Computer Games Development CW208
Technical Design Document
Year IV

Joshua Boyce Hyland
C00270197

Joshua Boyce Hyland
C00270917

11/4/2025

Contents

Architecture Overview:

The core game loop is centred around a modular scene-based system. Each scene is derived from an abstract `Scene` Class that requires an update, render and multiple handle inputs functions. This design allows for the different states in the game to encapsulate their specific logic cleanly and not interfere with each other.

The `Game` class holds a pointer to the current scene which then is called each frame. This allowed for a lot of modularity and separation of responsibility. Making sure a single scene isn't trying to accommodate to the function of multiple. This also allows for each gameplay mode to be developed and tested independently while adhering the structure of the game.

A key component shared between many scenes is the `GameObject` class. This base class defined 2 simple and common components shared across many classes. A Enum tag indicating what object is and a `sf::Sprite` which is its visual representation which also holds information of where it is in the world (position and rotation). This unified design helps with in limiting collision checks to only neighbouring cells to the object, as each cell holds a set of game objects currently occupying the cell, which avoided expensive global checks constantly between multiple different classes. It also allows for the mini map system to work easily, as most object derive from this class, it makes keeping track of them within the world very simple.

The world is broken up into cells which are encapsulated into a grid, each portion the game the player walks around in is made up into separate `Grids` (their own space station and other generated space stations); the cells in these grids hold `Nodes` which are used for pathfinding and searching. These `Nodes` hold information such as costs for A start heuristic such as Euclidian and Manhattan along with their indices based on the row and column. This setup allows for traversal through the `Grid` to be efficient and simple. Along with these `Nodes` the `Cells` also hold a set of `GameObjects` which as previously mentioned are for making collision check efficient and simple.

Resources are also efficiently managed in my game with the uses of multiple singletons which have the responsibility of centralising of game content These include the `TileLibrary`, WorldItemLibrary Each library handles loading, construction and categorizing of objects based on an index or enum class. This design leaves the construction of objects to one class which then hands out copies of objects to the users.

Similarly, all these classes rely on a `Loader` class, which ensures textures and fonts are only loaded once. The loader allows for loading many assets at once from a single folder and retrieving previously loaded assets by using the path as a key within a map of textures. This loader optimises memory usage and asset management.

The game also features an AI decision tree which is used to control the NPC behaviours. Each NPC get a custom-built decision tree based on its desired behaviour. Each NPC holds a pointer to a base node which is updated every frame to decide on the current behaviour of NPC based on certain conditions, such as whether they have been fired at, requiring them to dodge. The Decision tree is made up derived node types like `Sequence`, `Selector` and `Condition` which allow for complex behaviours to emerge from simple components.

When designing my architecture for the game I prioritized to keep modularity in mind making for uncomplicated structures which could be reused in many systems.

Decision Tree

Design Specification:

Purpose:

The decision tree system is used to decide on what the current behaviour of the NPC should be and check whether it needs to transition to a different behaviour such as attacking to dodging. This provides a flexible and scalable way of controlling NPC behaviour based on the games context while keeping the decision-making logic separate from the behaviour implementation logic.

System Overview:

Each NPC owns a decision tree which are made up of modular node types like `Selector`, `Sequence` and `Condition`, all derived from a `DecisionNode` base class.

The tree is traversed every frame, starting from a root node, and evaluates a series of conditions which will end up producing a action node which contains a behaviour, like `AttackingNode` which will be performed that frame.

The tree starts from a root `Selector` Node, which then evaluates its child nodes. Which could be any of the following types :

- Selector Node: Evaluates each of its children until one returns a non nullptr.
- Condition Nodes: Evaluates a Boolean function (e.g `attackPlayer()`, `bulletDetection()`)
- Sequence Node: Evaluates child nodes. Returning a succeeding behaviour node or a nullptr,
- Behaviour Action Node: The final action which holds a behaviour.

Each nodes shares the interface given by its base class `DecisionNode,` where they implement their versions of `decide()`, onEnter(), `preform()` and `onExit()`.

Key Components:

NPC: Owns and updates the decision tree.

Decision Node: Base for all nodes.

Selector/Sequence: Nodes which control the flow.

