UnitSystem

Table of Contents

Script Reference

BasicSO.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

BasicSO

```
public class BasicSO
```

Basic ScriptableObject that contains UUID for it. Identifiers are good practice for matching items, to avoid comparing multiple properties, structures or even references.

Constructors

BasicSO

```
protected  BasicSO()
```

Variables

UUID

```
public string UUID
```

Public getter for UUID.

Attribute

AttributeSO.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

AttributeSO

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

Defines base attribute for producables. To add additional fields or to add methods/functionality to attributes, derive from this class. Changing original files will result in override (lost changes) if package is ever updated.

Variables

Name

```
public string Name
```

Specifies the attribute name, such as "Health", "DamageToArchers", etc.

Description

```
public string Description
```

Specifies the description of the attribute. Generally this is also the description that player may read.

AttributeValue.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Enumerations

ValueType

```
public enum ValueType{
     Relative,
     Absolute,
     Percentage}
```

Relative: When value should manipulate current unit attribute value.

Absolute: When value should replace current unit attribute value.

Percentage: When value should manipulate current unit attribute with percentage value.

Classes

AttributeValue

```
[Serializable]
public struct AttributeValue
```

Representation of attribute, value and its value type (ValueType).

This allows relative values (example: increasing population capacity for 2), percentage value (example: where damage versus buildings is increased for 20%), and absolute value which is expected to override/replace the current value (example: new health for a unit is 200 regardless of the previous value).

Variables

Attribute

```
public AttributeSO Attribute
```

Specifies the attribute for the value.

Value

```
public float Value
```

Specifies the value of the attribute.

ValueType

```
public ValueType ValueType
```

Specifies the type of value and how it is applied.

Player

APlayer.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

APlayer

```
public abstract class APlayer
```

Definition of main players component that is referenced by units in order to communicate events from unit components such as ProductionUnit, where player is used to add newly produced units to the player.

Variables

OnUnitAdded

```
public Action<Unit> OnUnitAdded
```

Action invoked once a unit has been added to the Units collection.

OnUnitRemoved

```
public Action<Unit> OnUnitRemoved
```

Action invoked once a unit has been removed from the Units collection.

OnProducableAdded

```
public Action<ProducableSO,float> OnProducableAdded
```

Action invoked once a producable has been added to the player.

OnProducableRemoved

```
public Action<ProducableSO> OnProducableRemoved
```

Action invoked once a producable has been removed from the player.

Methods

GetUnits

```
public Unit[] GetUnits()
```

Public getter for all the players units. This converts list to an array so do not call it too often.

AddUnit

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

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.

RemoveUnit

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

Removes unit from the player by removing ownership and removing the unit from Units collection. ’ArgumentException’ will be thrown if player does not contain this unit.

unit: Unit to be removed, which will no longer be under ownership of it’s current player.

AddProducable

```
public virtual void AddProducable(
     ProducableSO producable)
```

Add a producable of quantity 1 to the player.

producable: Producable to be added.

AddProducable

```
public virtual void AddProducable(
     ProducableSO producable,
     float quantity)
```

Adds a producable with specified quantity to the player. Primarily this method only triggers action for other components to respond to, with the use of OnProducableAdded. Player production units use this interface to produce new units or resources for the unit owner (player).

producable: Producable to be added.
quantity: Quantity of producables to be added.

RemoveProducable

```
public virtual void RemoveProducable(
     ProducableSO producable)
```

Remove a producable from the player. Primarily this method only triggers action for other components to respod to, with the use of OnProducableRemoved.

producable: Producable to be removed.

ContainsProducable

```
public virtual bool ContainsProducable(
     ProducableSO producable)
```

Invokes method ContainsProducable(ProducableQuantity) with quantity of 1.

producable: Producable to be used for check.

Returns: Returns true if contains a producable of at least 1 quantity.

ContainsProducable

```
public abstract bool ContainsProducable(
     ProducableQuantity producableQuantity)
```

Implement this method to check whether a player contains the specified producable with quantity equal or larger.

producableQuantity: Producable quantity to be used for checking

ContainsUnit

```
public bool ContainsUnit(
     Unit unit)
```

Checks whether the unit exists within the Units collection.

