindBoat

This state is incharge of returning to the enemies set boat that it received when spawning into the world. The enemy keeps a reference to which node the boat was disembarked on, so that they can then pathfinder towards that specific node and trigger the board boat function which transfers the enemies control to their ship.

FollowPlayer

This state is responsible for following the player if the enemy has been hired. They will constantly check the neighbours of the node the player is currently occupying. Based on the direction the player is going, they will accordingly choose the nodes that are opposite of that player's heading direction, to give off the effect following behind the player. If they are too close to the player they will slow down, and if they are too far from the player, they will change their state back to Wander.

```
case NORTH:
    for (auto& node : playerRef->getCurrentNode()->getNeighbours()) {
        if (node.second == 2 || node.second == 5 || node.second == 8)
        {
            if (!node.first->getIsLand() && !node.first->isOccupied()) {
                destinationNode = (node.first);
                break;
            }
        }
    }
```

In case the player has landed on an island, the state can detect to dock at the island:

```
// Handle disembarking if the next node is land
if (path[currentNodeInPath]->getIsLand() && enemy.isOnBoat())
{
    enemy.disembarkBoat(path[currentNodeInPath]);
    path.clear();
    detectPlayerClock.restart();
    return;
}
```

This allows them to pathfind on the island rather than the ocean. The state can also detect if a player has reboarded their boat and then trigger the Find Boat state to return to their boat.

Reputation System

The reputation system handles how enemies interact with the player. The changes between states and the transition into the battle scene is reliant on the reputation that the enemy has towards the player. The system holds a value as a representation of the relationship to the player, this value can be checked and altered as the game progresses.
By calling these functions :

```
bool isFriendly() const {
    return allegianceLevel == AllegianceLevel::Friendly;
}

bool isHostile() const {
    return allegianceLevel == AllegianceLevel::Hostile;
}

vate:
```

The enemy can check if they are hostile or not towards a player before deciding their move. For example if the enemy is hostile towards the player and they are in their Land Wander State, they are able to check their allegiance to the player if the player is within range, and instantly change to the chase state :

```
if (enemy.GetPlayerAllegiance().isHostile()) {
    for (auto& node : enemy.getUpdateableArea()->getUpdateableNodes())
    {
        if (node == playerRef->getCurrentNode())
        {
            enemy.ChangeState(new ChaseState(playerRef));
            break;
        }
    }
}
```

The Reputation system is the factor used for the interactions between the player and enemy. If the reputation is low, the enemy will chase the player and attempt to trigger a battle, if the reputation is high, the player is able to interact with the enemy and hire them, this triggers the Follow player state which then has the enemy follow you, now hired on your side.
The system itself is dynamic and can be altered at run time.

```
void changeAllegiance(int change) {
    allegianceValue += change;
    updateAllegianceLevel();
}
```

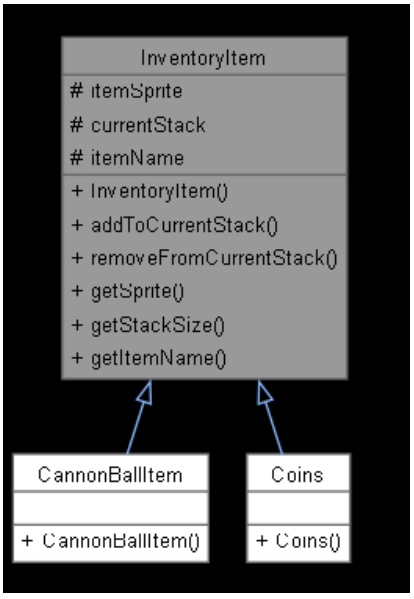
If the player has a currently following/Recruited enemy, the next time they enter the battle scene against an enemy, one of their recruited ally's units will join the player's army. The player is then free to use this unit to aid them in the battle, once the battle is over the unit is removed from the player's army, and returned to the ally's army if still alive. The end of the battle ends the allyship between the player and that current ally and the other Pirate returns to a wandering state.

```
void Player::addRandomEnemyUnitToArmy()
{
    int armySize = hiredEnemy->getArmy()->getArmy().size();
    auto unitRef = hiredEnemy->getArmy()->getArmy()[rand() % armySize];
    enemyUnit = hiredEnemy->getArmy()->removeUnit(unitRef);
    enemyUnit->updateUnitAllegiance(HUMAN_PLAYER);
    enemyUnit->updateUnitFacingDirection();
    playerArmy->addUnitNoCombine(enemyUnit);
}

void Player::removeEnemyUnitFromArmy()
{
    if (enemyUnit != nullptr)
    {
        playerArmy->removeUnit(enemyUnit);
        if(enemyUnit->unitStats.isActive)
            hiredEnemy->getArmy()->addUnitNoCombine(enemyUnit);
        enemyUnit->updateUnitAllegiance(hiredEnemy->GetEnemyTeam());
        enemyUnit->updateUnitFacingDirection();
        enemyUnit = nullptr;
    }
}
```

