Ales And Fables: Development Commentary

Project Student: Aimar Goñi

Course: FGCT6011 Final Major Project: Production

Date: 19/05/2025

Repository Link: https://github.com/Aimar-Goni/AG_AIMedievalSim

Itch.io Build Link: Build

Project Outline

AG_AIMedievalSim is an ambitious simulation project developed within Unreal Engine 5, designed to explore complex AI-driven behaviors. The core concept revolves around creating a multi-agent system where AI characters, or "pawns," autonomously manage their needs, execute tasks, and interact with a persistent world. The purpose extends beyond a simple game; it aims to serve as a flexible framework or plugin-set (`AlesAndFables` and `CustomMovementPlugin`), enabling users to rapidly prototype and deploy sophisticated AI simulations. This involves pawns that can gather resources like wood, berries, and water, construct buildings, farm, and engage in leisure activities, all while responding to emergent world states and system defined quests.

[Figure 1. Overview of the AG_AIMedievalSim environment, showcasing AI pawns engaged in various activities, highlighting the dynamic nature of the simulation.]

The initial goals for AG_AIMedievalSim were multi-faceted:

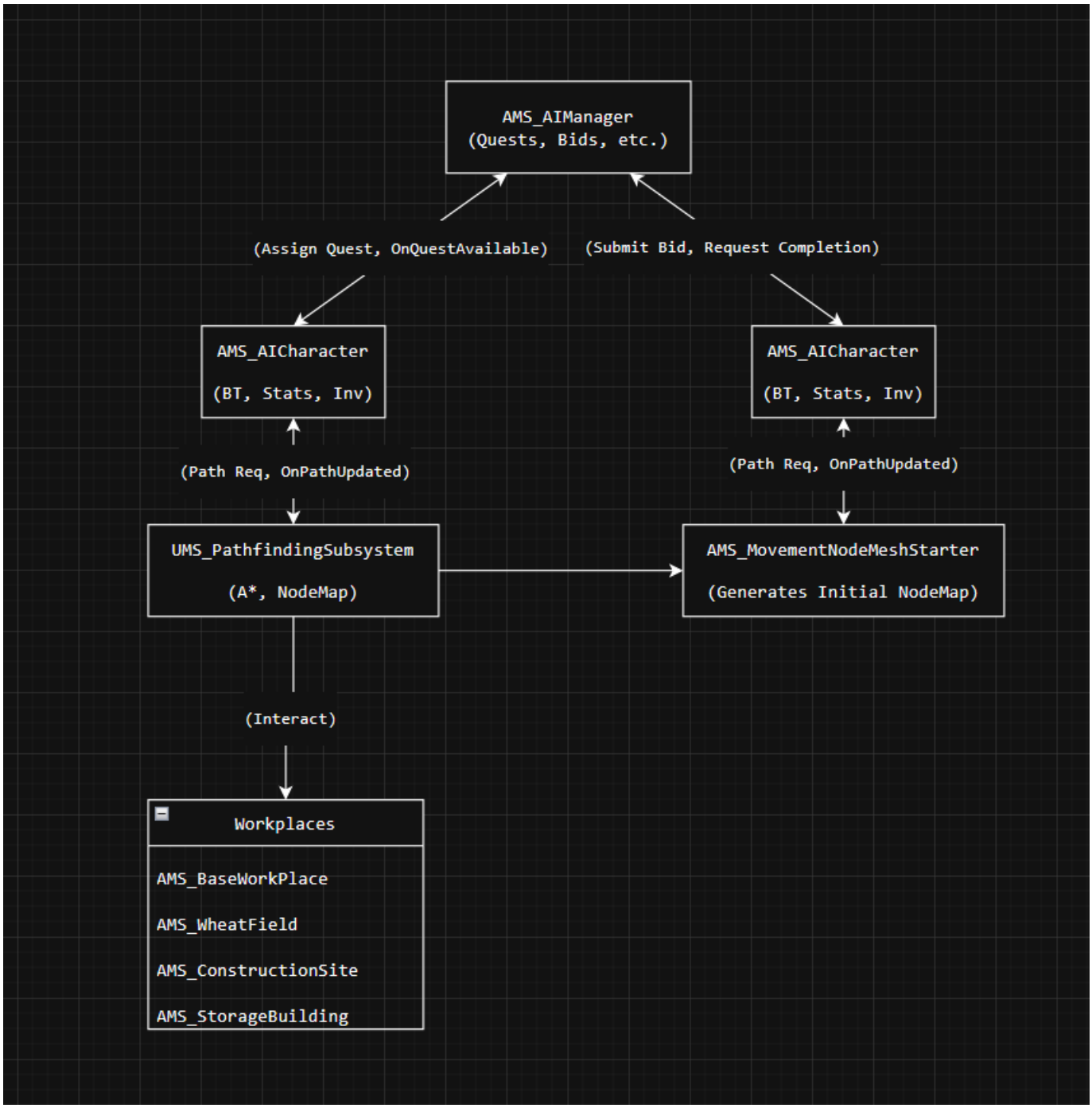
1. **Develop Robust AI Agents:**
   Create AI characters capable of independent decision-making using Unreal Engine's Behavior Trees and Blackboards. This included managing core needs and prioritizing actions accordingly.

2. **Implement a Dynamic Task & Resource Ecosystem:**
   Design a system where AI agents can identify resource scarcities, accept gathering or crafting quests, and interact with various modular workplaces. This involved an inventory system for both agents and storage locations.

3. **Create an Efficient Custom Pathfinding System:** Implement a grid-based A* pathfinding solution, optimized for a potentially large number of agents and dynamic obstacles.

4. **Establish a Quest Management and Bidding System:** Allow the `AMS_AIManager` to generate quests based on resource needs or construction projects, and enable AI agents to evaluate and "bid" on these quests, fostering a simple economic simulation.

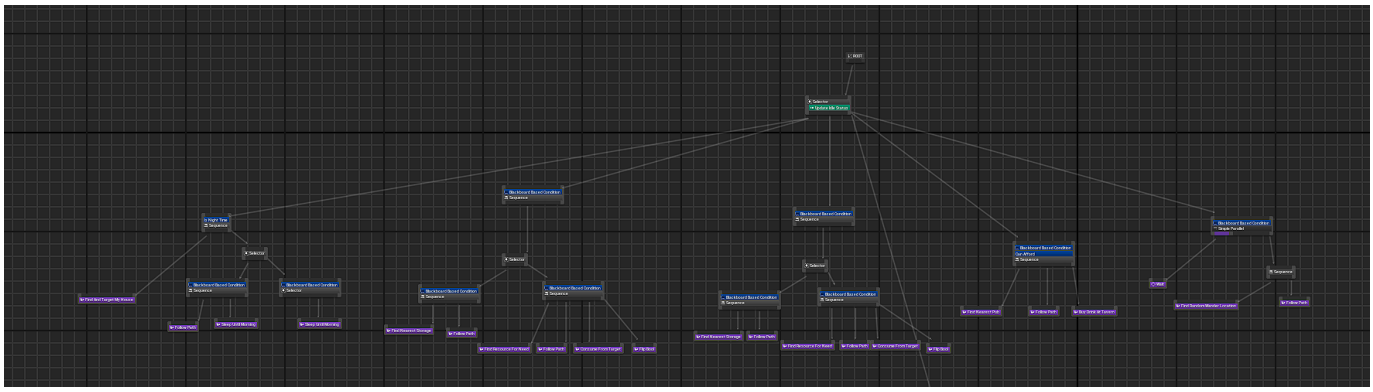
5. **Modular Plugin Architecture:** Structure the core functionalities into two distinct plugins: `AlesAndFables` for the simulation-specific logic (AI characters, manager, quests, medieval assets) and `CustomMovementPlugin` for the generic pathfinding subsystem, making the latter potentially reusable in other projects.



[Figure 2. System architecture diagram of AG_AIMedievalSim, showing the interaction between key components like the AI Manager, individual AI Characters, the Pathfinding Subsystem, and Workplaces.]

Several challenges were anticipated and encountered:

- **Synchronization and Race Conditions:** With multiple AI agents accessing shared resources, ensuring data integrity and preventing conflicts where multiple AIs target the same depleted resource node was a key concern. This was particularly relevant for the `AMS_WorkpPlacePool` and `AMS_StorageBuildingPool` in managing available instances.
- **Pathfinding Complexity and Performance:** Developing a custom A* pathfinding system that could handle dynamic changes and scale to numerous agents without significant performance degradation required careful optimization of node management and path recalculation logic.
- **Behavior Tree Complexity:** Designing and debugging intricate Behavior Trees for varied AI tasks (gathering, delivering, building) became increasingly complex. Ensuring logical flow and preventing AI agents from getting stuck in loops or making irrational decisions was a continuous effort.



[Figure 3. The root of the primary Behavior Tree for `AMS_AICharacter`. This selector node prioritizes satisfying needs (e.g., hunger), then active quests, then idle behaviors like wandering. Key Blackboard flags like `bIsHungry` and `bHasQuest` drive the execution flow.]

[CODE SNIPPET: AlesAndFables/Source/AlesAndFables/Public/AI/Characters/MS_AICharacter.h - AMS_AICharacter::PawnStats_ declaration.]

```
// AlesAndFables/Source/AlesAndFables/Public/AI/Characters/MS_AICharacter.h

// ...
UPROPERTY(EditAnywhere, Category = "Design|Stats")
TObjectPtr<UMS_PawnStatComponent> PawnStats_;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Design|Inventory")
TObjectPtr<UInventoryComponent> Inventory_;

UPROPERTY()
TObjectPtr<UMS_PathfindingSubsystem> PathfindingSubsystem;
// ...
```

[Figure 4. Header declaration for `AMS_AICharacter` showing core component pointers like `PawnStats_` for managing needs, `Inventory_` for resources, and `PathfindingSubsystem` for navigation. These components are central to the AI's autonomy.]

Research

Methodology

The development of Ales And Fables was designed to integrate insights from existing games, academic theory, and technical documentation. The initial phase involved a broad survey of medieval simulation games to identify common mechanics, AI behavioral patterns, and user expectations. This was followed by a deeper dive into academic literature on game AI, specifically focusing on agent architecture, decision-making algorithms, and pathfinding techniques. Concurrently, extensive use was made of Unreal Engine's official documentation, GDC talks, and community forums to understand best practices for implementing these systems within the engine. This iterative approach game analysis informing feature design, academic research guiding algorithmic choices, and technical documentation facilitating implementation allowed for a robust and well-informed development process.

Game Sources

1. **RimWorld (Ludeon Studios, 2018)**

    - **Description:** A colony simulation game driven by AI storytellers, where colonists have individual needs, skills, and work priorities.

    - **Relevance & Key Takeaways:** RimWorld's pawn management system was highly influential (Ludeon Studios, 2018). The concept of pawns having distinct needs (hunger, rest, joy) that they autonomously try to satisfy, alongside a prioritized list of work tasks (Ludeon Studios, 2018), directly inspired the `UMS_PawnStatComponent` in Ales And Fables and the logic for AI characters to balance self-preservation with assigned quests. The way RimWorld handles job assignments provided a model for how tasks could be generated and picked up by available pawns.

    - **Analysis & Influence:** RimWorld uses a sophisticated work prioritization system that players can configure. While Ales And Fables doesn't offer direct player control over priorities, the underlying principle of AI evaluating available tasks against their current needs and capabilities was adopted. For example, an AI in my project with low hunger (`PawnStats_->IsHungry()` is true) would favor a "Get Food" task over a "Gather Wood" quest. This is managed in the Behavior Tree using decorators that check Blackboard flags like `bIsHungry` and `bHasQuest`. The `CalculateBidValue` function in `AMS_AICharacter` also reflects this, where needs heavily penalize the bid for new quests.

