Unit System

Table of Contents

Script Reference

Scripts

Core

ManagedSO.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

ManagedSO

```
public class ManagedSO
: ScriptableObject
```

Base class for ScriptableObjects that require a unique identifier (UUID). Identifiers are used to efficiently match and compare items, avoiding reliance on multiple property checks, object references, or other complex comparisons.

Variables

invalidId

```
public static readonly int invalidId
```

Represents an invalid ID value. Used to signify that an ID has not been assigned.

Methods

ID

```
public int ID()
```

Gets the unique identifier (UUID) for this object.

InvalidateID

```
public void InvalidateID()
```

Invalidates the current ID, setting it back to invalidId.

AssignUniqueID

```
public void AssignUniqueID(
     int id)
```

Assigns a new unique ID to this object.

id: The unique ID to assign.

SystemDatabase.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

SystemDatabase

```
public class SystemDatabase
: ScriptableObject
```

Centralized database for managing and tracking ManagedSO assets. This database allows for efficient ID management, grouping, and retrieval of assets, ensuring that only registered assets are managed while allowing others to remain untouched.

Variables

OnDatabaseChange

```
public static System.Action OnDatabaseChange
```

Methods

GetInstance

```
public static SystemDatabase GetInstance()
```

Retrieves the instance of the SystemDatabase. Creates a new instance if none exists.

Returns: The singleton instance of the SystemDatabase.

GetAllAssets

```
public ManagedSO[] GetAllAssets()
```

Reset

```
public void Reset()
```

Clears all registered assets and resets the index to the starting value.

RemoveMissingAssets

```
public void RemoveMissingAssets()
```

Removes null references from the asset list. This should be called after assets are deleted to ensure the list remains clean.

FindAssetById

```
public ManagedSO FindAssetById(
     int id)
```

Retrieves a registered ManagedSO asset by its unique ID.

id: The ID of the asset to retrieve.

Returns: The corresponding ManagedSO asset, or null if not found.

IsRegistered

```
public bool IsRegistered(
     ManagedSO asset)
```

Checks if a given ManagedSO asset is registered in the database.

asset: The asset to check.

Returns: True if the asset is registered; otherwise, false.

Register

```
public void Register(
     ManagedSO managedObject)
```

Registers a ManagedSO asset in the database and assigns a unique ID if necessary.

managedObject: The asset to register.

Remove

```
public bool Remove(
     ManagedSO asset)
```

Removes a ManagedSO asset from the database.

asset: The asset to remove.

Returns: True if the asset was removed; otherwise, false.

RemoveAssetWithId

```
public bool RemoveAssetWithId(
     int id)
```

Removes an asset from the database using its unique ID.

id: The ID of the asset to remove.

Returns: True if the asset was removed; otherwise, false.

DatabaseCleanupProcessor

```
public class DatabaseCleanupProcessor
: AssetPostprocessor
```

Database cleanup processor triggered when an asset imported/created, moved or deleted and it will remove any missing assets from the database to avoid keeping null references.

UnitSystemConfig.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

UnitSystemConfig

```
public class UnitSystemConfig
: ScriptableObject
```

Configuration ScriptableObject for the Unit System. This configuration file defines paths for the database and asset creation within the Unit System. It is essential for managing the Unit System's behavior and ensuring that assets and database files are stored in appropriate locations.

Methods

DatabaseName

```
public string DatabaseName()
```

Gets the default name of the database file.

GetOrCreate

```
public static UnitSystemConfig GetOrCreate()
```

Retrieves or creates the configuration instance. If no configuration exists, a new one is created at the default path. Use this method to ensure the configuration is available.

Returns: The UnitSystemConfig instance.

SetCustomDatabasePath

```
public void SetCustomDatabasePath(
     string path)
```

SetCustomAssetsPath

```
public void SetCustomAssetsPath(
     string path)
```

SetCustomPrefabsPath

```
public void SetCustomPrefabsPath(
     string path)
```

GetAssetsFolderPath

```
public string GetAssetsFolderPath()
```

Gets the folder path for creating new assets. If no custom path is set, it uses the default path.

GetDatabaseFolderPath

```
public string GetDatabaseFolderPath()
```

Gets the folder path for the Unit System database. If no custom path is set, it uses the default path.

GetsPrefabsFolderPath

```
public string GetsPrefabsFolderPath()
```

Gets the folder path for creating new prefabs. If no custom path is set, it uses the default path.

CreateDatabasePath

```
public string CreateDatabasePath()
```

Constructs the full path for the database file based on the current database folder path.

Returns: The full database path.

CreateDatabasePath

```
public string CreateDatabasePath(
     string folder)
```

Constructs the full path for the database file based on a specified folder.

folder: The folder path to use for constructing the database path.

Returns: The full database path.

Entity

AEntityComponent.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

AEntityComponent

```
public abstract class AEntityComponent
: MonoBehaviour, IEntityComponent
```

Provides a abstract foundation for entity components in the framework. This class serves as a foundation for components attached to entities, offering basic lifecycle management and a mechanism to access the associated entity.

Methods

Entity

```
public IEntity Entity()
```

Gets the entity associated with this component. This is typically set automatically during initialization.

IsActive

```
public virtual bool IsActive()
```

Indicates whether the component is active and operational. This can be overridden by subclasses for custom logic.

Owner

```
public APlayer Owner()
```

Gets the owning player of the associated entity.

AEntitySO.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

AEntitySO

```
public abstract class AEntitySO
: AProducibleSO, IEntityPrefabCreatable
```

Represents the base ScriptableObject for defining entity data and capabilities. This class serves as a data layer for entities, allowing configuration of capabilities and integration with prefabs. It provides methods for creating and configuring prefabs dynamically.

Methods

Capabilities

```
public IEntityCapability[] Capabilities()
```

Gets the collection of capabilities associated with the entity.

TryGetCapability

```
public virtual bool TryGetCapability(
     out Capability capability)
```

Attempts to retrieve a capability of the specified type from the entity’s capabilities.

capability: The retrieved capability if it exists; otherwise, the default value for the type.

Returns: true if the capability exists; otherwise, false.

AddCapability

```
public virtual void AddCapability(
     IEntityCapability capability)
```

Adds a new capability to the entity’s list of capabilities.

capability: The capability to add.

AddCapability

```
public virtual CapabilityType AddCapability()
```

Create and add new capability to the entity’s list of capabilities of the specified type. Initializer with no parameters must be available for this method to work.

Returns: Returns capability if initialisation was successful; otherwise false..

CreatePrefab

```
public virtual GameObject CreatePrefab(
     string name)
```

Creates a prefab GameObject for the entity with the specified name.

name: The name of the GameObject to create.

Returns: A new GameObject configured as the entity’s prefab.

ConfigurePrefab

```
public virtual void ConfigurePrefab(
     GameObject prefab)
```

Configures a prefab GameObject by applying entity-specific data and capabilities.

prefab: The GameObject to configure.

Entity.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

Entity

```
public class Entity
: MonoBehaviour, IEntity
```

Represents a core entity in the framework, providing foundational functionality for managing entity data, ownership, components, capabilities, and lifecycle events.

Variables

OnDestroy

```
[Tooltip("Event invoked when unit is destroyed through
DestroyEntity(float) method.")]
public UnityEvent<Entity> OnDestroy
```

OnOperationsActive

```
[Tooltip("Event invoked when the entity becomes operational.")]
public UnityEvent<Entity> OnOperationsActive
```

Properties

Health

```
public IHealth Health
     { get;
     private set; }
```

Gets the health component of this entity, if one is present.

Garrison

```
public IGarrisonEntity Garrison
     { private set;
     get; }
```

Gets the garrison component if present.

ActiveProduction

```
public IActiveProduction ActiveProduction
     { private set;
     get; }
```

Gets the active production component if present.

UnitSpawn

```
public IUnitSpawn UnitSpawn
     { private set;
     get; }
```

Spawn point for units in case they are garrisoned, produced or otherwise spawned out of this entity.

EntityControl

```
public IControlEntity EntityControl
     { private set;
     get; }
```

Primary control for the entity. If entity supports movement and spawn point, this will be returning spawn point control. Otherwise, it will be returning the same value as for movement controls.

Buildable

```
public IBuildableEntity Buildable
     { private set;
     get; }
```

Gets the entity component if buildable.

Methods

Owner

```
public APlayer Owner()
```

Gets the player owner of this entity.

Data

```
public AEntitySO Data()
```

Gets the AEntitySO data associated with this entity.

IsOperational

```
public bool IsOperational()
```

OnInitialize

```
public virtual void OnInitialize()
```

Unity’s Awake lifecycle method. Ensures initialization of the entity.

SetOperational

```
public virtual void SetOperational(
     bool operational)
```

TryGetCapability

```
public bool TryGetCapability(
     out TCapability capability)
```

TryGetEntityComponent

```
public bool TryGetEntityComponent(
     out TEntityComponent component)
```

AssignOwner

```
public void AssignOwner(
     APlayer player)
```

RemoveEntityFromOwner

```
public virtual void RemoveEntityFromOwner()
```

DestroyEntity

```
public virtual void DestroyEntity(
     float delay = 0f)
```

EntityEvents.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

EntityEvents

```
public sealed class EntityEvents
```

Manages events related to entity. Provides a centralized event system for the entity mechanics in the game.

Variables

Instance

```
public static readonly EntityEvents Instance
```

OnEntityDestroy

```
public UnityEvent<Entity> OnEntityDestroy
```

Event invoked when an will be destroyed.

OnOwnerChanged

```
public UnityEvent<Entity> OnOwnerChanged
```

Event invoked when unit changes owners.

EntityPrefabHelper.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

EntityPrefabHelper

```
public static class EntityPrefabHelper
```

Utility class for loading entity prefabs. Provides methods to handle prefab loading for entities and units while ensuring the expected components are attached to the prefabs.

Methods

LoadUnitPrefab

```
public static void LoadUnitPrefab(
     this AUnitSO unit,
     System.Action<Unit> loadedPrefab)
```

Loads the prefab associated with a AUnitSO and invokes the provided callback with the loaded unit prefab.

unit: The unit scriptable object from which to load the prefab.
loadedPrefab: Callback to invoke with the loaded unit prefab.

LoadEntityPrefab

```
public static void LoadEntityPrefab(
     this AEntitySO entity,
     System.Action<IEntity> loadedPrefab)
```

Loads the prefab associated with an AEntitySO and invokes the provided callback with the loaded entity prefab.

entity: The entity scriptable object from which to load the prefab.
loadedPrefab: Callback to invoke with the loaded entity prefab.

EntityRelationshipHelper.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

EntityRelationshipHelper

```
public static class EntityRelationshipHelper
```

Provides utilities for determining relationships between entities, such as ally or enemy status.

Methods

IsEnemy

```
public static bool IsEnemy(
     this IEntity thisEntity,
     IEntity otherEntity)
```

Checks if two entities are considered enemies.

thisEntity: The first entity.
otherEntity: The second entity.

Returns: True if the entities are enemies; otherwise, false.

IsAlly

```
public static bool IsAlly(
     this IEntity thisEntity,
     IEntity otherEntity)
```

Checks if two entities are considered allies.

thisEntity: The first entity.
otherEntity: The second entity.

Returns: True if the entities are allies; otherwise, false.

AreEntitiesEnemies

```
public static bool AreEntitiesEnemies(
     IEntity entityA,
     IEntity entityB)
```

Checks if two entities are considered enemies.

entityA: The first entity.
entityB: The second entity.

Returns: True if the entities are enemies; otherwise, false.

AreEntitiesAllied

```
public static bool AreEntitiesAllied(
     IEntity entityA,
     IEntity entityB)
```

Checks if two entities are considered allies.

entityA: The first entity.
entityB: The second entity.

Returns: True if the entities are allies; otherwise, false.

IEntity.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

IEntity

```
public interface IEntity
```

Root interface for all entities in the game, representing any interactive game object, including units, resource nodes, towers, faction flags, etc.

Properties

transform

```
public Transform transform
     { get;
     }
```

Gets the root transform of the entity.

gameObject

```
public GameObject gameObject
     { get;
     }
```

Gets the root game object of the entity.

Owner

```
public APlayer Owner
     { get;
     }
```

Gets the owner of the entity.

IsOperational

```
public bool IsOperational
     { get;
     }
```

Indicates whether the entity is operational and ready to be used. This should only be false if entity requires construction or startup animation before it is ready for use.

Data

```
public AEntitySO Data
     { get;
     }
```

Gets the data associated with the entity, this would be a scriptable object defining its configuration.

Health

```
public IHealth Health
     { get;
     }
```

Gets the health system associated with the entity, if present.

Garrison

```
public IGarrisonEntity Garrison
     { get;
     }
```

Contract for entity garrison which enables other units to enter this one.

ActiveProduction

```
public IActiveProduction ActiveProduction
     { get;
     }
```

Contract for entity production component.

UnitSpawn

```
public IUnitSpawn UnitSpawn
     { get;
     }
```

Contract for entity spawn point component.

EntityControl

```
public IControlEntity EntityControl
     { get;
     }
```

Contract for entity control for direct command. This is used for non-automated behaviour, like initial movement on spawn or player commands.

Buildable

```
public IBuildableEntity Buildable
     { get;
     }
```

Contract for entity which requires building.

Methods

AssignOwner

```
public void AssignOwner(
     APlayer newOwner)
```

Assigns ownership of this entity to a specified player.

player: The player to whom ownership will be assigned.

RemoveEntityFromOwner

```
public void RemoveEntityFromOwner()
```

Removes this entity from its current owner's control. This method should be used to manually manage ownership before destruction.

DestroyEntity

```
public void DestroyEntity(
     float delay = 0f)
```

Destroys the entity and removes it from its owner's control.

delay: The delay, in seconds, before the GameObject is destroyed. Defaults to 0.

SetOperational

```
public void SetOperational(
     bool operational)
```

Sets the operational state of the entity. By default, all entities are operational; this can be useful for cases like building state of an entity.

operational: True if the entity is becoming operational, false otherwise.

TryGetCapability

```
public bool TryGetCapability(
     out Capability capability)
```

Attempts to retrieve a capability of the specified type from the entity's data.

capability: The retrieved capability, if available.

Returns: True if the capability exists; otherwise, false.

TryGetEntityComponent

```
public bool TryGetEntityComponent(
     out EntityComponent component)
```

Attempts to retrieve a component of the specified type attached to the entity.

component: The retrieved component, if available.

Returns: True if the component exists; otherwise, false.

IEntityCapability.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

IEntityCapability

```
public interface IEntityCapability
```

Represents a capability that can be added to or configured for an entity.

IEntityComponent.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

IEntityComponent

```
public interface IEntityComponent
```

Defines a contract for components that can be attached to entities within the framework.
Provides access to the entity, its transform, and its activity status.

Faction

AFactionSO.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

UnitQuantity

```
[System.Serializable]
public struct UnitQuantity
```

Represents the quantity of a specific unit type that can be spawned or produced.

Variables

count

```
[PositiveInt]
[Tooltip("The number of units of the specified type.")]
public int count
```

unit

```
[Tooltip("Specifies limited unit, this can be of any type that
player can spawn. " +
"When limit is reached it’s requirements for
production will not be fulfilled.")]
public AUnitSO unit
```

The unit type associated with this quantity. This can represent any unit type that a player can spawn or produce. If a limit is reached, its production requirements will not be fulfilled.

AFactionSO

```
public abstract class AFactionSO
: ManagedSO
```

Represents an abstract base class for faction definitions within the game. Stores essential faction details such as name, description, and unit limits.

Methods

Description

```
public string Description()
```

Gets the description of the faction.

Name

```
public string Name()
```

Gets the name of the faction.

UnitMaxLimit

```
public UnitLimit[] UnitMaxLimit()
```

Gets the maximum allowed limits for specific unit types in the faction.

SetFactionName

```
public void SetFactionName(
     string name)
```

Sets new faction name.

SetDescription

```
public void SetDescription(
     string description)
```

Sets new faction description.

ConfigurePlayer

```
public abstract void ConfigurePlayer(
     APlayer player)
```

Configures the faction for a specific player. This typically involves setting up resource limits, research, and other faction-specific parameters.

player: The player entity to configure.

GetStartingUnits

```
public abstract UnitQuantity[] GetStartingUnits()
```

Returns an array of unit prefabs and their count. This interface is generally used for spawning initial units of the player in world.

Returns: Array of starting units and their count

FactionSO.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

FactionSO

```
[CreateAssetMenu(fileName = "New faction", menuName = "Unit
System/Faction")]
public class FactionSO
: AFactionSO
```

Represents a concrete implementation of a faction, defining its starting configuration, default production actions, and resource setup.

Methods

DefaultActions

```
public ProductionAction[] DefaultActions()
```

Gets the default production actions available for the faction.

SetDefaultActions

```
public void SetDefaultActions(
     ProductionAction[] actions)
```

Sets new default actions for the faction.

SetStartData

```
public void SetStartData(
     FactionStartData data)
```

Sets new start data for the faction.

ConfigurePlayer

```
public override void ConfigurePlayer(
     APlayer player)
```

Configures the player’s faction-specific settings, such as resource limits, population, and starting research upgrades.

player: The player entity to configure.

FactionStartData.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

FactionStartData

```
[System.Serializable]
public struct FactionStartData
```

Represents the initial data configuration for a faction, including starting population, resources, and unit limits.

Variables

PopulationHardLimit

```
[Header("Population")]
[Tooltip("The maximum population capacity of the faction. This
is a hard limit of the faction.")]
public int PopulationHardLimit
```

PopulationCapacity

```
[FormerlySerializedAs("populationCapacity")]
[Tooltip("The additional starting population capacity of the
faction.\n" +
"This does not include the capacity contributed
by starting units.")]
public int PopulationCapacity
```

Methods

Units

```
public readonly UnitQuantity[] Units()
```

Gets the initial units available to the faction.

Resources

```
public readonly ResourceQuantity[] Resources()
```

Gets the initial resources available to the faction.

ResourceCapacities

```
public readonly ResourceQuantity[] ResourceCapacities()
```

Gets the maximum starting resources allowed for the faction.

Researches

```
public readonly ResearchSO[] Researches()
```

The initial researches already done by the faction.

Interactions

ALimitedInteractionTarget.cs

Namespaces

UnitSystem.Interactions

```
namespace UnitSystem.Interactions
```

Classes

ALimitedInteractionTarget

```
[System.Serializable]
public abstract class ALimitedInteractionTarget
: AEntityComponent, ILimitedUnitInteractionTarget
```

Entity component used to enforce limitations on concurrent interactions.

Variables

TargetingEntities

```
public readonly HashSet<IUnit> TargetingEntities
```

Tracks the units currently assigned to interact with this target.

Methods

InteractionLimit

```
public int InteractionLimit()
```

ActiveInteractions

```
public int ActiveInteractions()
```

RemainingInteractions

```
public int RemainingInteractions()
```

Gets the number of remaining interaction slots available.

Position

```
public Vector3 Position()
```

Gets the position of this interaction target in the world.

IsLimitActive

```
public bool IsLimitActive()
```

HasReachedLimit

```
public bool HasReachedLimit()
```

Reset

```
public void Reset()
```

Resets the target, clearing all active interactions.

DefaultMoveUnitForInteraction.cs

Namespaces

UnitSystem.Interactions

```
namespace UnitSystem.Interactions
```

Classes

DefaultMoveUnitForInteraction

```
public struct DefaultMoveUnitForInteraction
: IMoveUnitForInteraction
```

The default implementation of the IMoveUnitForInteraction interface. This class uses a helper class (InteractorPositionHelper) to determine and set the interaction position. It is assumed that the underlying movement system handles the actual unit movement.

ILimitedUnitInteractionTarget.cs

Namespaces

UnitSystem.Interactions

```
namespace UnitSystem.Interactions
```

Classes

ILimitedUnitInteractionTarget

```
public interface ILimitedUnitInteractionTarget
: IUnitInteracteeComponent
```

Defines an interaction target that imposes a limit on the number of units that can simultaneously interact with it. Extends the IUnitInteracteeComponent interface.

IMoveUnitForInteraction.cs

Namespaces

UnitSystem.Interactions

```
namespace UnitSystem.Interactions
```

Classes

IMoveUnitForInteraction

```
public interface IMoveUnitForInteraction
```

Defines a contract for managing unit movement once interaction is invoked. It mediates between interaction components and a movement system. Interactor should find best appropriate position to move to. The rest should be controlled by the movement components.

Methods

SetInteractionTarget

```
public bool SetInteractionTarget(
     IUnitInteractorComponent interactor,
     IUnitInteracteeComponent interactee)
```

Sets the interaction target, moving the interactor unit to an appropriate position for interaction.

interactor: The unit initiating the interaction (e.g., the unit performing an action).
interactee: The unit being interacted with (e.g., the unit receiving the action).

Returns: True if the interaction target was successfully set and movement was initiated; otherwise, false.

RemoveInteractionTarget

```
public bool RemoveInteractionTarget(
     IUnitInteractorComponent interactor,
     IUnitInteracteeComponent interactee)
```

Removes the interaction target.

interactor: The unit that was interacting.
interactee: The unit that was being interacted with.

Returns: True if the interaction target was successfully removed; otherwise, false.

IPredefinedInteractionPositionProvider.cs

Namespaces

UnitSystem.Interactions

```
namespace UnitSystem.Interactions
```

Classes

IPredefinedInteractionPositionProvider

```
public interface IPredefinedInteractionPositionProvider
```

Defines the contract for providing predefined interaction positions for units attempting to interact with other units in the system.