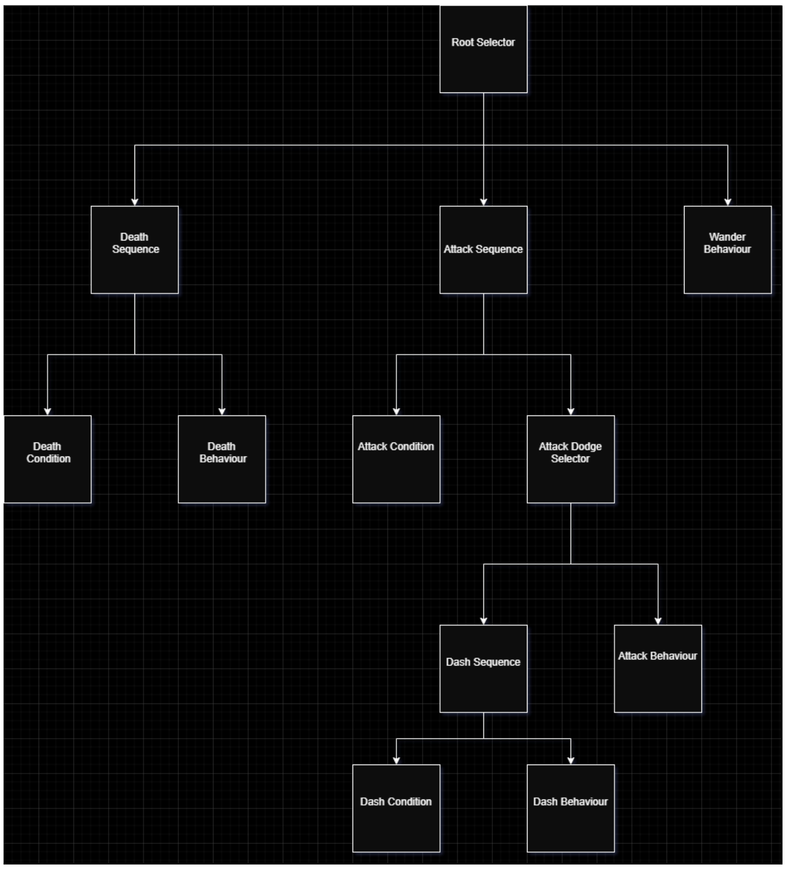
Condition: Wraps logic checks using `std::function`

(Action) Behaviour Node Classes: Perform all the actual in game behaviours.

Design Rationale:

This modular, behaviour driven system separates the actual decision making from the implementation. Making it so the behaviours themselves are self-contained and handle only their own logic. New behaviours or changes to the logic of a decision can be added without changing existing code, simply add a new branch. NPCs can also have completely different tress if needed as creating one is just a case of selecting the components you want to it. It scales quite well and makes setting individual behaviours quite simple, being able to single them out.

Example Flow:



Class Specification:

DecisionNode

Purpose: Base Class for all decision tree nodes

Key Functions:

`-virtual DecisionNode* decide(float deltaTime)`

`-virtual void onEnter()`

`-virtual void perform(float deltaTime)`

`-virtual void onExit()`

`-virtual Behaviour* getBehaviour()`

Condition

Purpose: wraps a Boolean condition using `std::function`

Key Functions:

`Condition(std::function<bool>() condition)`

- Takes in condition as argument

`Void Decide()`

- Returns itself if the condition is true, other wise nullptr indicating the condition has failed

Selector

Purpose: Returns the first successful child node.

Key Variables:

- `Std::vector<std::unique_ptr<DecisionNode>> m_decisions` – Child nodes

Key Functions:

- `Void decide()` - Iterates through its children nodes (m_decisions) and returns the first one that doesn't return a nullptr

Sequence

Purpose: returns the last succeeding child node, being the node containing the behaviour or a nullptr.

Key Varibles:

`std::vector<std::unique_ptr<DecisionNode>> m_children`

Key Functions

`Void decide()`

- Returns nullptr if any child fails; otherwise returns last successful child

WanderNode, DashNode, AttackingNode, DeathNode

Purpose: these are our action nodes. Each wraps its individual behaviour like Wandering Attacking etc

Key Functions:

`Void decide()`

- Returns itself.