[Figure 5. A screenshot from *RimWorld* (Ludeon Studios, 2018) displaying the colonist interface with visible need bars and work priorities. This visual representation of pawn state directly inspired the `UMS_PawnStatComponent` in AG_AIMedievalSim.]

[CODE SNIPPET: AlesAndFables/Source/AlesAndFables/Private/AI/Characters/MS_AICharacter.cpp - AMS_AICharacter::CalculateBidValue - NeedsPenalty calculation.]

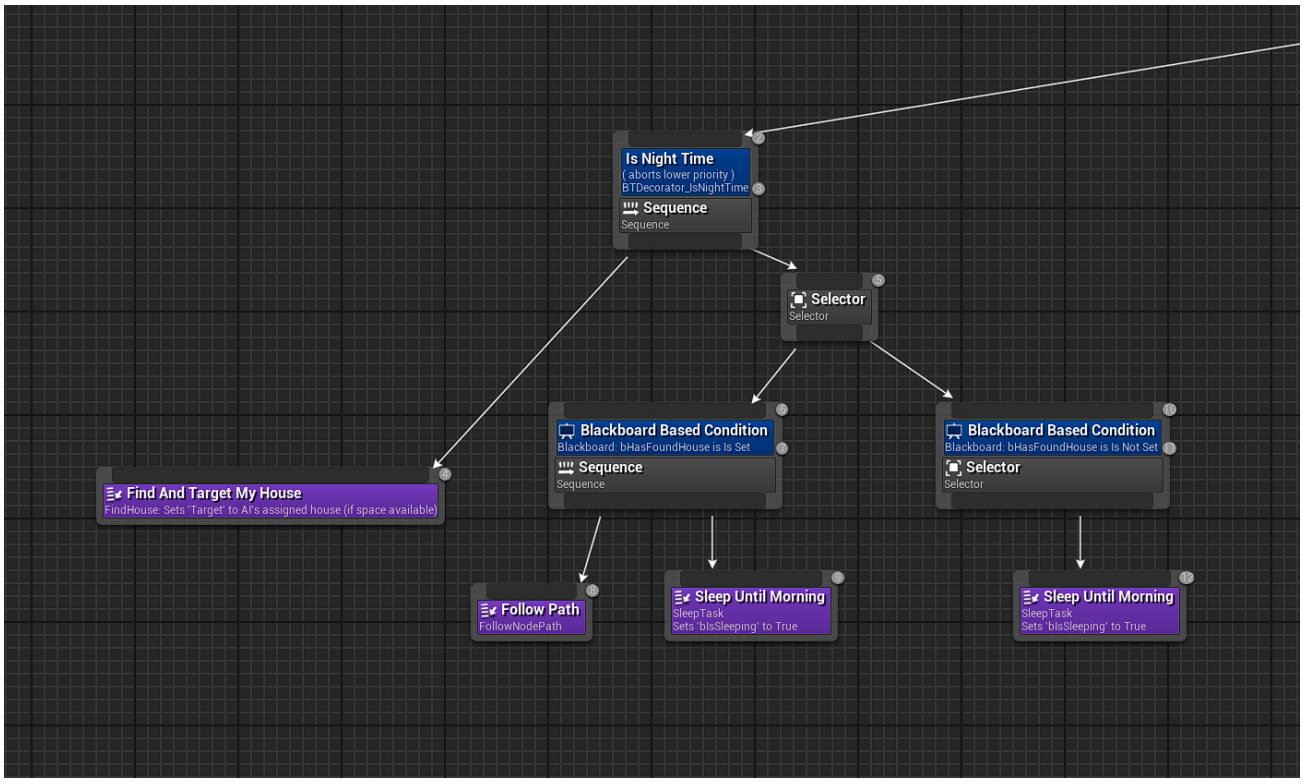
```
// AlesAndFables/Source/AlesAndFables/Private/AI/Characters/MS_AICharacter.

float AMS_AICharacter::CalculateBidValue(const FQuest& Quest)
{
    // ... other factors ...
    float HungerPenalty = FMath::Clamp(1.0f - (PawnStats_->GetHunger() / 10
    float ThirstPenalty = FMath::Clamp(1.0f - (PawnStats_->GetThirst() / 10
```

```
        if (PawnStats_->IsHungry()) HungerPenalty *= 3.0f; // Drastic penalty i
        if (PawnStats_->IsThirsty()) ThirstPenalty *= 3.0f;

        // Combine needs penalties - Higher penalty = lower multiplier, making
        NeedsPenalty = 1.0f / FMath::Max(1.0f, 1.0f + HungerPenalty + ThirstPer
        // ...
        float CalculatedValue = RewardFactor * NeedsPenalty * DistanceFactor;
        return CalculatedValue;
    }
```

[Figure 6. Code snippet from `AMS_AICharacter::CalculateBidValue`. This function determines how "attractive" a quest is to an AI. The `NeedsPenalty` significantly reduces the bid value if the AI is hungry or thirsty, mirroring how a *RimWorld* colonist prioritizes survival.]

2. **The Guild 2 / The Guild 3 (4HEAD Studios / GolemLabs)**
    - **Description:** Life simulation games set in the late Middle Ages, focusing on economic and social progression, where characters run businesses, engage in politics, and manage daily routines.
    - **Relevance & Key Takeaways:** The Guild series (4HEAD Studios / GolemLabs, 2006-2017) excels at portraying AI characters with daily schedules, moving between home, work, and market. This influenced the desire for Ales And Fables's characters to have a sense of routine (4HEAD Studios / GolemLabs, 2006-2017), such as working during the day and potentially seeking shelter or rest at night (using `UMS_BTDecorator_IsNightTime` and `MS_SleepTask`). The concept of production chains in The Guild, while more complex, inspired the simpler resource flow in my project: gather raw resource -> deliver to storage/construction.
    - **Analysis & Influence:** The Guild's AI often appears to follow predefined schedules for their roles. I adapted this by having the `AMS_AIManager` generate resource-specific quests which AI then bid on. This creates a more dynamic "work schedule" than a fixed one, driven by systemic needs. The `MS_TimeSubsystem` and its `OnDayStart` / `OnNightStart` delegates were foundational for allowing AI behavior to change based on the time of day, such as prioritizing going home to sleep using `MS_FindHouse` task and `MS_SleepTask` when `IsNightTime()` is true. The `AMS_House` class itself provides a basic shelter and occupancy system.

[Figure 7. Gameplay screenshot from *The Guild 2* (4HEAD Studios, 2006), illustrating AI characters performing daily routines and moving between various town locations. This informed the desire for AG_AIMedievalSim's AI to exhibit similar scheduled behaviors, particularly with the day/night cycle.]



[Figure 8. Blueprint graph for `UMS_BTDecorator_IsNightTime`. This decorator queries the `UMS_TimeSubsystem` to check if `IsNightTime()` is true. It's used in the Behavior Tree to gate behaviors like sleeping, ensuring AIs only attempt to sleep during appropriate hours, similar to scheduled activities in *The Guild*.]

3. **Banished (Shining Rock Software, 2014)**

- **Description:** A city-building strategy game where players guide a group of exiled travellers to grow and maintain a settlement. Resource management and citizen well-being are central.
- **Relevance & Key Takeaways:** Banished (Shining Rock Software, 2014) emphasises the importance of resource logistics and the roles of citizens. The way citizens fetch materials for construction sites (Shining Rock Software, 2014) was a direct inspiration for the construction quest loop in Ales And Fables: `AMS_AIManager` identifies a need, spawns an `AMS_ConstructionSite`, and generates delivery quests for resources like wood. AI characters then fetch these from `AMS_StorageBuilding` and deliver them.
- **Analysis & Influence:** In Banished, if resources aren't available or if labourers are too busy, construction halts. This concept of resource dependency is mirrored in my `CheckAndInitiateConstruction` logic in `AMS_AIManager`. The task flow for an AI completing a construction delivery quest involves:

    1. Accepting a quest (e.g., Deliver 15 Wood to Site X).
    2. Pathing to `CentralStorageBuilding` (`MS_FindNearestStorage`).
    3. Fetching materials (`MS_FetchFromStorage`).
    4. Pathing to `AMS_ConstructionSite`.
    5. Delivering materials (handled in `AMS_AICharacter::OnOverlapBegin` with the site).
    6. `AMS_ConstructionSite` updates its `CurrentAmount` and completes construction if `AmountRequired` is met.

[Figure 9. A scene from *Banished* (Shining Rock Software, 2014) where citizens are actively transporting resources to a construction site. This logistical loop directly inspired the multi-step construction quest system in AG_AIMedievalSim.]

[CODE SNIPPET: AlesAndFables/Source/AlesAndFables/Private/AI/Manager/MS_AIManager.cpp - AMS_AIManager::StartBuildingProject - Delivery quest generation loop.]

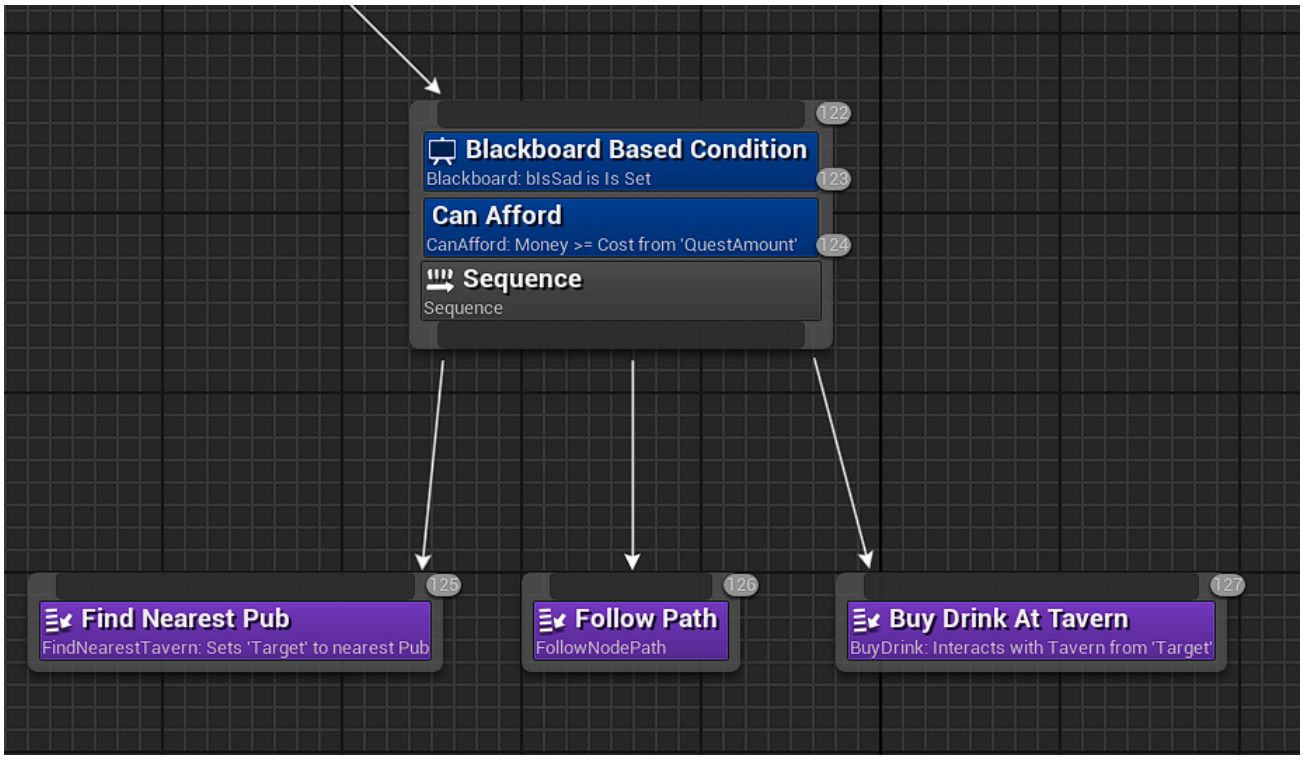
```
// AlesAndFables/Source/AlesAndFables/Private/AI/Manager/MS_AIManager.cpp

void AMS_AIManager::StartBuildingProject(TSubclassOf<AActor> BuildingClassT
{
```

```
        // ... (spawn AMS_ConstructionSite NewSite) ...
        if (NewSite)
        {
            // ... (configure NewSite) ...
            int32 RemainingCost = ResourceCost;
            while (RemainingCost > 0)
            {
                int32 deliveryAmount = FMath::Min(RemainingCost, DeliveryCarryC
                FQuest deliveryQuest(RequiredResource, deliveryAmount, Calculat
                AvailableQuests_.Add(deliveryQuest);
                StartBidTimer(deliveryQuest);
                OnQuestAvailable.Broadcast(deliveryQuest);
                RemainingCost -= deliveryAmount;
            }
            ActiveConstructionDeliveryQuests.Add(NewSite, GeneratedQuestIDs);
        }
        // ...
    }
```