Methods

GetAvailableInteractionPosition

```
public Vector3 GetAvailableInteractionPosition(
     IUnitInteractorComponent interactor,
     IUnitInteracteeComponent interactee,
     bool reserve)
```

Retrieves an available interaction position for the given interactor and interactee.

interactor: The unit attempting to interact.
interactee: The unit being interacted with.
reserve: Whether to reserve the position for the interactor.

Returns: A Vector3 representing the available interaction position.

ReleaseInteractionPosition

```
public bool ReleaseInteractionPosition(
     IUnitInteractorComponent interactor,
     IUnitInteracteeComponent interactee)
```

Releases a previously reserved interaction position.

interactor: The unit that occupied the position.
interactee: The unit being interacted with.

Returns: True if the position was successfully released, otherwise false.

IUnitInteracteeComponent.cs

Namespaces

UnitSystem.Interactions

```
namespace UnitSystem.Interactions
```

Classes

IUnitInteracteeComponent

```
public interface IUnitInteracteeComponent
: IEntityComponent
```

Defines an object that can be interacted with by an IUnitInteractorComponent.

Properties

Position

```
public Vector3 Position
     { get;
     }
```

Current position of the interactee.

IUnitInteractingPosition.cs

Namespaces

UnitSystem.Interactions

```
namespace UnitSystem.Interactions
```

Classes

IUnitInteractingPosition

```
public interface IUnitInteractingPosition
: IUnitInteracteeComponent
```

Represents an object that can provide a specific interaction position for a unit. Extends the IUnitInteracteeComponent interface. This allows units to define their own interaction positions instead of interactors simply positioning themselves around it.

IUnitInteractorComponent.cs

Namespaces

UnitSystem.Interactions

```
namespace UnitSystem.Interactions
```

Classes

IUnitInteractorComponent

```
public interface IUnitInteractorComponent
: IUnitComponent
```

Defines a unit capable of initiating interactions with an IUnitInteracteeComponent.

StaticPositions

GeneratedPositionsData.cs

Namespaces

UnitSystem.Interactions.PredefinedPositions

```
namespace UnitSystem.Interactions.PredefinedPositions
```

IPositionGenerator.cs

Namespaces

UnitSystem.Interactions.PredefinedPositions

```
namespace UnitSystem.Interactions.PredefinedPositions
```

PredefinedPositionGenerator.cs

Namespaces

UnitSystem.Interactions.PredefinedPositions

```
namespace UnitSystem.Interactions.PredefinedPositions
```

Classes

PredefinedPositionGenerator

```
[System.Serializable, DisallowMultipleComponent]
public class PredefinedPositionGenerator
: IPositionGenerator
```

A class that generates predefined positions around a target area. The target area can be a sphere or a box, and the positions can be generated at a specified distance.

Constructors

PredefinedPositionGenerator

```
public  PredefinedPositionGenerator(
     Transform transform)
```

Initializes a new instance of the PredefinedPositionGenerator class.

transform: The transform of the target area.

Variables

transform

```
[SerializeField, HideInInspector]
public Transform transform
```

Properties

Bounds

```
public Bounds Bounds
     { get;
     }
```

Computed bounds of the target area based on its size and configuration.

Methods

TryGeneratePositionsInRange

```
public bool TryGeneratePositionsInRange(
     float distance,
     out Vector3[] result)
```

Attempts to generate positions around the target area within the specified range.

distance: The range within which positions should be generated.
result: An array of positions generated within the specified range. Returns an empty array if no positions are generated.

Returns: True if positions were generated successfully; otherwise, false.

DrawGizmos

```
public void DrawGizmos()
```

Draws gizmos in the editor to visualize the target area's shape and generated positions.

StaticInteractionPositionProvider.cs

Namespaces

UnitSystem.Interactions.PredefinedPositions

```
namespace UnitSystem.Interactions.PredefinedPositions
```

Classes

StaticInteractionPositionProvider

```
[DisallowMultipleComponent]
public class StaticInteractionPositionProvider
: MonoBehaviour, IPredefinedInteractionPositionProvider
```

Provides predefined interaction positions for interactors engaging with an interactee. This class utilizes a predefined position generator and manages interaction positions dynamically.

Utility

CirclePositioner.cs

Namespaces

UnitSystem.Interactions.PredefinedPositions

```
namespace UnitSystem.Interactions.PredefinedPositions
```

Classes

CirclePositioner

```
public static class CirclePositioner
```

Utility class for generating evenly distributed positions around a sphere collider or circular area.

Methods

GeneratePositionsAroundSphereCollider

```
public static List<Vector3>
GeneratePositionsAroundSphereCollider(
     Vector3 center,
     float radius,
     Vector3 scale,
     float unitRadius,
     float additionalDistance = 0)
```

Generates positions around a sphere collider or circular area.

center: The center point of the circle in local space.
radius: The radius of the circle or sphere collider.
scale: The scale of the object. Used to adjust the radius dynamically.
unitRadius: The radius of the unit or object to be positioned.
additionalDistance: Additional distance to be added to the calculated radius. Defaults to 0.

Returns: A list of Vector3 positions representing evenly distributed points around the circle.

RectanglePositioner.cs

Namespaces

UnitSystem.Interactions.PredefinedPositions

```
namespace UnitSystem.Interactions.PredefinedPositions
```

Classes

RectanglePositioner

```
public static class RectanglePositioner
```

Utility class for generating evenly distributed positions around the bounds of a rectangle or cuboid.

Methods

GeneratePositionsAroundBounds

```
public static List<Vector3>
GeneratePositionsAroundBounds(
     Bounds bounds,
     float unitRadius,
     float distance = 0)
```

Generates positions along the perimeter of a rectangular bounds.

bounds: The bounds defining the rectangular area.
unitRadius: The spacing between positions, based on the radius of the unit or object.
distance: An additional offset distance from the bounds. Defaults to 0.

Returns: A list of Vector3 positions evenly distributed around the rectangle’s perimeter.

Modules

Build

BuildEvents.cs

Namespaces

UnitSystem.Build

```
namespace UnitSystem.Build
```

Classes

BuildEvents

```
public sealed class BuildEvents
```

Centralized event manager for handling the build process of units, including notifications for starting, updating, canceling, and completing builds.

Constructors

BuildEvents

```
private  BuildEvents()
```

Private constructor to enforce the singleton pattern.

Variables

Instance

```
public static readonly BuildEvents Instance
```

Singleton instance of the BuildEvents class, ensuring a single point for managing build-related events.

OnBuilderFinished

```
public UnityEvent<Builder,IEntity> OnBuilderFinished
```

Invoked when a builder finishes their assigned task for building an entity.

OnBuildCompleted

```
public UnityEvent<EntityBuilding> OnBuildCompleted
```

Invoked when the building process for an entity is fully completed.

OnBuildStarted

```
public UnityEvent<EntityBuilding> OnBuildStarted
```

Invoked when the building process for an entity begins.

OnBuildCanceled

```
public UnityEvent<EntityBuilding> OnBuildCanceled
```

Invoked when the build process for an entity is canceled.

OnBuildUpdated

```
public UnityEvent<EntityBuilding,float> OnBuildUpdated
```

Invoked periodically to provide updates on the progress of an entity's build process.

BuildableEntitySOHelper.cs

Namespaces

UnitSystem.Build

```
namespace UnitSystem.Build
```

Classes

BuildableEntitySOHelper

```
public static class BuildableEntitySOHelper
```

Provides extension methods for AUnitSO to determine whether a unit requires building or supports automatic building.

Methods

RequiresBuilding

```
public static bool RequiresBuilding(
     this AEntitySO entity)
```

Determines if the unit requires a build process by checking for the IBuildableEntityCapability.

entity: The entity ScriptableObject to evaluate.

Returns: true if the entity has the IBuildableEntityCapability; otherwise, false.

BuildingModule.cs

Namespaces

UnitSystem.Build

```
namespace UnitSystem.Build
```

Classes

BuildingModule

```
[DisallowMultipleComponent]
public class BuildingModule
: APlayerModule
```

A module for managing building operations within a player's system. Handles assigning builders to building tasks, managing resource consumption, and integrating with other modules such as CollectionModule.

Variables

BuildingFinder

```
public IBuildableEntityFinder BuildingFinder
```

Methods

StartBuilding

```
public bool StartBuilding(
     GameObject newObject)
```

Starts the building process of a new entity if eligible.

newObject: The GameObject representing the new entity to build.

Returns: True if building started successfully; otherwise, false.

StartBuilding

```
public bool StartBuilding(
     Entity entity)
```

Starts the building process of a new entity if eligible.

entity: The Entity to build.

Returns: True if building started successfully; otherwise, false.

GetBuilderCount

```
public int GetBuilderCount(
     EntityBuilding entity)
```

Gets the number of assigned builders on a specific entity in building process.

entity: A building entity.

Returns: Number of assigned builders.

GetAvailableBuilderCount

```
public int GetAvailableBuilderCount()
```

Returns the count of currently available builders.

FindClosestUnitBuilding

```
public bool FindClosestUnitBuilding(
     Builder builder,
     bool filterFull,
     out IBuildableEntity buildable)
```

Finds the closest entity with an active building process.

builder: Builder to find the entity for building
filterFull: If enabled, filters out full units (limited by Interactions.ILimitedUnitInteractionTarget).
buildable: Closest entity in building process.

Returns: Returns true if entity was found; otherwise false.

IBuildableEntityFinder.cs

Namespaces

UnitSystem.Build

```
namespace UnitSystem.Build
```

Classes

IBuildableEntityFinder

```
public interface IBuildableEntityFinder
```

Provides a contract for finding buildable units nearby.

Methods

FindNearby

```
public IBuildableEntity[] FindNearby(
     Vector3 position,
     float range,
     bool filterFull)
```

Finds nearby unfinished buildable entities within a specified range.

position: The position to search from.
range: The maximum search radius.
filterFull: Specifies whether to exclude units building that have reached their build limit.

Returns: An array of IBuildableEntity instances that match the criteria, sorted by proximity to the given position.

FindClosest

```
public bool FindClosest(
     Vector3 position,
     float range,
     bool filterFull,
     out IBuildableEntity buildable)
```

Finds the closest unfinished buildable entity within a specified range.

position: The position to search from.
range: The maximum search radius.
filterFull: Specifies whether to exclude entities that have reached their build interaction limit.
buildable: When the method returns true, contains the closest buildable instance; otherwise, null.

Returns: true if an eligible unit building is found; otherwise, false.

BuildableEntityFinder

```
[System.Serializable]
public class BuildableEntityFinder
: IBuildableEntityFinder
```

Default implementation for IBuildableEntityFinder where it takes players list of entities and searches for the nearby buildable entities.

Constructors

BuildableEntityFinder

```
public  BuildableEntityFinder(
     APlayer player)
```

Methods

FindNearby

```
public IBuildableEntity[] FindNearby(
     Vector3 position,
     float range,
     bool filterFull)
```

FindClosest

```
public bool FindClosest(
     Vector3 position,
     float range,
     bool filterFull,
     out IBuildableEntity buildable)
```

Buildable

ABuildingAnimation.cs

Namespaces

UnitSystem.Build

```
namespace UnitSystem.Build
```

BuildStateToggler.cs

Namespaces

UnitSystem.Build

```
namespace UnitSystem.Build
```

Classes

BuildStateToggler

```
[DisallowMultipleComponent]
public class BuildStateToggler
: MonoBehaviour
```

A component that visualizes the build state of a unit by enabling or disabling a specified GameObject (e.g., a sprite or mesh) when build events occur.

BuildableEntityCapability.cs

Namespaces

UnitSystem.Build

```
namespace UnitSystem.Build
```

Classes

BuildableEntityCapability

```
public struct BuildableEntityCapability
: IBuildableEntityCapability
```

Defines the implementation for the IBuildableEntityCapability interface. Encapsulates the configuration required for enabling the building functionality on an entity.

Methods

AutoBuild

```
public readonly bool AutoBuild()
```

UsesHealth

```
public readonly bool UsesHealth()
```

EntityBuilding.cs

Namespaces

UnitSystem.Build

```
namespace UnitSystem.Build
```

Classes

EntityBuilding

```
[DisallowMultipleComponent]
public class EntityBuilding
: ALimitedInteractionTarget, IBuildableEntity,
IUnitInteractingPosition
```

Manages the building process for a unit. Tracks build progress, handles build events, and integrates with unit capabilities to determine build behavior.

Variables

OnBuildProgress

```
[Header("Events")]
public UnityEvent<float> OnBuildProgress
```

Event invoked when the building progress updates. Parameter: Current build progress as a float between 0 and 1.

OnBuildStarted

```
public UnityEvent OnBuildStarted
```

Event invoked when the building process starts.

OnBuildCompleted

```
public UnityEvent OnBuildCompleted
```

Event invoked when the building process is completed.

OnBuildCanceled

```
public UnityEvent OnBuildCanceled
```

Event invoked when the building process is canceled before completion.

PredefinedPositionProvider

```
public IPredefinedInteractionPositionProvider
PredefinedPositionProvider
```

Position provider reference. This is an optional provider.

Properties

Progress

```
public float Progress
     { get;
     private set; }
```

Methods

BuildAutomatically

```
public bool BuildAutomatically()
```

IsBuilt

```
public bool IsBuilt()
```

Indicates whether the unit is fully built and operational.

GetAvailableInteractionPosition

```
public virtual Vector3 GetAvailableInteractionPosition(
     IUnitInteractorComponent interactor,
     bool reserve)
```

ReleaseInteractionPosition

```
public virtual bool ReleaseInteractionPosition(
     IUnitInteractorComponent interactor)
```

StartBuilding

```
public void StartBuilding()
```

Starts the building process for the unit. This is called automatically if manuallyStartBuild is set to false.

BuildWithPower

```
public void BuildWithPower(
     float power)
```

FinishBuilding

```
public void FinishBuilding()
```

Instantly finishes the building process and marks the unit as operational.

IBuildableEntity.cs

Namespaces

UnitSystem.Build

```
namespace UnitSystem.Build
```

Classes

IBuildableEntity

```
public interface IBuildableEntity
: IUnitInteracteeComponent
```

Represents a unit that can undergo a building process. Provides mechanisms to track and manage the state of the build.

IBuildableEntityCapability.cs

Namespaces

UnitSystem.Build

```
namespace UnitSystem.Build
```

Classes

IBuildableEntityCapability

```
public interface IBuildableEntityCapability
: IUnitCapability
```

Represents the capability of a unit to be built. Provides metadata for whether the building process should start automatically.

Properties

AutoBuild

```
public bool AutoBuild
     { get;
     }
```

Gets a value indicating whether the building process should start automatically when the unit is initialized.

UsesHealth

```
public bool UsesHealth
     { get;
     }
```

Gets a value indicating whether the building process should be using health or independent progress value.

RaiseBuildingAnimation.cs

Namespaces

UnitSystem.Build

```
namespace UnitSystem.Build
```

Builder

Builder.cs

Namespaces

UnitSystem.Build

```
namespace UnitSystem.Build
```

Classes

Builder

```
[DisallowMultipleComponent]
public class Builder
: AUnitComponent, ITaskInput, ITaskProvider, IUnitBuilder
```

Represents a component that enables a unit to act as a builder, managing building tasks and interactions with buildable entities.

Methods

PickUpWorkWhenMoving

```
public bool PickUpWorkWhenMoving()
```

Gets whether the unit can pick up new work while moving.

BuildPower

```
public int BuildPower()
```

Gets the build power of the unit.

BuildWithIntervals

```
public bool BuildWithIntervals()
```

Gets whether the unit builds with intervals based on reload time.

BuildInterval

```
public float BuildInterval()
```

Gets the reload time between build updates.

Position

```
public Vector3 Position()
```

Gets the unit’s current position in the world.

CurrentAssignment

```
public EntityBuilding CurrentAssignment()
```

Gets the unit’s current build assignment.

AutoPickUpWorkRadius

```
public float AutoPickUpWorkRadius()
```

Gets the radius for automatic task pickup.

MinInteractionRange

```
public float MinInteractionRange()
```

Gets the minimum range for interactions.

MaxInteractionRange

```
public float MaxInteractionRange()
```

Gets the maximum range for interactions.

GoBuildUnit

```
public bool GoBuildUnit(
     IBuildableEntity buildable)
```

ShouldAutoPickupJob

```
public bool ShouldAutoPickupJob(
     Vector3 jobPosition)
```

Determines whether the unit should automatically pick up tasks based on its position.

jobPosition: The position of the task to evaluate.

Returns: True if the unit should pick up the task, false otherwise.

BuilderCapability.cs

Namespaces

UnitSystem.Build

```
namespace UnitSystem.Build
```

Classes

BuilderCapability

```
[System.Serializable]
public struct BuilderCapability
: IBuilderCapability
```

Concrete implementation of the IBuilderCapability interface. Defines serialized properties for configuring the building capabilities of units.

Methods

AutoPickup

```
public readonly bool AutoPickup()
```

PickupRadius

```
public readonly float PickupRadius()
```

MinRange

```
public readonly float MinRange()
```

Range

```
public readonly float Range()
```

BuildInterval

```
public readonly float BuildInterval()
```

Power

```
public readonly int Power()
```

IBuilder.cs

Namespaces

UnitSystem.Build

```
namespace UnitSystem.Build
```

Classes

IUnitBuilder

```
public interface IUnitBuilder
: IUnitInteractorComponent
```

Represents a unit that has the capabilities of building another entity in world space (construction).

IBuilderCapability.cs

Namespaces

UnitSystem.Build

```
namespace UnitSystem.Build
```

Classes

IBuilderCapability

```
public interface IBuilderCapability
: IUnitCapability
```

Represents the capability for units to build other entities. Includes attributes for automated task pickup, interaction ranges, and build power and frequency.

Task

BuildTargetInput.cs

Namespaces

UnitSystem.Build

```
namespace UnitSystem.Build
```

Classes

BuildTargetInput

```
public struct BuildTargetInput
: ITaskInput
```

Represents input data required for executing a build task.

Constructors

BuildTargetInput

```
public  BuildTargetInput(
     Builder builder)
```

Creates a new instance with the specified builder component.

builder: The Builder component initiating the task.

Variables

builder

```
public Builder builder
```

The build component providing data and behavior for the task.

BuildTargetTask.cs

Namespaces

UnitSystem.Build

```
namespace UnitSystem.Build
```

Classes

BuildTargetTask

```
public class BuildTargetTask
: ITask
```

Represents a task for building a specific target unit. This task ensures the builder reaches the target and performs the build process.

Methods

CanExecuteTask

```
public bool CanExecuteTask(
     ITaskContext context,
     ITaskInput input)
```

CreateHandler

```
public ITaskHandler CreateHandler()
```

BuildTargetTaskHandler.cs

Namespaces

UnitSystem.Build

```
namespace UnitSystem.Build
```

Classes

BuildTargetTaskHandler

```
public class BuildTargetTaskHandler
: ITaskHandler
```

Handles the execution of a build task, moving the builder to the target unit and performing the build process.

Methods

CurrentTarget

```
public EntityBuilding CurrentTarget()
```

The current target being built.

IsBuilding

```
public bool IsBuilding()
```

Indicates whether the builder is actively building the target.

IsFinished

```
public bool IsFinished()
```

StartTask

```
public void StartTask(
     ITaskContext context,
     ITaskInput input)
```

UpdateTask

```
public void UpdateTask(
     float deltaTime)
```

EndTask

```
public void EndTask()
```

Collection

CollectionEvents.cs

Namespaces

UnitSystem.Collection

```
namespace UnitSystem.Collection
```

Classes

CollectionEvents

```
public class CollectionEvents
```

Manages events related to resource collection, including resource deposits, depletion of resource nodes, and missing resource depots. Provides a centralized event system for the collection mechanics in the game.

Variables

OnResourceDeposited

```
public UnityEvent<IResourceCollector,ProducibleQuantity>
OnResourceDeposited
```

Event invoked when a resource collector deposits resources into a depot.

OnNodeDepleted

```
public UnityEvent<IResourceNode> OnNodeDepleted
```

Event invoked when a resource node is depleted.

OnMissingDepot

```
public UnityEvent<ResourceCollector> OnMissingDepot
```

Event invoked when collector wants to deposit his resources, but no depot is found.

Properties

Instance

```
public static CollectionEvents Instance
     { get;
     }
```

Singleton instance of the CollectionEvents class. Ensures centralized event management.

CollectionModule.cs

Namespaces

UnitSystem.Collection

```
namespace UnitSystem.Collection
```

Classes

CollectionModule

```
[DisallowMultipleComponent]
public class CollectionModule
: APlayerModule
```

Manages the resource collection process for a player, including tracking resource collectors and depots, and handling interactions with resource nodes.

Methods

GetAvailableCollectors

```
public IResourceCollector[] GetAvailableCollectors()
```

Gets the available collectors that are idle and capable of collecting resources.

FindNearbyNode

```
public bool FindNearbyNode(
     Vector3 position,
     ResourceSO resource,
     out IResourceNode node)
```

Finds a nearby resource node for a specified resource type. It uses the default nextNodeRange for the search.

position: The position from which to search.
resource: The resource type to search for.
node: The closest or randomly selected resource node found.

Returns: True if a resource node was found; otherwise, false.