FulfillsRequirements

```
public virtual bool FulfillsRequirements(
     ProducableSO producable,
     float quantity)
```

Checks if the requirements for a producable are fullfilled, by checking if the player contains all the requirements defined on producable with the help of existing method ContainsProducable(ProducableQuantity). To customise this override the method.

producable: Producable with requirements
quantity: Quantity of the producable, this will be multipled requirements quantity.

Returns: If there are no requirements or they are fullfilled TRUE is returned, otherwise FALSE.

Player.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

Player

```
public class Player
```

Next to responsibilities of APlayer this component also connects various management components that allow player to track it’s stats from a single component. Each player in game should have a component that implements APlayer in order to attach information to Units spawned in scene.

If you wish to customise behaviour of the player completely, derive APlayer to use it with the UnitSystem components.

Variables

ResearchManager

```
[Tooltip("Specifies the research manager for the player.")]
public ResearchManager ResearchManager
```

Specifies the research manager for the player.

ResourceManager

```
[Tooltip("Specifies the resource manager for the player.")]
public ResourceManager ResourceManager
```

Specifies the resource manager for the player.

PopulationManager

```
[Tooltip("Specifies the population manager for the player.")]
public PopulationManager PopulationManager
```

Specifies the population manager for the player.

Methods

AddUnit

```
public override void AddUnit(
     Unit unit)
```

RemoveUnit

```
public override void RemoveUnit(
     Unit unit)
```

AddResource

```
public float AddResource(
     ProducableSO producable,
     float quantity)
```

ContainsProducable

```
public override bool ContainsProducable(
     ProducableQuantity producableQuantity)
```

Population

PopulationManager.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

PopulationManager

```
[RequireComponent(typeof(APlayer))]
public class PopulationManager
```

Component responsible for managing population of a player. Many RTS games use population to control the number of units that can be spawned in a game.

This component handles attributes of units! If you do not use maximal population and do not want to limit max population, then you may add only an attribute on the unit definition for population consumption (with this attribute you can easily track players current population count - populationConsumptionAttributeUUID). If the player can increase maximal population by constructing or producing other units, then make sure to set populationCapacityAttributeUUID for manager to handle this automatically.

Variables

MaxPopulationEnabled

```
public bool MaxPopulationEnabled
```

Returns true if the maximal population is enabled, which means population is capped.

CurrentPopulation

```
public int CurrentPopulation
```

Current population count.

OnPopulationUpdate

```
public System.Action<int,int> OnPopulationUpdate
```

Action invoked when population is updated, either current or maximal.

Properties

MaxPopulation

```
public int MaxPopulation
     { get;
     }
```

Maximal population allowed. If maximal population is disabled (not used), then '-1' will be returned.

Methods

ShouldFinishProductionFor

```
public bool ShouldFinishProductionFor(
     ProducableQuantity producableQuantity)
```

Checks if production should be finished for a certain producable regarding the population consumption attribute.

producableQuantity: Producable to produce

Returns: Returns ’false’ if population requirements are not fulfilled (not enough population capacity for the producables quantity).

Production

IProduce.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

IProduce

```
public interface IProduce
```

Production interface used for updating current production with time/progress. You can use time passed (example: Time.deltaTime) for realtime production or a custom delta value that can represent a turn based progress (example: 1 for when player ends his turn).

Methods

Produce

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

Called each time production needs to apply the delta. Implement this to handle production progress of components.

delta: Value change that will be applied to production. If its realtime it will probably be Time.deltaTime, otherwise it can be turn based like 1, 2, etc...

ProducableQuantity.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

ProducableQuantity

```
[Serializable]
public struct ProducableQuantity
```

Representation of producable quantities with float.

Constructors

ProducableQuantity

```
public  ProducableQuantity(
     ProducableSO producable,
     float quantity)
```

Variables

Producable

```
public ProducableSO Producable
```

Specifies the producable.

Quantity

```
public float Quantity
```

Specifies quantity of the producable.