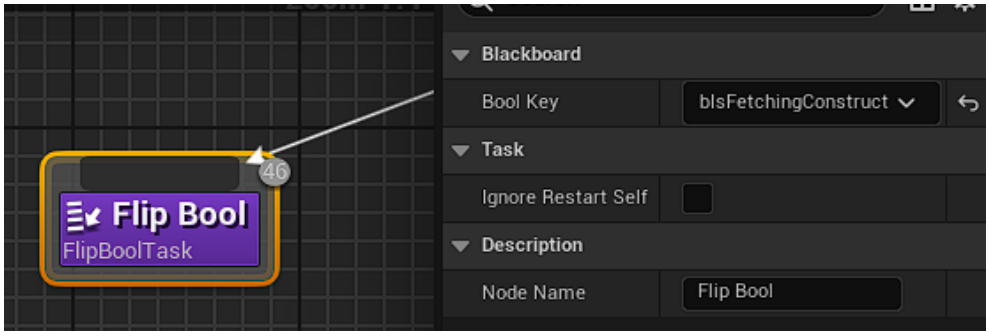
[Figure 10. Snippet from `AMS_AIManager::StartBuildingProject`. When a construction project begins, the AIManager breaks down the total `ResourceCost` into smaller `deliveryQuest` chunks based on `DeliveryCarryCapacity`. Each chunk becomes a separate quest for AIs to fetch and deliver, mimicking the incremental resource delivery seen in *Banished*.]

Academic Sources

1. **Millington, I., & Funge, J. *Artificial Intelligence for Games*.**
    - **Summary:** This book provides a comprehensive overview of AI techniques used in games, covering pathfinding, decision-making, movement, and agent architecture.
    - **Relevance & Application:** Chapter 4 (Decision Making) and Chapter 5 (Behavior Trees) were particularly influential (Millington & Funge, 2009). The book's explanation of Behavior Trees nodes like Sequences, Selectors, Decorators, and Tasks (Millington & Funge, 2009) provided the foundational knowledge for designing the AI logic in `AlesAndFables`.
    - **Analysis & Influence:** Millington and Funge (2009) emphasise the modularity and reusability of Behavior Tree tasks and services. This principle (Millington & Funge, 2009) guided the creation of many specific task nodes in my project, such as `MS_FindNearestWorkSite, and` FlipBoolTask`. Each task is designed to be a self-contained unit of behaviour. For example,` UMS_BTDecorator_NeedToGatherItems `checks if the AI's inventory (`AIChar->Inventory_->GetResourceAmount(QuestType)`) is less than the` QuestAmount` on the Blackboard. This decorator, combined with a Sequence node, ensures the AI only attempts to gather if needed.



[Figure 11. A conceptual diagram illustrating basic Behavior Tree components (Selector, Sequence, Task, Decorator) as described by Millington & Funge (2009). This structure forms the basis of AI decision-making in AG_AIMedievalSim.]



[Figure 12. The `UFlipBoolTask` ExecuteTask implementation. This simple C++ task allows the Behavior Tree to toggle a boolean value on the Blackboard, a utility inspired by the modular task design advocated in "AI for Games."]

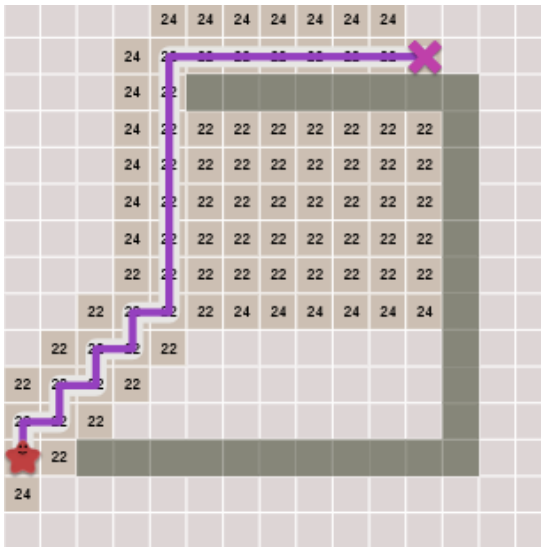
```
// AlesAndFables/Source/AlesAndFables/Private/AI/TaskNodes/FlipBoolTask.cpp
EBTNodeResult::Type UFlipBoolTask::ExecuteTask(UBehaviorTreeComponent& Owne
{
    UBlackboardComponent* BlackboardComp = OwnerComp.GetBlackboardComponent
    if (!BlackboardComp || !BoolKey.SelectedKeyName.IsValid())
    {
        return EBTNodeResult::Failed;
    }
    const bool bCurrentValue = BlackboardComp->GetValueAsBool(BoolKey.Selec
    BlackboardComp->SetValueAsBool(BoolKey.SelectedKeyName, !bCurrentValue)
    return EBTNodeResult::Succeeded;
}
```

2. *Patel, A. A Pathfinding for Beginners.**

- **Summary:** Amit Patel's website offers exceptionally clear explanations and interactive diagrams for various pathfinding algorithms, especially A*. It breaks down concepts like heuristics, cost functions, and the open/closed set management.

- **Relevance & Application:** This was the primary guide for implementing the A* algorithm (Patel, n.d.) within `UMS_PathfindingSubsystem::FindPathNodes`. The use of `TSet<TSharedPtr<FMoveNode>> OpenSet`, `TSet<TSharedPtr<FMoveNode>> ClosedSet`, `TMap<TSharedPtr<FMoveNode>, float> GScore`, and `TMap<TSharedPtr<FMoveNode>, float> FScore` directly follows Patel's (n.d.) described A* implementation. The Manhattan distance heuristic was chosen for its efficiency on a grid.

- **Analysis & Influence:** Patel's guide (n.d.) stresses the importance of an admissible and consistent heuristic for A* to find the optimal path. My `Heuristic` function calculates Manhattan distance, a common heuristic for grid-based pathfinding defined as the sum of the absolute differences of their Cartesian coordinates (DataCamp, n.d.; Patel, n.d.): `FMath::Abs(NodeA->GridPosition.X - NodeB->GridPosition.X) + FMath::Abs(NodeA->GridPosition.Y - NodeB->GridPosition.Y);`. This choice was made because my `FMoveNode` structures are on a grid, and Manhattan distance is computationally cheaper than Euclidean while still being admissible for grid movement (Patel, n.d.), where diagonal moves are not significantly cheaper or disallowed. The ability to handle blocked nodes (`BlockedNodes` TSet in `UMS_PathfindingSubsystem`) and update paths dynamically (`OnPathUpdated` delegate) was an extension inspired by the need for a responsive world, going beyond the basic A* tutorial.



[Figure 13. An illustrative screenshot from Amit Patel's A* Pathfinding guide on RedBlobGames.com, showing the visualization of the open set, closed set, and path. This clear explanation was crucial for implementing the A* algorithm in `UMS_PathfindingSubsystem`.]

[CODE SNIPPET: CustomMovementPlugin/Source/CustomMovementPlugin/Private/Movement/MS_PathfindingSubsystem.cpp - UMS_PathfindingSubsystem::FindPathNodes - A* main loop structure.]

```
// CustomMovementPlugin/Source/CustomMovementPlugin/Private/Movement/MS_Pat
TArray<TSharedPtr<FMoveNode>> UMS_PathfindingSubsystem::FindPathNodes(TShar
{
    // ... (initialization of OpenSet, ClosedSet, GScore, FScore, CameFrom,
```

```
        while (PriorityQueue.Num() > 0)
        {
            TSharedPtr<FMoveNode> CurrentNode = PriorityQueue[0]; // Get node w
            PriorityQueue.RemoveAt(0);

            if (CurrentNode == GoalNode)
            {
                // Reconstruct path
                // ...
                return Path;
            }
            ClosedSet.Add(CurrentNode);
            for (const TPair<TSharedPtr<FMoveNode>, bool>& NeighborPair : Curre
            {
                TSharedPtr<FMoveNode> Neighbor = NeighborPair.Key;
                bool bIsAccessible = NeighborPair.Value;
                if (!bIsAccessible || ClosedSet.Contains(Neighbor)) continue;

                float TentativeGScore = GScore[CurrentNode] + FVector::Dist(Cur
                if (!GScore.Contains(Neighbor) || TentativeGScore < GScore[Neig
                {
                    CameFrom.Add(Neighbor, CurrentNode);
                    GScore.Add(Neighbor, TentativeGScore);
                    FScore.Add(Neighbor, TentativeGScore + Heuristic(Neighbor,
                    if (!PriorityQueue.Contains(Neighbor))
                    {
                        PriorityQueue.Add(Neighbor);
                    }
                }
            }
            PriorityQueue.Sort([&](const TSharedPtr<FMoveNode>& A, const TShare
        }
        return TArray<TSharedPtr<FMoveNode>>(); // No path found
    }
```

[Figure 14. Core A* search loop from `UMS_PathfindingSubsystem::FindPathNodes`. This implements the open/closed set management, g-score/f-score calculation, and neighbor exploration as detailed in Patel's A* guide. The `Heuristic` function (Manhattan distance) is used for `FScore` calculation.]

Documentation Sources

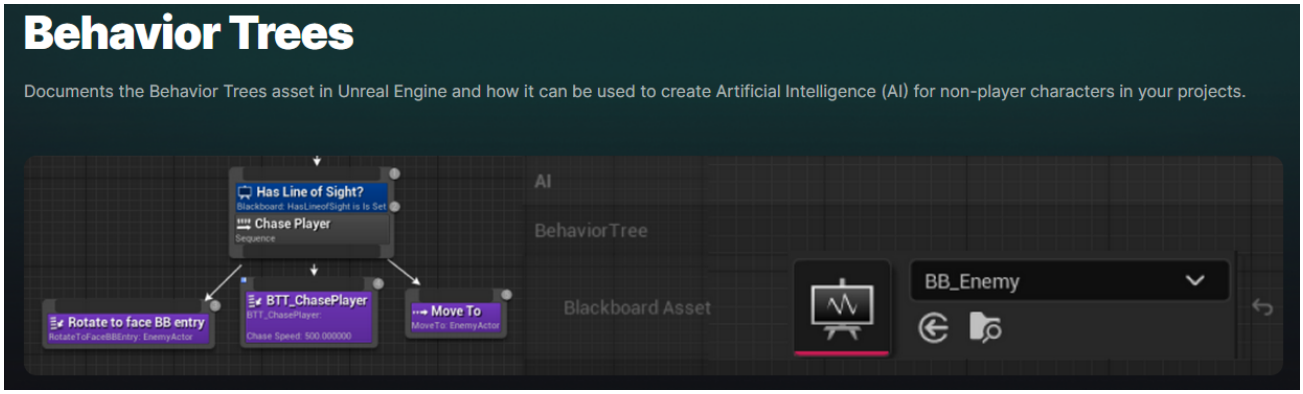
1. . ***Behavior Trees*** **(Epic Games, n.d.a).**
    - **Summary:** The official UE documentation provides a comprehensive guide to using Behavior Trees, including explanations of node types, Blackboards, AI Controllers, and how they integrate with Pawns.
    - **Relevance & Application:** This was the go-to resource for the technical implementation (Epic Games, n.d.a) of all AI behaviors. It guided the creation of custom C++ BT nodes like `UFlipBoolTask`,

decorators like `UMS_BTDecorator_IsNightTime`, and services like `UMS_BTService_UpdateIdleStatus`. The correct way to override functions (Epic Games, n.d.a) like `ExecuteTask` was learned directly from these pages.