`Void onEnter()`

- Called just before the change over of behaviours and allows for the each behaviour to be entered and prepped in its own way if anything needs to be prepared or changed before changing state.

`Void perform()`

- Preforms the behaviour every frame

`Void onExit()`

- Called just before the exiting the behaviour and Allows for any final actions to be specified before the change of state.

Animator

Design Specification:

Purpose:

The Animator is responsible for the loading, managing and updating sprites animations for various game objects through out the game such as NPCs, Weapons and the player. It decouples animation from gameplay logic and ensures game actions are synched with animations without cluttering the behaviour or weapon code

System Overview

The Animator is initialized with:

- A base path for texture folders
- A list of folder names where the animations are held (e.g “/Idle”, “/Run”)
- A reference to an sf::Sprite to animate

- An origin offset

The textures are all loaded in using the loader singleton. And turned into an `Animation.` The animation class represents a single animation. It holds the current frame the animation is on and the number of frames in the animation along with a vector of the textures.

We then have a vector of animations which we access with an index to get the desired animation. The public variable `m_currentState` is then accessed by the user of the animator to decide which animation is getting currently played.

The `animate()` function is called every frame to animate the sprite onto its next frame if enough time has passed since the previous frame. Once completing the animation rests. This function also returns a Boolean which indicates whether the animation has been completed. This allows for other systems to react when the animation is complete. In the case of weapons, we want to fire the bullet on the last from of the animation.

The animator is used across classes:

- BasePlayer uses it to animate its sprite
- `NPC` uses it within it behaviour to play different animation depending on the behaviour and also to determine when we exit our dash behaviour being at the end once the animation is done playing.
- Weapon and all its derived classes use the animator to animate its sprite and to decide when a bullet gets fired.

Key Components:

- `Animation`: class used to store its textures and corresponding information
- `Loader`: Singleton used to load textures and prevent duplication.

Design Rationale:

The animator was designed for modularity and reuse in mind. It allows for any single system to specify any textures for its animation while not requiring them to manually load or manage them. This keeps responsibility away from the class using it and allows the animator to be hugely reusable.

By returning the animation status through the animate function, it also allows other classes to synchronise events to then animations like shooting or dashing. Which allows for more customizable and specified interactions.

Class specification:

Animator

Purpose: manage and load sprites and animation frame by frame.

Key Variables:

`sf::Sprite& m_sprite`

- reference to sprite to animate

`std::vector<animation> m_animations`

- all animations

`int m_currentState`

- index for m_aniamtion.

`int m_elapsedTime, m_elapseReset`

- timing controls for playing next frame

`bool m_assigneLastFrame`

- whether to assign reset the animation back to frame 0 after finishing (used primarly for weapon so we reset after firing)

Key Functions:

```
Animator(std::string t_texturesPathBase, std::vector<std::string>
t_animationFolderNames, sf::Sprite& t_sprite,  sf::Vector2f
t_spriteOrigin )
```

- constructor which takes in the paths for textures, reference to sprite which will be animated and the sprite origin.

`Bool animate()`

- Advances out animation forward and return true if we finish a loop of an animation

Animation struct

Purpose:

Represents a single animation with frames

Key Variables:

`Int m_currentFrame`

- the current frame Index

`Int m_maxFrames`

- totale frames in the animation

`std::vector<Texture*> m_sprites`

- pointers to loaded frame textures

Pathfinding Agent

Design Specification:

Purpose: The Agent class is responsible for handling the NPCs movement across a grid-based world. It calculates and follows a path to a specified goal position.

System Overview:

Each `NPC` has an instance of the `Agent` class, which initialized with a pointer to the current game world `Grid` and the NPCs starting position.

When a goal is set, usually by a behaviour, with the `pathFindTo(Node* t_goalNode)` function:

- `Search::AStart()` is used to find the best path to the goal
- The result is then stored in a dequeue of `Node*` targets (`m_currentPath`)
- Agent then starts moving toward its first target (`m_target`) which gets popped from the top of the `m_currentPath` queue.