ProducableSO.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

ProducableSO

```
public abstract class ProducableSO
```

Base producable object, it is primarily used to define producables for the project.. Any gameObject/unit should use this class or derive from this class in order to utilise UnitSystem components.

All producables should originate from this scriptable object and should primarily differentiate between other producables with the UUID. You may either subclass this to define new types (like structures) or you may also add a new field like ‘Type‘ with an enum to differentiate between such units (like workers vs structures).

Variables

Name

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

Specifies the name of the producable, examples: "Solider", "House".

Description

```
public string Description
```

Specifies the description of the producable.

Attributes

```
public List<AttributeValue> Attributes
```

Specifies producable attributes.

Requirements

```
public List<ProducableQuantity> Requirements
```

Specifies producable objects requirements, other than cost. These should not be consumed before production starts.

Cost

```
public List<ProducableQuantity> Cost
```

Specifies the resource requirements to produce this. These will be consumed before the production starts.

ProductionDuration

```
[Range(0, 99_999)]
public float ProductionDuration
```

Specifies the time required to produce this (examples: training or construction). Values allowed should be positive numbers, max on inspector range is 99_999 and it can be increased here if needed.

ProductionAction.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

ProductionActionGroup

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

Representation of production actions group. A group will define a list of actions under a certain name. Example: Research, Units, Default or for workers a groups like Housing, Food Production structures, etc.

Variables

Name

```
public string Name
```

Specifies the name of the production action group.

Actions

```
public List<ProductionAction> Actions
```

Specifies the list of actions in the group.

ProductionAction

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

Represents a production action result and its quantity. Quantity specifies number of producables to be produced and consequently how much resources is required for the production.

Variables

Producable

```
public ProducableSO Producable
```

Producable item that will be the result of production.

Quantity

```
[Range(0.1f, 99_999f)]
public float Quantity
```

Quantity of the ’Producable’ this action produces.

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.

Methods

ShouldFinishProductionFor

```
public bool ShouldFinishProductionFor(
     ProducableQuantity producableQuantity)
```

Method invoked before producing the latest item. Implement this to return ’false’ and prevent production from finishing. This will then be attempted for each ProductionQueue.Produce(float) iteration.

producableQuantity: Producable and its quantity

Returns: Return ’true’ to finish production, return ’false’ to prevent it.

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

Queue

```
public ProducableQuantity[] Queue
```

Collection of the current production items.

OnProductionFinished

```
public Action<ProducableQuantity> OnProductionFinished
```

Action invoked once a producable 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

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(
     ProducableSO producable)
```

Checks if the production queue contains this producable. Matching is done with BasicSO.UUID.

producable: Producable used for matching.

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

AddProductionOrder

```
public void AddProductionOrder(
     ProducableSO producable,
     float quantity,
     bool queueMultipleOrders = false)
```

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

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

CancelProductionOrder

```
public bool CancelProductionOrder(
     ProducableSO producable)
```

Cancels the first producable that matches the parameter.

producable: Producable that will be canceled if present in queue.

Returns: Returns 'true' if production order was canceled. Returns 'false' only when producable with the same UUID is not present in queue.

CancelProductionOrder

```
public bool CancelProductionOrder(
     string producableUUID)
```

Cancel the first producable with UUID that matches the argument.

producableUUID: UUID that will be used for finding a matching producable.

Returns: Returns 'true' if production order was canceled. Returns 'false' only when no producable was found with matching UUID.

CancelProductionOrder

```
public bool CancelProductionOrder(
     int index)
```

Cancels the producable on specified index.

index: Index on which the producable should be canceled.

Returns: Returns 'true' if item was successfully removed from the list. Returns 'false' when index is out of bounds.

CancelProduction

```
public void CancelProduction()
```

Cancels all production orders in the queue.

Research

ResearchManager.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

ResearchManager

```
public class ResearchManager
```

Simple research managment component, responsible for keeping track of currently completed researches for a player. These are generally less complex than for resource management. Actual production of research should be processed as part of production.

Variables

OnResearchFinished