Other Systems

Inventory System

There is an inventory system that follows the structure of how the Army system is set up. It follows a similar inheritance where more items can easily be created and added to the players inventory.



Here you are able to add/remove from the current item you are trying to edit. This way if you had 4 cannon balls, you would be able to shoot 1, leaving you with 3 or collect 2 leaving you with 6. Each inventory is unique and only altered by their owner.
Barrels are equipped with an inventory of their own as they act as a chest in the game. This allows the transfer of items between inventories, where items can be removed from one inventory and added onto the other.

```
void GamePlayScene::transferInventoryItems()
{
    sf::Vector2f mousePos = static_cast<sf::Vector2f>(Mouse::getInstance().getMousePosition());
    for (auto& slot : RenderableInventory::getInstance().getSlots())
    {
        if (Mouse::getInstance().LeftClicked() && slot->getIsOccupied())
        {
            if (slot->getBackgroundSprite().getGlobalBounds().contains(mousePos))
            {
                std::unique_ptr<InventoryItem> item = currentObjectInteract->getInventory()->removeItem(slot->getOccupiedBy());
                slot->clearSlot();
                myPlayer->getInventory()->addItem(std::move(item));
            }
        }
    }
}
```

Here you can see the item being removed from the inventory that is opened, and being added to the players inventory, where addItem is responsible for either adding a new item or performing a search to see if it can combine items.

Interactable Game Objects

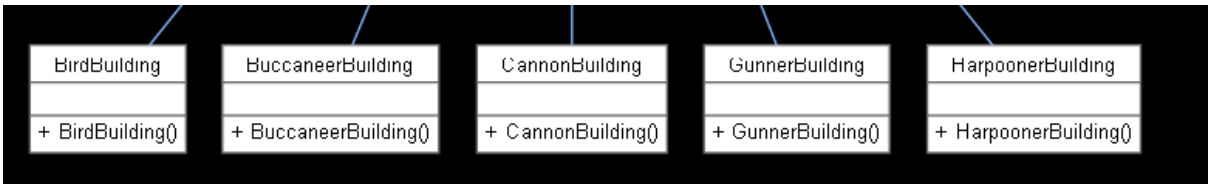
There are Two interactable game objects inside of the game, Buildings and barrels

Barrels

Barrels are used as a look chest and are responsible for storing items. These items are randomly assigned at generation between all of the available inventory items in the game. The player may then interact with the barrel in order to transfer the items from one inventory to another (See Inventory System)

Buildings

Buildings are responsible for the purchase and growth of different units across the game.