On every frame, the `update()` function gets called and if the agent has a target it will:

1. Move towards the target using the `VectorMath::directionVector()` function to move in the right direction.
2. Once close enough to the target, it will pop the next node from the `m_currentPath` dequeue for its next target.
3. Upon reaching its last target (when the dequeue is empty) it will notify its user

Every agent has a pointer to a `AgentUser` which is an abstract class for users to derive from and implement their own `reachTarget()` function. In our case the Behaviour class inherits from the AgentUser Class while classes like `Wander`, `Attack` and `Dash` implement their own `reachTarget()` implementation. Allowing for all these behaviours to specify their own following action upon reaching a target in sort of a subscriber design.

Key Components:

- Grid/ Cell/ Node: environment for movement and pathfinding system to navigate
- Search: Provides search algorithm A* for pathfinding.
- AgentUser: Interface for notifying behaviours when goals are reached.
- Behaviour: Derived classes implement AgentUser virtual functions to coordinate animation and pathfinding

Design Rational:

The agent system cleanly separate movement logic from behaviour logic, allowing for the NPCs single Agent instance to be used across behaviours.

By using the `AgentUser::reachedTarget()` function, behaviours stay loosely coupled with each behaviour getting to its own unique action.

This system allows for ease of testing and extension, the algorithm could easily be switched from A* to a different algorithm without having to change the rest of the systems surrounding it.

Class Specification

Agent:

Purpose: Handle Pathdinifng and movement for NPCS using a grid base A* system.

Key Variables:

- Sf::Vector2f m_position – current position
- Sf::Vector2f m_direction – current direction vector for movement
- Flat m_speed – current speed
- Node* m_currentNode – current node/ cell we are situated in
- Node* m_target – target node we are currently heading for
- Std::deque<Node*> m_currentPath – remaining nodes in path to go
- Std::queue<Node*> m_previousPath – used for visualisation of path
- AgentUser* m_user – pointer to current user (behaviour) to notify of target being reached.
- Grid* m_map – pointer to current grid being traversed

Key Methods:

- Agent(Grid* t_grid, sf::Vector2f t_positon) – Takes in the grid we will be working on and a start position
- Void update(flaot deltaTime) – updates our position and moves us along the us along the path if there is currently
- Void followPath(float deltaTime) – calculates a direction for agent to go in and manages whether we are close enough to our target to get the next node in our path list. Along with notifying the user upon reaching the goal node.
- Deque followPath(Node* t_goalNode) – finds a path to a goal node using A* star if the goal is reachable
- Void reset() – resets the agent, used in `onEnter()` functions for behaviour action nodes before moving to next behaviour to clear the current path and target so the agent is ready to use again.

AgentUser (Interface)

Purpose:

Allows the agent to notify its current user of when it reached its designated goal/

Method:

- Virtual void reachedTarget() = 0 – called by agent when its reached its target. Implemented by all Behaviours.

The Agent class handles movement and pathfinding for NPCS. Its designed to work in the grid-based structure of the game world and moves the NPC using a precomputed path using A* algorithm. Each agent stores its current position, movement direction, target node and path queue.

Pathfinding is started through the pathFindTo() function, which uses the Search class to create a path from the current node to the goal node using A*. once the path has been created the agent follows it step by step using the followPath() function updating position based on direction vectors and the chosen speed.

The agent also acts as a subscriber to it behaviour logic through eh AgentUser interface. When an agent reaches its target it notifies its assigned behaviour through the reachedTarget() function. This design decouples movement from the behaviour logic and allows for each behaviour to specify what it want to do upon arriving at its target position. Maing the agent very reusable across different behaviours.

This system works directly with the Grid, Node and GameoBject systems also.

Grid World:

Design Specification:

Purpose:

To spatially partition the world up into cells so that it the game world can be managed efficiently while keeping track of where objects are within the game world. Splitting the world up into cells also empowers other systems such as pathfinding, tile editing, object placement, Dungeon generation and collision management. Making it a core component to the game that ties many systems together.