FindNearbyNode

```
public bool FindNearbyNode(
     Vector3 position,
     ResourceSO resource,
     float range,
     out IResourceNode node)
```

Finds a nearby resource node for a specified resource type.

position: The position from which to search.
resource: The resource type to search for.
node: The closest or randomly selected resource node found.
range: Maximal range of the search.

Returns: True if a resource node was found; otherwise, false.

FindNearestDepot

```
public bool FindNearestDepot(
     Vector3 position,
     ResourceSO resource,
     out IResourceDepot depot)
```

Finds the nearest resource depot capable of accepting a specified resource.

position: The starting position for the search.
resource: The resource to deposit.
depot: The closest resource depot found.

Returns: True if a resource depot was found; otherwise, false.

Collector

CollectType.cs

Namespaces

UnitSystem.Collection

```
namespace UnitSystem.Collection
```

Enumerations

CollectType

```
public enum CollectType{
     RealtimeCollect,
     StackAndCollect,
     GatherAndDeposit}
```

Enum specifying different types of collection behaviors for resource collectors.

RealtimeCollect: Collects resources directly into the player’s resource pool without requiring transport or capacity management.

StackAndCollect: Collects resources into the collector’s storage until capacity is reached, then automatically adds them to the player’s resource pool without requiring transport.

GatherAndDeposit: Collects resources into the collector’s storage and requires transporting them to a resource depot for depositing.

IResourceCollector.cs

Namespaces

UnitSystem.Collection

```
namespace UnitSystem.Collection
```

Classes

IResourceCollector

```
public interface IResourceCollector
: IUnitComponent
```

Defines the interface for resource collectors, outlining their interaction with resource nodes and supported resource types.

IResourceCollectorCapability.cs

Namespaces

UnitSystem.Collection

```
namespace UnitSystem.Collection
```

Classes

IResourceCollectorCapability

```
public interface IResourceCollectorCapability
: IEntityCapability
```

Defines the capability of a resource collector, specifying its collection behavior, range, capacity, and other collection-related attributes.

ResourceCollector.cs

Namespaces

UnitSystem.Collection

```
namespace UnitSystem.Collection
```

Classes

ResourceCollector

```
[DisallowMultipleComponent]
public class ResourceCollector
: AUnitComponent, IResourceCollector,
IUnitInteractorComponent, ITaskProvider
```

Represents a unit component responsible for collecting resources from resource nodes and depositing them into the player's resource system.

Variables

OnCollectedResource

```
public UnityEvent<IResourceNode,long> OnCollectedResource
```

Event invoked when a resource is collected from the node. This does not imply that the resource is deposited to the unit's owner, but only that the transaction of the resources from node to collector was done.

OnResourceDeposited

```
public UnityEvent<ProducibleQuantity> OnResourceDeposited
```

Event invoked when a resource is successfully deposited.

Properties

CollectedResource

```
public ResourceQuantity CollectedResource
     { get;
     set; }
```

The currently collected resource and its quantity.

Methods

IsCapacityFull

```
public bool IsCapacityFull()
```

Checks if carrying capacity is full.

SupportedResources

```
public ResourceSO[] SupportedResources()
```

The types of resources that the collector can gather.

CollectInterval

```
public float CollectInterval()
```

The time interval between resource collection actions.

CollectAmount

```
public int CollectAmount()
```

The amount of resource collected in a single action.

FollowTarget

```
public bool FollowTarget()
```

Whether the resource collector should follow the target node dynamically.

CollectType

```
public CollectType CollectType()
```

The type of collection behavior used by the collector.

DelayAfterDeposit

```
public float DelayAfterDeposit()
```

Delay after depositing resources.

MinInteractionRange

```
public float MinInteractionRange()
```

The minimum range required to interact with a resource node.

MaxInteractionRange

```
public float MaxInteractionRange()
```

The maximum range within which the collector can interact with a resource node.

Position

```
public Vector3 Position()
```

The position of the resource collector in world space.

ClearResource

```
public void ClearResource()
```

Clears the currently collected resources from the collector.

CollectResource

```
public void CollectResource(
     IResourceNode node)
```

Collects resources from a specified resource node.

node: The resource node to collect resources from.

CanCollect

```
public bool CanCollect(
     ResourceSO resource)
```

Checks whether the collector can gather a specific type of resource.

resource: The resource to check.

Returns: Returns true if the resource can be collected, otherwise false.

DepositResource

```
public long DepositResource()
```

Deposits the currently held resources directly into the owner’s resource pool.

Returns: The remainder of resources that could not be deposited.

GoCollectResource

```
public bool GoCollectResource(
     IResourceNode node)
```

Initiates a collection task for a specified resource node.

node: The resource node to collect from.

Returns: True if the task was successfully scheduled, otherwise false.

GoDepositResource

```
public bool GoDepositResource(
     IResourceDepot depot)
```

Initiates a deposit task for a specified resource depot.

node: The resource depot to deposit to.

Returns: True if the task was successfully scheduled, otherwise false.

ResourceCollectorCapability.cs

Namespaces

UnitSystem.Collection

```
namespace UnitSystem.Collection
```

Classes

ResourceCollectorCapability

```
public struct ResourceCollectorCapability
: IResourceCollectorCapability
```

Defines the capability of a resource collector, specifying its attributes and behavior for collecting resources in the game.

Methods

CollectType

```
readonly public CollectType CollectType()
```

Capacity

```
readonly public int Capacity()
```

SupportedResources

```
readonly public ResourceSO[] SupportedResources()
```

MinRange

```
readonly public float MinRange()
```

Range

```
readonly public float Range()
```

CollectInterval

```
readonly public float CollectInterval()
```

CollectAmount

```
readonly public int CollectAmount()
```

Depot

IResourceDepot.cs

Namespaces

UnitSystem.Collection

```
namespace UnitSystem.Collection
```

Classes

IResourceDepot

```
public interface IResourceDepot
: IUnitInteracteeComponent
```

Interface representing a resource depot that interacts with resource-carrying units. Allows validation and deposit of resources into the player's resource system.

Properties

SupportedResources

```
public ResourceSO[] SupportedResources
     { get;
     }
```

Array of resources supported for depositing. If empty, all resources are accepted.

Methods

CanDepositResource

```
public bool CanDepositResource(
     ResourceSO resource)
```

Determines if a specific resource type can be deposited in this depot.

resource: The resource to check.

Returns: Returns true if the resource can be deposited, false otherwise.

DepositResources

```
public long DepositResources(
     ResourceQuantity resourceAmount)
```

Deposits a specified amount of resources into the player's resource system.

resourceAmount: The amount and type of resource to deposit.

Returns: The remaining resource amount that could not be deposited due to capacity limitations.

IResourceDepotCapability.cs

Namespaces

UnitSystem.Collection

```
namespace UnitSystem.Collection
```

Classes

IResourceDepotCapability

```
public interface IResourceDepotCapability
: IEntityCapability
```

Capability interface for resource depots, defining the types of resources they support for deposit.

ResourceDepot.cs

Namespaces

UnitSystem.Collection

```
namespace UnitSystem.Collection
```

Classes

ResourceDepot

```
[DisallowMultipleComponent]
public class ResourceDepot
: AEntityComponent, IResourceDepot,
IUnitInteractingPosition
```

Component representing a resource depot, which allows resource collection units to deposit gathered resources. Supports filtering by resource type and integrates with the player’s resource system.

Methods

Position

```
public Vector3 Position()
```

Position of the depot in the world.

SupportedResources

```
public ResourceSO[] SupportedResources()
```

CanDepositResource

```
public bool CanDepositResource(
     ResourceSO resource)
```

DepositResources

```
public long DepositResources(
     ResourceQuantity resourceAmount)
```

GetAvailableInteractionPosition

```
public virtual Vector3 GetAvailableInteractionPosition(
     IUnitInteractorComponent interactor,
     bool reserve)
```

Returns closests position around the object based on dropPoint configuration.

ReleaseInteractionPosition

```
public virtual bool ReleaseInteractionPosition(
     IUnitInteractorComponent interactor)
```

ResourceDepotCapability.cs

Namespaces

UnitSystem.Collection

```
namespace UnitSystem.Collection
```

Classes

ResourceDepotCapability

```
public struct ResourceDepotCapability
: IResourceDepotCapability
```

Implementation of the resource depot capability, specifying supported resources.

Methods

SupportedResources

```
readonly public ResourceSO[] SupportedResources()
```

Node

IResourceNode.cs

Namespaces

UnitSystem.Collection

```
namespace UnitSystem.Collection
```

Classes

IResourceNode

```
public interface IResourceNode
: IUnitInteracteeComponent
```

Interface for resource nodes, defining behaviors and properties for resource interactions.

IResourceNodeCapability.cs

Namespaces

UnitSystem.Collection

```
namespace UnitSystem.Collection
```

Classes

IResourceNodeCapability

```
public interface IResourceNodeCapability
: IEntityCapability
```

Capability interface for resource nodes, defining the resource that the node provides.

IResourceNodeFinder.cs

Namespaces

UnitSystem.Collection

```
namespace UnitSystem.Collection
```

Classes

IResourceNodeFinder

```
public interface IResourceNodeFinder
```

Methods

FindNearby

```
public IResourceNode[] FindNearby(
     Vector3 position,
     ResourceSO resource,
     float range)
```

Finds nearby resource nodes within the specified range.

position: The position from which to search for resource nodes.
resource: The specific resource type to filter by, or null to include all types.
range: The search radius in units.

Returns: An array of ResourceNode objects within the specified range, sorted by distance.

FindClosest

```
public bool FindClosest(
     Vector3 position,
     ResourceSO resource,
     float range,
     out IResourceNode node)
```

Finds the closest resource node within the specified range.

position: The position from which to search for the closest resource node.
resource: The specific resource type to filter by, or null to include all types.
range: The search radius in units.
node: The closest ResourceNode found, or null if no matching resource nodes are found.

Returns: True if a matching resource node is found, otherwise false.

ResourceNode.cs

Namespaces

UnitSystem.Collection

```
namespace UnitSystem.Collection
```

Classes

ResourceNode

```
[DisallowMultipleComponent]
public class ResourceNode
: ALimitedInteractionTarget, IResourceNode,
IUnitInteractingPosition
```

Component representing a resource node in the game world, handling resource depletion and interactions.

Variables

PredefinedPositionProvider

```
public IPredefinedInteractionPositionProvider
PredefinedPositionProvider
```

Position provider reference. This is an optional provider.

Methods

ResourceAmount

```
public ResourceQuantity ResourceAmount()
```

IsDepleted

```
public bool IsDepleted()
```

ReduceResource

```
public void ReduceResource(
     long amount)
```

GetAvailableInteractionPosition

```
public virtual Vector3 GetAvailableInteractionPosition(
     IUnitInteractorComponent interactor,
     bool reserve)
```

ReleaseInteractionPosition

```
public virtual bool ReleaseInteractionPosition(
     IUnitInteractorComponent interactor)
```

ResourceNodeCapability.cs

Namespaces

UnitSystem.Collection

```
namespace UnitSystem.Collection
```

Classes

ResourceNodeCapability

```
public struct ResourceNodeCapability
: IResourceNodeCapability
```

Implementation of the resource node capability, specifying the resource type and quantity.

Methods

Resource

```
public readonly ResourceQuantity Resource()
```

ResourceNodeFinder.cs

Namespaces

UnitSystem.Collection

```
namespace UnitSystem.Collection
```

Classes

ResourceNodeFinder

```
[System.Serializable]
public class ResourceNodeFinder
: IResourceNodeFinder
```

Utility class for scanning and finding nearby resource nodes within a specified range. Provides methods to search for all resource nodes or the closest one based on specified criteria.

Constructors

ResourceNodeFinder

```
public  ResourceNodeFinder(
     LayerMask layerMask,
     bool useNonAllocating = false,
     int allocationSize = 100)
```

Methods

FindNearby

```
public IResourceNode[] FindNearby(
     Vector3 position,
     ResourceSO resource,
     float range)
```

FindClosest

```
public bool FindClosest(
     Vector3 position,
     ResourceSO resource,
     float range,
     out IResourceNode node)
```

ResourceNodeSO.cs

Namespaces

UnitSystem.Collection

```
namespace UnitSystem.Collection
```

Classes

ResourceNodeSO

```
public class ResourceNodeSO
: AEntitySO, IProvidesEntityPrefab
```

Scriptable object defining a resource node in the game world. This class is responsible for providing and managing the associated prefab for resource nodes and ensuring required capabilities are present.

Methods

IsPrefabSet

```
public bool IsPrefabSet()
```

IsPrefabLoaded

```
public bool IsPrefabLoaded()
```

HasPrefab

```
public bool HasPrefab()
```

SetPrefab

```
public void SetPrefab(
     Entity entity)
```

GetAssociatedPrefab

```
public Entity GetAssociatedPrefab()
```

LoadAssociatedPrefabAsync

```
public Task<Entity> LoadAssociatedPrefabAsync()
```

LoadAssociatedPrefab

```
public void LoadAssociatedPrefab(
     Action<Entity> prefabLoaded)
```

SetAssociatedPrefab

```
public void SetAssociatedPrefab(
     Entity entity)
```

Task

CollectNodeResourceTask.cs

Namespaces

UnitSystem.Collection

```
namespace UnitSystem.Collection
```

Classes

CollectNodeResourceTask

```
public class CollectNodeResourceTask
: ITask
```

Represents a task for a resource collector to gather resources from a specified resource node.

Methods

CanExecuteTask

```
public bool CanExecuteTask(
     ITaskContext context,
     ITaskInput input)
```

CreateHandler

```
public ITaskHandler CreateHandler()
```

CollectNodeResourceTaskHandler

```
public class CollectNodeResourceTaskHandler
: ITaskHandler
```

Handles the logic for executing a resource collection task, including movement to the node and collecting resources based on the collector's configuration.

Methods

IsFinished

```
public bool IsFinished()
```

StartTask

```
public void StartTask(
     ITaskContext context,
     ITaskInput input)
```

UpdateTask

```
public void UpdateTask(
     float deltaTime)
```

EndTask

```
public void EndTask()
```

DepositResourceTask.cs

Namespaces

UnitSystem.Collection

```
namespace UnitSystem.Collection
```

Classes

DepositResourceTask

```
public class DepositResourceTask
: ITask
```

Represents a task for a resource collector to deposit resources into a valid resource depot.

Methods

CanExecuteTask

```
public bool CanExecuteTask(
     ITaskContext context,
     ITaskInput input)
```

CreateHandler

```
public ITaskHandler CreateHandler()
```

DepositResourceTaskHandler

```
public class DepositResourceTaskHandler
: ITaskHandler
```

Handles the logic for executing a resource deposit task, including movement to the depot and the deposit operation itself.

Constructors

DepositResourceTaskHandler

```
public  DepositResourceTaskHandler(
     float targetPositionUpdateInterval)
```

Initializes a new instance of the DepositResourceTaskHandler class.

targetPositionUpdateInterval: Interval at which the target position is updated while depositing resources. Target position is updated only if target supports movement.

Methods

IsFinished

```
public bool IsFinished()
```

StartTask

```
public void StartTask(
     ITaskContext context,
     ITaskInput input)
```

EndTask

```
public void EndTask()
```

UpdateTask

```
public void UpdateTask(
     float deltaTime)
```

ResourceAreaCollectingTask.cs

Namespaces

UnitSystem.Collection

```
namespace UnitSystem.Collection
```

ResourceCollectingTask.cs

Namespaces

UnitSystem.Collection

```
namespace UnitSystem.Collection
```

Classes

ResourceCollectingTask

```
public class ResourceCollectingTask
: ITask
```

Represents a task for managing the collection and depositing of resources by a resource collector. This task handles the logic for switching between collecting resources from nodes and depositing them at depots.

Methods

CanExecuteTask

```
public bool CanExecuteTask(
     ITaskContext context,
     ITaskInput input)
```

CreateHandler

```
public ITaskHandler CreateHandler()
```

ResourceCollectingTaskHandler

```
public class ResourceCollectingTaskHandler
: ITaskHandler
```

Manages the execution of a resource collecting task, including transitions between collecting from resource nodes and depositing resources at depots.

Constructors

ResourceCollectingTaskHandler

```
public  ResourceCollectingTaskHandler(
     ITask collectTask,
     ITask depositTask)
```

Initializes a new instance of the ResourceCollectingTaskHandler.

collectTask: The task for collecting resources from a node.
depositTask: The task for depositing resources at a depot.

Methods

IsDepositing

```
public bool IsDepositing()
```

IsCollecting

```
public bool IsCollecting()
```

TargetDepot

```
public IResourceDepot TargetDepot()
```

TargetNode

```
public IResourceNode TargetNode()
```

IsFinished

```
public bool IsFinished()
```

StartTask

```
public void StartTask(
     ITaskContext context,
     ITaskInput input)
```

UpdateTask

```
public void UpdateTask(
     float deltaTime)
```

EndTask

```
public void EndTask()
```

ResourceCollectorInput.cs

Namespaces

UnitSystem.Collection

```
namespace UnitSystem.Collection
```

Classes

ResourceCollectorInput

```
public struct ResourceCollectorInput
: ITaskInput
```

Represents input data required for a resource collection task.

Constructors

ResourceCollectorInput

```
public  ResourceCollectorInput(
     ResourceCollector collector)
```

Initializes a new instance of ResourceCollectorInput with the specified collector.

collector: The resource collector executing the task.

Variables

collector

```
public ResourceCollector collector
```

The resource collector executing the task.

Combat

CombatEvents.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Classes

CombatEvents

```
public class CombatEvents
```

Singleton class for handling combat-related events in the game. Provides event hooks for when health changes or is depleted during combat.

Constructors

CombatEvents

```
private  CombatEvents()
```

Private constructor to enforce singleton pattern.

Variables

Instance

```
public static readonly CombatEvents Instance
```

Singleton instance of the CombatEvents class.

OnHitpointDecreased

```
public UnityEvent<IHealth,AUnitComponent,int>
OnHitpointDecreased
```

Event triggered when a entity's health decreases.

OnHitpointIncreased

```
public UnityEvent<IHealth,AUnitComponent,int>
OnHitpointIncreased
```

Event triggered when a entity's health increases (e.g., from healing or regeneration).

OnHealthDepleted

```
public UnityEvent<IHealth,APlayer,AUnitComponent>
OnHealthDepleted
```

Event triggered when an entity's health is fully depleted, causing it to die or be destroyed.

CombatModule.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Classes

CombatModule

```
[DisallowMultipleComponent]
public class CombatModule
: APlayerModule
```

Represents a combat module that manages unit attacks and enemy interactions.

Variables

HealthFinder

```
public IHealthFinder HealthFinder
```

The health scanning system used by the combat module for identifying potential targets.

AttackDistanceHandler

```
[SerializeReference, HideInInspector]
public IAttackDistanceHandler AttackDistanceHandler
```

AttackTargetPositionGetter

```
[SerializeReference, HideInInspector]
public IAttackTargetPositionGetter
AttackTargetPositionGetter
```

Methods

TotalUnitsLost

```
public int TotalUnitsLost()
```

ResetAttackOnExit

```
public bool ResetAttackOnExit()
```

GetAttackPosition

```
public Vector3 GetAttackPosition(
     IUnitAttack attack,
     IHealth target)
```

IsTargetInAttackRange

```
public bool IsTargetInAttackRange(
     IUnitAttack attack,
     IHealth target)
```

FindNearestEnemy

```
public bool FindNearestEnemy(
     IUnitAttack attack,
     out IHealth health)
```

Finds the nearest enemy for the unit that is attempting to attack.

attack: The attacking unit.
health: The health of the nearest enemy found.

Returns: True if an enemy is found; otherwise, false.

FindNearestEnemyByStance

```
public bool FindNearestEnemyByStance(
     IUnitAttack attack,
     IHealth ignoringTarget,
     out IHealth health)
```

Finds the nearest enemy while considering the attack stance of the unit.

attack: The attacking unit.
ignoringTarget: A target to ignore in the search.
health: The health of the nearest enemy found.

Returns: True if an enemy is found; otherwise, false.

FindNearestEnemy

```
public bool FindNearestEnemy(
     IUnitAttack attacker,
     float range,
     out IHealth health)
```

Finds the nearest enemy within a specified range.

attacker: The unit attacking.
range: The range within which to find the enemy.
health: The health of the nearest enemy found.

Returns: True if an enemy is found; otherwise, false.

FindNearbyEnemies

```
public IHealth[] FindNearbyEnemies(
     IUnitAttack attacker,
     Vector3 position,
     float range)
```

Finds all nearby enemies within a specific range.

attacker: The unit attacking.
position: The position from which to find nearby enemies.
range: The range within which to find enemies.

Returns: An array of nearby enemies.

Attack

AttackCapability.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Classes

AttackCapability

```
[System.Serializable]
public struct AttackCapability
: IUnitCapability, IAttackCapability
```

Represents a combat capability for a unit, defining attributes such as attack range, reload time, and damage. This capability enables a unit to engage in combat and provides damage customization based on the target's type or specific entity data.