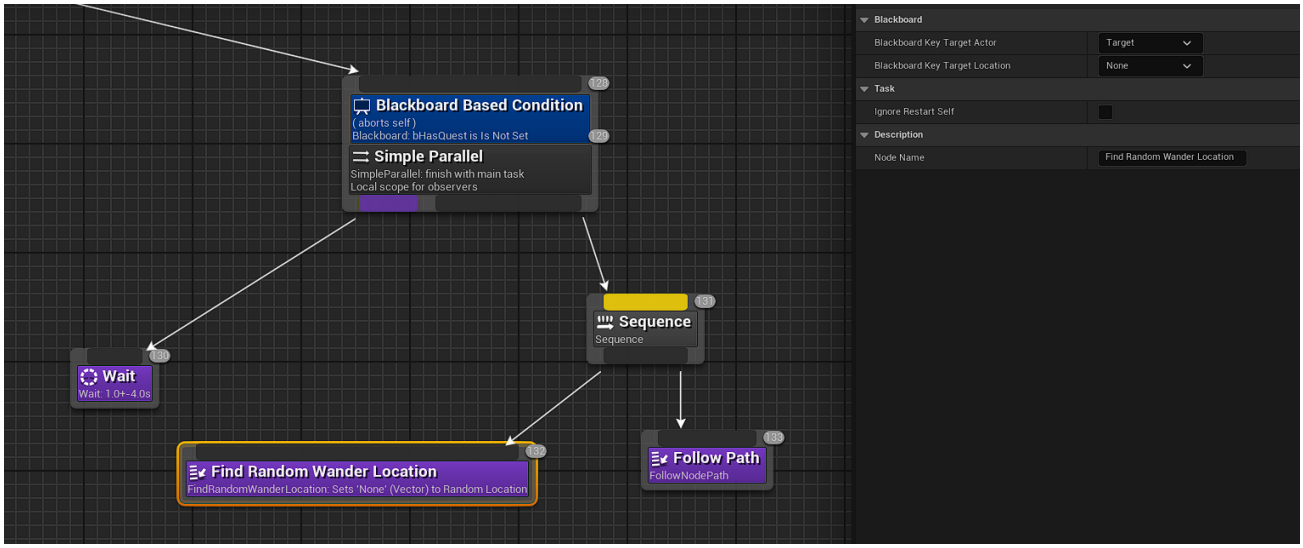
- **Analysis & Influence:** The documentation's (Epic Games, n.d.a) emphasis on creating custom nodes in C++ for performance and complex logic was key. For example, `UMS_FindRandomWanderLocation` uses the `UMS_PathfindingSubsystem` to find a valid, unblocked node and sets it on the Blackboard. This involves engine/game-specific logic that is best handled in C++.

```
// UMS_FindRandomWanderLocation::ExecuteTask - getting a random node
if (PathSubsystem->GetRandomFreeNode(RandomLocation, RandomGridLocation))
{
    Blackboard->SetValueAsVector(BlackboardKey_TargetLocation.SelectedKeyNa
    // ... path generation logic ...
    return EBTNodeResult::Succeeded;
}
```

The documentation on Blackboard Key Selectors (`FBlackboardKeySelector`) and how to filter them (e.g., `AddObjectFilter`, `AddBoolFilter`) was essential for making custom nodes configurable from the Behavior Tree editor. This allowed tasks like `UMS_CopyBlackboardValue` to be generic.



[Figure 15. A screenshot of the official Unreal Engine Behavior Tree documentation. This resource was fundamental for understanding how to create and integrate custom C++ nodes.]



[Figure 16. `UMS_FindRandomWanderLocation` task node as seen in the Behavior Tree editor. It uses

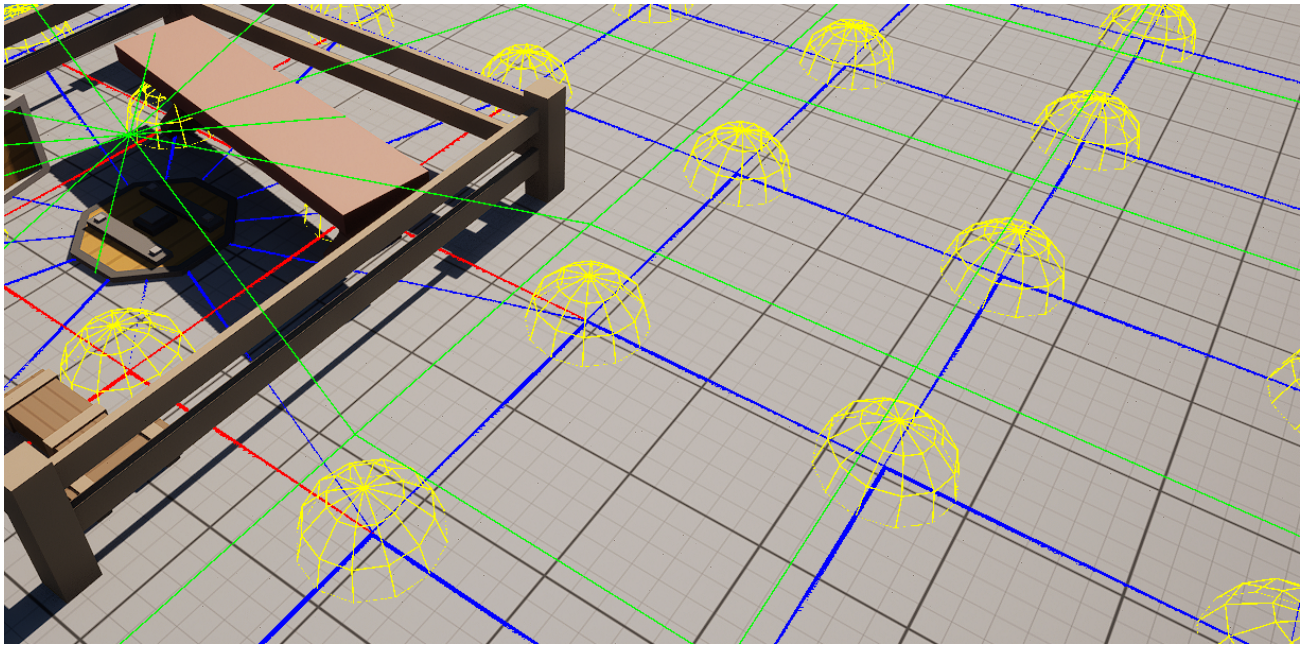
`FBlackboardKeySelector` for `BlackboardKey_TargetLocation`, allowing designers to specify the output key directly in the editor, a feature detailed in UE documentation.]

2. . ***Custom C++ Pathfinding with A* .(Epic Games Community, n.d.b)***. Epic Games Forums / Community Wiki (Often more practical examples here).

    - **Summary:** While official docs focus on UE's NavMesh, community tutorials and forum discussions (Epic Games Community, n.d.b) often detail how to implement fully custom pathfinding systems, including grid generation, A* logic, and integrating it with AI movement components.

    - **Relevance & Application:** Implementing `UMS_PathfindingSubsystem` and `AMS_MovementNodeMeshStarter` required going beyond standard NavMesh. Community examples of A* (Epic Games Community, n.d.b) in C++ for Unreal, often found on forums or developer blogs, likely provided patterns for managing the node graph (`TMap<FIntPoint, TSharedPtr<FMoveNode>> NodeMap`), neighbour connections, and the A* algorithm itself.

    - **Analysis & Influence:** The challenge with custom pathfinding is integrating it smoothly. Community resources often show how to make a subsystem accessible globally, which is how `PathfindingSubsystem` is used by AI characters and the `AMS_AIManager`. The `AMS_MovementNodeMeshStarter`'s approach of raycasting to generate a grid of `FMoveNode`s and then connecting neighbours is a common technique discussed in such custom pathfinding tutorials. The dynamic update, allowing the pathfinding grid to react to in-game changes is an advanced feature that makes the custom system more robust than a static grid.



[Figure 17. Debug visualization of the pathfinding node grid generated by `AMS_MovementNodeMeshStarter`. Nodes are placed based on raycasts, and connections (not shown here) are established if paths between them are clear. This custom grid approach was informed by community examples.]

[CODE SNIPPET: CustomMovementPlugin/Source/CustomMovementPlugin/Private/Movement/MS_PathfindingSubsystem.cpp - UMS_PathfindingSubsystem::SetNodeBlockedStatus - Broadcasting path updates.]

```
// CustomMovementPlugin/Source/CustomMovementPlugin/Private/Movement/MS_Pat
void UMS_PathfindingSubsystem::SetNodeBlockedStatus(const FIntPoint& GridPo
{
    // ... (logic to update Node->Neighbors and BlockedNodes set) ...
    bool bStatusChanged = false; // Assume this is set based on actual char
    // ...
    if (bBlocked)
    {
        if (!BlockedNodes.Contains(GridPosition))
        {
            BlockedNodes.Add(GridPosition);
            bStatusChanged = true;
            // Update neighbors of 'Node' to mark paths to/from it as inacc
        }
    }
    else // Unblocking
    {
        if (BlockedNodes.Remove(GridPosition) > 0) // If an element was act
        {
            bStatusChanged = true;
            // Update neighbors of 'Node' to re-evaluate paths to/from it
        }
    }

    if (bStatusChanged)
    {
        OnPathUpdated.Broadcast(GridPosition); // Notify listeners (like AI
    }
}
```

[Figure 18. Code snippet from `UMS_PathfindingSubsystem::SetNodeBlockedStatus`. When a node's traversability changes (e.g., a building is constructed), this function updates the `BlockedNodes` set and broadcasts the `OnPathUpdated` delegate (Gamma et al., 1994). AI characters subscribed to this delegate can then trigger a path recalculation if their current path is affected.]

Implementation

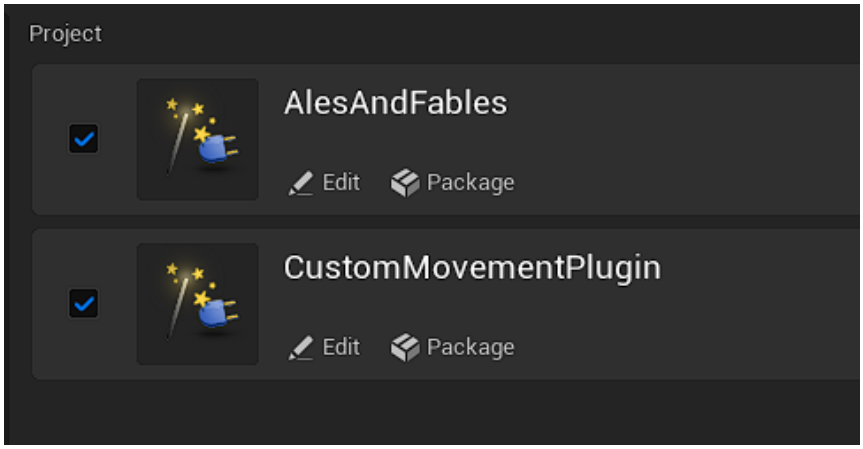
Process

The development of Ales And Fables was an iterative process, structured into several key phases, primarily using C++ within Unreal Engine 5, with Blueprints for initial prototyping and some UI elements. Version control was managed using Git.