System Overview:

Each Grid consists of a 2D vector of `Cell` objects. With each Cell containing:

- A `Node` for pathfinding
- A visual representation to assign textures to (`sf::Rectangle`)
- An optional job or world item pointer
- A `TraversalProperty` to indicate its walkability

The grid can be dynamically edited using the `TileEditorBox` and supports the:

- Real time placing of tiles (`placePiece()`)
- Deletion of tiles or cell jobs (`deletePiece()`)
- Position based tile selection (`cellSelection()`) which is used across the system to get the current location of many different type of game objects.

To aid performic ethe grid only draw visible cells based on current camera view and when gameplay starts, `setForGameplay()` collects the valid walkable `Cells` and builds the `Node` neighbours for efficient A* pathfinding.

Key Components:

- `Cell` – Contains visual data, properties and a `Node`
- `Node` - holds all data relevant to A* start pathfinding such as costs, neighbours and also its row and cell in the grid.
- `TraversalProperty` – Enum which defines whether a cell is Walkable or not
- `TileEditorBox` - Tool used to modify the grid during editing
- `Agent`/ `Behaviours`: interacts with grid for movement and decision making

Design Rationale:

The grid was built to be modular, takin to account for performance and also support editing:

- It allows for the tiled level design which allows the player to edit tiles as they wish
- `Cells` were used to as they simplify collision and selection logic
- The grids make the dungeon generation much easier as they are very customisable in size.
- `setUpNeighbours()` only links walkable neighbours to `Nodes` ensure pathfinding remains optimised
- Rendering only the visible `cells` improved the performance in such large environments like the `Dungeons`.

Class Specification:

Grid

Purpose:

Manage all cells in a way which take performance into account and supports the game world

Key Methods:

`Grid( int rows, int cols, float width, float height, sf::Vector2f startPos)`

- constructor specifies the dimensions of the grid and the starting position

`Void draw( sf::RenderWindow& window)`

- draw the relevant cells within the cameras view

`Void placePiece( sf::Vector2f t_mouseClick, EditorItem* t_tile)`

- places a tile on the cell at the mouse click giving it the tiles texture, traversal properties or job.

`Void deletePiece( sf::Vector2f t_mouseClick, TraversalProperty t_currentEditorSection)`

- deletes the tile or the job at the mouse click depending on what section of the editor you are currently in

`Void setForGameplay()`

- sets up the grid for gameplay by deactivating cell which don't have textures.

`Void setUpNeighbours( bool t_requireWalkable)`

- sets each node in the grids neighbours nodes to its surrounding walkable nodes, if the Boolean is set to true the nodes do not need to be walkable, this is used during the dungeon generation.

`Cell* cellSelection(sf::Vector t_positon)`

- returns a cell in the grid at the passed through position If there is one, if there is not a cell at that position meaning it is out side the grid a nullptr will be returned

`Cell* getRandomTraverableCell()`

- gets a random cell which is walkable in the grid, used to set up enemies.

Cell

Purpose:

Key Variables:

`Sf::RectangleShap m_body`

– visual representation

`Node m_node`

- node with pathfinding information nodes

`TraversalProperty m_property`

- whether the cell is walkable

`WorldItem* m_cellJob` –

- job object (if there is one)

`std::unordered_set<GameObject*> m_gameObjects`

- `GameObjects` currently in the cell, used a set for easy removal and addition as `GameObjects` will adding and removing constantly.

Node

Purpose: Used for pathfinding and holds A* data and neighbouring connections.

Key Variables:

- Std::vector<Node*> m_neighbors – surrounding nodes
- `Int m_row, m_column` – indexes in the grid
- `Float m_euclidian, m_manhattan, m_heuristic` – A* star data
- `Node* m_previous, bool m_marked, bool m_beingChecked –` Node traversal variables.

// pictures examples, uml

Base Builder

Purpose:

To allow the player the ability to edit their base using the grid system in place that would affect how they walk around their base.

System Overview:

The base building system in the game works off a `TileEditorBox` class which holds a selection of `Tiles` for the user choose from. The items in available for selection all derive from a `EditorItem` base class, the editor return these items make it easier to have a range of items in the editor.

The TileEditor Class allows the player to choose a **`Tile`** or a `WorldItem` from the editor. The tile will replace the texture on a Cell from the grid while the WorldItem will place it above the `Cell.`The TileEditor loads in all the tiles from a singleton class called TileLibrary which is the centralised point for accessing tiles. Similar it does the same for world items using the WorldItemLibrary. The Tile editor display a simple UI with the tiles depending on the current tab you are in which is split up into different sections.