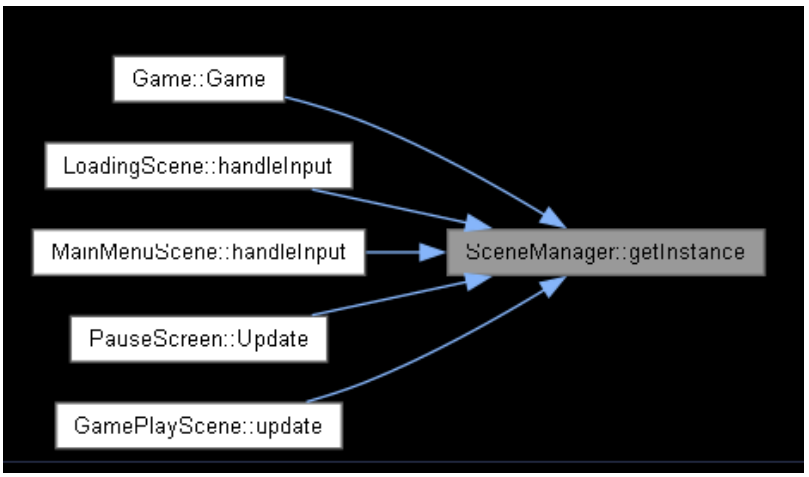


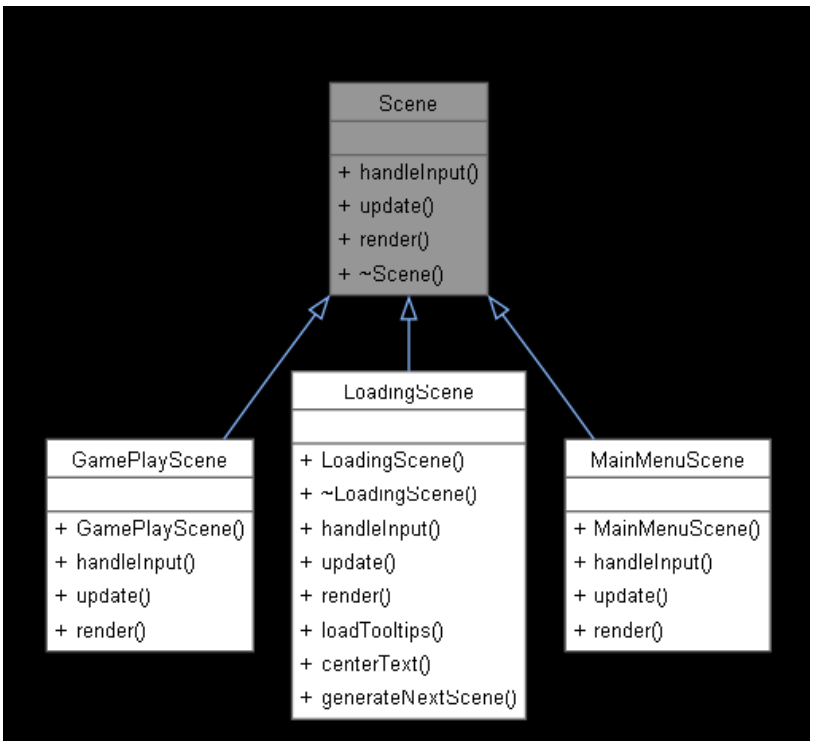
Each building displayed here is responsible for producing their specific unit. Upon interacting with a building, the building's information is passed to a UI where the player may choose to buy a unit. The building's UI is responsible for checking the players inventory when they click to buy a unit to ensure if they have sufficient gold in their inventory. If they do, the building removes that amount of gold from the players inventory and adds the designated amount of units into the players Army, starting a timer for when more units are available at said building.

```
if (purchase->IsTriggered() && availableUnits > 0)
{
    auto it = std::ranges::find_if(playerRef->getInventory()->getItems(), [&](const std::unique_ptr<InventoryItem>& item) {
        return item->getItemName() == COINS;
        });

    if (it != playerRef->getInventory()->getItems().end()) {
        if ((it->get()->getStackSize() - amountSlider->getValue() * costPerUnit) >= 0) {
            std::cout << "prev stack size : " << it->get()->getStackSize() << "\n";
            it->get()->removeFromCurrentStack(amountSlider->getValue() * costPerUnit);
            std::cout << "new stack size : " << it->get()->getStackSize() << "\n";
            availableUnits -= amountSlider->getValue();
            available->setText("Available", std::to_string(availableUnits));
            AddCharacterToPlayer();
            amountSlider->updateMaxValue(availableUnits);
            purchase->ResetTrigger();
```

Individual Screens

The Scene Manager is responsible for the handling of which scene is currently active.