Methods

DefaultStance

```
public readonly AttackStance DefaultStance()
```

MinRange

```
public readonly float MinRange()
```

Range

```
public readonly float Range()
```

ReloadSpeed

```
public readonly float ReloadSpeed()
```

Damage

```
public readonly int Damage()
```

LineOfSight

```
public readonly float LineOfSight()
```

InvalidTargetTypes

```
public readonly UnitTypeSO[] InvalidTargetTypes()
```

GetDamage

```
public readonly int GetDamage(
     IEntity target)
```

Retrieves the damage value based on the target's entity or unit type. Falls back to the default damage if no specific configuration exists.

target: The target entity.

Returns: The damage value to apply to the target.

AttackStance.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Enumerations

AttackStance

```
public enum AttackStance{
     Aggressive,
     Defensive,
     StandGround,
     NoAttack}
```

Determines the unit behaviour managed by attack tasks.

Aggressive: Aggressive and default stance. Unit will engage any enemy within Line of Sight and follow until they are killed.

Defensive: Unit will attack enemies within range, will follow certain distance before returning to their position.

StandGround: Units do not move, but will attack any enemy in their attack range.

NoAttack: Units do not attack on their own.

IAttackCapability.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Classes

IAttackCapability

```
public interface IAttackCapability
: IEntityCapability
```

Defines the capabilities of a unit or entity for performing attacks.

IDefendPosition.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Classes

IDefendPosition

```
public interface IDefendPosition
```

Interface defining a defensive position for a unit. Allows the unit to defend a specific position and orientation with a configurable defensive range.

Methods

UpdateStance

```
public static void UpdateStance(
     IUnitAttack unitAttack,
     Vector3 position,
     Vector3 direction)
```

Checks if unit attack is in defensive stance and if it also implements IDefendPosition interface, then it updates it's position and direction. The move parameter on interface will always be false when this helper is used.

unitAttack: Attack component
position: Defend position
direction: Defend facing direction

IUnitAttack.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Classes

IUnitAttack

```
public interface IUnitAttack
: IUnitInteractorComponent
```

Defines the interface for components that manage a unit’s attack capabilities.

UnitAttack.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Classes

UnitAttack

```
[DisallowMultipleComponent]
public class UnitAttack
: AUnitComponent, IUnitAttack, IDefendPosition,
ITaskProvider
```

Represents a unit’s attack capability, providing mechanisms for managing attack tasks, reloading, and interactions with targets. Supports dynamic behaviors such as switching between attack modes and managing attack-specific settings.

Variables

OnAttack

```
public UnityEvent<UnitAttack,IHealth,int> OnAttack
```

Event triggered when the unit attacks, providing details such as damage dealt.

OnStanceChanged

```
public UnityEvent<IUnitAttack,AttackStance>
OnStanceChanged
```

Event triggered when the unit stance has changed.

Properties

ManuallyTriggerAttack

```
public bool ManuallyTriggerAttack
     { get;
     set; }
```

Whether attacks are triggered manually instead of automatically.

Methods

Stance

```
public AttackStance Stance()
```

The current stance of the unit, determining its behavior in combat.

LineOfSight

```
public float LineOfSight()
```

GetDamage

```
public int GetDamage(
     IEntity entity)
```

Retrieves the damage dealt to a specific target.

Position

```
public Vector3 Position()
```

MinInteractionRange

```
public float MinInteractionRange()
```

MaxInteractionRange

```
public float MaxInteractionRange()
```

InvalidTargetTypes

```
public UnitTypeSO[] InvalidTargetTypes()
```

HasReloaded

```
public bool HasReloaded()
```

Determines whether the unit has reloaded and is ready to attack again.

StartReloading

```
public void StartReloading()
```

Starts the reload timer, marking the beginning of the reload period.

ResetReload

```
public void ResetReload()
```

Resets the reload timer, making the unit ready to attack immediately.

DefendPosition

```
public Vector3 DefendPosition()
```

The position unit defends if AttackStance.Defensive stance is active. If stance is not active this position may be stale.

DefendDirection

```
public Vector3 DefendDirection()
```

The direction in which unit defends if AttackStance.Defensive stance is active. If stance is not active this direction may be stale.

DefensiveRange

```
public float DefensiveRange()
```

The range unit may follow when defending position in AttackStance.Defensive stance.

SetStance

```
public virtual void SetStance(
     AttackStance stance)
```

Defend

```
public virtual void Defend(
     Vector3 position,
     Vector3 direction,
     bool move)
```

Configures the unit to adopt a defensive stance and defend a specific position and direction. Optionally moves the unit to the specified position.

position: The position the unit should defend.
direction: The direction the unit should face while defending.
move: Whether the unit should move to the defensive position.

GoAttackEntity

```
public bool GoAttackEntity(
     IEntity entity)
```

Initiates an attack on a specific entity if it is a valid target.

entity: The target entity to attack.

Returns: True if the attack was successfully initiated; otherwise, false.

GoAttackEntity

```
public bool GoAttackEntity(
     IHealth health)
```

Initiates an attack on a specific health component.

health: The target health component to attack.

Returns: True if the attack was successfully initiated; otherwise, false.

Health

Health.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Classes

Health

```
[DisallowMultipleComponent]
public class Health
: AEntityComponent, IHealth
```

Handles the health system of an entity, including damage, healing, and health events. Implements invulnerability, health depletion logic, and optional destruction upon depletion. This will also manage self regeneration when enabled.

Methods

OnHitpointDecreased

```
public UnityEvent<IHealth,UnitAttack,int>
OnHitpointDecreased()
```

OnHitpointIncreased

```
public UnityEvent<IHealth,IEntity,int>
OnHitpointIncreased()
```

OnHealthDepleted

```
public UnityEvent<IHealth,UnitAttack> OnHealthDepleted()
```

OnHealthSet

```
public UnityEvent<IHealth,int> OnHealthSet()
```

IsDepleted

```
public bool IsDepleted()
```

IsInvulnerable

```
public bool IsInvulnerable()
```

Regenerates

```
public int Regenerates()
```

CurrentHealthPercentage

```
public float CurrentHealthPercentage()
```

CurrentHealth

```
public int CurrentHealth()
```

MaxHealth

```
public int MaxHealth()
```

Position

```
public Vector3 Position()
```

DestroyGameObject

```
public void DestroyGameObject()
```

Destroys the entire entity GameObject.

DestroyScript

```
public void DestroyScript()
```

Removes the health component from the entity.

SetCurrentHealth

```
public void SetCurrentHealth(
     int amount)
```

Damage

```
public void Damage(
     UnitAttack attacker,
     int damage)
```

Heal

```
public void Heal(
     IEntity entity,
     int amount)
```

HealthCapability.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Classes

HealthCapability

```
public struct HealthCapability
: IHealthCapability
```

Represents the capability of a unit or entity to have health-related behaviors.

Methods

HealthPoints

```
readonly public int HealthPoints()
```

Gets the maximum hit points for the entity.

CanDecrease

```
readonly public bool CanDecrease()
```

Determines if the entity’s health can decrease.

CanIncrease

```
readonly public bool CanIncrease()
```

Determines if the entity’s health can increase.

Regeneration

```
readonly public int Regeneration()
```

Gets the regeneration rate of the entity’s health.

HealthFinder.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Classes

HealthFinder

```
[System.Serializable]
public class HealthFinder
: IHealthFinder
```

Provides utilities for scanning health components in the game world, with methods for finding nearby and closest health components. This is a default implementation and can be simply replaced by implementing IHealthFinder.

Constructors

HealthFinder

```
public  HealthFinder(
     LayerMask layerMask,
     bool useNonAllocating = false,
     int allocationSize = 100)
```

Methods

FindNearbyHealth

```
public IHealth[] FindNearbyHealth(
     IUnitAttack attacker,
     float range)
```

FindNearbyHealth

```
public IHealth[] FindNearbyHealth(
     IUnitAttack attacker,
     Vector3 position,
     float range)
```

FindNearestHealth

```
public bool FindNearestHealth(
     IUnitAttack attacker,
     Vector3 position,
     float range,
     out IHealth closestHealth)
```

IHealth.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Classes

IHealth

```
public interface IHealth
: IUnitInteracteeComponent
```

Defines the interface for health components in the combat system.

IHealthCapability.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Classes

IHealthCapability

```
public interface IHealthCapability
: IEntityCapability
```

Defines the health-related capabilities of an entity. This includes the initial health points, whether health can be decreased or increased, and the entity's health regeneration rate.

Properties

HealthPoints

```
public int HealthPoints
     { get;
     }
```

The maximum health points the entity starts with.

CanDecrease

```
public bool CanDecrease
     { get;
     }
```

Indicates whether the entity's health can decrease (i.e., take damage).

CanIncrease

```
public bool CanIncrease
     { get;
     }
```

Indicates whether the entity's health can increase (i.e., heal or repair).

Regeneration

```
public int Regeneration
     { get;
     }
```

The health regeneration rate per tick (e.g., per second). Set to 0 if no regeneration is required. Regeneration may be applied regardless of the CanIncrease flag.

IHealthFinder.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Classes

IHealthFinder

```
public interface IHealthFinder
```

Interface defining methods for scanning and identifying health components within a specific range and conditions.

Methods

FindNearbyHealth

```
public IHealth[] FindNearbyHealth(
     IUnitAttack attacker,
     Vector3 position,
     float range)
```

Finds all nearby health components within a specified range.

attacker: The unit performing the scan.
position: The center of the scan area.
range: The maximum range to scan.
layer: The layer mask to filter targets.

Returns: An array of Health components within the range.

FindNearestHealth

```
public bool FindNearestHealth(
     IUnitAttack attacker,
     Vector3 position,
     float range,
     out IHealth closestHealth)
```

Finds the nearest health component within a specified range.

attacker: The unit performing the scan.
position: The center of the scan area.
range: The maximum range to scan.
layer: The layer mask to filter targets.
closestHealth: Outputs the closest health component found.

Returns: True if any health nearby is found; otherwise, false.

Projectile

IProjectile.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Classes

IProjectile

```
public interface IProjectile
```

Defines the basic functionality of a projectile. Implementations specify how projectiles behave upon being launched.

ProjectileLauncher.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Classes

ProjectileLauncher

```
public class ProjectileLauncher
: AUnitComponent
```

Handles launching projectiles from a unit, integrating with the UnitAttack component.

Methods

Attack

```
public UnitAttack Attack()
```

LaunchProjectile

```
public void LaunchProjectile(
     UnitAttack attacker,
     IHealth target,
     int damage)
```

DestroyProjectiles

```
public void DestroyProjectiles()
```

Cleans up all active projectiles.

Basic

BallisticMovement.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Classes

BallisticMovement

```
[System.Serializable]
public class BallisticMovement
: IProjectileMovement
```

Handles arc-based projectile movement, where the projectile reaches a peak height at the midpoint before descending toward the target. The height and speed adjust dynamically based on distance.

Methods

Start

```
public void Start(
     UnitAttack owner,
     Transform transform,
     Vector3 start,
     Vector3 target)
```

Initializes the projectile's trajectory based on the start and target positions. Here the initial rotation will also be set for the projectile.

transform: Transform of the projectile
start: The starting position of the projectile.
target: The target position the projectile will aim for.

Update

```
public void Update(
     Transform transform,
     Vector3 targetPosition,
     float deltaTime)
```

Updates the projectile's position, height, and rotation based on its trajectory progress.

transform: The transform of the projectile.
targetPosition: The current target position (can move over time).
deltaTime: The time elapsed since the last update.

UpdateSpeed

```
public void UpdateSpeed(
     float deltaTime)
```

Updates the projectile's speed based on the configured speed curve and trajectory progress.

deltaTime: The time elapsed since the last update.

HasReachedTarget

```
public bool HasReachedTarget()
```

Determines whether the projectile has reached its target.

Returns: True if the projectile has completed its trajectory, otherwise false.

BasicProjectile.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Classes

BasicProjectile

```
public class BasicProjectile
: MonoBehaviour, IProjectile
```

A basic implementation of the IProjectile interface. Handles launching, movement, and collision behavior for a projectile.

IProjectileMovement.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Classes

IProjectileMovement

```
public interface IProjectileMovement
```

LinearProjectileMovement.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Classes

LinearProjectileMovement

```
[System.Serializable]
public class LinearProjectileMovement
: IProjectileMovement
```

Handles linear movement for a projectile, ensuring it moves directly toward a target at a constant speed.

Methods

Start

```
public void Start(
     UnitAttack owner,
     Transform transform,
     Vector3 start,
     Vector3 target)
```

Update

```
public void Update(
     Transform transform,
     Vector3 targetPosition,
     float deltaTime)
```

Updates the projectile's position, moving it toward the target. Ensures the projectile doesn't overshoot the target.

transform: The transform of the projectile.
targetPosition: The current target position (if dynamic).
deltaTime: The time elapsed since the last update.

HasReachedTarget

```
public bool HasReachedTarget()
```

Checks if the projectile has reached the target.

Returns: True if the projectile is close enough to the target; otherwise, false.

StraightMovement.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Classes

StraightMovement

```
[System.Serializable]
public class StraightMovement
: IProjectileMovement
```

Direct movement with speed defined by a curve.

Methods

Start

```
public void Start(
     UnitAttack owner,
     Transform transform,
     Vector3 start,
     Vector3 target)
```

Update

```
public void Update(
     Transform transform,
     Vector3 targetPosition,
     float deltaTime)
```

HasReachedTarget

```
public bool HasReachedTarget()
```

TimedDirectMovement.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Classes

TimedDirectMovement

```
public class TimedDirectMovement
: IProjectileMovement
```

Implements projectile movement over a fixed duration, interpolating between the start and target positions. This movement type ensures the projectile reaches its destination at a consistent pace regardless of the distance to the target.

Methods

HasReachedTarget

```
public bool HasReachedTarget()
```

Start

```
public void Start(
     UnitAttack owner,
     Transform transform,
     Vector3 start,
     Vector3 target)
```

Update

```
public void Update(
     Transform transform,
     Vector3 targetPosition,
     float deltaTime)
```

Task

AAttackTargetTaskHandler.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Classes

AAttackTargetTaskHandler

```
public abstract class AAttackTargetTaskHandler
: ITaskHandler
```

Abstract base class for handling attack tasks targeting specific units or entities. Implements the ITaskHandler interface to define behavior for initiating, updating, and completing attack-related tasks.

Variables

RequestsNewTarget

```
public bool RequestsNewTarget
```

Determines if a new target should be requested when the current task is complete.

Methods

CurrentTarget

```
public IHealth CurrentTarget()
```

Gets the current attack target's Health component.

IsAttacking

```
public bool IsAttacking()
```

Indicates whether the unit is currently engaged in attacking.

IsFinished

```
public bool IsFinished()
```

Checks if the task is finished and ready for cleanup or replacement.

Returns: True if the task is complete; otherwise, false.

StartTask

```
public virtual void StartTask(
     ITaskContext context,
     ITaskInput input)
```

Initializes the attack task with the provided context and input.

context: The task context containing necessary interaction information.
input: The input data for the task, such as attack parameters.

UpdateTask

```
public virtual void UpdateTask(
     float deltaTime)
```

Updates the attack task logic each frame.

deltaTime: The time elapsed since the last frame.

EndTask

```
public virtual void EndTask()
```

Ends the attack task, cleaning up target references and resetting state if necessary.

AttackTaskInput.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Classes

AttackTaskInput

```
public readonly struct AttackTaskInput
: ITaskInput
```

Represents input data required for executing an attack task.

Constructors

AttackTaskInput

```
public  AttackTaskInput(
     UnitAttack attack)
```

Creates a new instance with the specified attack component.

attack: The UnitAttack component initiating the task.

Variables

attack

```
public readonly UnitAttack attack
```

The attack component providing data and behavior for the task.

MobileAttackTargetTask.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Classes

MobileAttackTargetTask

```
public class MobileAttackTargetTask
: ITask
```

Represents a task for attacking a specified target. This task checks if the attack can be executed based on the attacker, the target's state, and other conditions. It uses an interval to update target positions for moving targets.

Methods

CanExecuteTask

```
public bool CanExecuteTask(
     ITaskContext context,
     ITaskInput input)
```

CreateHandler

```
public ITaskHandler CreateHandler()
```

MobileAttackTargetTaskHandler

```
public class MobileAttackTargetTaskHandler
: AAttackTargetTaskHandler
```

Constructors

MobileAttackTargetTaskHandler

```
public  MobileAttackTargetTaskHandler(
     float checkInterval,
     float closerEnemyRange,
     float closerEnemyCheckInterval)
```

Methods

StartTask

```
public override void StartTask(
     ITaskContext context,
     ITaskInput input)
```

Initializes the task handler and assigns the target if possible. Marks the task as complete if the target cannot be assigned.

context: The context containing interaction details, such as the attacker and target.
input: The input data for the task, including attack-related information.

EndTask

```
public override void EndTask()
```

MoveAndAttackTask.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Classes

MoveAndAttackTask

```
public class MoveAndAttackTask
: ITask
```

A composite task that allows a unit to move towards a position and attack any encountered enemies within its sight range. If no enemies are present, the unit will proceed to the final destination.

Methods

CanExecuteTask

```
public bool CanExecuteTask(
     ITaskContext context,
     ITaskInput input)
```

CreateHandler

```
public ITaskHandler CreateHandler()
```

MoveAndAttackTaskHandler

```
public class MoveAndAttackTaskHandler
: ITaskHandler
```

Handles the movement and attack behavior for the MoveAndAttackTask. This handler transitions between states: moving to a destination, engaging enemies within range, and resuming movement if no enemies are present.

Constructors

MoveAndAttackTaskHandler

```
public  MoveAndAttackTaskHandler()
```

Initializes a new instance of the MoveAndAttackTaskHandler class.

Methods

IsFinished

```
public bool IsFinished()
```

StartTask

```
public void StartTask(
     ITaskContext context,
     ITaskInput input)
```

Begins the task, setting up the movement to the target position.

context: The task context containing the target position.
input: The task input containing attack information.

UpdateTask

```
public void UpdateTask(
     float deltaTime)
```

Updates the task, transitioning between states based on movement and combat conditions.

deltaTime: The time elapsed since the last update.

EndTask

```
public void EndTask()
```

Cleans up task-related components and stops movement.

StationaryAttackTargetTask.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Classes

StationaryAttackTargetTask

```
public class StationaryAttackTargetTask
: ITask
```

Represents a task for a stationary unit to attack a specified target. This task ensures the target is within the attack range and validates conditions such as the attacker’s state and the target’s health.

Methods

CanExecuteTask

```
public bool CanExecuteTask(
     ITaskContext context,
     ITaskInput input)
```

CreateHandler

```
public ITaskHandler CreateHandler()
```

StationaryAttackTargetTaskHandler

```
public class StationaryAttackTargetTaskHandler
: AAttackTargetTaskHandler
```

Handles the execution of a stationary attack task. This handler ensures the target remains within the attack range and performs attacks at appropriate intervals.

Constructors

StationaryAttackTargetTaskHandler

```
public  StationaryAttackTargetTaskHandler(
     float distanceValidationCheckInterval)
```

Methods

StartTask

```
public override void StartTask(
     ITaskContext context,
     ITaskInput input)
```

Initializes the task handler and assigns the target if possible. Marks the task as complete if the target cannot be assigned.

context: The context containing interaction details, such as the attacker and target.
input: The input data for the task, including attack-related information.

Utility

IAttackDistanceHandler.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Classes

IAttackDistanceHandler

```
public interface IAttackDistanceHandler
```

Contract for checking distance between attacker and his target and determine if attacker is within the attack range.

Methods

IsTargetInAttackRange

```
public bool IsTargetInAttackRange(
     IUnitAttack attack,
     Vector3 attackPosition,
     IHealth target)
```

Checks if the attacker is within the attack range.

attack: Attacker to check for
attackPosition: Attack position itself, from where attacker must attack.
target: Target of the attacker

Returns: Returns true if attacker is within attack range to the attack the target; Otherwise false.

UnitAttackDistanceHandler

```
public class UnitAttackDistanceHandler
: IAttackDistanceHandler
```

Default implementation for checking distance for attack. Uses attackers position relative to it’s attack position.

Methods

IsTargetInAttackRange

```
public bool IsTargetInAttackRange(
     IUnitAttack attack,
     Vector3 attackPosition,
     IHealth target)
```

IAttackTargetPositionGetter.cs

Namespaces

UnitSystem.Combat

```
namespace UnitSystem.Combat
```

Classes

IAttackTargetPositionGetter

```
public interface IAttackTargetPositionGetter
```

Contract for finding the most optimal position to attack, based on current placement of attacker/target.

Methods

GetAttackPosition

```
public Vector3 GetAttackPosition(
     IUnitAttack attacker,
     IHealth target)
```

Gets the closest position near the target for the attacker to position itself before performing an attack.

UnitAttackTargetPositionGetter

```
public class UnitAttackTargetPositionGetter
: IAttackTargetPositionGetter
```

Default implementation of IAttackTargetPositionGetter that uses collider on the object to get the closest point on it. If this position is used directly, with no modifications, make sure stopping distance is set to prevent attempting to walk into the collider.

Methods

GetAttackPosition

```
public Vector3 GetAttackPosition(
     IUnitAttack attacker,
     IHealth target)
```

Garrison

GarrisonEvents.cs

Namespaces

UnitSystem.Garrison

```
namespace UnitSystem.Garrison
```

Classes

GarrisonEvents

```
public class GarrisonEvents
```

Manages events related to garrisons.