When the `EditorItem` (base class from both `WorldItem` and `Tile`) is selected from the `TileEditorBox` it is tracked to the mouse position until it is placed on the grid.

The static `MapSaver` Class, which uses `nlohmann::json` library to to save and load the players space station to a json file.

Key Components

- `TileEditorBox` – UI tool which the player interacts with to access the `Tiles` and `WorldItems`
- `TileLibrary` / `WorldItemLibrary` – The centralised libraries which construct our `Tiles` and `WorldItems` the `TileEidtorBox` uses to get the items.
- `Grid` / `Cell` – The base where the tiles will get places on.
- `TraversalProperty` – used to indicate the type of tile you are placing and the section for the `TileEditorBox`
- `MovableCamera` – A movable Camera which is Used in the view move around the base via the middle mouse click

Design Rationale:

Base builder was designed around the used of the TileLibrary and `WorldItemLibrary` which are used by the editor box to construct all the items it will have available in its editor box. The Grid system is also important in its design with functions like `placePiece()` and `deletePiece()` being key to the Tile Editors function.

Example Flow:

1. Player selects a wall from Ui (`TileEditorBox`)
2. They will use the middle mouse button to navigate to where on the grid they want to place the cell (`MovableCamera`)
3. Player clicks on a tile (`placePiece()`) which update the cell texture and `TraversalProperty`
4. Grid and gameplay logic immediately then reflect the updated state in gameplay

Class Specification

TileBoxEditor

Purpose: Holds has 3 section tabs which can be switched between using the arrow buttons by the section header. The first 2 sections holding `TraversalProperty::Walkable` and `TraversalProperty::Unwalkable` `Tiles` and the third holding the `WorldItems`.

`TileEditorBox` derives from the `UIEditorBox` which the `ShipPartEditorBox` also derives from. This class implements all the UI and sprites aspect of the UI box which the player interacts with to select a part.

Key Variables:

- **`UIEditorBox::std::vector<std::vector<sf::Sprite>>> m_uiSprites`** – these sprites are used as the buttons for getting the `EditorItems`. They the `EditorItems` are loaded from their respective libraries on construction and their textures copied to these sprites. These sprites then correspond to Items in the TileLibrary and `WorldItemLibrary.`

- **`UIEditorBox::`**sf::CirceShape m_button, m_button2 – Used to navigate the sections

Key Functions:

- **`UIEditorBox::checkForInteraction(sf::Vector2f t_mousePosition)`** – checks for interactions with the section buttons
- **`EditorItem* partSelectionCheck(sf::Vector2f t_mousePosition) -`** – checks for interactions with the `EditorItems`. sprites representation and then if one was clicked on it returns the corresponding `EditorItem` from either the TileLibrary or the `WorldItemLibrary`

TileLibrary

Purpose: To centralize the construction of all tiles in one singleton class.

Key Variables:

- `Static TileLibrary* instance` – the single instance of the class.
- `Std::map<TraveralProperty, std::vector<Tile>> m_tiles –` this stores every `Tile` of type in a map keyed to its `TraversalProperty`. Using a map here made sense as it made it easy to associate tiles to functionality also.

Key Functions:

- `Tile* getTile(TraversalProperty, Int)` – this is the function that gets the tiles from the map with eh passed through key and index.

WorldItemLibrary

Purpose: To centralize the construction of all `WorldItems` in one singleton class.

Key Variables:

`Static WorldItemLibrary* instance`

- the single instance of the class.

`Std::vector<WorldItem*> m_items`

- this stores every type of world type.

`std::function<void(SceneType)> m_sceneChangeFunction`

- This is a function that gets passed though and is needed by the world Items so they can change scene.

Key Functions:

`Tile* getTile(TraversalProperty, Int)`

- this is the function that gets the tiles from the map with eh passed through key and index.

MovableCamera:

Purpose: A movable sf::View which can zoom, follow positions and also implements a drag movement with the mouse.

Key Variables:

`Sf::Renderwindow& m_window`

- Reference to main window to set view to.