Menu Screen

The Main Menu screen is responsible for kicking off the loading screen and closing the application.

Loading Screen

Screen that is displayed while the generation of the gameplay scene is complete. The Gameplay Scene is loaded onto another thread while the Loading scene is updated.

```
generationThread = std::thread([this]() {
    Scene* newScene = generateNextScene();

    {
        std::lock_guard<std::mutex> lock(sceneMutex);
        nextScene = newScene;
        sceneReady = true;
    }
    });
```

This allows the generation to be done simultaneously and the transition from scene to scene only possible when the gameplay scene is ready.

Gameplay Screen

In charge of all aspects of gameplay as well as the generation of the map.

Mini Map

A minimap is able to be displayed as the player progresses through the game. The minimap is created through individual nodes that are updated as the player traverses the map. The

minimap dictates the color of each node based on the status of the nodes that are being seen currently by the player.

```
MiniMapMenu::getInstance().GetMiniMap()->updateColor(index, node);
MiniMapMenu::getInstance().GetMiniMap()->updatePlayerLocation(myPlayer->getCurrentNode()->getID());
```

Land nodes are displayed green, water blue, sand yellow. There is a red indicator for the player's current position on the screen. The mini map reacts to the player's location as since the map is big, the map needs to be discovered in order to be revealed. This is done by presetting every node to be black and only updating the colors of the nodes that were already seen.

Animations

Animations are mostly handled through the Animator Singleton which is responsible for animating sprite sheets for pirate units.

```
void Animator::AnimateSprite(sf::Sprite& _sprite, AnimationState& _state, bool& _animationComplete, int _colAmt, int _rowNum, float _dt)
{
    const float frameDuration = 0.1f;

    _state.elapsedTime += _dt / 1000.0f;

    _animationComplete = false;

    if (_state.elapsedTime >= frameDuration)
    {
        _state.currentFrame++;
        if (_state.currentFrame > _colAmt - 1)
        {
            _state.currentFrame = 0;
            _animationComplete = true;
        }
        _state.elapsedTime = 0;
    }
    int col = _state.currentFrame % _colAmt;
    int row = _rowNum;

    sf::IntRect rectSourceSprite;
    rectSourceSprite.height = _sprite.getLocalBounds().height;
    rectSourceSprite.width = _sprite.getLocalBounds().width;
    rectSourceSprite.left = col * rectSourceSprite.width;
    rectSourceSprite.top = row * rectSourceSprite.height;
    _sprite.setTextureRect(rectSourceSprite);
}
```

This function for example ensures the accurate sprite sheet animation of the pirate unit while keeping it at a constant time no matter the specifications of your computer. This can be seen by keeping an elapsed time variable and dividing it by delta time.

Other animated features of the game include the FloatingNumber class, which is responsible for displaying the damage a unit did to another during the battle scene. This spawns in a number that will float out and fade away.

The initiative bar also animates depending on how the battle scene progresses. The bar moves right to left as the turns change, giving a smooth transition. It also removes units that have been destroyed throughout the battle, and updates the individual slots accordingly in a clear manner before continuing with the battle. It was challenging to come up with how to animate the bar itself as it would need to clearly move while also indicating and filling the next unit at the end of the initiative bar to keep things consistent. I also face a challenge of if a unit was duplicated on the bar, I would have to remove those two positions, while keeping the turn order correct and re adding on the next units towards the end of the bar.

I decided to go with an approach of 2 sprites in each slot, Where the main sprite would move as the bar updates, and to cover any empty slots, each slot would store the sprite of the turn after theirs, so it could fade in as the bar moves. This approach helped me indicate which slots should fade, and how far the others should move, before the bar would update everything, snapping back the original values, which would not be noticed.

```
void TacticsArmyUI::AnimateRemoveUnit(double dt)
{
    if (!armySlots.back()->isStillMoving())
    {
        animateRemoveUnit = false;
        finsihedAnimation = true;
        for (int i = armySlots.size() - (mutliplier - 1) ; i < armySlots.size() - 1; ++i)
        {
            //pushes back all slots between the last slot still rendered and the last slot
            slotsToFade.push_back(armySlots[i]);
        }
        return;
    }
    int multiplier = 1;
    for (int i = removedUnitIndex; i < armySlots.size(); i += extendedDupeUnitIndex) {
        armySlots[i]->FadeOut(dt);
        for (int j = i; j < i + extendedDupeUnitIndex && j < armySlots.size(); j++)
        {
            armySlots[j]->MoveSlot(multiplier); //moves all slots after the fading one
        }
        multiplier++;
    }
    mutliplier = multiplier;
}
```