1. **Phase 1: Core Systems & AI Character Foundation**
    - **Decision:** Establish the foundational classes.
    - **Implementation:**

- Created `AlesAndFables` (main game logic) and `CustomMovementPlugin` (pathfinding) modules.
- Developed `AMS_AICharacter` with basic skeletal structure, linking it to `AMS_AICharacterController`.
- Implemented `UInventoryComponent` for managing resources (`TMap<ResourceType, int32> Resources_`).
- Created `UMS_PawnStatComponent` to handle needs like hunger and thirst, with a timer for decay (`DecreaseStats`) and delegates for state changes (`OnStateChanged`).
- Basic Behavior Tree and Blackboard setup, with `SelfActor` key (Epic Games, n.d.a).

- **Tools:** Visual Studio for C++, Unreal Engine editor for Blueprint/BT setup.



[Figure 18. Unreal Engine project browser highlighting the `AlesAndFables` and `CustomMovementPlugin` modules, demonstrating the modular architecture established early in development.]

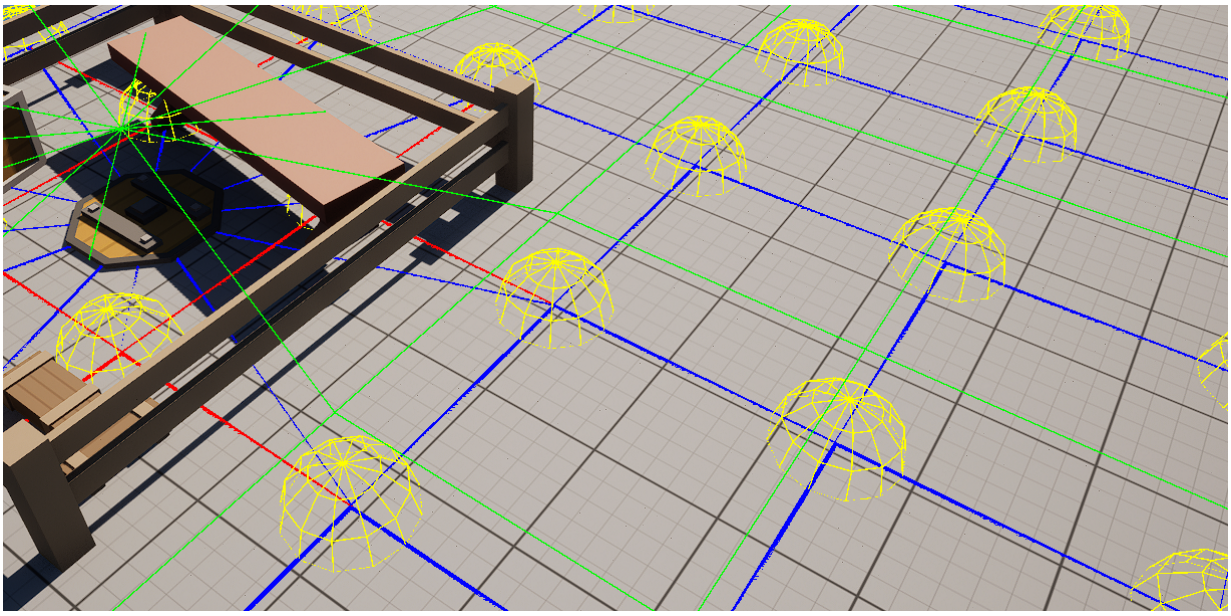
[CODE SNIPPET: AlesAndFables/Source/AlesAndFables/Public/Systems/MS_PawnStatComponent.h - UMS_PawnStatComponent needs variables and delegates.]

```
// AlesAndFables/Source/AlesAndFables/Public/Systems/MS_PawnStatComponent.h
// ...
UPROPERTY(EditAnywhere, Category = "Stats")
float Hunger = 100.0f;
UPROPERTY(EditAnywhere, Category = "Stats")
float Thirst = 100.0f;
// ... other needs ...

UPROPERTY(EditAnywhere, Category = "Stats")
float HungryThreshold = 30.0f;
// ... other thresholds ...

DECLARE_DYNAMIC_MULTICAST_DELEGATE(FOnStateChanged);
UPROPERTY(BlueprintAssignable, Category = "Events")
FOnStateChanged OnStateChanged;
// ...
```

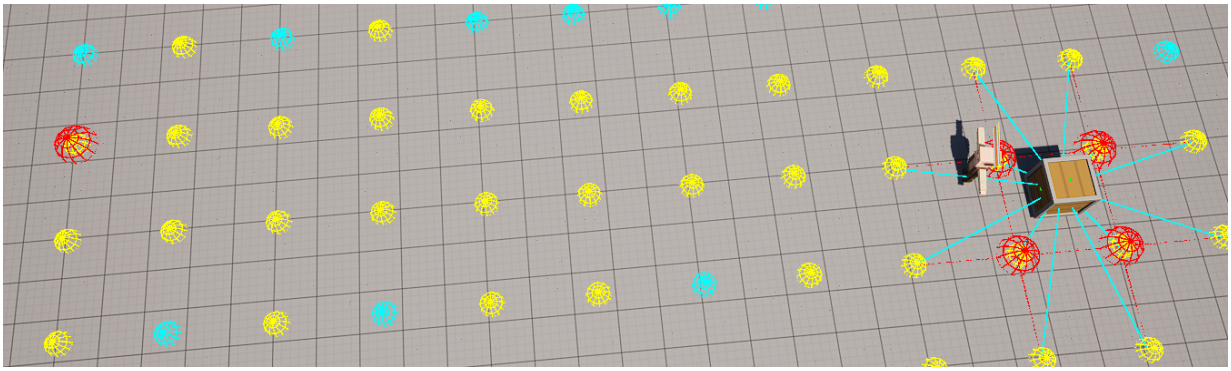
[Figure 19. Key stat variables (`Hunger`, `Thirst`) and the `OnStateChanged` delegate in `UMS_PawnStatComponent.h`. This component manages AI needs, broadcasting updates when critical thresholds are crossed, driving need-fulfillment behaviors.]

2. **Phase 2: Pathfinding Implementation**

- **Decision:** Implement a custom A* pathfinding solution for greater control over dynamic environments.
- **Implementation:**
  - Created `FMoveNode` struct within `MS_MovementNode.h`.
  - Developed `AMS_MovementNodeMeshStarter` to procedurally generate a grid of `FMoveNode`s at game start by raycasting down (`PerformRaycastAtPosition`) and connecting neighbors if paths are clear (`PerformRaycastToPosition`).
  - Implemented `UMS_PathfindingSubsystem` (as a `UGameInstanceSubsystem`) to store the `NodeMap` and contain the A* algorithm (`FindPathNodes`, `FindPathPoints`) (Patel, n.d.c). This used `TSet` for open/closed sets and `TMap` for scores as per A* theory.
  - Added functionality to `BlockNode`, `UnblockNode`, and the `OnPathUpdated` delegate (Gamma et al., 1994) to handle dynamic changes.



[Figure 20. Image demonstrating the `AMS_MovementNodeMeshStarter`'s raycasting process during initialization. Green lines indicate successful floor detection for node placement, while red might indicate no suitable surface. (This is illustrative; actual debug may vary).]



[Figure 21. In-game screenshot with pathfinding debug visualization. The yellow sphere represents the A* path calculated by `UMS_PathfindingSubsystem` for an AI agent moving towards its target (red sphere).]

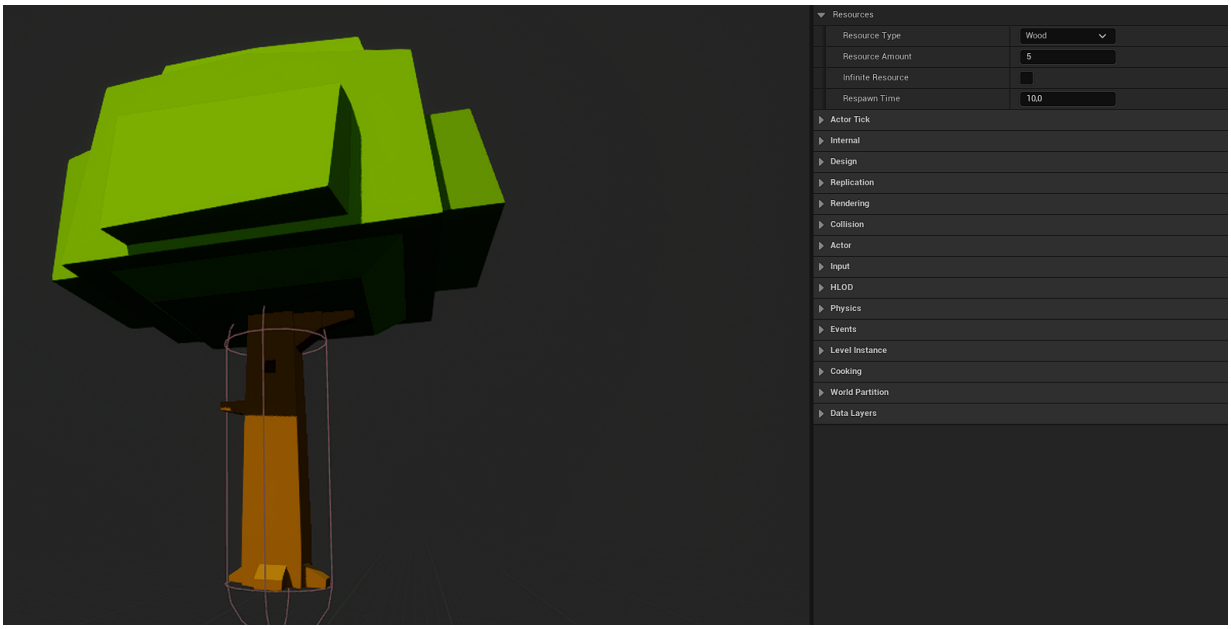
- **Techniques:** A* algorithm (Patel, n.d.c; Millington & Funge, 2009), grid generation, raycasting for obstacle detection.

3. **Phase 3: Workplaces & Basic AI Interaction**

   - **Decision:** Create modular workplaces and enable AI to interact with them.
   - **Implementation:**

     - Defined `AMS_BaseWorkPlace` as a base class for all resource spots (e.g., trees, berry bushes) with `ResourceType_`, `ResourceAmount_`, and `TakeResources()` virtual method.
     - Created derivatives like `AMS_TreeWorkPlace`, `MS_BushWorkPlace`.
     - Implemented `AMS_WorkpPlacePool` to manage instances of workplaces, including spawning and deactivating them.
     - Developed Behavior Tree tasks: `MS_FindNearestWorkSite` (queries `WorkPlacesPool_` on `AMS_AICharacter`), `MS_GeneratePathToTarget`, `MS_FollowNodePath`, `MS_PerformWorkAction` (calls `TakeResources()` and adds to AI's `Inventory_`)(Millington & Funge, 2009).
     - AI characters could now find a workplace of a specific type, path to it, "work" for a duration, and collect resources.

       [BLUEPRINT IMAGE: AlesAndFables/Placeables/Interactables/BP_TreeWorkPlace.uasset - Blueprint for a TreeWorkPlace, showing its StaticMesh and configuration of `ResourceType_` and `ResourceAmount_`.]

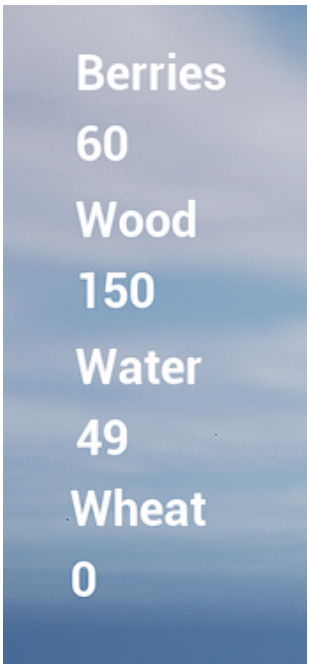
       

       [Figure 22. Blueprint editor view of `BP_TreeWorkPlace` (derived from `AMS_BaseWorkPlace`). It shows the assigned tree mesh and default properties like `ResourceType_` set to `WOOD` and `ResourceAmount_` set to a default value, making it a configurable resource node.]