Variables

OnUnitEnterGarrison

```
[Tooltip("Event invoked when a unit enters the garrison.")]
public UnityEvent<IGarrisonEntity,IUnit>
OnUnitEnterGarrison
```

OnUnitExitGarrison

```
[Tooltip("Event invoked when a unit exits the garrison.")]
public UnityEvent<IGarrisonEntity,IUnit> OnUnitExitGarrison
```

Properties

Instance

```
public static GarrisonEvents Instance
     { get;
     private set; }
```

GarrisonModule.cs

Namespaces

UnitSystem.Garrison

```
namespace UnitSystem.Garrison
```

Classes

GarrisonModule

```
[DisallowMultipleComponent]
public class GarrisonModule
: APlayerModule
```

Manages garrisonable units and garrison structures for a player. Tracks all available garrisons and units that can enter them, providing functions to manage garrison interactions.

Variables

OnGarrisonListChanged

```
public UnityEvent OnGarrisonListChanged
```

Event triggered when the list of garrisons changes (addition/removal).

OnGarrisonableListChanged

```
public UnityEvent OnGarrisonableListChanged
```

Event triggered when the list of garrisonable units changes (addition/removal).

Methods

GarrisonUnits

```
public IGarrisonEntity[] GarrisonUnits()
```

Gets an array of all garrison units currently managed by this module.

GarrisonableUnits

```
public IGarrisonableUnit[] GarrisonableUnits()
```

Gets an array of all garrisonable units currently managed by this module.

OnDisable

```
public void OnDisable()
```

ReleaseGarrisonedUnits

```
public void ReleaseGarrisonedUnits(
     IGarrisonEntity garrison)
```

Releases all units currently inside the specified garrison.

garrison: The garrison to be emptied.

EmptyAllGarrisons

```
public void EmptyAllGarrisons()
```

Empties all garrisons owned by the player, releasing their units.

CallInAllGarrisons

```
public virtual void CallInAllGarrisons(
     float range,
     bool cancelActiveTasks)
```

Calls all garrisons to bring in nearby garrisonable units within a specified range.

range: The maximum distance from the garrison to consider units.
cancelActiveTasks: Whether to cancel active tasks before garrisoning units. Otherwise those with active tasks are ignored.

CallInNearbyUnits

```
public virtual void CallInNearbyUnits(
     IGarrisonEntity garrison,
     float range,
     bool cancelActiveTasks)
```

Calls in nearby garrisonable units to enter a specific garrison.

garrison: The target garrison.
range: The maximum distance from the garrison to consider units.
cancelActiveTasks: Whether to cancel active tasks before garrisoning units. Otherwise those with active tasks are ignored.

Garrison Unit

GarrisonEntity.cs

Namespaces

UnitSystem.Garrison

```
namespace UnitSystem.Garrison
```

Classes

GarrisonEntity

```
[DisallowMultipleComponent]
public class GarrisonEntity
: AEntityComponent, IGarrisonEntity,
IUnitInteractingPosition
```

Garrison component designed for units that can accommodate other units within them. This component manages the behavior, events, and capacity for units entering and exiting the garrison. Requires the AEntitySO.capabilities to include GarrisonUnitsCapability or a custom IGarrisonUnitsCapability implementation.

Methods

GarrisonedUnits

```
public List<Unit> GarrisonedUnits()
```

Read-only list of units currently in the garrison.

Position

```
public Vector3 Position()
```

The position of the garrison in world space.

IsActive

```
public override bool IsActive()
```

Specifies whether the garrison is active and operational.

GarrisonCapacity

```
public int GarrisonCapacity()
```

Capacity of the garrison, as defined by its capability.

GetAvailableInteractionPosition

```
public Vector3 GetAvailableInteractionPosition(
     IUnitInteractorComponent _,
     bool reserve)
```

ReleaseInteractionPosition

```
public virtual bool ReleaseInteractionPosition(
     IUnitInteractorComponent interactor)
```

IsEligibleToEnter

```
public virtual bool IsEligibleToEnter(
     Unit unit)
```

Determines if a given unit is eligible to enter the garrison.

unit: The unit to check.

Returns: True if the unit is eligible to enter, false otherwise.

AddUnit

```
public virtual bool AddUnit(
     Unit unit)
```

Adds a unit to the garrison if there is available capacity.

unit: The unit to add.

Returns: True if the unit was successfully added, false otherwise.

RemoveUnit

```
public virtual bool RemoveUnit(
     Unit unit)
```

Removes a unit from the garrison.

unit: The unit to remove.

Returns: True if the unit was successfully removed, false otherwise.

RemoveAllUnits

```
public void RemoveAllUnits()
```

Removes all units currently in the garrison.

GarrisonUnitsCapability.cs

Namespaces

UnitSystem.Garrison

```
namespace UnitSystem.Garrison
```

Classes

GarrisonUnitsCapability

```
public struct GarrisonUnitsCapability
: IGarrisonUnitsCapability
```

Specifies the capacity of this unit to garrison other units and the permissions for it.

Methods

Capacity

```
public readonly int Capacity()
```

IsEligibleToEnter

```
public readonly bool IsEligibleToEnter(
     AUnitSO unit)
```

Determines if a specific unit is eligible to enter the garrison.

unit: The unit to check.

Returns: Returns true if the unit is eligible to enter, false otherwise.

IGarrisonEntity.cs

Namespaces

UnitSystem.Garrison

```
namespace UnitSystem.Garrison
```

Classes

IGarrisonEntity

```
public interface IGarrisonEntity
: IUnitInteracteeComponent
```

Interface used for entities that have capability to garrison units.

IGarrisonUnitsCapability.cs

Namespaces

UnitSystem.Garrison

```
namespace UnitSystem.Garrison
```

Classes

IGarrisonUnitsCapability

```
public interface IGarrisonUnitsCapability
: IEntityCapability
```

Defines the capability for a unit to act as a garrison, allowing other units to enter and be stored within it.

Garrisonable Unit

GarrisonableUnit.cs

Namespaces

UnitSystem.Garrison

```
namespace UnitSystem.Garrison
```

Classes

GarrisonableUnit

```
[System.Serializable]
[DisallowMultipleComponent]
public class GarrisonableUnit
: AUnitComponent, IGarrisonableUnit, ITaskProvider
```

Implements behavior for units that can interact with and enter garrison units.

Methods

IsGarrisoned

```
public bool IsGarrisoned()
```

Position

```
public Vector3 Position()
```

MinInteractionRange

```
public float MinInteractionRange()
```

MaxInteractionRange

```
public float MaxInteractionRange()
```

SetMinRange

```
public void SetMinRange(
     float minRange)
```

Sets the minimum range for garrison interaction.

minRange: The minimum range.

SetMaxRange

```
public void SetMaxRange(
     float maxRange)
```

Sets the maximum range for garrison interaction.

maxRange: The maximum range.

GoEnterGarrison

```
public bool GoEnterGarrison(
     IGarrisonEntity garrison)
```

EnterGarrison

```
public virtual void EnterGarrison()
```

ExitGarrison

```
public virtual void ExitGarrison()
```

GarrisonableUnitCapability.cs

Namespaces

UnitSystem.Garrison

```
namespace UnitSystem.Garrison
```

Classes

GarrisonableUnitCapability

```
public struct GarrisonableUnitCapability
: IGarrisonableUnitCapability
```

Adds the capability for a unit to enter a garrison unit. Configures the unit with garrison-related behaviors and settings.

IGarrisonableUnit.cs

Namespaces

UnitSystem.Garrison

```
namespace UnitSystem.Garrison
```

Classes

IGarrisonableUnit

```
public interface IGarrisonableUnit
: IUnitInteractorComponent
```

Defines behavior for units that can enter and exit garrisons.

IGarrisonableUnitCapability.cs

Namespaces

UnitSystem.Garrison

```
namespace UnitSystem.Garrison
```

Classes

IGarrisonableUnitCapability

```
public interface IGarrisonableUnitCapability
: IUnitCapability
```

Defines a capability for a unit to enter a garrison. This interface is implemented by data structures that specify garrison-related settings for a unit. Behavior is managed by components such as AUnitComponent subclasses.

Task

EnterGarrisonInput.cs

Namespaces

UnitSystem.Garrison

```
namespace UnitSystem.Garrison
```

Classes

EnterGarrisonInput

```
public struct EnterGarrisonInput
: ITaskInput
```

Represents the input required for executing the EnterGarrisonTask.

Constructors

EnterGarrisonInput

```
public  EnterGarrisonInput(
     IGarrisonableUnit garrisonableUnit)
```

Initializes a new instance of the EnterGarrisonInput struct.

garrisonableUnit: The garrisonable unit providing task input.

Variables

garrisonableUnit

```
public IGarrisonableUnit garrisonableUnit
```

The unit that is attempting to hide in a garrison.

EnterGarrisonTask.cs

Namespaces

UnitSystem.Garrison

```
namespace UnitSystem.Garrison
```

Classes

EnterGarrisonTask

```
[Serializable]
public class EnterGarrisonTask
: ITask
```

Task for directing a unit to enter a garrison. The task ensures that both the unit and the garrison meet eligibility requirements before execution.

Methods

CanExecuteTask

```
public bool CanExecuteTask(
     ITaskContext context,
     ITaskInput input)
```

CreateHandler

```
public ITaskHandler CreateHandler()
```

EnterGarrisonTaskHandler.cs

Namespaces

UnitSystem.Garrison

```
namespace UnitSystem.Garrison
```

Classes

EnterGarrisonTaskHandler

```
public class EnterGarrisonTaskHandler
: ITaskHandler
```

Handles the execution of the EnterGarrisonTask, directing the unit to move to the garrison and enter it.

Constructors

EnterGarrisonTaskHandler

```
public  EnterGarrisonTaskHandler(
     float targetPositionUpdateInterval = 0.5f)
```

Methods

CurrentTarget

```
public IGarrisonEntity CurrentTarget()
```

The current garrison target that the unit should approach.

IsFinished

```
public bool IsFinished()
```

StartTask

```
public void StartTask(
     ITaskContext context,
     ITaskInput input)
```

UpdateTask

```
public void UpdateTask(
     float deltaTime)
```

EndTask

```
public void EndTask()
```

Limitation

UnitLimit.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

UnitLimit

```
[System.Serializable]
public struct UnitLimit
```

Represents a unit type and its associated limit.

Variables

unit

```
[Tooltip("The unit type for which the limit applies.")]
public AUnitSO unit
```

limit

```
[Tooltip("The maximum allowable count for the specified
unit type.")]
[PositiveInt]
public int limit
```

UnitLimitAttributeSO.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

UnitLimitAttributeSO

```
[CreateAssetMenu(fileName = "New unit limit attribute", menuName
= "Unit System/Unit Limit Attribute")]
public class UnitLimitAttributeSO
: ProductionAttributeSO
```

A ScriptableObject representing a unit limit attribute. This associates a specific unit type with its production or operational limits.

Variables

unit

```
public AUnitSO unit
```

The unit type associated with this limit attribute.

UnitLimitationModule.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

UnitLimitationModule

```
public class UnitLimitationModule
: APlayerModule
```

Manages the operational and production limits for units within a player’s system. Tracks unit counts and enforces both global and faction-specific limitations.

Variables

OnUnitCountChange

```
public UnityEvent OnUnitCountChange
```

Event invoked when the unit count changes.

Methods

HasReachedLimit

```
public bool HasReachedLimit(
     int unitId)
```

Checks if a specific unit type has reached its limit.

unitId: The ID of the unit type.

Returns: True if the limit is reached; otherwise, false.

Population

PopulationModule.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

PopulationModule

```
[DisallowMultipleComponent]
public class PopulationModule
: APlayerModule
```

A module responsible for managing the population system of a player in the game. Population is a common mechanic in RTS games used to control the number of units a player can spawn. This module handles attributes such as population capacity and population consumption for units and other producibles.

If the player does not need to track or limit the population, only the populationConsumptionAttribute can be used for tracking purposes. If population can increase dynamically (e.g., through certain buildings or upgrades), the populationCapacityAttribute should also be set to enable automatic handling.

Variables

OnPopulationUpdate

```
[Header("Events")]
public UnityEvent<int,int> OnPopulationUpdate
```

Event invoked whenever the population count or maximum population changes. Parameters: - Current population - Maximum population

Methods

MaxPopulationEnabled

```
public bool MaxPopulationEnabled()
```

Indicates whether population is capped at the maximum allowed population.

MaxPopulation

```
public int MaxPopulation()
```

Gets the maximum population allowed, accounting for any hard cap restrictions.

CurrentPopulation

```
public int CurrentPopulation()
```

Gets the current population count.

SetPopulationHardCap

```
public void SetPopulationHardCap(
     int hardCapacity)
```

Sets a hard cap for the population. This ensures the maximum population cannot exceed the specified limit.

hardCapacity: The new hard cap value.

SetMaxPopulation

```
public void SetMaxPopulation(
     int population)
```

Sets the maximum population to a specified value. Only positive values will be applied.

population: The new maximum population value.

IncreaseMaxPopulation

```
public void IncreaseMaxPopulation(
     int population)
```

Increases the maximum population.

AddPopulation

```
public void AddPopulation(
     int population)
```

Adds population amount to the current population. Only positive values are applied, others ignored.

population: Population amount to add.

HasPopulationCapacity

```
public bool HasPopulationCapacity(
     ProducibleQuantity producibleQuantity)
```

Determines whether a producible can be added to the player's population without exceeding the maximum population capacity.

producibleQuantity: The producible and its quantity to evaluate.

Returns: Returns true if the producible can be added without exceeding the maximum population capacity; otherwise, false. If population limits are not enabled or the producible does not have a population consumption attribute, this method returns true.

HasPopulationCapacity

```
public bool HasPopulationCapacity(
     int populationConsumption)
```

Determines whether a population can be added to the player's population without exceeding the maximum population capacity.

populationConsumption: The population amount to evaluate.

Returns: Returns true if the producible can be added without exceeding the maximum population capacity; otherwise, false. If population limits are not enabled or the producible does not have a population consumption attribute, this method returns true.

Production

ProductionModule.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

ProductionModule

```
[DisallowMultipleComponent]
public class ProductionModule
: APlayerModule, IActiveProductionDelegate
```

Module for managing production processes associated with a player. Tracks and updates components implementing IProduce on player units.

Variables

OnStartPlacementRequest

```
[Tooltip("This event will be invoked to manage the
placement of a production object.")]
public UnityEvent<PlacementRequiredInfo>
OnStartPlacementRequest
```

If production is requested for an entity or unit with building capabilities, they should be placed in world space instead of produced by unit itself. This event will be invoked to manage the placement of a production object.

Properties

ProduceManually

```
public bool ProduceManually
     { get;
     set; }
```

Enables or disables manual production. If enabled production will run in realtime using tickInterval.

Methods

ShouldFinishProductionFor

```
public bool ShouldFinishProductionFor(
     ProducibleQuantity producibleQuantity)
```

Checks if the producible can finish production by validating population consumption, if requirements of the producible are still fulfilled and for resource producible it also checks if storage will be full.

producibleQuantity: Producible with quantity to check

Returns: true if production may finish; otherwise false.

Produce

```
public void Produce(
     float delta)
```

Applies production updates to all active production components.

delta: The time or turn increment to apply to all production processes.

RequestProduction

```
public void RequestProduction(
     ProducibleQuantity productionQuantity,
     IEntity entity)
```

SetProductionRequestHandler

```
public void SetProductionRequestHandler(
     IProductionRequestHandler customHandler)
```

Active

ActiveProduction.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

IActiveProductionDelegate

```
public interface IActiveProductionDelegate
: IProductionQueueDelegate
```

Delegate that allows implementation to prevent production of certain units when they reached the very end of production.

Example: This may be used if some resources like population needs to be above 0 before spawning/producing a unit is possible.

ActiveProduction

```
[DisallowMultipleComponent]
public class ActiveProduction
: AEntityComponent, IActiveProduction,
IProductionQueueDelegate
```

Feature rich component for active production of any AProducibleSO. Handles queue of production requests, supports cancelation, garrison produced units, spawning produced units, collect right away or stash produced items before player collects.

Variables

GarrisonUnitsAfterProduction

```
[Tooltip("Set this to true if unit should garrison
produced units. " +
"Which means that they will need to be
manually spawned." +
"Garrison maximal capacity is still used.")]
public bool GarrisonUnitsAfterProduction
```

Specifies behaviour for produced units. If this is set to ‘true‘ then IGarrisonEntity (if present) will be used to garrison units after production.

Properties

Delegate

```
public IActiveProductionDelegate Delegate
     { get;
     set; }
```

Current production delegate.

Methods

ProducedStash

```
public ProducibleQuantity[] ProducedStash()
```

Currently stashed producibles.

CurrentProductionProgress

```
public float CurrentProductionProgress()
```

CollectedProducedStash

```
public ProducibleQuantity[] CollectedProducedStash()
```

Clears produced stash and returns it. This method does not apply the produced stash to player itself.

CollectFromProducedStash

```
public bool CollectFromProducedStash(
     int index,
     out ProducibleQuantity collected)
```

Collects stashed production on specified index.

index: Index on which to collect from
collected: Collected producible quantity

Returns: Returns true if collecting was successful.

CollectFromProducedStash

```
public bool CollectFromProducedStash(
     AProducibleSO requested,
     out ProducibleQuantity collected)
```

Collects stashed production for requested producible.

requested: Producible to collect
collected: Collected producible quantity

Returns: Returns true if collecting was successful.

CollectFromProducedStash

```
public bool CollectFromProducedStash(
     ProducibleQuantity requested,
     out ProducibleQuantity collected)
```

Collects stashed production for requested producible and it's quantity. Collecting will be successful even if it was not collected in full.

requested: Requested producible and it's quantity
collected: Collected producible quantity

Returns: Returns true if collecting was successful. If requested quantity was 5 but produced stash was 4, it will retrieve the 4 and output it.

StartProduction

```
public void StartProduction(
     ProducibleQuantity productionQuantity,
     bool queueMultipleOrders = false)
```

Queues production order on the unit and if set, splits it into multiple production orders.

productionQuantity: Production quantity to be queued
queueMultipleOrders: Split production quantity into multiple orders

StartProduction

```
public void StartProduction(
     AProducibleSO producible,
     long quantity,
     bool queueMultipleOrders = false)
```

Queues the production of producible with quantity or splits it into multiple productions.

producible: Producible to be produced
quantity: Number of producibles
queueMultipleOrders: Split production quantity into multiple orders

CancelProduction

```
public ResourceQuantity[] CancelProduction()
```

Convenience method to cancel all production orders. Specific production can be cancelled on ProductionQueue directly.

CancelProductionOrder

```
public long CancelProductionOrder(
     AProducibleSO producible)
```

Cancels the first production with specified producible.

producible: Producible to cancel production for.

Returns: Returns quantity of the producible order cancelled.

ShouldFinishProductionFor

```
public bool ShouldFinishProductionFor(
     ProducibleQuantity producibleQuantity)
```

Produce

```
public void Produce(
     float delta)
```

Updates production time on production queue. This can only produce while unit is operational.

delta: Amount of progression for the queue

IsProducing

```
public bool IsProducing(
     AProducibleSO producible)
```

GetProductionQueue

```
public ProducibleQuantity[] GetProductionQueue()
```

ActiveProductionCapability.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

ActiveProductionCapability

```
public struct ActiveProductionCapability
: IActiveProductionCapability
```

Implementation of the IActiveProductionCapability interface, defining the production actions a unit can perform.

Methods

ProductionActions

```
public readonly ProductionAction[] ProductionActions()
```

DefaultProductionRequestHandler.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

DefaultProductionRequestHandler

```
public class DefaultProductionRequestHandler
: IProductionRequestHandler
```

Default implementation for handling production requests, including resource checks, unit limitations, and production initiation.

Constructors

DefaultProductionRequestHandler

```
public  DefaultProductionRequestHandler(
     APlayer player)
```

Methods

HandleProductionRequest

```
public void HandleProductionRequest(
     ProducibleQuantity productionQuantity,
     IEntity entity,
     Action<PlacementRequiredInfo> placementRequired)
```

IActiveProduction.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

IActiveProduction

```
public interface IActiveProduction
: IEntityComponent, IProduce
```

Defines the behavior for unit production in a system. Extends IProduce to provide methods and properties for managing, tracking, and delegating production processes.

IActiveProductionCapability.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

IActiveProductionCapability

```
public interface IActiveProductionCapability
: IEntityCapability
```

Defines a capability for units to actively produce producibles on demand.

IProductionRequestHandler.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

PlacementRequiredInfo

```
public struct PlacementRequiredInfo
```

Contains the necessary information for placing an entity in the game.

Constructors

PlacementRequiredInfo

```
public  PlacementRequiredInfo(
     APlayer owner,
     IEntity prefab,
     long quantity)
```

Initializes a new instance of the PlacementRequiredInfo struct.

owner: The owner of the entity.
prefab: The prefab of the entity.
quantity: The number of entities to be placed.

Variables

owner

```
public APlayer owner
```

The player who owns the entity being placed.

prefab

```
public IEntity prefab
```

The entity prefab to be placed.

quantity

```
public long quantity
```

The quantity of the entity to be placed.

IProductionRequestHandler

```
public interface IProductionRequestHandler
```

Defines a contract for handling production requests within the production system.

Methods

HandleProductionRequest

```
public void HandleProductionRequest(
     ProducibleQuantity productionQuantity,
     IEntity entity,
     Action<PlacementRequiredInfo> placementRequired)
```

Handles the production request by verifying limits, checking and consuming resources, initiating production or placement, etc.

productionQuantity: The production quantity.
entity: The entity requesting production.
placementRequired: Action callback for placement when required.