```
public Action<ProducableSO> OnResearchFinished
```

Action invoked when research is added to the collection via AddFinishedResearch(ProducableSO).

Methods

AddFinishedResearch

```
public void AddFinishedResearch(
     ProducableSO research)
```

Adds new research to the completedResearches collection and invokes the OnResearchFinished action.

research: Research to be added.

IsResearchComplete

```
public bool IsResearchComplete(
     ProducableSO 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(
     ProducableSO 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
```

Specifies a default research scriptable object. The ResearchManager is dependent on the ProducableSO so this is just a convenience for type casting in Player for specific behaviour.

Resource

ResourceManager.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

ResourceManager

```
public class ResourceManager
```

Resource managment component, responsible for keeping track of current and maximal resources available for a player. Provides a set of methods that allow simple modification of resources.

Variables

CommonMaxResourceQuantity

```
public float CommonMaxResourceQuantity
```

Specifies default resource capacity. For all resources specified in maxResources list, this value is ignored.

UnlimitedResourcesQuantity

```
[Tooltip("Specifies if the max resources are ignored.")]
public bool UnlimitedResourcesQuantity
```

Specifies if the max resources are ignored, therefore resources can store to "unlimited" amount. Then the resources amount is limited by float.MaxValue.

UseUniqueMaxResources

```
[SerializeField]
public bool UseUniqueMaxResources
```

This flag enables configuration of max resources per resource. If this is set to FALSE, resources will share a single storage and all resources combined should NOT exceed the common max storage (CommonMaxResourceQuantity). In such case to modify max common storage of all resources you need to adjust the CommonMaxResourceQuantity property by invoking ModifyCommonMaxResourceQuantity(float).

OnResourceFull

```
[Header("Events")]
public Action<ProducableSO> OnResourceFull
```

Action invoked when resource was updated and its maximal capacity was reached.

OnResourceUpdate

```
public Action<ProducableSO> OnResourceUpdate
```

Action invoked when resource quantity has changed.

OnMaxResourceUpdate

```
public Action<ProducableSO> OnMaxResourceUpdate
```

Action invoked when maximal resource quantity has changed.

Methods

AddResources

```
public ProducableQuantity[] AddResources(
     ProducableQuantity[] resources)
```

Modifies existing resource quantity or adds a new one if none exists yet. Internally calls AddResource(ProducableQuantity) for each resource quantity.

resources: Resources to be added.

Returns: Returns list of remaining resources. If this is empty, then all resources were added to the manager.

AddResource

```
public float AddResource(
     ProducableQuantity resourceQuantity)
```

Modifies existing resource quantity or adds a new one if none exists yet. Also invokes OnResourceUpdate after resource quantity is modified and invokes OnResourceFull when resource has reached its max capacity.

resourceQuantity: Resource to be added.

Returns: Returns a value of remaining resource quantity. If this value is 0, then full quantity was applied to the manager.

ConsumeResource

```
public bool ConsumeResource(
     ProducableQuantity 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 manager.

Returns: Returns ’false’ if there is not enough resource available.

ConsumeResources

```
public bool ConsumeResources(
     List<ProducableQuantity> 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 manager.

Returns: Returns ’false’ if there is not enough resources available.

ModifyMaxResourceQuantity

```
public void ModifyMaxResourceQuantity(
     ProducableQuantity resourceQuantity)
```

Applies changes to current maximal resource quantity available in maxResources. If there is no existing maximal resource, then this quantity is simply set as initial value.

resourceQuantity: Resource quantity used matching and modifying.

ModifyCommonMaxResourceQuantity

```
public void ModifyCommonMaxResourceQuantity(
     float quantity)
```

Modify CommonMaxResourceQuantity.

quantity: Quantity to add

SetCommonMaxResourceQuantity

```
public void SetCommonMaxResourceQuantity(
     float newQuantity)
```

Set the CommonMaxResourceQuantity.

newQuantity: New quantity

SetMaxResourceQuantity

```
public void SetMaxResourceQuantity(
     ProducableQuantity resourceQuantity)
```

Setting maximal resource quantity by overriding existing one, or adding a new one if one does not exist.

resourceQuantity: Overriding maximal resource quantity.

SetResourceQuantity

```
public void SetResourceQuantity(
     ProducableQuantity 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(
     ProducableSO resource,
     float 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(
     ProducableQuantity 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(
     List<ProducableQuantity> 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.