This function for example would fade out any slots that were meant to be removed, while moving the rest, increasing the multiplier to indicate to the slot how far to move to the left and using the amount on the multiplier to help fade in incoming slots if more than one slot was removed.

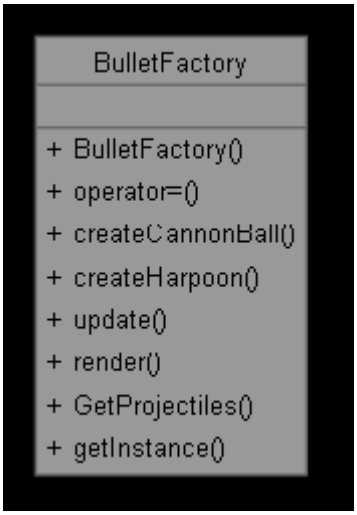
Factory Classes

Bullet Factory

Bullet factory is responsible for the creation/Update of projectiles inside of the game. These projectiles include Cannonballs and harpoons.

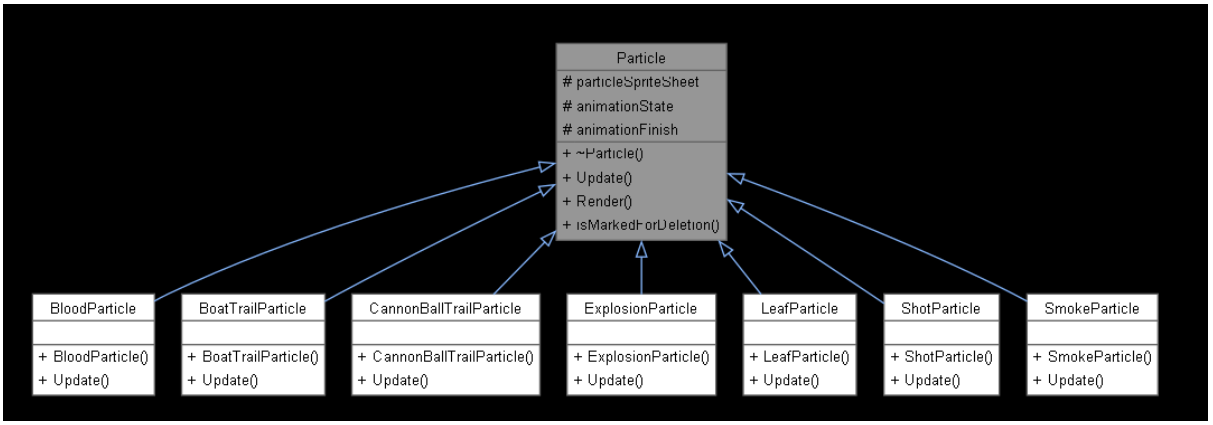
```
void update(double dt) {
    for (auto& proj : projectiles) {
        if (proj->getIsActive()) {
            proj->update(dt);
        }
    }

    std::erase_if(projectiles, [](const std::unique_ptr<Projectile>& projectile) {
        return !projectile->getIsActive();
        });
}
```

The projectiles are seen in the game inside of the battle system when the ranged units fire their weapons and while onboard a boat if the player fires a cannonball.