ProductionEvents.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

ProductionEvents

```
public class ProductionEvents
```

Singleton class responsible for handling high-level production-related events in the system. This class acts as a centralized point for raising and observing production events, which can be utilized by various systems or modules. For example, you can use these events to notify modules, UI components, or other listeners when a production process is completed.

Variables

Instance

```
public static readonly ProductionEvents Instance
```

Singleton instance of the ProductionEvents class. Use this instance to subscribe to or invoke production events.

OnNewProductionScheduled

```
public readonly
UnityEvent<ActiveProduction,AProducibleSO,long>
OnNewProductionScheduled
```

Event invoked when a new production is scheduled on a unit-

OnProductionCancelled

```
public readonly
UnityEvent<ActiveProduction,AProducibleSO,long>
OnProductionCancelled
```

Event invoked when specific production order is cancelled on the unit.

OnAllProductionCancelled

```
public readonly UnityEvent<ActiveProduction>
OnAllProductionCancelled
```

Event invoked when all productions have been cancelled on the unit.

OnProductionFinished

```
public readonly
UnityEvent<ActiveProduction,ProducibleQuantity>
OnProductionFinished
```

Event invoked when production order has been finished.

ProductionQueue.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

IProductionQueueDelegate

```
public interface IProductionQueueDelegate
```

Delegate that allows implementation to prevent production of certain units when they reached the very end of production.

Example: This may be used if some resources like population needs to be above 0 before spawning/producing a unit is possible.

ProductionQueue

```
public class ProductionQueue
```

Class used for producing items from a list, one by one. First item in queue is producing, the rest wait. It supports starting, queueing or canceling production.

Variables

OnProductionFinished

```
public Action<ProducibleQuantity> OnProductionFinished
```

Action invoked once a producible finishes production.

Delegate

```
public IProductionQueueDelegate Delegate
```

Optional delegate that allows pausing of ongoing production.

Properties

CurrentProductionProgress

```
public float CurrentProductionProgress
     { private set;
     get; }
```

Returns range from 0 to 1 when an item is in production. When there is no production -1 is returned.

Methods

Queue

```
public ProducibleQuantity[] Queue()
```

Collection of the current production items. First one in queue is currently in production.

Produce

```
public void Produce(
     float delta)
```

Applies delta changes to current production queue. Only a single item can be produced at a time (first one in queue). If delta is larger than the remaining time of the current production, remainder is applied to the next production item (if exists).

delta: Value applied to the production time. This can be either an actual time or time period value.

IsProducing

```
public bool IsProducing(
     AProducibleSO producible)
```

Checks if the production queue contains this producible. Matching is done with ManagedSO.ID.

producible: Producible used for matching.

Returns: Returns 'true' only if this item is in production.

AddProductionOrder

```
public void AddProductionOrder(
     AProducibleSO producible,
     long quantity,
     bool queueMultipleOrders = false)
```

Adds a producible and its quantity to the end of production queue.

producible: Producible to be added.
quantity: Quantity to be produced.
queueMultipleOrders: If quantity should be split into multiple orders, diving by quantity 1.

CancelProductionOrder

```
public long CancelProductionOrder(
     AProducibleSO producible)
```

Cancels the first producible that matches the parameter.

producible: Producible that will be canceled if present in queue.

Returns: Returns quantity of the producible order cancelled.

CancelProductionOrder

```
public long CancelProductionOrder(
     int producibleID)
```

Cancel the first producible with ID that matches the argument.

producibleID: ID that will be used for finding a matching producible.

Returns: Returns quantity of the producible order cancelled.

CancelProductionOrderIndex

```
public long CancelProductionOrderIndex(
     int index)
```

Cancels the producible on specified index.

index: Index on which the producible should be canceled.

Returns: Returns quantity of the producible order cancelled.

CancelProduction

```
public ResourceQuantity[] CancelProduction()
```

Cancels all production orders in the queue. Returns all the accumulating resources freed duo to cancalled productions.

Core

AProducibleSO.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

AProducibleSO

```
public abstract class AProducibleSO
: ManagedSO
```

Base producible object, primarily used to define producibles for the project. Any gameObject/unit should use this class or derive from this class to utilize UnitSystem components.

All producibles should originate from this scriptable object and should primarily differentiate between others using the ID. You may either subclass this to define new types (e.g., structures) or add a new field like ‘Type‘ with an enum to differentiate between such units (e.g., workers vs. structures).

Variables

Name

```
[Header("General")]
public string Name
```

Specifies the name of the producible.

Description

```
[TextArea(1, 5)]
public string Description
```

Specifies the description of the producible.

Sprite

```
public Sprite Sprite
```

Specifies the sprite of the producible.

ProductionAttributes

```
public ProductionAttributeValue[] ProductionAttributes
```

Specifies production attributes of the producible.

Methods

TryGetAttribute

```
public bool TryGetAttribute(
     int attributeID,
     out ProductionAttributeValue value)
```

Searches for a valid attribute with a value on this producible.

attributeID: ID of the attribute.
value: Value returned if found.

Returns: Returns true if the attribute was found; otherwise, false.

IProduce.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

IProduce

```
public interface IProduce
```

Interface for managing production progress using a delta value, which can represent real-time updates (e.g., Time.deltaTime) or turn-based increments.

ProducibleQuantity.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

ProducibleQuantity

```
[Serializable]
public struct ProducibleQuantity
```

Representation of producible quantities with float.

Constructors

ProducibleQuantity

```
public  ProducibleQuantity(
     AProducibleSO producible,
     long quantity)
```

Variables

Producible

```
[Tooltip("Specifies the producible.")]
public AProducibleSO Producible
```

Specifies the producible.

Quantity

```
[Tooltip("Specifies the quantity of the producible.")]
[PositiveLong]
public long Quantity
```

Specifies quantity of the producible.

ProductionAction.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

ProductionAction

```
[System.Serializable]
public struct ProductionAction
: IEntityUIAction
```

Represents an action that can be performed to produce a specified quantity of a producible item. Typically used for binding production actions to user input (e.g., keyboard shortcuts).

Methods

KeyCode

```
public readonly KeyCode KeyCode()
```

Gets the key associated with this production action.

ProducibleQuantity

```
public readonly ProducibleQuantity ProducibleQuantity()
```

Gets the producible item and quantity associated with this action.

Execute

```
public readonly void Execute(
     IEntity entity)
```

Attribute

ProductionAttributeSO.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

ProductionAttributeSO

```
[CreateAssetMenu(fileName = "New attribute", menuName =
"Unit System/Attribute")]
public class ProductionAttributeSO
: ManagedSO
```

Defines a base production attribute, such as "Population" or "Unit Limit."

Variables

Name

```
public string Name
```

The name of the attribute (e.g., "Population" or "Unit Limit").

Description

```
[TextArea(2, 5)]
public string Description
```

A description of the attribute, generally visible to the player.

ProductionAttributeValue.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

ProductionAttributeValue

```
[Serializable]
public struct ProductionAttributeValue
```

Represents an attribute and its associated value.

Variables

Attribute

```
public ProductionAttributeSO Attribute
```

Specifies the attribute associated with this value.

Value

```
public float Value
```

The numeric value of the attribute.

Passive

IPassiveResourceProductionCapability.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

IPassiveResourceProductionCapability

```
public interface IPassiveResourceProductionCapability
: IEntityCapability
```

Defines passive resource production capability, used for constant and passive resource production without manually queuing them.

Properties

ProducesResource

```
public ResourceQuantity[] ProducesResource
     { get;
     }
```

Resources this unit produces and their quantity per second.

DepositResourceQuantity

```
public long DepositResourceQuantity
     { get;
     }
```

Resource quantity at which the produced resource will be deposited to the players storage.

PassiveResourceProduction.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

PassiveResourceProduction

```
[DisallowMultipleComponent]
public class PassiveResourceProduction
: AEntityComponent, IProduce
```

Unit component for producing resources passively. Resources to produce are retrieved from units Entity.data and its capability defined by IPassiveResourceProductionCapability interface.

PassiveResourceProductionCapability.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

PassiveResourceProductionCapability

```
public struct PassiveResourceProductionCapability
: IPassiveResourceProductionCapability
```

Methods

ProducesResource

```
public readonly ResourceQuantity[] ProducesResource()
```

Specifies the resources this unit produces per game defined period. Multiple resources are supported.

DepositResourceQuantity

```
public readonly long DepositResourceQuantity()
```

Specifies the resource amount at which it should be deposited to the player's resources storage.

Research

ResearchModule.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

ResearchModule

```
[DisallowMultipleComponent]
public class ResearchModule
: APlayerModule
```

Simple research managment module, responsible for keeping track of currently completed researches for a player. These are generally less complex than for resource management.

Variables

OnResearchFinished

```
[Header("Events")]
public UnityEvent<ResearchSO> OnResearchFinished
```

Event invoked when research is added to the collection via AddFinishedResearch(ResearchSO).

Methods

AddFinishedResearch

```
public void AddFinishedResearch(
     ResearchSO research)
```

Adds new research to the completedResearches collection and invokes the OnResearchFinished event.

research: Research to be added.

IsResearchComplete

```
public bool IsResearchComplete(
     ResearchSO research)
```

Checks if the completedResearches collection contains a research with the same UUID.

research: Research used for matching UUIDs.

Returns: Returns ’true’ if collection contains research.

RemoveCompletedResearch

```
public bool RemoveCompletedResearch(
     ResearchSO research)
```

Removes research from completedResearches collection if it contains one. Matching is done with UUIDs.

research: Research to be removed.

Returns: Returns ’true’ when research is removed. Returns ’false’ if collection does not contain the research and therefore could not be removed. Only produced/completed are present.

ResearchSO.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

ResearchSO

```
[CreateAssetMenu(fileName = "New research", menuName = "Unit
System/Research")]
public class ResearchSO
: AProducibleSO
```

Specifies a default research scriptable object. Derived from AProducibleSO without additional fields.

Resource

ResourceModule.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

ResourceModule

```
[DisallowMultipleComponent]
public class ResourceModule
: APlayerModule
```

Resource management component responsible for tracking current resources and their capacities for a player. Provides methods for modifying resources.

Variables

UnlimitedResources

```
[Tooltip("Specifies if the resource capacity is ignored.")]
public bool UnlimitedResources
```

If true, resource capacity is ignored, allowing "unlimited" storage up to long.MaxValue.

UseUniqueResourceCapacities

```
public bool UseUniqueResourceCapacities
```

When false, all resources share a single common capacity defined by CommonResourceCapacity. When true, each resource can have its unique capacity.

OnResourceFull

```
[Header("Events")]
public UnityEvent<ResourceSO> OnResourceFull
```

Event invoked when a resource reaches its capacity.

OnResourceUpdate

```
public UnityEvent<ResourceSO> OnResourceUpdate
```

Event invoked when a resource’s quantity changes.

OnResourceCapacityUpdate

```
public UnityEvent<ResourceSO> OnResourceCapacityUpdate
```

Event invoked when a resource’s capacity changes.

Methods

CommonResourceCapacity

```
public long CommonResourceCapacity()
```

Specifies the default resource capacity. For all resources specified in resourceCapacities, this value is ignored.

AddResources

```
public ResourceQuantity[] AddResources(
     ResourceQuantity[] resources)
```

Modifies existing resource quantity or adds a new one if none exists yet.

AddResource

```
public long AddResource(
     ResourceQuantity resourceQuantity)
```

Adds a specific quantity of a resource. If capacity is reached, returns the remainder.

RemoveResource

```
public bool RemoveResource(
     ResourceQuantity resourceQuantity)
```

Removes the provided resource quantity from existing resource if there is enough of it. Also invokes OnResourceUpdate if resource was modified.

resourceQuantity: Resource to remove from module.

Returns: Returns 'false' if there is not enough resource available.

RemoveResources

```
public bool RemoveResources(
     ResourceQuantity[] resourceQuantities)
```

Removes the provided resources quantity from existing resources if there is enough resources available. Also invokes OnResourceUpdate for each resource that was modified.

resourceQuantity: Resource to remove from module.

Returns: Returns 'false' if there is not enough resources available.

RemoveAllResources

```
public void RemoveAllResources()
```

Removes all present resources.

ModifyResourceCapacity

```
public void ModifyResourceCapacity(
     ResourceQuantity resourceCapacity)
```

Applies changes to current resource capacity. If there is no existing capacity specifies for this resource resource, then this capacity will be set as initial value.

resourceCapacity: Resource capacity to modify with.

ModifyCommonResourceCapacity

```
public void ModifyCommonResourceCapacity(
     long capacity)
```

Modifies CommonResourceCapacity.

capacity: Capacity to add

SetCommonResourceCapacity

```
public void SetCommonResourceCapacity(
     long newCapacity)
```

Set the CommonResourceCapacity.

newCapacity: New capacity

SetResourceCapacity

```
public void SetResourceCapacity(
     ResourceQuantity resourceQuantity)
```

Setting resource capacity by overriding existing one or adding a new one if one does not exist.

resourceQuantity: New resource capacity.

SetResourceQuantity

```
public void SetResourceQuantity(
     ResourceQuantity resourceQuantity)
```

Setting current resource quantity by overriding existing one, or adding a new one if one does not exist.

resourceQuantity: Overriding resource quantity.

HasEnoughStorage

```
public bool HasEnoughStorage(
     AProducibleSO resource,
     long quantity)
```

Compares the quantity of the currently available resource.

resource: Resource used for matching
quantity: Quantity used for comparison

Returns: Returns ’false’ if there is not enough resource available.

HasEnoughResource

```
public bool HasEnoughResource(
     ResourceQuantity resourceQuantity)
```

Compares the quantity of the currently available resource.

resourceQuantity: Resource and its quantity used for matching.

Returns: Returns ’false’ if there is not enough resource available.

HasEnoughResources

```
public bool HasEnoughResources(
     ResourceQuantity[] resources)
```

Checks for quantity of the resources available.

resources: Resources and its quantities used for matching.

Returns: Returns ’false’ if there is not enough resources available.

GetResourceCapacity

```
public long GetResourceCapacity(
     int resourceID)
```

Retrieves the current capacity for a specific resource.

resourceID: ID used for matching with resourceCapacities

Returns: Returns the resource capacity allowed for this resource module.

GetResourceQuantity

```
public long GetResourceQuantity(
     int resourceID)
```

Returns the current resource quantity for the provided resource ID.

resourceID: ID used for matching the resource

Returns: Returns the current resource quantity for the given ID.

IsResourceCapacityReached

```
public bool IsResourceCapacityReached(
     int resourceID)
```

Checks if the resource capacity is reached for the requested resource

resourceID: Resource ID to check.

GetResourcesQuantityForLimitedResources

```
public long GetResourcesQuantityForLimitedResources()
```

Calculates the sum of all resources that are not unlimited in capacity. This is generally useful when these resources share a common storage.

Returns: Returns sum of all limited resources.

GetResources

```
public ResourceQuantity[] GetResources()
```

Current resources.

This quantity can be set below 0 and will generally behave as it would with 0. This cannot happen with taking resources, but only with SetResourceQuantity(ResourceQuantity) method if this is ever desired.

GetResourceCapacity

```
public ResourceQuantity[] GetResourceCapacity()
```

Current resources capacity. If resource is not specified here, default value is used (CommonResourceCapacity) for storage capacity.

ResourceQuantity.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

ResourceQuantity

```
[System.Serializable]
public struct ResourceQuantity
```

Represents a specific quantity of a resource.

Constructors

ResourceQuantity

```
public  ResourceQuantity(
     ResourceSO resource,
     long quantity)
```

Initializes a new instance of the ResourceQuantity struct.

resource: The type of resource.
quantity: The quantity of the resource.

Variables

EmptyArray

```
public static readonly ResourceQuantity[] EmptyArray
```

A reusable, immutable empty array of resource quantities. Use this to avoid allocating new empty arrays in scenarios where an empty array must be returned by design.

Quantity

```
[PositiveLong]
public long Quantity
```

The quantity of the specified resource.

Methods

GetFullCost

```
public readonly ResourceQuantity[] GetFullCost()
```

Computes the full cost of the resource based on its base cost and the current quantity.

Returns: An array of ResourceQuantity representing the total cost for the current quantity.

GetFullCost

```
public readonly ResourceQuantity[] GetFullCost(
     long quantity)
```

Computes the full cost of the resource based on its base cost and a specified quantity.

quantity: The specified quantity to calculate the cost for.

Returns: An array of ResourceQuantity representing the total cost for the specified quantity.

GetFullCost

```
public static ResourceQuantity[] GetFullCost(
     ResourceQuantity[] cost,
     long quantity)
```

Computes the full cost of a set of resources based on a specified quantity.

cost: The base cost as an array of ResourceQuantity.
quantity: The specified quantity to calculate the cost for.

Returns: An array of ResourceQuantity representing the total cost for the specified quantity.

ResourceSO.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

ResourceSO

```
[CreateAssetMenu(fileName = "New resource", menuName = "Unit
System/Resource")]
public class ResourceSO
: AProducibleSO
```

Specifies a resource scriptable object asset. Derived from AProducibleSO without additional fields.

Navigation

BuildableEntityNavMeshManager.cs

Namespaces

UnitSystem.Navigation

```
namespace UnitSystem.Navigation
```

Classes

BuildableEntityNavMeshManager

```
[RequireComponent(typeof(EntityBuilding))]
public class BuildableEntityNavMeshManager
: MonoBehaviour
```

Manages the activation and deactivation of Unity’s NavMesh components (NavMeshAgent and NavMeshObstacle) for buildings during their construction lifecycle.

GroundValidationManager.cs

Namespaces

UnitSystem.Navigation

```
namespace UnitSystem.Navigation
```

Classes

IValidateWorldPositions

```
public interface IValidateWorldPositions
```

Interface for validation world positions.

Methods

ValidatePosition

```
public bool ValidatePosition(
     Vector3 position,
     out Vector3 validPosition)
```

Validates position, ensuring its on valid ground.

position: World position to validate
validPosition: Validated position

Returns: Returns true if position was validated; otherwise false.

GroundValidationManager

```
[ExecuteAlways]
public class GroundValidationManager
: MonoBehaviour
```

Manages ground validation, ensuring objects have valid world positions.

Methods

Get

```
public static GroundValidationManager Get()
```

GetOrCreate

```
public static GroundValidationManager GetOrCreate()
```

Gets or creates a singleton instance of the GroundValidationManager.

Returns: The singleton instance.

HasValidator

```
public bool HasValidator()
```

Gets whether a validator is set.

SetValidator

```
public void SetValidator(
     IValidateWorldPositions validator)
```

Sets the validator for position validation.

validator: The validator to set.

SetValidator

```
public void SetValidator(
     GameObject gameObject)
```

Sets the validator using a GameObject reference.

gameObject: The GameObject containing the validator component.

ClearValidator

```
public void ClearValidator()
```

Clears the currently assigned validator.

ValidatePositions

```
public Vector3[] ValidatePositions(
     Vector3[] positions)
```

Validates an array of world positions using the assigned validator.

positions: The positions to validate.

Returns: The validated positions.

ValidatePosition

```
public bool ValidatePosition(
     Vector3 position,
     out Vector3 validPosition)
```

Validates an array of world positions using the assigned validator.

position: Position to validate
validPosition: Validated position

Returns: Returns true if position was validated; otherwise false.

NavMeshPositionValidator.cs

Namespaces

UnitSystem.Navigation

```
namespace UnitSystem.Navigation
```

Classes

NavMeshPositionValidator

```
[ExecuteAlways]
public class NavMeshPositionValidator
: MonoBehaviour, IValidateWorldPositions
```

Validates world positions against the NavMesh to ensure they are within navigable areas.

Methods

ValidatePositions

```
public Vector3[] ValidatePositions(
     Vector3[] positions)
```

Validates an array of world positions, ensuring they are within valid NavMesh areas.

positions: The array of positions to validate.

Returns: An array of validated positions that conform to the NavMesh constraints.

ValidatePosition

```
public bool ValidatePosition(
     Vector3 position,
     out Vector3 validPosition)
```

Validates a world position, ensuring it is within a valid NavMesh area.

position: Position to validate
validPosition: Validated position

Returns: Returns true if position was validated; otherwise false.

Player

APlayer.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

APlayer

```
public abstract class APlayer
: MonoBehaviour
```

Defines the main player component, representing a player in the game. This component manages the player's units, entities, modules, and producibles.

Variables

OnUnitRemoved

```
public UnityEvent<Unit> OnUnitRemoved
```

Event invoked when a unit is removed from the player's collection.

OnEntityAdded

```
public UnityEvent<Entity> OnEntityAdded
```

Event invoked when a entity is added to the player's collection.

OnEntityRemoved

```
public UnityEvent<Entity> OnEntityRemoved
```

Event invoked when a entity is removed from the player's collection.

OnRegisterProducible

```
public UnityEvent<AProducibleSO,long> OnRegisterProducible
```

Event invoked when a producible is added to the player.

OnUnregisterProducible

```
public UnityEvent<AProducibleSO> OnUnregisterProducible
```

Event invoked when a producible is added to the player.

Properties

Faction

```
public AFactionSO Faction
     { get;
     set; }
```

Gets or sets the player's faction.

Methods

Modules

```
public APlayerModule[] Modules()
```

Gets all modules attached to the player.