`Sf::View m_camera`

- the view that will follow our player or be moved around the base editor scene.

`Bool m_mouseDown`

- Indicates if the middle mouse button is pressed down, meaning we start to move relative to the `m_mouseClickPoint`.

`Sf::Vector2f m_mouseClickPoint`

- Where the mouse was clicked last.

`Sf::Vector2f m_mouseDelta`

- The Difference between the `m_mouseCurrentPosition` and the `m_mouseClickPoint`.

`Sf::Vector2f m_mouseCurrentPositon`

- he current position of the mouse.

Key Functions:

`void update()`

- Updates the `m_windows` view to `m_camera` and tracks where the mouse moves to if the middle button has been pressed

`void startMove()`

- This starts the dragging movement that get used in the editor to navigate around the world. It set the `m_mouseDown` to true and assigns the `m_mouseClickPoint`

`void move()`

- Gets the mouse current position and if the middle mouse has been pressed it calculates the delta between the `m_mouseCurrentPostion` and `m_mouseClickPoint` to create a delta which then moves the camera.

`void endMove()`

- Once the middle mouse has been release the movement is over and is sets `m_mouseDown` to false

`void follow(sf::Vector2f t_position)`

- Sets `m_cameras` centre to t_positon

`float zoom(float t_mouseWheelDelta)`

- Calculates a zoom from `t_mouseWheelDelta` and zooms `m_camera`

Dungeon Generation:

Design Specification:

Purpose:

The purpose of the dungeon generation system to generate a dungeon which is navigable using a procedural generation algorithm with the goal to produce unique interesting layouts using multiple generation steps including Delauney Triangulation and Minimum Spanning tree.

System Overview:

The DungeonGeneration class has a multistage process to generate its dungeon:

1. Initial Grid Generation - Spawns a specified number of grids at a random point within a radius of a point. These grids will have a random min and max width and height which is parameterized. Each `Grid` get spawned with a collider twice the size of the grid.
2. Room Separation – Here the rooms are separated until none of their colliders are over lapping
3. Room Culling – Rooms are removed that fall below a minimum size to keep rooms that are large enough to be usable.
4. Delauney Tringulation – Creates a super Triangle that encapsulates all the remaining rooms and creates triangulations between all room centres, then using circumcircles it removes any triangulations that don't meet the Delauney criteria.
5. Minimum Spanning Tree – converts edges created by Delauney Triangulation into a minimal set of edges between rooms using prims algorithm. Giving us the final edges which will become hallways.
6. Grid Placement – we then place the rooms within a enclosing grid.
7. Hallway Generation – Converts the edges into hallways between rooms using horizontal and vertical strips of cells.
8. Texture application – Textures are applied to the `cells` with traversal values

Key Components

- `Room` – has `Point` which stored all the edge and triangle information of where the room is connected to and stores cell where the room occupies on grid
- `Point` – holds `PointEdges` and `triangles` between points
- `Grid` – initial structures spawning in during the spawning and separation stage
- `TileLibrary` – Used to texture cells during the texturing stage
- `Dungeon`

Design Rationale

The design rational behind the dungeon generation was to make it possible to step thought different stages of the generation to get to see each of the stages in action. This is what brought along the switch statement of stages.

Class Specification

DungeonGeneration

Purpose: To create a procedurally generated `Dungeon` which can be traversed by `BasePlayer` and `NPCs`

Key Variables:

`Grid* m_dungeon`

- Grid which will be return as a `Dungeon`

`std::vector<Room*> m_mainRooms`

- Final rooms made from remaining `m_gridsGenerated` after culling stage

`std::vector<Grid*> m_gridsGenerated`

- These are initial rooms generated which will be cut down on later in generation, remaining get turned into `m_mainRooms`

`std::vector<sf::RectangleShape> m_roomCollider`

- Colliders used for `m_gridGenerated` during separation stage of generation

`std::vector<Triangle> m_finalTriangulations`

- Valid Triangulations after Delaunay triangulation stage

`std::vector<sf::Vector2f> m_separitionForces`

- The separation forces for each grid generated during the separation stage `int m_hallWayPadding`

- This specifies how big you want hallways to be in the `Dungeon`.