4. **Phase 4: Quest Management & Economy**

   - **Decision:** Introduce a system for dynamic task generation and assignment.
   - **Implementation:**

- Defined `FQuest` struct (`MS_ResourceSystem.h`) with ID, type, amount, reward, and optional target.
- Created `AMS_AIManager` to:
  - Monitor resource levels in a central storage (`CentralStorageBuilding`).



[Figure 23. A UI overlay displaying the `AMS_AIManager` 's status: current central storage inventory levels.]

  - Generate gathering quests (`GenerateQuestsForResourceType`) if resources are low.
  - Broadcast `OnQuestAvailable` delegate (Gamma et al., 1994).
  - Manage a bidding system: AI characters `EvaluateQuestAndBid` (based on distance, needs, reward), AIManager `ReceiveBid` and `SelectQuestWinner_Internal` after a `BidDuration` .

[CODE SNIPPET: AlesAndFables/Source/AlesAndFables/Private/AI/Manager/MS_AIManager.cpp - AMS_AIManager::SelectQuestWinner_Internal - Logic for choosing the best bid.]

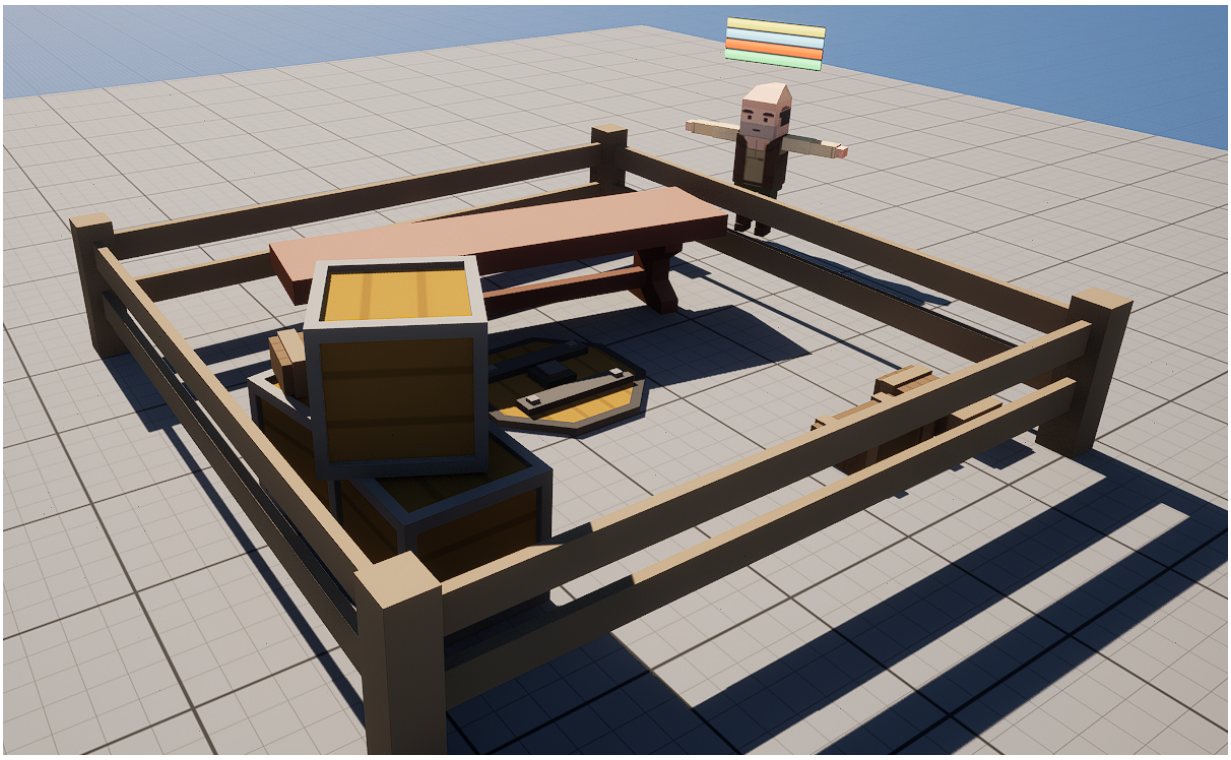
```
// AlesAndFables/Source/AlesAndFables/Private/AI/Manager/MS_AIManag
void AMS_AIManager::SelectQuestWinner_Internal(FGuid QuestID)
{
    // ... (find originalQuest) ...
    if (CurrentBids.Contains(QuestID))
    {
        TArray<FBidInfo>& bids = CurrentBids[QuestID];
        AMS_AICharacter* winner = nullptr;
        float highestBid = -1.0f;
        float winningBidTimestamp = FLT_MAX; // For tie-breaking

        for (const FBidInfo& bid : bids)
        {
```

```
                    if (bid.Bidder.IsValid()) // Ensure bidder is still va
                    {
                        if (bid.BidValue > highestBid)
                        {
                            highestBid = bid.BidValue;
                            winner = bid.Bidder.Get();
                            winningBidTimestamp = bid.BidTimestamp;
                        }
                        // Tie-breaking: first come, first served (earlier
                        else if (FMath::IsNearlyEqual(bid.BidValue, highest
                        {
                            winner = bid.Bidder.Get();
                            winningBidTimestamp = bid.BidTimestamp;
                        }
                    }
                }
                if (winner && !winner->AssignedQuest.QuestID.IsValid()) //
                {
                    // Assign quest to winner
                    winner->AssignQuest(originalQuest);
                    // ... (move quest from AvailableQuests_ to AssignedQue
                }
                // ... (cleanup bids and timer) ...
            }
            // ... (handle no bids) ...
        }
```

[Figure 24. `AMS_AIManager::SelectQuestWinner_Internal` function. This logic iterates through bids received for a specific `QuestID`, selecting the `winner` based on the `highestBid`. It includes a tie-breaking mechanism using `BidTimestamp`. The winning AI is then assigned the `originalQuest`.]

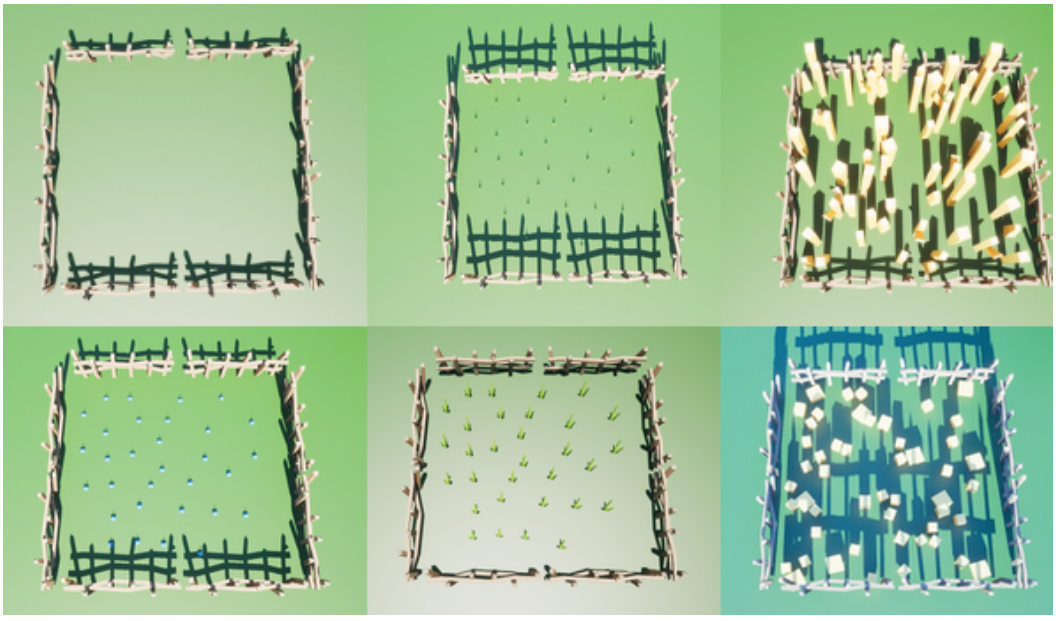
    - Handle quest completion (`RequestQuestCompletion`).
  - AI characters' `AssignQuest` method updates their Blackboard (Epic Games, n.d.a) (e.g., `bHasQuest`, `QuestType`).
- **Techniques:** Delegate-based event system (Gamma et al., 1994), timer-based bidding.

5. **Phase 5: Advanced Behaviors & World Systems**
   - **Decision:** Expand AI capabilities to include construction, farming, housing, and leisure.
   - **Implementation:**
     - **Construction:** `AMS_ConstructionSite` actor. `AMS_AIManager` initiates construction (`StartBuildingProject`), generating delivery quests. AI fetches from storage (`MS_FetchFromStorage`), delivers to site. Site completes (`CompleteConstruction`) and spawns final building.



[Figure 25. An `AMS_ConstructionSite` in-game, visually represented by scaffolding. An AI character is shown interacting or moving towards it, fulfilling a delivery quest generated by the `AMS_AIManager`.]

- **Farming:** `AMS_WheatField` with states (`EFieldState`). AIManager generates quests for planting, watering, harvesting based on field state and `OnFieldNeedsPlanting` etc. delegates. `MS_PerformWorkAction` handles field-specific actions.



[Figure 26. A visual progression of the `AMS_WheatField` through its different states (`EFieldState`). Each state (Constructed, Planted, Growing, ReadyToHarvest) is represented by a different static mesh, updated via `AMS_WheatField::ChangeState`.]

- **Housing:** `AMS_House` class. AIManager assigns houses (`UpdateHousingState`). AI uses `MS_FindHouse` and `MS_SleepTask` (triggered by `MS_BTDecorator_IsNightTime`) to sleep and regain energy.

- **Tavern/Leisure:** `AMS_Tavern` . AIManager checks `ShouldBuildTavern` . AI uses `MS_FindNearestTavern` and `MS_BuyDrink` to increase happiness.
- **Time & Sky:** `MS_TimeSubsystem` for day/night cycle and `MS_SkyController` to update directional light.

[Figure 27. Comparison of the game world's lighting at midday versus nighttime. The `AMS_SkyController` adjusts the directional light's rotation, intensity, and color based on the `UMS_TimeSubsystem` 's current hour.]