GetMaxResourceQuantity

```
public float GetMaxResourceQuantity(
     string resourceUUID)
```

Retrieves the current maximal resource quantity for the provided resource UUID, or the default value (CommonMaxResourceQuantity) when UseUniqueMaxResources is set to FALSE or the custom max resource is not set for this resource.

resourceUUID: UUID used for matching with maxResources

Returns: Returns the max resource quantity allowed for this resource manager.

GetResourceQuantity

```
public float GetResourceQuantity(
     string resourceUUID)
```

Returns the current resource quantity for the provided resource UUID.

resourceUUID: UUID used for matching the resource

Returns: Returns the current resource quantity for the given UUID.

IsResourceMaxedOut

```
public bool IsResourceMaxedOut(
     string resourceUUID)
```

Checks if the resource is maxed out based on its maximal capacity.

GetResourcesQuantityForLimitedResources

```
public float 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 ProducableQuantity[] 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(ProducableQuantity) method if this is ever desired.

GetMaxResources

```
public ProducableQuantity[] GetMaxResources()
```

Current maximal resources. If resource is not specified here, default value is used (CommonMaxResourceQuantity) for maximal storage of the resource.

ResourceSO.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

ResourceSO

```
[CreateAssetMenu(fileName = "New resource", menuName = "Unit
System/Resource")]
public class ResourceSO
```

Specifies a default resource scriptable object. The ResourceManager is dependent on the ProducableSO so this is just a convenience for type casting in Player for specific behaviour.

Time Management

RTSTimeController.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

RTSTimeController

```
public class RTSTimeController
```

Component for updating production for real time strategy. Each player has a collection of units and if any of units components implement <see cref="IProduce"/ then production time will be applied to them each frame.

Variables

IsGameRunning

```
public bool IsGameRunning
```

Specifies if the game is running. Set this to 'false' when game is paused and production will be paused as well.

Players

```
public List<APlayer> Players
```

TBSTimeController.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

TBSTimeController

```
public class TBSTimeController
```

Component for updating production for turn based strategy. Each player has a collection of units and if any of units components implement <see cref="IProduce"/ then production time will be applied to them on end of turn.

Variables

Players

```
public List<APlayer> Players
```

CurrentPlayer

```
public APlayer CurrentPlayer
```

Current player on turn. If there are no players in the collection (Players), this getter will crash.

singleTurnTimeIncrement

```
public float singleTurnTimeIncrement
```

Delta applied for a single production cycle (one turn).

AllTurnsEnded

```
public Action AllTurnsEnded
```

Action invoked when all players have ended their turn.

Methods

EndTurnForCurrentPlayer

```
public void EndTurnForCurrentPlayer()
```

Call this method when current player ends its turn.

Unit

AUnitSO.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

AUnitSO

```
public abstract class AUnitSO
```

Core scriptable object for units (producables that are spawned in Scene). Any producable that creates a game object should derive from this class. Unit components are primarily tied to this abstract class. To learn how to use this in custom iplementation check out UnitSO.

Methods

GetAssociatedPrefab

```
public abstract IEnumerator GetAssociatedPrefab(
    Action<Unit> unitLoaded)
```

Implement this method to provide prefab for the unit. Interface is structured for async loading, but you can load prefab right away and invoke the argument unitLoaded.

unitLoaded: Action for invoking once prefab is loaded. If this is never called, unit will never be spawned.

Unit.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

Unit

```
[DisallowMultipleComponent]
public class Unit
```

Core Unit component used for attaching unit data and player owner to the unit. This way unit can access it’s own data and it’s owner for various events.

Variables

Owner

```
public APlayer Owner
```

Specifies the player owner of the unit.

Data

```
public AUnitSO Data
```

Specifies the data information about the Unit.

UnitSO.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

UnitSO