GetUnits

```
public Unit[] GetUnits()
```

Retrieves all units currently owned by the player.

GetEntities

```
public Entity[] GetEntities()
```

Retrieves all entities currently owned by the player.

ArePlayersAllied

```
public abstract bool ArePlayersAllied(
     APlayer other)
```

Checks if the players are allies.

other: Player to check.

Returns: Returns true if players are allies; otherwise false.

AddModule

```
public void AddModule(
     APlayerModule module)
```

Adds a module to the player.

module: The module to be added.

CreateModule

```
public Module CreateModule()
```

Creates a new module and adds it to the player.

Returns: Returns created module.

CreateModule

```
public APlayerModule CreateModule(
     Type type)
```

Creates a new module and adds it to the player.

type: Type of the module to create. This type must be a subclass of APlayerModule

Returns: Returns created module.

RemoveModule

```
public void RemoveModule(
     APlayerModule module)
```

Removes a specific module from the player.

module: The module to be removed.

DestroyModule

```
public void DestroyModule()
```

Removes all modules of the specified type from the player.

GetModule

```
public Module GetModule()
```

Retrieves a module of the specified type from the player.

Returns: The first module of the specified type, or null if no such module exists.

TryGetModule

```
public bool TryGetModule(
     out Module behaviour)
```

Attempts to retrieve a module of the specified type from the player.

behaviour: The retrieved module, or null if no such module exists.

Returns: true if a module of the specified type is found; otherwise, false.

AddEntity

```
public virtual void AddEntity(
     Entity entity,
     bool register)
```

Adds an entity to the player by assigning ownership and managing related attributes or producibles based on the entity type.

entity: The entity to be added.
register: Specifies whether to register producible for this entity.

RemoveEntity

```
public virtual void RemoveEntity(
     Entity entity,
     bool unregister)
```

Removes an entity from the player by revoking ownership and managing related attributes or producibles based on the entity type.

entity: The entity to be removed.
unregister: Specifies whether to unregister producible for this entity.

AddUnit

```
public virtual void AddUnit(
     Unit unit,
     bool registersProducible)
```

Adds unit to the player by assigning ownership and storing the reference into Units collection. This allows quick access and matching for players units.

unit: New unit that will to be added to the player.
registersProducible: Determines if the unit should also be registered (invoking register event) for modules to respond to.

RemoveUnit

```
public virtual void RemoveUnit(
     Unit unit,
     bool unregistersProducible)
```

Removes unit from the player by removing ownership and removing the unit from Units collection.

unit: Unit to be removed, which will no longer be under ownership of it’s current player.
unregistersProducible: Determines if the unit should also be unregistered (invoking unregister event) for modules to respond to.

RefundPlayer

```
public abstract void RefundPlayer(
     Unit unit)
```

Refunds the player for the destruction of the unit. This is generally used for when player destroys the unit manually.

unit: Unit for which to refund cost.

AddResource

```
public virtual long AddResource(
     ResourceQuantity resourceQuantity)
```

Adds a producible of resource type with the specified quantity to the player. If the player has a resource module, the resource will be added; otherwise, the full quantity remains unadded.

resourceQuantity: The resource and quantity to add.

Returns: Returns the remaining quantity if not all resources could be added/deposited; otherwise, returns 0.

RegisterProducible

```
public virtual void RegisterProducible(
     AProducibleSO producible)
```

Registers a producible to the player with a default quantity of 1.

producible: The producible to add.

RegisterProducible

```
public virtual void RegisterProducible(
     AProducibleSO producible,
     long quantity)
```

Registers a producible to the player with the specified quantity.

producible: The producible to add.
quantity: The quantity of the producible to add.

UnregisterProducible

```
public virtual void UnregisterProducible(
     AProducibleSO producible)
```

Removes a registered producible from the player.

producible: The producible to remove.

HasRegisteredProducible

```
public abstract bool HasRegisteredProducible(
     ProducibleQuantity producibleQuantity)
```

Checks if the player has registered the specified producible in the required quantity.

producibleQuantity: The producible and quantity to check for.

Returns: Returns true if the player has registered the producible in at least the specified quantity; otherwise, false.

FulfillsRequirements

```
public virtual bool FulfillsRequirements(
     AProducibleSO producible,
     long quantity)
```

Checks if the requirements for a producible are fullfilled, by checking if the player contains all the requirements defined on producible with the help of existing method HasRegisteredProducible(ProducibleQuantity). To customise this override the method.

producible: Producible with requirements
quantity: Quantity of the producible, this will be multipled requirements quantity.

Returns: If there are no requirements, or they are fullfilled TRUE is returned, otherwise FALSE.

APlayerModule.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

APlayerModule

```
public abstract class APlayerModule
: MonoBehaviour
```

Represents a base script for player modules, providing a flexible framework for adding custom functionality to players.

Methods

IsEnabled

```
public bool IsEnabled()
```

Indicates whether the module is currently enabled.

ActivePlayersManager.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

ActivePlayersManager

```
public class ActivePlayersManager
: MonoBehaviour
```

Manages the active players in the game, ensuring a centralized reference for adding, removing, and tracking players during gameplay.

Variables

OnPlayersChange

```
public UnityEvent<APlayer[]> OnPlayersChange
```

Invoked whenever the list of active players changes.

Methods

GetActivePlayers

```
public APlayer[] GetActivePlayers()
```

Gets the array of active players. Do not modify this, it will create unexpected behaviour.

ActivePlayerCount

```
public int ActivePlayerCount()
```

Gets the number of active players.

GetOrCreate

```
public static ActivePlayersManager GetOrCreate()
```

Retrieves the singleton instance of ActivePlayersManager, creating a new one if it does not already exist.

Returns: The singleton instance of ActivePlayersManager.

AddPlayer

```
public void AddPlayer(
     APlayer newPlayer)
```

Adds a player to the active players list.

newPlayer: The player to add.

RemovePlayer

```
public void RemovePlayer(
     APlayer player)
```

Removes a player from the active players list.

player: The player to remove.

Player.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

Player

```
public class Player
: APlayer
```

Default implementation of APlayer. Provides integration with standard player modules, such as resource management, population control, and research. Allows for managing player units, resources, and production.

This implementation can be replaced with a custom version by deriving from APlayer.

Methods

ResourceModule

```
public ResourceModule ResourceModule()
```

Gets the player’s resource management module.

ArePlayersAllied

```
public override bool ArePlayersAllied(
     APlayer other)
```

RefundPlayer

```
public override void RefundPlayer(
     Unit unit)
```

AddResource

```
public override long AddResource(
     ResourceQuantity resourceQuantity)
```

HasRegisteredProducible

```
public override bool HasRegisteredProducible(
     ProducibleQuantity producibleQuantity)
```

PlayersRelationshipManager.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Enumerations

RelationshipState

```
[System.Serializable]
public enum RelationshipState{
     Neutral,
     Ally,
     Hostile,}
```

Defines possible states between players.

Neutral: Default state with no hostility or alliance.

Ally: Players are friendly and cooperative.

Hostile: Players are enemies and can attack each other.

Classes

PlayersRelationshipManager

```
[DisallowMultipleComponent]
public class PlayersRelationshipManager
: MonoBehaviour
```

Manages relationships between players.

Methods

GetOrCreate

```
public static PlayersRelationshipManager GetOrCreate()
```

Retrieves the singleton instance of PlayersRelationshipManager, creating a new one if it does not already exist.

Returns: The singleton instance of PlayersRelationshipManager.

SetRelationship

```
public void SetRelationship(
     APlayer p1,
     APlayer p2,
     RelationshipState state)
```

Sets the relationship state between two players. If a relationship already exists, it is updated.

p1: The first player.
p2: The second player.
state: The new relationship state.

GetRelationship

```
public RelationshipState GetRelationship(
     APlayer p1,
     APlayer p2)
```

Gets the relationship state between two players. If no relationship is found, defaults to defaultState.

p1: The first player.
p2: The second player.

Returns: The current relationship state.

AreAllied

```
public bool AreAllied(
     APlayer p1,
     APlayer p2)
```

Determines whether two players are allies.

p1: The first player.
p2: The second player.

Returns: True if the players are allied; otherwise, false.

AreEnemies

```
public bool AreEnemies(
     APlayer p1,
     APlayer p2)
```

Determines whether two players are enemies.

p1: The first player.
p2: The second player.

Returns: True if the players are enemies; otherwise, false.

Task

ITask.cs

Namespaces

UnitSystem.Task

```
namespace UnitSystem.Task
```

Classes

ITask

```
public interface ITask
```

Represents the contract for a task, providing methods to validate execution and create handlers for task logic.

Methods

CanExecuteTask

```
public abstract bool CanExecuteTask(
     ITaskContext context,
     ITaskInput input)
```

Determines whether the task can be executed in the specified context with the given input.

context: The context in which the task will be executed.
input: The input data required for task execution.

Returns: True if the task can be executed, otherwise false.

CreateHandler

```
public abstract ITaskHandler CreateHandler()
```

Creates a handler to manage the task's execution process.

Returns: An instance of ITaskHandler for task execution.

ITaskContext.cs

Namespaces

UnitSystem.Task

```
namespace UnitSystem.Task
```

Classes

ITaskContext

```
public interface ITaskContext
```

Represents the context for executing a task. Provides contextual information needed for task validation and execution.

ITaskHandler.cs

Namespaces

UnitSystem.Task

```
namespace UnitSystem.Task
```

Classes

ITaskHandler

```
public interface ITaskHandler
```

Handles the execution lifecycle of a task, including starting, updating, and completing the task.

ITaskInput.cs

Namespaces

UnitSystem.Task

```
namespace UnitSystem.Task
```

Classes

ITaskInput

```
public interface ITaskInput
```

Represents the input data required for a task's execution. Extend this interface to define specific input requirements for tasks.

ITaskOwner.cs

Namespaces

UnitSystem.Task

```
namespace UnitSystem.Task
```

Classes

ITaskOwner

```
public interface ITaskOwner
```

Defines a contract for components that manage ownership of a single active task. This interface is typically implemented by components attached to game objects to track and manage tasks assigned to them. TaskSystem will update component that implements this interface with the currently active task handler for its game object.

ITaskProvider.cs

Namespaces

UnitSystem.Task

```
namespace UnitSystem.Task
```

Classes

ITaskProvider

```
public interface ITaskProvider
```

Interface for validating and executing tasks within a specific context.

PositionContext.cs

Namespaces

UnitSystem.Task

```
namespace UnitSystem.Task
```

Classes

PositionContext

```
public class PositionContext
: ITaskContext
```

A task context for tasks that require a target position for execution, such as movement or positioning tasks.

Constructors

PositionContext

```
public  PositionContext(
     Vector3 position)
```

Initializes a new instance of the PositionContext class with the specified target position.

position: The target position for the task.

Properties

TargetPosition

```
public Vector3 TargetPosition
     { get;
     }
```

Gets the target position for the task.

TaskHelper.cs

Namespaces

UnitSystem.Task

```
namespace UnitSystem.Task
```

Classes

TaskHelper

```
public static class TaskHelper
```

Provides helper methods for evaluating and running tasks based on a given context and input.

Methods

CanRunTask

```
public static bool CanRunTask(
     ITaskContext context,
     ITaskInput input,
     ITask[] tasks,
     out ITask taskToRun)
```

Determines whether any task in the provided array can be executed based on the given context and input.

context: The task context for evaluation. This typically specifies the situation in which the task might run.
input: The task input, containing data and configurations for the task.
tasks: An array of tasks to evaluate.
taskToRun: Outputs the first task from the array that can be executed, or null if no task is valid.

Returns: true if a valid task is found and assigned to ; otherwise, false.

TaskSystem.cs

Namespaces

UnitSystem.Task

```
namespace UnitSystem.Task
```

Classes

TaskParameters

```
public struct TaskParameters
```

Represents parameters for a task, including its context and input.

Variables

Context

```
public ITaskContext Context
```

The context in which the task is executed.

Input

```
public ITaskInput Input
```

The input required for the task.

TaskSystem

```
public class TaskSystem
: MonoBehaviour
```

Manages the lifecycle of tasks for various objects, including queuing, execution, and cleanup.

Classes

QueuedTask

```
public struct QueuedTask
```

Represents a queued task with its parameters and handler.

Constructors

QueuedTask

```
public  QueuedTask(
     TaskParameters param,
     ITaskHandler handler)
```

Initializes a new instance of the QueuedTask struct.

param: The parameters for the task.
handler: The handler to execute the task.

Variables

param

```
public TaskParameters param
```

The parameters associated with the task.

handler

```
public ITaskHandler handler
```

The handler responsible for executing the task.

Methods

GetOrCreate

```
public static TaskSystem GetOrCreate()
```

Retrieves or creates the singleton instance of the TaskSystem.

Returns: The singleton TaskSystem instance.

TryGetOrCreate

```
public static bool TryGetOrCreate(
     out TaskSystem system)
```

Attempts to retrieve or create the singleton TaskSystem.

system: The retrieved or created TaskSystem instance.

Returns: True if the instance was successfully retrieved or created; otherwise, false.

ScheduleTask

```
public void ScheduleTask(
     GameObject gameObject,
     ITaskContext context,
     ITaskInput input,
     ITask task)
```

Adds a task to the system for the specified GameObject.

gameObject: The owner of the task.
context: The context for the task.
input: The input data for the task.
task: The task to add.

RemoveTasks

```
public void RemoveTasks(
     GameObject owner)
```

Removes all tasks associated with the specified owner.

owner: The owner whose tasks will be removed.

GetActiveTask

```
public ITaskHandler GetActiveTask(
     GameObject owner)
```

Gets the active task for the specified owner.

owner: The owner whose active task is retrieved.

Returns: The active task handler, or null if none is active.

GetPendingTasks

```
public QueuedTask[] GetPendingTasks(
     GameObject owner)
```

Gets all pending tasks for the specified owner.

owner: The owner whose pending tasks are retrieved.

Returns: An array of pending tasks.

HasActiveTask

```
public bool HasActiveTask(
     GameObject owner)
```

Determines whether the specified owner has an active task.

owner: The owner to check.

Returns: True if the owner has an active task, otherwise false.

UnitInteractionContext.cs

Namespaces

UnitSystem.Task

```
namespace UnitSystem.Task
```

Classes

UnitInteractionContext

```
public class UnitInteractionContext
: ITaskContext
```

A task context for unit interactions that require a specific interactee target.

Constructors

UnitInteractionContext

```
public  UnitInteractionContext(
     IUnitInteracteeComponent target,
     bool commanded = false)
```

Initializes a new instance of the UnitInteractionContext class with the specified interaction target.

target: The target of the interaction task.
commanded: If the player/owner has commanded the movement. Some tasks might behave differently based on this.

Properties

Target

```
public IUnitInteracteeComponent Target
     { get;
     }
```

Gets the target of the interaction context.

Commanded

```
public bool Commanded
     { get;
     }
```

Specifies if the interaction was commanded by the owner.

UI

AEntityActionUIElement.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

AEntityActionUIElement

```
public abstract class AEntityActionUIElement
: AEntityUIElement
```

Represents a UI element capable of performing an action.

Methods

SetAction

```
public abstract void SetAction(
     IEntityUIAction action)
```

Assigns an action to this UI element.

action: The action to be executed when interacted with.

CheckActionHotKey

```
public abstract bool CheckActionHotKey()
```

Checks if the assigned action can be triggered via a hotkey.

Returns: True if a valid hotkey is pressed, otherwise false.

IEntityUIAction

```
public interface IEntityUIAction
```

Defines an interface for an actionable entity UI element.

AEntityUIElement.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

AEntityUIElement

```
public abstract class AEntityUIElement
: MonoBehaviour
```

Represents a base class for all UI elements related to an entity.

Methods

Configure

```
public abstract void Configure(
     IEntity entity)
```

Configures the UI element with the provided entity data.

entity: The entity whose information will be displayed.

IEntityUIHandler.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

EntityUILayoutData

```
public struct EntityUILayoutData
```

Data for the entity UI layout handler.

Variables

SectionIndex

```
public int SectionIndex
```

Section of the container.

Container

```
public Transform Container
```

Reference to the layout root element / container.

IEntityUIHandler

```
public interface IEntityUIHandler
```

Defines an interface for handling UI elements associated with an entity.

Unit

AUnitComponent.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

AUnitComponent

```
[System.Serializable]
public abstract class AUnitComponent
: AEntityComponent, IUnitComponent
```

Abstract base class for components associated with units.

Methods

Unit

```
public Unit Unit()
```

AUnitSO.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

AUnitSO

```
public abstract class AUnitSO
: AEntitySO
```

Core scriptable object for units, representing producibles that are spawned into the Scene.

Methods

DoesMatchAnyType

```
public bool DoesMatchAnyType(
     UnitTypeSO[] types)
```

Determines if the unit matches any of the specified types.

types: The types to check against.

Returns: Returns true if the unit matches any of the specified types; otherwise, false.

CreatePrefab

```
public override GameObject CreatePrefab(
     string name)
```

Creates a prefab instance of this unit with the specified name.

name: The name of the prefab instance.

Returns: The created prefab game object.

IUnit.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

IUnit

```
public interface IUnit
: IEntity
```

Interface representing core unit functionality.

Properties

InteractionMovement

```
public IMoveUnitForInteraction InteractionMovement
     { get;
     }
```

Contract for movement to interact. This does not itself implement the movement so it depends on Movement heavily. By default this

Movement

```
public IUnitMovement Movement
     { get;
     }
```

Contract for unit movement.

GarrisonableUnit

```
public IGarrisonableUnit GarrisonableUnit
     { get;
     }
```

Contract for garrisonable unit which allows this unit to enter a garrison.

Builder

```
public IUnitBuilder Builder
     { get;
     }
```

Contract for unit which supports building other units.

UnitAttack

```
public IUnitAttack UnitAttack
     { get;
     }
```

Contract for unit attacks.

IUnitCapability.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

IUnitCapability

```
public interface IUnitCapability
: IEntityCapability
```

Represents a capability specific to units, extending the base entity capability.

IUnitComponent.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

IUnitComponent

```
public interface IUnitComponent
: IEntityComponent
```

Interface for components associated with a unit.

Unit.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

Unit

```
[DisallowMultipleComponent]
public class Unit
: Entity, IUnit, ITaskOwner
```

Core component representing a unit in the game.

Variables

OnActiveTaskChange

```
public UnityEvent<ITaskHandler,ITaskHandler> OnActiveTaskChange
```

Event invoked when the active task assigned to the unit changes.

Properties

ActiveTask

```
public ITaskHandler ActiveTask
     { set;
     get; }
```

Gets or sets the active task handler for this unit.

InteractionMovement

```
public IMoveUnitForInteraction InteractionMovement
     { private set;
     get; }
```

This reference allows you to fully control where unit is positioned for interaction. This way you can improve obstacle avoidance, walk away, etc. This is where position and direction for interaction is computed and then passed to Movement behaviour.

Movement

```
public IUnitMovement Movement
     { private set;
     get; }
```

GarrisonableUnit

```
public IGarrisonableUnit GarrisonableUnit
     { private set;
     get; }
```

Builder

```
public IUnitBuilder Builder
     { private set;
     get; }
```

UnitAttack

```
public IUnitAttack UnitAttack
     { private set;
     get; }
```

Methods

AUnitSO Data

```
public new AUnitSO Data()
```

OnInitialize

```
public override void OnInitialize()
```

DestroyUnit

```
public virtual void DestroyUnit(
     bool refundPlayer = false,
     float delay = 0f)
```

Destroys the unit, optionally refunding the owner and applying a delay before destruction.

refundPlayer: Whether the owner should be refunded for this unit.
delay: The delay in seconds before the unit is destroyed.

UnitSO.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

UnitSO

```
[CreateAssetMenu(fileName = "New unit", menuName = "Unit
System/Units/Unit")]
public class UnitSO
: AUnitSO, IProvidesEntityPrefab
```

Represents a unit definition in the game, deriving from AUnitSO. This scriptable object provides a basic prefab-based implementation for spawning units as game objects within the scene. Customizations can be achieved by subclassing AUnitSO.

Methods

IsPrefabSet

```
public bool IsPrefabSet()
```

IsPrefabLoaded

```
public bool IsPrefabLoaded()
```

GetAssociatedPrefab

```
public Entity GetAssociatedPrefab()
```

LoadAssociatedPrefabAsync

```
public Task<Entity> LoadAssociatedPrefabAsync()
```

LoadAssociatedPrefab

```
public void LoadAssociatedPrefab(
     Action<Entity> prefabLoaded)
```

SetAssociatedPrefab

```
public void SetAssociatedPrefab(
     Entity entity)
```

UnitTypeSO.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

UnitTypeSO

```
[CreateAssetMenu(fileName = "New Unit Type", menuName = "Unit
System/Unit Type")]
public class UnitTypeSO
: ManagedSO
```

Represents a unit type that can be assigned to one or more AUnitSO. Unit types are used to define classifications and rules for units, such as permissions for specific features or interactions.

For example, unit types can restrict which units can garrison in a Garrison.GarrisonEntity or define unit behavior based on their type.

Methods

TypeName

```
public string TypeName()
```

Gets the name of the unit type.

Description

```
public string Description()
```

Gets a brief description of the unit type.