Particle Manager

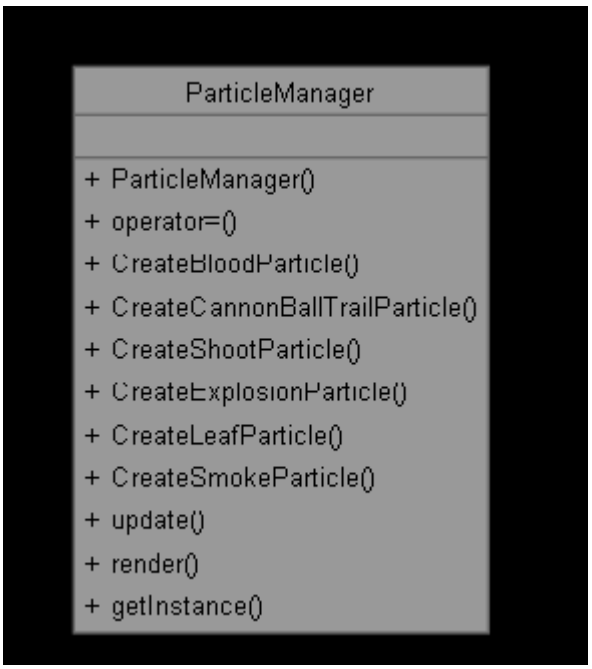
Particle Manager is responsible for Creating/Updating any in-game particles that are generated. Unique particles are created using the particle base class.



They are given a time time live on implementation and are monitored by the Particle manager, which will then delete them if they are flagged for deletion:

```
void update(float _dt) {
    for (auto& particle : particles) {
        particle->Update(_dt);
    }

    // Remove particles that are marked for deletion
    std::erase_if(particles, [](const std::unique_ptr<Particle>& particle) {
        return particle->isMarkedForDeletion();
        });
}
```

The particle manager can be called universally as it's a singleton class and has the capability of specifying which particle needs to be created.



Game Optimization

Singleton Classes

I use singleton classes for each of access to specific components in the game, Like the Camera and the Mouse as there is only one of them in my game, while also utilizing them for UI Screens such as player/enemy Interactions and different menus in the game. Instead of giving each building/Barrel their own individual renderable interaction screen. I use the singleton and pass information into it so it changes. This also saves searching for which objects UI is currently being rendered as I can call their singleton to close the menu from the gameplay scene in order to continue on with the game. This function showcases the many UI singletons used and the ability to easily close the UI .

```
void GamePlayScene::HandlePauseScreen()
{
    bool currentEscapePressed = sf::Keyboard::isKeyPressed(sf::Keyboard::Escape);

    if (currentEscapePressed && !previousEscapePressed)
    {
        if (Inventory::isInventoryOpen())
        {
            currentObjectInteract->getInventory()->closeInventory();
        }
        else if (HireRecruitUI::IsUIOpen())
        {
            HireRecruitUI::getInstance().CloseUI();
        }
        else if (PlayerTabMenu::isMenuOpen())
        {
            PlayerTabMenu::CloseMenu();
        }
        else if (AllianceDialogueUI::getInstance().isMenuOpen())
        {
            AllianceDialogueUI::getInstance().CloseMenu();
        }
        else if (MiniMapMenu::getInstance().isOpened())
        {
            MiniMapMenu::getInstance().CloseMenu();
        }
        else if (!PauseScreen::getInstance().isOpened())
        {
            PauseScreen::getInstance().OpenMenu();
        }

    }

    previousEscapePressed = currentEscapePressed;
}
```

Selective Rendering

As the map may be very large in terms of nodes, to save memory I calculate which world nodes are currently visible by the player. This is done by taking into account the camera class as well as the default size of the nodes. Each node The screen can see is then added to a set which can then be selectively updated and rendered, saving memory by not updating the possible thousands of nodes and off screen game objects / enemies etc.

```
void GamePlayScene::updateVisableNodes()
{
    sf::FloatRect viewBounds(Camera::getInstance().getCamera().getCenter() - (Camera::getInstance().getCamera().getSize() + sf::Vector2f(128, 128)) / 2.0f,
        Camera::getInstance().getCamera().getSize() + sf::Vector2f(128, 128));

    int minX = std::max(0, static_cast<int>(viewBounds.left / NODE_SIZE));
    int maxX = std::min(NODE_SIZE * mapSize - 1, static_cast<int>((viewBounds.left + viewBounds.width) / NODE_SIZE));
    int minY = std::max(0, static_cast<int>(viewBounds.top / NODE_SIZE));
    int maxY = std::min(NODE_SIZE * mapSize - 1, static_cast<int>((viewBounds.top + viewBounds.height) / NODE_SIZE));

    std::set<int> newVisibleNodes;

    for (int y = minY; y <= maxY; ++y) {
        for (int x = minX; x <= maxX; ++x) {
            int index = y * (NODE_SIZE * mapSize) + x;
            newVisibleNodes.insert(index);
        }
    }

    visibleNodes = std::move(newVisibleNodes);
}
```