```
[CreateAssetMenu(fileName = "New unit", menuName = "Unit
System/Units/Unit")]
public class UnitSO
```

Scriptable object deriving from AUnitSO and contains implementation for the basic prefab approach. Prefab is required for spawning units (game objects) within world scene. If more customisation is needed on the scriptable object, modify this one or create your own subclass of AUnitSO.

Methods

GetAssociatedPrefab

```
public override IEnumerator GetAssociatedPrefab(
     Action<Unit> unitLoaded)
```

Control

AUnitSpawnPoint.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

AUnitSpawnPoint

```
public abstract class AUnitSpawnPoint
```

Definition for spawning point on the unit, where other units can be spawned or respawned. Generally spawn points have option to set target location for the spawned units.

Methods

SpawnUnit

```
public abstract Unit SpawnUnit(
     Unit prefab)
```

Implement this to spawn a unit from prefab.

prefab: Prefab used for spawning the new unit.

Returns: Returns newly instantiated unit.

RespawnUnit

```
public abstract void RespawnUnit(
     Unit sceneUnit)
```

Implement this to respawn a unit.

sceneUnit: Instance of the unit that needs to be respawned.

SetTargetPosition

```
public abstract void SetTargetPosition(
     Vector3 target)
```

Implement this to set the target position of the spawn. Commonly units spawn on spawn point, are then moved to the target spawn position.

target: New target position for the unit.

IControlUnit.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

IControlUnit

```
public interface IControlUnit
```

Interface used for communicating commands to units that a new target position has been assigned to them.

UnitSpawnPoint.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

UnitSpawnPoint

```
public class UnitSpawnPoint
```

Basic implementation of the AUnitSpawnPoint abstract component. This spawn point supports spawning new units from prefab, respawning existing units and moving them to target position using the IControlUnit interface.

Variables

gizmoColor

```
[SerializeField]
public Color gizmoColor
```

Specifies the color of the gizmo on target position.

Methods

SpawnUnit

```
public override Unit SpawnUnit(
     Unit prefab)
```

RespawnUnit

```
public override void RespawnUnit(
     Unit sceneUnit)
```

SetTargetPosition

```
public override void SetTargetPosition(
     Vector3 target)
```

Garrison

Garrison.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

Garrison

```
[RequireComponent(typeof(Unit), typeof(Collider))]
public class Garrison
```

Garrison component is designed to be used for units that have garrison attribute which allows other units to enter this one. Max capacity is determined by the attribute attached to the Unit.Data and its UUID is matched by Garrison.GarrisonCapacityAttributeUUID. This allows you to create your own capacity attribute and still use this component.

Variables

MaxCapacity

```
public int MaxCapacity
```

Get maximal capacity available for this garrison.

GarrisonEntrancePosition

```
public Vector3 GarrisonEntrancePosition
```

Position of the garrison entrance.

GarrisonCapacityAttributeUUID

```
[Tooltip("Specifies the UUID of the garrison capacity
attribute.")]
public string GarrisonCapacityAttributeUUID
```

Specifies the ’garrison’ attribute reference. Set this in order to compare with attribute on units. If this is left on ’null’, it will not work.

OnUnitEnter

```
public System.Action<Unit> OnUnitEnter
```

Action invoked when unit enters the garrison.

OnUnitExit

```
public System.Action<Unit> OnUnitExit
```

Action invoked when unit exits the garrison.

Properties

GarrisonUnits

```
public List<Unit> GarrisonUnits
     { get;
     private set; }
```

Specifies the list of units currently occupying space within this unit. Maximal number of units is defined by the garrison attribute on the unit.

Methods

AddUnit

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

RemoveUnit

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

RemoveAllUnits

```
public void RemoveAllUnits()
```

GarrisonEntrance.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

GarrisonEntrance

```
public class GarrisonEntrance
```

Entrance for Garrison that handles adding units to the garrison once they are within range.

Setup the position and size of the entrance for units to be picked up when in range. Make sure the entrace collision detection is not inside garrison units' main collider. Make sure to select a desired collisionDetectionMode in order to check collisions. Support is for collider trigger or manual detection of overlapping colliders.

Methods

Update