`m_minWidthGridGen, m_maxWidthGridGen, m_minHeightGridGen, m_maxHeightGridGen`

- These are parameters which specify the random min and max dimensions of the initial grids that get spawned in the start of generation in `GenerateInitalGrids()`

`int m_minRoomWidthFinal, m_minRoomHeightFinal`

- These are the minimum dimensions the grids must be to not be culled during `cullRooms()`

Key Functions:

`Void generateInitialGrids()`

- Generates a specified number of initial `m_gridsGenerated` that get spawn at a random point within a radius.

`Bool allRoomsAreSeperated()`

- Checks the `m_separitionForces` to see if they are 0,0 to confirm all rooms have been successfully separated and are not longer moving

`Void calculateSeperation()`

- Calculates a separation force for each room based on the rooms colliders, its collider is intersecting with.

`Void seperateRooms()`

- Calls `calculateSeperation()` and applies a separation force to all rooms.

`Void cullRooms()`

- Culls rooms that don't meet the `m_minRoomWidthFinal, m_minRoomHeightFinal` values and creates rooms from the grids which get added to `m_mainRooms`

`Void delauneyTriangle()`

- Performs the Delaney triangle by creating a super triangle and creating a trianglulation between every room.
- A circumcircle is then created for each of these triangles and evaluates whether it meets the criteria of being a valid Delauney Triangle
- Valid triangles get added to `m_finalTriangulations`

`Void AddEdgesToRooms`

- The `Triangle` edges are then given to the `m_mainRooms` std::vector<PointEdge> ***minSpanning()***

- The minimum spanning tree is performed on the `m_mainRooms` edges
- returns a set of edges that are visualised

`void placeEnclosingGrid()`

- Places all the `m_mainRooms` in the final dungeon grid.

`Void generateHallways()`

- Generates hallways based on the angle of the edge indicating what pattern of horizontal and vertical strips need to be created to form a hallway.

`Void applyTextures()`

- Traverse the grid through the `Nodes` Neighbours and assigns the `Cells` textures based on their `TraversalProperty`

Rooms

Purpose: point to cells on a grid where the toom is located and keep information on what rooms it is connected to.

Key Variables:

`Point point`

- Holds all the information on edge connections for the room

`std::vector<std::vector<Cell*>> cellsOccupied`

- Points to the cell the room occupied on a grid

Key Functions:

```
bool emplaceOnGrid(Grid* t_backgroundGrid, sf::Vector2f
t_mosuePosition)
```

- places another the room onto another grid and keep track of where it was placed on that grid with `cellsOccupied.`

CircumCircle

Purpose: to evaluate whether a point lies with the circum circle, to evaluate if a triangle meets the Delauney criteria.

Key Functuons:

`CircumCircle(sf::Vector2f A, sf::Vector2f B, sf::Vector2f C)`

- This function Is AI generated as did not fully grasp the understanding of creating a circumcircle programmatically

`Bool inCircumCircle(sf::Vector2f point)`

- Tells us whether a point lies with the circle or not

References:

Computerphile (2014) *A* (A Star) Search Algorithm - Computerphile*. [YouTube video] 26 November. Available at: https://www.youtube.com/watch?v=ySN5Wnu88nE (Accessed: 20 January 2025).

Johnson, D., *Procedural Dungeon Generation Algorithm*, Game Developer, Available at: https://www.gamedeveloper.com/programming/procedural-dungeon-generation-algorithm [Accessed March 4 2025].https://gwlucastrig.github.io/TinfourDocs/DelaunayIntro/index.html

Triggs, G.W.L., *Introduction to Delaunay Triangulation*, Tinfour Documentation, Available at: https://gwlucastrig.github.io/TinfourDocs/DelaunayIntro/index.html [Accessed 11 March 2025]

Vedantu (n.d.) *About Circumcircle of a Triangle*. Available at: https://www.vedantu.com/maths/about-circumcircle-of-a-triangle (Accessed: March 18 2025).

GeeksforGeeks, *What is Minimum Spanning Tree (MST)?*, Available at: https://www.geeksforgeeks.org/what-is-minimum-spanning-tree-mst/ [Accessed: 24 March]