DoesMatchAnyType

```
public static bool DoesMatchAnyType(
     UnitTypeSO[] typesA,
     UnitTypeSO[] typesB)
```

Control

AUnitMovement.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

AUnitMovement

```
public abstract class AUnitMovement
: AUnitComponent, IUnitMovement
```

Abstract class that defines the basic contract for unit movement behavior. By implementing IUnitMovement it provides methods and properties that define how a unit can move, rotate, and respond to commands in the game world.

Properties

IsMoving

```
public abstract bool IsMoving
     { get;
     }
```

Destination

```
public abstract Vector3 Destination
     { get;
     }
```

Methods

GetControlPosition

```
public Vector3 GetControlPosition()
```

Returns the move target destination.

SetControlPosition

```
public virtual void SetControlPosition(
     Vector3 groundPosition)
```

HasReachedDestination

```
public abstract bool HasReachedDestination()
```

SetDestination

```
public virtual void SetDestination(
     Vector3 groundPosition)
```

Sets a new destination for the unit, optionally validating the ground position. Clears any previously set target direction before setting the new destination.

groundPosition: The target position to set as the destination.

SetDestinationAndDirection

```
public virtual void SetDestinationAndDirection(
     Vector3 position,
     Vector3 direction)
```

StopInCurrentPosition

```
public abstract void StopInCurrentPosition()
```

IsFacingTargetDirection

```
public abstract bool IsFacingTargetDirection()
```

SetTargetDirection

```
public abstract void SetTargetDirection(
     Vector3 direction)
```

Sets the target direction for the unit.

direction: The target direction.

BaseUnitMovement.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

BaseUnitMovement

```
public abstract class BaseUnitMovement
: AUnitMovement
```

Abstract base class for unit movement behavior that provides functionality for rotation and direction handling. Intended for extension by specific movement implementations.

Methods

IsFacingTargetDirection

```
public override bool IsFacingTargetDirection()
```

Checks if the unit is currently facing a specific target direction. This method indicates whether the unit is in the process of rotating towards its designated target direction.

Returns: True if the unit is facing the target direction; otherwise, false.

SetTargetDirection

```
public override void SetTargetDirection(
     Vector3 direction)
```

RotationUtils

```
public static class RotationUtils
```

Methods

IsLookingAt

```
public static bool IsLookingAt(
     Transform transform,
     Vector3 targetDirection,
     Vector3 rotationThreshold,
     bool useYAxisOnly = false)
```

IControlEntity.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

IControlEntity

```
public interface IControlEntity
```

Interface for unit's main control which sets the target position of an entity. Designed to accommodate both movable and non-movable entities, such as stationary units with target positions for spawn points or moving units to a new position controlled by the player. Control contract is intended to be invoked by player's interactions and not designed to be used for regular unit movement controls.

IUnitMovement.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

IUnitMovement

```
public interface IUnitMovement
: IUnitComponent, IControlEntity
```

Interface defining movement behavior for units within the UnitSystem. Enables interaction with unit positioning, movement, and rotation.

Properties

Destination

```
public Vector3 Destination
     { get;
     }
```

Gets the current destination of the unit.

IsMoving

```
public bool IsMoving
     { get;
     }
```

Checks if the unit is moving or standing still.

UnitTaskHelper.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

UnitTaskHelper

```
public static class UnitTaskHelper
```

Static helper class for managing tasks on entities.

Methods

ScheduleTask

```
public static void ScheduleTask(
     this IUnit unit,
     ITaskContext context,
     ITaskInput input,
     ITask task)
```

Schedules a task for the specified entity.

unit: The unit to schedule the task for.
context: The task context.
input: The task input.
task: The task to schedule.

TryScheduleTask

```
public static bool TryScheduleTask(
     this IUnit unit,
     ITaskContext context,
     bool cancelsExistingTasks)
```

Attempts to schedule a task for the specified entity using the provided context.

unit: The unit to schedule the task for.
context: The task context.
cancelsExistingTasks: Cancels existing tasks before scheduling a new one.

Returns: True if a task was successfully scheduled; otherwise, false.

TryScheduleTask

```
public static bool TryScheduleTask(
     this IUnit unit,
     IEntity target,
     bool command = false)
```

Attempts to schedule a task for the specified entity targeting another entity.

unit: The unit initiating the task.
target: The target entity.
command: Whether the interaction task is a command. If true this will also cancel previous tasks.

Returns: True if a task was successfully scheduled; otherwise, false.

TryScheduleTask

```
public static bool TryScheduleTask(
     this IUnit unit,
     IUnitInteracteeComponent interactee,
     bool command)
```

Attempts to schedule a task for the specified entity interacting with an interactee.

unit: The unit initiating the task.
interactee: The interactee.
command: Whether the interaction task is a command. If true this will also cancel previous tasks.

Returns: True if a task was successfully scheduled; otherwise, false.

TryScheduleTask

```
public static bool TryScheduleTask(
     this IUnit unit,
     bool command,
     List<IEntity> potentialTargets,
     out IEntity target)
```

Attempts to schedule a task for the specified entity targeting a list of potential targets.

unit: The unit initiating the task.
command: Whether the interaction task is a command. If true this will also cancel previous tasks.
potentialTargets: The list of potential target entities.

Returns: True if a task was successfully scheduled; otherwise, false.

RemoveAllTasks

```
public static void RemoveAllTasks(
     this IUnit entity)
```

Removes all tasks associated with the specified entity.

entity: The entity to clear tasks from.

HasActiveTask

```
public static bool HasActiveTask(
     this IUnit unit)
```

Determines whether the specified unit has an active task.

unit: The unit to check.

Returns: True if the unit has an active task; otherwise, false.

Spawn

ABuildingSpawner.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

ABuildingSpawner

```
public abstract class ABuildingSpawner
: MonoBehaviour
```

Abstract base class defining the contract for spawning buildings.

Methods

SpawnBuilding

```
public abstract bool SpawnBuilding(
     AUnitSO unit,
     System.Action<Unit> unitSpawned)
```

Spawns a building using the specified unit data.

unit: The unit data representing the building to spawn.
unitSpawned: Callback invoked after the building is successfully spawned.

Returns: Returns true if building was spawned, otherwise false.

APlayerUnitSpawner.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

APlayerUnitSpawner

```
public abstract class APlayerUnitSpawner
: MonoBehaviour
```

Base class for handling unit spawning logic for a player. Supports both direct unit spawning and construction-based spawning.

Methods

GetBuildingSpawner

```
public abstract ABuildingSpawner GetBuildingSpawner()
```

Retrieves the building spawner responsible for constructing buildings.

Returns: The ABuildingSpawner instance associated with the player.

GetUnitSpawnPoint

```
public abstract UnitSpawnPoint GetUnitSpawnPoint()
```

Retrieves the spawn point used for direct unit spawning.

Returns: The UnitSpawnPoint instance used for unit placement.

SpawnUnits

```
public void SpawnUnits(
     UnitQuantity[] unitQuantity,
     Action<Unit[]> unitsSpawned = null)
```

Spawns units based on the specified quantities and calls a callback once all units are spawned.

unitQuantity: An array defining the units and their respective counts to spawn.
unitsSpawned: Optional callback invoked when all units are spawned.

FactionPlayerUnitSpawner.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

FactionPlayerUnitSpawner

```
public class FactionPlayerUnitSpawner
: APlayerUnitSpawner
```

A spawner for player units that belongs to a specific faction. Responsible for initializing units and managing spawning mechanisms for units and buildings.

Methods

GetUnitSpawnPoint

```
public override UnitSpawnPoint GetUnitSpawnPoint()
```

Gets the unit spawn point used for placing units.

Returns: The UnitSpawnPoint instance associated with this spawner.

GetBuildingSpawner

```
public override ABuildingSpawner GetBuildingSpawner()
```

Gets the building spawner used for constructing buildings.

Returns: The ABuildingSpawner instance associated with this spawner.

IUnitSpawn.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

IUnitSpawn

```
public interface IUnitSpawn
: IControlEntity
```

Defines a spawning point for units, responsible for spawning and respawning units at a designated location. Spawn points may also specify a target location for spawned units to move towards.

PlayerBuildingsSpawner.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

PlayerBuildingsSpawner

```
public class PlayerBuildingsSpawner
: ABuildingSpawner
```

Handles the spawning of buildings for a specific player, using predefined plot positions.

Methods

SpawnBuilding

```
public override bool SpawnBuilding(
     AUnitSO unit,
     System.Action<Unit> unitSpawned)
```

RadiusUnitSpawnPoint.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

RadiusUnitSpawnPoint

```
public class RadiusUnitSpawnPoint
: UnitSpawnPoint
```

A spawn point implementation that spawns units within a defined radius from the base spawn point.

Methods

GetRandomPoint

```
public static Vector3 GetRandomPoint(
     float radius)
```

Generates a random point within a circle of a given radius on the XZ plane.

radius: The radius of the circle.

Returns: A Vector3 representing a random point within the circle.

UnitSpawnPoint.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

UnitSpawnPoint

```
public class UnitSpawnPoint
: MonoBehaviour, IUnitSpawn
```

A MonoBehaviour that implements the IUnitSpawn interface, responsible for spawning and respawning units at a designated position and optionally moving them to a specified target location.

Methods

SpawnPoint

```
public Vector3 SpawnPoint()
```

SpawnUnit

```
public void SpawnUnit(
     AUnitSO unitData,
     bool moveToTarget,
     System.Action<Unit> unitSpawned)
```

SpawnUnit

```
public Unit SpawnUnit(
     Unit prefab,
     bool moveToTarget)
```

RespawnUnit

```
public void RespawnUnit(
     Unit sceneUnit,
     bool moveToTarget)
```

SetControlPosition

```
public void SetControlPosition(
     Vector3 target)
```

GetControlPosition

```
public Vector3 GetControlPosition()
```

Returns the target position of the spawn point.

Refund

DestructionRefundCapability.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

DestructionRefundCapability

```
[System.Serializable]
public struct DestructionRefundCapability
: IDestructionRefundCapability
```

Defines destruction refund capability for when a unit is destroyed.

Methods

DestructionRefund

```
public readonly ResourceQuantity[] DestructionRefund()
```

ApplyOnlyIfNonOperational

```
public readonly bool ApplyOnlyIfNonOperational()
```

IDestructionRefundCapability.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

IDestructionRefundCapability

```
public interface IDestructionRefundCapability
: IEntityCapability
```

Defines the refund capability when a unit is destroyed.

UnitRefund.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

UnitRefund

```
public static class UnitRefund
```

Methods

GetRefundResources

```
public static ResourceQuantity[] GetRefundResources(
     this IUnit unit)
```

Calculates and returns the resources to be refunded based on the unit's state.

Returns: The resources to be refunded.

Utility

AssetDatabaseHelper.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

AssetDatabaseHelper

```
public static class AssetDatabaseHelper
```

Methods

CreateFolders

```
public static bool CreateFolders(
     string fullPath)
```

DisableInInspectorAttribute.cs

Namespaces

UnitSystem.Utility

```
namespace UnitSystem.Utility
```

Classes

DisableInInspectorAttribute

```
public class DisableInInspectorAttribute
: PropertyAttribute
```

Disables property editing in Unity Editor Inspector, but still allows it to be visible or editable in DEBUG Inspector view.

DisableInInspectorDrawer

```
[CustomPropertyDrawer(typeof(DisableInInspectorAttribute))]
public class DisableInInspectorDrawer
: PropertyDrawer
```

Methods

OnGUI

```
public override void OnGUI(
     Rect position,
     SerializedProperty property,
     GUIContent label)
```

FloatingPointPrecision.cs

Namespaces

UnitSystem.Utility

```
namespace UnitSystem.Utility
```

Classes

FloatingPointPrecision

```
public static class FloatingPointPrecision
```

Provides methods for converting integer/long values to floating-point representations and formatted string outputs with configurable precision.

Methods

GetFloat

```
public static float GetFloat(
     long value,
     int precision = 2)
```

Converts a long value into a floating-point number with the specified precision.

value: The integer/long value to convert.
precision: The number of decimal places to represent. Default is 2.

Returns: A float representing the input value with the specified precision.

GetString

```
public static string GetString(
     long value,
     int precision = 2)
```

Converts a long value into a formatted string representation with integer and decimal parts.

value: The integer/long value to convert.
precision: The number of decimal places to represent. Default is 2.

Returns: A string representing the value as "integer,decimal".

GameObjectHelper.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

GameObjectHelper

```
public static class GameObjectHelper
```

Provides extension methods for managing components and child GameObjects in Unity.

Methods

AddComponentIfNotPresent

```
public static Component AddComponentIfNotPresent(
     this GameObject gameObject)
```

Adds a component of the specified type to the GameObject if it does not already exist.

gameObject: The GameObject to which the component will be added.

Returns: The existing component if present, otherwise the newly added component.

AddComponentIfNotPresent

```
public static Component AddComponentIfNotPresent(
     this GameObject gameObject,
     bool includesChildren)
```

Adds a component of the specified type to the GameObject or its children if it does not already exist.

gameObject: The GameObject to which the component will be added.
includesChildren: If true, checks the GameObject and all its children for the component.

Returns: The existing component if present in the GameObject or its children, otherwise the newly added component.

AddChildGameObject

```
public static GameObject AddChildGameObject(
     this GameObject parent,
     string childName)
```

Creates a new child GameObject under the specified parent GameObject.

parent: The parent GameObject to which the child will be added.
childName: The name to assign to the new child GameObject.

Returns: The newly created child GameObject.

IEntityPrefabCreatable.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

IEntityPrefabCreatable

```
public interface IEntityPrefabCreatable
```

Provided interface for creating a prefab from a AEntitySO in the Unit System Manager Window.

IProvidesEntityPrefab.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

IProvidesEntityPrefab

```
public interface IProvidesEntityPrefab
: IEntityPrefabCreatable
```

Provided interfaces for accessing a prefab from AEntitySO. Because loading a prefab can be done in various ways from directly referencing an asset in project, loading it from path or using Addressables, this interfaces provides multiple ways of accessing it. If GetAssociatedPrefab is called and null is returned, then the spawning system will assume that prefab must be loaded in an async manner. Manager window uses IsPrefabSet to determine if prefab should be created for the data set.

InfoLogger.cs

Namespaces

UnitSystem.Utility

```
namespace UnitSystem.Utility
```

Classes

InfoLogger

```
public static class InfoLogger
```

Class responsible for printing info logs in the console produced by the UnitSystem logic. It is using Debug.Log(object) when flag Enabled is true.

Variables

Enabled

```
public static bool Enabled
```

Logging by default is enabled. To disable it set this to false.

Methods

Log

```
public static void Log(
     string log)
```

Print info text in the console.

log: Text to print

InteractorPositionHelper.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

InteractorPositionHelper

```
public static class InteractorPositionHelper
```

Provides utility methods for handling interaction positioning between IUnitInteractorComponent and IUnitInteracteeComponent objects.

Methods

Vector3) CalculateInteractionPositionAndDirection

```
public static (Vector3, Vector3)
CalculateInteractionPositionAndDirection(
     this IUnitInteractorComponent interactor,
     IUnitInteracteeComponent interactee)
```

Calculates the interaction position and direction for a given interactor and interactee.

interactor: The unit initiating the interaction.
interactee: The unit being interacted with.

Returns: A tuple containing: - The calculated destination position for the interactor. - The direction the interactor should face towards the interactee.

Vector3) CalculateInteractionPositionAndDirection

```
public static (Vector3, Vector3)
CalculateInteractionPositionAndDirection(
     this IUnitInteractorComponent interactor,
     IUnitInteracteeComponent interactee,
     Bounds interacteeBoundingBox)
```

Calculates the interaction position and direction, taking the interactee's bounding box into account.

interactor: The unit initiating the interaction.
interactee: The unit being interacted with.
interacteeBoundingBox: The bounding box of the interactee. This is used to calculate the closest point for the interaction.

Returns: A tuple containing: - The calculated destination position for the interactor. - The direction the interactor should face towards the interactee.

MathUtils.cs

Namespaces

UnitSystem.Utility

```
namespace UnitSystem.Utility
```

Classes

MathUtils

```
public static class MathUtils
```

Utility class providing common mathematical operations for long integers.

Methods

Min

```
public static long Min(
      long a,
      long b)
```

Returns the smaller of two long values.

a: The first value to compare.
b: The second value to compare.

Returns: The smaller of the two input values.

Max

```
public static long Max(
      long a,
      long b)
```

Returns the larger of two long values.

a: The first value to compare.
b: The second value to compare.

Returns: The larger of the two input values.

NearbyColliderFinder.cs

Namespaces

UnitSystem.Utility

```
namespace UnitSystem.Utility
```

Classes

NearbyColliderFinder

```
[System.Serializable]
public class NearbyColliderFinder
```

Generic wrapper for using Physics sphere overlaps. Can use non allocating methods, based on configuration.

Constructors

NearbyColliderFinder

```
public  NearbyColliderFinder()
```

Methods

Validate

```
public delegate bool Validate(
     TComponentType component)
```

PositiveFloatAttribute.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

PositiveFloatAttribute

```
public class PositiveFloatAttribute
: PropertyAttribute
```

Prevents negative value for float types.

Constructors

PositiveFloatAttribute

```
public  PositiveFloatAttribute()
```

UnitSystem

```
namespace UnitSystem
```

Classes

PositiveFloatAttributeDrawer

```
[CustomPropertyDrawer(typeof(PositiveFloatAttribute))]
public class PositiveFloatAttributeDrawer
: PropertyDrawer
```

Methods

OnGUI

```
public override void OnGUI(
     Rect position,
     SerializedProperty property,
     GUIContent label)
```

PositiveIntAttribute.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

PositiveIntAttribute

```
public class PositiveIntAttribute
: PropertyAttribute
```

Prevents negative value for int types.

Constructors

PositiveIntAttribute

```
public  PositiveIntAttribute()
```

UnitSystem

```
namespace UnitSystem
```

Classes

PositiveIntAttributeDrawer

```
[CustomPropertyDrawer(typeof(PositiveIntAttribute))]
public class PositiveIntAttributeDrawer
: PropertyDrawer
```

Methods

OnGUI

```
public override void OnGUI(
     Rect position,
     SerializedProperty property,
     GUIContent label)
```

PositiveLongAttribute.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

PositiveLongAttribute

```
public class PositiveLongAttribute
: PropertyAttribute
```

Prevents negative value for long types.

Constructors

PositiveLongAttribute

```
public  PositiveLongAttribute()
```

UnitSystem

```
namespace UnitSystem
```

Classes

PositiveLongAttributeDrawer

```
[CustomPropertyDrawer(typeof(PositiveLongAttribute))]
public class PositiveLongAttributeDrawer
: PropertyDrawer
```

Methods

OnGUI

```
public override void OnGUI(
     Rect position,
     SerializedProperty property,
     GUIContent label)
```

RequiresTypeAttribute.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

RequiresTypeAttribute

```
[AttributeUsage(AttributeTargets.Field)]
public class RequiresTypeAttribute
: PropertyAttribute
```

Attribute to specify that a serialized field requires a component implementing a specific interface or inheriting a specific type.

Constructors

RequiresTypeAttribute

```
public  RequiresTypeAttribute(
     Type expectedInterfaceType)
```

Initializes a new instance of the RequiresTypeAttribute class.

expectedInterfaceType: The Type that the field’s assigned value must implement or inherit from.

Variables

InterfaceType

```
public Type InterfaceType
```

Gets the required type or interface that the assigned field must implement.

SingletonHandler.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

SingletonHandler

```
public static class SingletonHandler
```

A utility class for creating and managing singleton instances of MonoBehaviour types.

Methods

Create

```
public static SingletonType Create(
     string objectName)
```

Creates a new singleton instance of the specified MonoBehaviour type, if one does not already exist.

objectName: The name to assign to the newly created singleton GameObject.

Returns: A singleton instance of the specified type, or null if the application is quitting.

RunOnStart

```
[RuntimeInitializeOnLoadMethod]
public static void RunOnStart()
```

Registers the application quitting event to handle cleanup logic.

UnityNullChecks.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

UnityObjectNullChecks

```
public static class UnityObjectNullChecks
```

Provides extension methods for null checks on Unity objects, addressing issues with Unity's special handling of the equality operator for destroyed objects.

Methods

IsNull

```
public static bool IsNull(
     this T obj)
```

Determines whether the given Unity object is effectively null.

obj: The Unity object to check.

Returns: true if the object is null or has been destroyed; otherwise, false.

IsNull

```
public static bool IsNull(
     this object obj)
```

Determines whether the given object is effectively null, including destroyed Unity objects.

obj: The object to check.

Returns: true if the object is null or has been destroyed; otherwise, false.

IsNotNull

```
public static bool IsNotNull(
     this object obj)
```

Determines whether the given object is not null, including checking for destroyed Unity objects.

obj: The object to check.

Returns: true if the object is not null and has not been destroyed; otherwise, false.

Vector3Distance.cs

Namespaces

UnitSystem.Utility

```
namespace UnitSystem.Utility
```

Classes

Vector3Distance

```
public static class Vector3Distance
```

Methods

GetClosestPointIndex

```
public static int GetClosestPointIndex(
     Vector3 targetPosition,
     Vector3[] points)
```

Finds the index of the point in a given array of Vector3 points that is closest to a specified target position.

targetPosition: The position to which the closest point is calculated.
points: An array of points to search for the closest one. The array can contain any number of points.

Returns: The index of the closest point in the array. Returns -1 if the array is null or empty.