```
public void Update()
```

IGarrisonUnit.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

IGarrisonUnit

```
public interface IGarrisonUnit
```

Interface used for units that have capacity to garrison other units.

IGarrisonableUnit.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

IGarrisonableUnit

```
public interface IGarrisonableUnit
```

Interface used for units that can be garrisoned into a garrison unit (IGarrisonUnit).

Production

ProductionUnit.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

ProductionUnit

```
public class ProductionUnit
```

Component used for handling unit production with ProductionQueue. It will take care of spawning units after they are produced with the use of spawnPoint.

It will also attempt to garrison units if GarrisonUnitsAfterProduction is enabled and if garrisonUnit is not ’NULL’.

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 IGarrisonUnit (if present) will be used to garrison units after production.

ProductionQueue

```
public readonly ProductionQueue ProductionQueue
```

Queue for producing units, resources, etc.

Methods

StartProduction

```
public void StartProduction(
     ProductionAction action,
     bool queueMultipleOrders = false)
```

Queues production action on the unit or splits it into multiple productions.

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

StartProduction

```
public void StartProduction(
     ProducableSO producable,
     float quantity,
     bool queueMultipleOrders = false)
```

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

producable: Producable to be produced
quantity: Number of producables
queueMultipleOrders: Split production quantity into multiple orders

CancelProduction

```
public void CancelProduction()
```

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

Produce

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

Updates production time on production queue.

delta: Amount of progression for the queue

SetTargetPosition

```
public void SetTargetPosition(
     Vector3 groundPosition)
```

Sets the spawn point target position.

groundPosition: New target position for spawned units.

ProductionUnitSO.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

ProductionUnitSO

```
[CreateAssetMenu(fileName = "New unit", menuName = "Unit
System/Units/Production Unit")]
public class ProductionUnitSO
```

Component derives from UnitSO and it is intended to be used in combination with the ProductionUnit. Production units can produce other producables (units, resources and more). If more complex class is required, modify this one or create your own subclass of either UnitSO or AUnitSO. ProductionUnit can be used completely independently even if this SO is not used.

Variables

GroupedActions

```
[Header("Production"), Tooltip("Action groups of the unit.
This way " +
"you can separate units by context or type.")]
public List<ProductionActionGroup> GroupedActions
```

Specifies group of production actions available on this unit. This can include any ProducableSO like resource, unit, research, etc.

ProducesResource

```
[Tooltip("List of resources this unit produces and their
quantity per " +
"second, per turn or per custom time based
period.")]
public List<ProducableQuantity> ProducesResource
```

Specifies the resources this unit produces per game defined period. This can be per one second or per turn, or any other custom time behaviour. Multiple resources are supported.

Methods

GetAllProductionActions

```
public List<ProductionAction> GetAllProductionActions()
```

Iterates through GroupedActions and creates a flat list of the actions of all groups.

Returns: Returns a flat list of all production actions within the actions groups.

UnitResourceProduction.cs

Namespaces

UnitSystem

```
namespace UnitSystem
```

Classes

UnitResourceProduction

```
[RequireComponent(typeof(Unit))]
public class UnitResourceProduction
```

Simple component for producing resources. Resources to produce are retrieved from units DATA on ProductionUnitSO and the production of those resources is handled with implementation of the IProduce interface.

It can support realtime or turn based production, depending on time manager behaviour.

Methods

Start

```
public void Start()
```

GetProducingResources

```
public virtual List<ProducableQuantity>
GetProducingResources()
```

Retrieves production resources from Unit.Data, it expects a type of ProductionUnitSO. Override this method to return custom production resources.

Returns: Returns resources to produce by the unit.

Produce

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

Utility

DisableInInspectorAttribute.cs

Namespaces

UnitSystem.Utility

```
namespace UnitSystem.Utility
```

Classes

DisableInInspectorAttribute

```
public class DisableInInspectorAttribute
```

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
```

Methods

OnGUI

```
public override void OnGUI(
     Rect position,
     SerializedProperty property,
     GUIContent label)
```

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