For example here I render just the gameobject and building that may be housed in the nodes currently seen on the screen, while avoiding all of the other ones in the game :

```
for (int index : visibleNodes) {
    std::shared_ptr<Node> node = myMap->getFullMap()[index];

    if (node->GetObject()) {
        node->GetObject()->render(window);
    }

    if (node->GetBuilding()) {
        node->GetBuilding()->Render(window);
    }
}
```

Updateable Area

In order to not constantly check interactions between the player and game objects/building, and to limit the search area for the enemies to either select a new node to wander to or chase the player, I implement an updateable area that is stored in the player and each enemy.

This area is updated every time the player or enemy changes their position from one node to another.

```cpp
//searches neighbours of a start node based on the depth and adds them to a set to know which nodes to immediately update
void UpdateableArea::updateVisibleNodes(const std::shared_ptr<Node>& _startNode, int depth)
{
    updateArea.clear();

    //determines area, 3x3, 5x5 if depth of 1 then area is 3x3
    int iterations = (1 + 2 * depth) * (1 + 2 * depth);
    int iterCount = 1;

    std::queue<std::shared_ptr<Node>> nodeQueue;
    nodeQueue.push(_startNode);
    updateArea.push_back(_startNode);

    while (!nodeQueue.empty() && iterCount < iterations) {
        std::shared_ptr<Node> currentNode = nodeQueue.front();
        nodeQueue.pop();

        auto neighbours = currentNode->getNeighbours();
        for (auto& neighbour : neighbours) {
            //add node if it's not already in my area and i haven't reached my limit of iterations of neighbours
            if (std::ranges::find(updateArea.begin(), updateArea.end(), neighbour.first) == updateArea.end() && iterCount < iterations) {

                iterCount++;

                updateArea.push_back(neighbour.first);
                nodeQueue.push(neighbour.first);
            }
        }
    }
}
```

Here you can see me update the area around the player on a specific depth so that I can expand or lessen the area for different purposes. For example the player's updatable area will be small (Only the neighbours of his current Node). This is done so that aspects like collision checks between nodes/objects or the interaction of the player between enemies/Objects are done much faster as only 9 nodes need to be processed instead of a larger area. This greatly improves the run time of my game.
The enemies also use this for their behaviours. Since they are constantly checking to see if the player is within their area, as soon as he is, the A star pathfinding doesn't need to be across the entire map and is confined to the general search area.

References

Web-site

Craft of Coding. (2021, July 12). The Square-Diamond Algorithm for 2D Surfaces I: Basics. [Blog post]. Retrieved from https://craftofcoding.wordpress.com/2021/07/12/the-square-diamond-algorithm-for-2d-surfaces-i-basics/

Sebastian Lague. (2019). Coding Adventure: Wave Function Collapse [Video]. YouTube. Retrieved from https://www.youtube.com/watch?v=slTEz6555Ts

The Coding Train. (2018). 10.6: Diamond-Square Algorithm - Nature of Code [Video]. YouTube. Retrieved from https://www.youtube.com/watch?v=dFYMOzoSDNE

sighack. (n.d.). Poisson Disk Sampling: Bridson’s Algorithm. Retrieved from https://sighack.com/post/poisson-disk-sampling-bridsons-algorithm

Heaton, R. (2018, December 17). WaveFunction Collapse Algorithm. Retrieved from https://robertheaton.com/2018/12/17/wavefunction-collapse-algorithm/