New Approaches

1. **Dynamic Pathfinding Grid with On-Demand Node Creation:**
   - Instead of pre-defining all possible movement nodes, the `AMS_MovementNodeMeshStarter` dynamically generates an initial grid based on raycasts against "Floor" tagged geometry. More significantly, systems like `AMS_WorkpPlacePool::SpawnWorkplaceAtRandomNode` or building construction (`AMS_ConstructionSite`) can request the `UMS_PathfindingSubsystem` to `AddNodeAtPosition` . This creates a new `FMoveNode` at the exact location of the new interactive object (e.g., a spawned berry bush or a newly built house's door) and connects it to nearby existing nodes.
   - **Why Chosen:** This approach offers flexibility for dynamically changing environments (van den Berg et al., 2006). Pre-baking a huge grid for a large map is inefficient, and static grids don't adapt well to player/AI-driven construction or procedural content. On-demand node creation ensures pathable access to newly spawned entities without needing a full grid regeneration.
   - **Evaluation:** This was highly successful for interactive objects. It reduced initial setup time and allowed for more organic placement of dynamic elements. The main challenge was ensuring robust neighbor connection logic for newly added nodes, especially near the edges of the existing grid or in dense areas. The `PathfindingSubsystem->DeactivateClosestNodes` was also an important part of this, ensuring that when a large building was placed, the underlying general-purpose nodes were correctly blocked.

```
// UMS_PathfindingSubsystem::AddNodeAtPosition - Core Logic
FIntPoint GridPosition = FIntPoint(FMath::RoundToInt(Position.X ), FMath::F
// ... (create NewNode) ...
NodeMap.Add(GridPosition, NewNode);
// Connect to neighbours within a certain distance
for (auto& Pair : NodeMap) {
    TSharedPtr<FMoveNode> ExistingNode = Pair.Value;
    if (FVector::Dist(NewNode->Position, ExistingNode->Position) <= NodeSep
```

```
            // Check clear path before adding as neighbor
            if (PerformRaycastToPosition(NewNode->Position, ExistingNode->Posit
                NewNode->Neighbors.Add(ExistingNode,true);
                ExistingNode->Neighbors.Add(NewNode, true);
            }
        }
    }
```

2. **AI Quest Bidding System:**
    - The `AMS_AIManager` broadcasts available quests, and idle AI characters (`AMS_AICharacter::IsIdle()`) can `EvaluateQuestAndBid`. The bid value (`CalculateBidValue`) is a heuristic combining expected reward, distance to task (resource node and delivery point), and the AI's current needs (hunger, thirst). The AIManager then assigns the quest to the highest bidder (with tie-breaking).
    - **Why Chosen:** This was an attempt to create a more decentralized and emergent task allocation system than simply assigning tasks to the closest or first available AI (Parsons et al., 2003). It introduces a simple form of "economic" decision-making and allows AIs that are better suited (e.g., less needy, closer) to preferentially take tasks.
    - **Evaluation:** The system works well for distributing simple gathering and delivery tasks. It prevents one AI from being overloaded while others are idle. However, `CalculateBidValue` heuristic needed careful tuning. Initially, distance was over-weighted, leading to AIs far away never getting quests. The "NeedsPenalty" was crucial to ensure AIs prioritized survival. A future improvement could be to factor in AI "skills" or specialized tools if those were implemented.

```
// AMS_AICharacter::CalculateBidValue - Simplified
float AMS_AICharacter::CalculateBidValue(const FQuest& Quest) {
    float DistanceFactor = 1.0f / FMath::Max(1.0f, TotalEstimatedDistance /
    float NeedsPenalty = 1.0f / FMath::Max(1.0f, 1.0f + HungerPenalty + Thi
    float RewardFactor = static_cast<float>(Quest.Reward);
    return RewardFactor * NeedsPenalty * DistanceFactor;
}
```

Testing

User testing and performance profiling were critical throughout the development of Ales And Fables.

1. **Unit & Integration Testing (Automated & Manual):**
    - **Type:** Primarily manual checks during development, with some ad-hoc automated tests using Unreal's console commands or simple test actors.
    - **Process:**
        - **Pathfinding:** Tested `UMS_PathfindingSubsystem::FindPathPoints` (Patel, n.d.c) by spawning two cubes and requesting a path between them, visualizing with debug lines. Dynamically placed obstacles to test `BlockNode` and path recalculation via `OnPathUpdated`.
        - **AI Needs:** Used console commands to manually set AI hunger/thirst (`SetPawnStat` if I had added such a command, or by temporarily modifying decrease rates) to observe if they correctly prioritized

finding food/water over other tasks. Verified `UMS_PawnStatComponent::OnStateChanged` (Gamma et al., 1994) delegate fired.

- **Quest System:** Forced `AMS_AIManager` to generate specific quests and observed if AI characters bid, were assigned, and correctly executed the quest steps (e.g., gather wood, deliver to storage). Log messages were heavily used here.

- **Feedback/Issues:**
  - Pathfinding sometimes failed if start/end nodes were too close to complex unblocked geometry that still occluded direct node-to-node raycasts. Solution: Refined `PerformRaycastToPosition` in `AMS_MovementNodeMeshStarter` to use a slightly elevated raycast.
  - AIs could get stuck in a loop bidding for quests they couldn't immediately start due to low needs. Solution: Added a stronger `NeedsPenalty` in `CalculateBidValue` and ensured the Behavior Tree had higher priority branches for satisfying critical needs.

2. **Performance Profiling:**

- **Type:** Used Unreal Engine's built-in profiling tools (`Stat Unit` , `Stat Game` , Unreal Insights) (Epic Games, n.d.d, "Performance and Profiling").
- **Process:** Ran simulations with increasing numbers of AI agents (e.g., 5, 10, 20, 50) and monitored frame times, game thread, and render thread performance. Focused on:
  - `UMS_PathfindingSubsystem::FindPathNodes` (A* execution).
  - `AMS_AICharacter::Tick` and Behavior Tree execution.
  - `AMS_AIManager::Tick` (quest generation, bid evaluation).
- **Feedback/Issues & Optimizations:**
  - **Initial Issue:** A* pathfinding became a bottleneck with >20 agents frequently recalculating paths.
    - **Optimization 1:** Cached `UMS_PathfindingSubsystem` pointer in `AMS_AICharacter` instead of getting it from GameInstance every time.
    - **Optimization 2:** AI characters only recalculate paths if their current path is invalidated by `OnPathUpdated` or if their primary target changes, not on every minor movement adjustment.
    - **Optimization 3:** The `NodeMap` in `UMS_PathfindingSubsystem` uses `TSharedPtr<FMoveNode>` , which has some overhead. For extreme scale, a plain C-style array or direct indexing might be faster, but `TSharedPtr` was kept for easier memory management and more complex node data.
  - **Initial Issue:** Frequent overlap events from `AMS_AICharacter::ShopCollision` (renamed from an earlier concept probably) if many AIs were near each other or interactive objects.
    - **Optimization:** Ensured collision profiles were set correctly to minimize unnecessary overlap checks. Overlap logic in `OnOverlapBegin` was streamlined, e.g., checking `CurrentTarget == OtherActor` before processing complex interactions.
  - **Initial Issue:** `AMS_AIManager::Tick` checking all managed resource types every frame for `GenerateQuestsForResourceType` .
    - **Optimization:** This wasn't a major bottleneck in tests up to 50 AIs, but for hundreds, this loop could be staggered (e.g., check one resource type per frame, or check less frequently). `DoesIdenticalQuestExist` also iterates lists, which could be optimized with `TSet` for active quest IDs if it became an issue.

Technical Difficulties

1. **Circular Dependencies and Initialization Order:**
    - **Difficulty:** `AMS_AIManager` needs references to pools like `AMS_StorageBuildingPool`. AI Characters need a reference to `AMS_AIManager` and `WorkPlacesPool_` / `StorageBuldingsPool_`. The `PathfindingSubsystem` needs its `NodeMap` generated by `AMS_MovementNodeMeshStarter` before it can be used.
    - **Diagnosis:** Crashes or null pointer exceptions during `BeginPlay` if an actor tried to access another system that wasn't fully initialized.
    - **Resolution:**
        - Heavy reliance on `UGameplayStatics::GetActorOfClass` or `GetSubsystem` within `BeginPlay` or on-demand, with null checks.
        - For `AMS_MovementNodeMeshStarter` and `UMS_PathfindingSubsystem`, the `OnNodeMapReady` delegate (Gamma et al., 1994) was crucial. Systems like `AMS_StorageBuildingPool` subscribe to this delegate and only perform actions requiring the node map (like `FindStorageBuildingsOnScene` which might call `PathfindingSubsystem->AddNodeAtPosition`) *after* the delegate fires.
        - For AI accessing `AIManager`, it's generally safe if AIManager is placed in the level and AI are spawned later or get it in their `BeginPlay`.
    - **Reflection:** A more formal dependency injection system (Fowler, 2004) or a multi-stage initialization process for game systems could make this more robust for larger projects.

    [CODE SNIPPET: AlesAndFables/Source/AlesAndFables/Private/Placeables/Buildings/MS_StorageBuildingPool.cpp - MS_StorageBuildingPool::BeginPlay - Subscribing to OnNodeMapReady.]

```
// AlesAndFables/Source/AlesAndFables/Private/Placeables/Buildings/MS_Stora
void AMS_StorageBuildingPool::BeginPlay()
{
    Super::BeginPlay();
    // Attempt to find the NodeMeshStarter
    for (TActorIterator<AMS_MovementNodeMeshStarter> It(GetWorld()); It; ++
    {
        AMS_MovementNodeMeshStarter* NodeMeshStarter = *It;
        if (NodeMeshStarter)
        {
            // Bind to the OnNodeMapReady delegate if pathfinding isn't rea
            if (!NodeMeshStarter->bNodeMapReady)
            {
                NodeMeshStarter->OnNodeMapReady.AddDynamic(this, &AMS_Stora
            }
            else // Pathfinding map is already ready, initialize immediatel
            {
                OnNodeMapInitialized();
            }
            break; // Found the starter, no need to continue loop
```

```
            }
        }
    }

    void AMS_StorageBuildingPool::OnNodeMapInitialized()
    {
        UE_LOG(LogTemp, Log, TEXT("StorageBuildingPool: Node Map is ready. Init
        FindStorageBuildingsOnScene(); // Now safe to potentially use pathfindi
    }
```

[Figure 28. `AMS_StorageBuildingPool::BeginPlay` and `OnNodeMapInitialized`. The pool subscribes to `AMS_MovementNodeMeshStarter::OnNodeMapReady` (Gamma et al., 1994). Critical initialization logic like `FindStorageBuildingsOnScene` (which might interact with the pathfinding grid) is deferred until `OnNodeMapInitialized` is called, ensuring the pathfinding system is ready. This delegate-based approach helps manage initialization order dependencies.]

Outcomes

Source Code/Project Files

The complete source code for Ales And Fables is publicly available on GitHub, organized into two primary Unreal Engine plugins:

- **Repository Link:** *(Add link)*

1. `AlesAndFables` **Plugin:** Contains the core simulation logic, including:
    - AI Character (`MS_AICharacter.h/.cpp`): Manages AI state, needs, inventory, quest handling, and interactions.
    - AI Controller (`MS_AICharacterController.h/.cpp`): Hosts the Behavior Tree and Blackboard components.
    - AI Manager (`MS_AIManager.h/.cpp`): Oversees quest generation, bidding, construction projects, and population/housing.
    - Workplaces (`MS_BaseWorkPlace.h/.cpp` , `MS_WheatField.h/.cpp` , etc.): Define interactive resource spots and production buildings.
    - Building & Pool Actors (`MS_House.h/.cpp` , `MS_StorageBuildingPool.h/.cpp` , `MS_WorkpPlacePool.h/.cpp` , etc.): Manage instances and availability of structures.
    - Data Structures (`MS_ResourceSystem.h` for `FQuest` , `MS_InventoryComponent.h` for `ResourceType`): Define core data types.
    - Behavior Tree Nodes (Tasks, Decorators, Services): Located in `AI/TaskNodes` , `AI/Decorators` , `AI/Services` subfolders.

2. `CustomMovementPlugin` **Plugin:** Contains the reusable pathfinding system:
    - Pathfinding Subsystem (`MS_PathfindingSubsystem.h/.cpp`): Implements A* pathfinding and dynamic node grid management.

- Node Generation (`MS_MovementNodeMeshStarter.h/.cpp`): Procedurally creates the initial pathfinding grid.
- Node Data (`MS_MovementNode.h` for `FMoveNode`): Defines the structure of a pathfinding node.

The project can be compiled and run using Unreal Engine 5.4.

Build Link

A playable build of Ales And Fables demonstrating the AI behaviors, resource management, and dynamic world can be downloaded from Itch.io:

- **Build Link:** Build

Instructions to Run:

1. Download the `.zip` file from the Itch.io page.
2. Extract the contents to a folder on your computer.
3. Run the `AG_MedievalSim.exe` located in the extracted folder.
4. System Requirements: Windows 10/11, DirectX 11/12 compatible GPU, ~4GB RAM.

Video Demonstration

A video demonstrating key features of Ales And Fables in action is embedded below / available at the following link:

Video Link: https://youtu.be/LFalMgVUfE0

Project Schedule

Card Name	Start Date	Finish Date	Checklist Item	Item State
Add New Job Types	2025-02-03	2025-05-20	Farm Vegetables	complete
Add New Job Types	2025-02-03	2025-05-20	Water Vegetables	complete
Add New Job Types	2025-02-03	2025-05-20	Plant Vegetables	complete
Agent Behaviour	2025-02-03	2025-05-20	Add economy	complete
Agent Behaviour	2025-02-03	2025-05-20	Add day/night cicle and sleep	complete
Building construction	2025-03-23	2025-02-03		complete
Movement system	2025-02-03	2025-05-20	Improve the point generation to work inside homes and with slopes	incomplete
Movement system	2025-02-03	2025-04-06	Optimize the pathfinding	complete

Card Name	Start Date	Finish Date	Checklist Item	Item State
Movement system	2025-02-03	2025-04-06	Add dynamic path modification with enviroment	complete
Job assigment	2024-10-01	2025-05-20	Implement bidding logic for agents	complete
Job assigment	2024-10-01	2025-05-20	Prioritize bids based on needs and distance	complete
Job assigment	2024-10-01	2024-10-19	Create a "Billboard" system for job postings	complete
Level design	2024-10-01	2025-05-20	Add enviroment	complete
Level design	2024-10-01	2025-05-20	Add main buildings	complete
Level design	2024-10-01	2024-10-19	Add task completion static positions	complete
Agent Movement	2024-10-01	2025-02-03		complete
Agent Movement	2024-10-01	2025-02-03	Use eqs to add points to the navmesh	complete
Agent Movement	2024-10-01	2025-02-03	Write the movement algorithm that thakes the points and creates roads	complete
Agent Movement	2024-10-01	2024-10-01	Add the navmesh to the worldmap	complete
Core Manager AI	2024-10-01	2024-11-04	Design the Manager class	complete
Core Manager AI	2024-10-01	2024-11-04	Implement job postings based on village needs	complete
Core Manager AI	2024-10-01	2024-11-04	Track village resources	complete
Village Statistics System	2024-10-01	2024-10-29	Update and monitor stats in real-time	complete
Village Statistics System	2024-10-01	2024-10-29	Track village-wide stats (population, resources)	complete
Base AI Agent Class	2024-10-01	2024-10-26	Add functions to update needs over time	complete
Base AI Agent Class	2024-10-01	2024-10-26	Add properties: needs (hunger, thirst, energy, money)	complete
Base AI Agent Class	2024-10-01	2024-10-26	Implement basic agent behaviors (idle, walk, gather)	complete
Base AI Agent Class	2024-10-01	2024-10-26	Create Agent C++ class	complete
Village resources	2024-10-01	2024-10-22	Define resource types (wood, food, water)	complete

Card Name	Start Date	Finish Date	Checklist Item	Item State
Village resources	2024-10-01	2024-10-22	Create UI to display village resources	complete
Village resources	2024-10-01	2024-10-22	Implement resource gathering and storage	complete
Job Classes	2024-10-01	2024-10-22	Implement resource collection logic	complete
Job Classes	2024-10-01	2024-10-22	Implement job actions	complete
Job Classes	2024-10-01	2024-10-22	Create job classes	complete
AI Controller (MANAGER)	2024-10-01	2024-10-19	Create an AIController class	complete
Agent behaviours	2024-10-01	2024-10-07	Implement tasks and conditions for Behavior Tree	complete
Agent behaviours	2024-10-01	2024-10-07	Create a basic Behavior Tree	complete
Agent behaviours	2024-10-01	2024-10-07	Set up Blackboard for agent-specific data	complete
Agent Stats natural decline	2024-10-01	2025-05-20		complete
AI Agent Base behaviours	2024-10-01	2025-05-20		complete
Add Agent Stats	2024-10-01	2025-04-06		complete
Basic MANAGER	2024-10-01	2025-04-06		complete
BASIC CHARACTER	2024-09-30	2025-05-20		complete
Create an AIController to manage agent decision-making	2024-09-30	2025-05-20		complete
Add functions to update needs over time	2024-09-30	2024-10-19		complete
Implement basic agent behaviors (idle, walk)	2024-09-30	2025-05-20		complete
Create a base Agent C++ class	2024-09-30	2025-05-20		complete
Add properties: needs (hunger, thirst, energy, money)	2024-09-30	2024-10-19		complete

Reflection

Research Effectiveness

The research for *Ales and Fables* was highly effective, directly shaping design and implementation.

- **Game Sources (*RimWorld*, *Banished*):** These inspired high-level AI design, especially task management and resource logistics. *RimWorld's* mood and work prioritization system influenced the `UMS_PawnStatComponent` and AI need-based decisions in the Behavior Tree arguably the most impactful reference overall.
- **Academic Sources (Millington & Funge, Patel's A* Guide):** These provided the theoretical base. Patel's guide directly informed the `UMS_PathfindingSubsystem`, while Millington & Funge clarified Behavior Tree logic, resulting in cleaner, modular AI systems.

Positive Analysis

1. **Modular AI Behavior System:** The use of Behavior Trees with custom C++ tasks and Blackboards enabled AI to handle varied actions (gathering, building, self-care) through reusable components like `MS_PerformWorkAction`, which adjusts behavior via Blackboard keys and internal config.
2. **Dynamic Pathfinding & Interaction:** The `UMS_PathfindingSubsystem` and dynamic node system allowed AI to path around changing environments. The `OnPathUpdated` delegate ensured routes updated automatically when obstacles appeared or vanished.
3. **Emergent Complexity:** Interplay between `AMS_AIManager` (quest generation), AI needs (`UMS_PawnStatComponent`), and bidding (`CalculateBidValue`) led to organic behavior. For example, AIs with urgent needs would avoid quests, leaving them to others better suited naturally distributing labor.

Negative Analysis

1. **Performance Scalability:** The custom A* system and Behavior Tree updates limited scalability. While functional with ~50 AIs, performance may degrade with 100+. Improvements like hierarchical pathfinding, better BT pruning, and further C++ optimization are needed. `AMS_AIManager`'s `Tick` could also bottleneck with more resources or projects.
2. **AI Clumping & Collision:** Unreal's built-in avoidance works, but large groups still bunch up, especially at shared access points. A more robust local avoidance or group movement system wasn't implemented, causing some jamming.

Next Time

1. **Earlier Performance Profiling:** Profiling pathfinding and AI decisions earlier would help spot bottlenecks before layering features.
2. **Use GAS:** Unreal's Gameplay Ability System could better manage AI actions and effects (e.g., hunger reducing work speed) than custom solutions.
3. **Advanced Pathfinding:** Techniques like jump point search or hierarchical pathfinding would enhance performance in larger maps.
4. **AI Debug Tools:** Building in-game tools to visualize AI perception, goals, path queries, or failed Behavior Tree paths would greatly ease debugging and refinement.

Bibliography

- 4HEAD Studios / GolemLabs. (2006-2017). *The Guild 2 / The Guild 3* [PC Games]. JoWooD Productions / THQ Nordic.

- Botea, A., Müller, M., & Schaeffer, J. (2004). Near Optimal Hierarchical Pathfinding. *Journal of Game Development, 1*(1).
- DataCamp. (n.d.). *Manhattan Distance*. Retrieved from https://www.datacamp.com/tutorial/manhattan-distance (Accessed [Your Access Date])
- Epic Games. (n.d.a). *Behavior Trees*. Unreal Engine Documentation. Retrieved from https://docs.unrealengine.com/en-US/InteractiveExperiences/ArtificialIntelligence/BehaviorTrees/ (Accessed [Your Access Date])
- Epic Games. (n.d.b). *AI Controller*. Unreal Engine Documentation. Retrieved from https://docs.unrealengine.com/en-US/InteractiveExperiences/ArtificialIntelligence/AIController/ (Accessed [Your Access Date])
- Epic Games. (n.d.c). *Gameplay Ability System*. Unreal Engine Documentation. Retrieved from https://docs.unrealengine.com/en-US/InteractiveExperiences/GameplayAbilitySystem/ (Accessed [Your Access Date])(Note: URL is a guess, replace with actual GAS doc link)
- Epic Games. (n.d.d). *Performance and Profiling*. Unreal Engine Documentation. Retrieved from https://docs.unrealengine.com/en-US/TestingAndOptimization/PerformanceAndProfiling/ (Accessed [Your Access Date])(Note: URL is a guess, replace with actual profiling doc link)
- Epic Games Community. (n.d.). Various Forum Posts and Wiki Articles on Custom Pathfinding. *Unreal Engine Forums/Wiki*. (Accessed [Your Access Date])(Note: This is a general citation; be more specific if possible)
- Fowler, M. (2004). *Inversion of Control Containers and the Dependency Injection pattern*. Retrieved from https://martinfowler.com/articles/injection.html (Accessed [Your Access Date])
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- Harabor, D., & Grastien, A. (2011). Online Graph Pruning for Pathfinding on Grid Maps. *In Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence (AAAI'11).*
- Ludeon Studios. (2018). *RimWorld* [PC Game]. Ludeon Studios.
- Millington, I., & Funge, J. (2009). *Artificial Intelligence for Games* (2nd ed.). CRC Press.
- Parsons, S., Rodriguez-Aguilar, J. A., & Klein, M. (2003). Auctions and bidding: A guide for computer scientists. *ACM Computing Surveys (CSUR), 35*(1), 29-79. (Note: This is an example for auction theory; replace if you used a different specific source or if it's too general).
- Patel, A. (n.d.c). *A* Pathfinding for Beginners*. Red Blob Games. Retrieved from https://www.redblobgames.com/pathfinding/a-star/introduction.html (Accessed [Your Access Date])
- Russell, S. J., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.
- Scott, M. L. (2001). *Programming Language Pragmatics*. Morgan Kaufmann. (Note: This is a general reference for memory management trade-offs; might be too broad unless a specific concept was drawn).
- Shining Rock Software. (2014). *Banished* [PC Game]. Shining Rock Software LLC.
- van den Berg, J., Lin, M., & Manocha, D. (2006). Real-time navigation of independent agents using adaptive roadmaps. *In Proceedings of the ACM SIGGRAPH/Eurographics symposium on Computer animation* (pp. 57-65).
- van den Berg, J., Patil, S., Sewall, J., Manocha, D., & Lin, M. (2011). ORCA: Optimal Reciprocal Collision Avoidance. *IEEE Transactions on Robotics, 27*(4), 834-844.

Declared Assets

- **3D Models & Environment Assets:**
  - Synty Studios: POLYGON - Fantasy Kingdom pack. Used for majority of buildings, props, and character models. Available at: https://syntystore.com/products/polygon-fantasy-kingdom-pack
- **Engine:** Unreal Engine 5.4.
- **IDE:** Microsoft Visual Studio Community 2022.

- **Version Control:** Git, hosted on GitHub.
- **AI Tools:**
    - **ChatGPT (OpenAI):** Used for brainstorming solutions to specific C++ coding problems, generating boilerplate code for simple UObject classes or functions, and assisting in rephrasing/clarifying text for this development commentary. For example, I might have asked "How to properly override TickTask in a latent BTTaskNode in Unreal Engine C++?" or "Suggest alternative ways to structure a resource management system." The AI-generated code/suggestions were always reviewed, adapted, and integrated manually.
- **External Code/Libraries:** All C++ code for AI logic, pathfinding, resource systems, and gameplay mechanics within the `AlesAndFables` and `CustomMovementPlugin` plugins was written by myself, Aimar Goñi, with the aforementioned AI tool assistance for specific snippets or problem-solving. No external pre-existing game-logic libraries were directly integrated. Standard Unreal Engine modules were used as expected.