Master of Mayhem – Technical Documentation

Contents

[GameManager 7](#_Toc174448953)

[GameManager 7](#_Toc174448954)

[PlacedObjectLog 8](#_Toc174448955)

[GeneralBalanceSettler 8](#_Toc174448956)

[BuildingBalanceSettler 8](#_Toc174448957)

[BuildingDisabler 9](#_Toc174448958)

[BuildingEnabler 9](#_Toc174448959)

[OneOffBuildingResourceSettler 9](#_Toc174448960)

[OperationBalanceSettler 9](#_Toc174448961)

[MinionBalanceSettler 9](#_Toc174448962)

[OneOffMinionResourceSetller 9](#_Toc174448963)

[General UI System 10](#_Toc174448964)

[UIManager 10](#_Toc174448965)

[Static Canvas 10](#_Toc174448966)

[DynamicCanvas 10](#_Toc174448967)

[DynamicUIElement 10](#_Toc174448968)

[MenuDrawer<T> 10](#_Toc174448969)

[Card<T> 10](#_Toc174448970)

[DiscardConfirmationPanel 11](#_Toc174448971)

[DescriptionSystem 11](#_Toc174448972)

[DescriptionSystem 11](#_Toc174448973)

[DescriptionWindow 11](#_Toc174448974)

[DescriptionProperty 11](#_Toc174448975)

[Building System 12](#_Toc174448976)

[BuildingSystem 12](#_Toc174448977)

[BuildingSystemManager 12](#_Toc174448978)

[BuildingSystemMenu 12](#_Toc174448979)

[ConstructionPanel 12](#_Toc174448980)

[PlaceableObjectManager 12](#_Toc174448981)

[PlaceableObjectManager 12](#_Toc174448982)

[ConstructionTaskIssuer 13](#_Toc174448983)

[PlacementSystem 13](#_Toc174448984)

[PlaceableObject 13](#_Toc174448985)

[ConstructionHandler 13](#_Toc174448986)

[UndergroundConstrcutionHandler 14](#_Toc174448987)

[CollisionHandlingZone 14](#_Toc174448988)

[TileSelector 14](#_Toc174448989)

[PlacementCondition 15](#_Toc174448990)

[Placeable Object Prefab Configuration 15](#_Toc174448991)

[PlaceableObjectMenuDrawer 15](#_Toc174448992)

[PlaceableObjectCard 16](#_Toc174448993)

[Buildings 16](#_Toc174448994)

[Building 16](#_Toc174448995)

[ResourceYieldingBuidling 16](#_Toc174448996)

[WorkingPlace 16](#_Toc174448997)

[ObjectSpawningBuilding 16](#_Toc174448998)

[BuildingData 17](#_Toc174448999)

[UndergroundEntrance 17](#_Toc174449000)

[ClearanceZone 17](#_Toc174449001)

[NoneIdleZone 17](#_Toc174449002)

[UndergroundEntranceDoor 17](#_Toc174449003)

[UndergroundEntranceFloor 17](#_Toc174449004)

[RoomManager 17](#_Toc174449005)

[RoomManager 17](#_Toc174449006)

[Selector 18](#_Toc174449007)

[SelectionCounter 18](#_Toc174449008)

[AccessibilityVerifier 18](#_Toc174449009)

[Prebuilder 18](#_Toc174449010)

[Restorer 18](#_Toc174449011)

[RoomData 19](#_Toc174449012)

[RoomMenuDrawer 19](#_Toc174449013)

[RoomCard 19](#_Toc174449014)

[UndergroundConstructionArea (UCA) 19](#_Toc174449015)

[UndergroundConstructionArea 19](#_Toc174449016)

[MainAreaGenerator 19](#_Toc174449017)

[EdgeGenerator 20](#_Toc174449018)

[BottomGenerator 20](#_Toc174449019)

[UCA Blocks 20](#_Toc174449020)

[Block 20](#_Toc174449021)

[ClearableBlock 20](#_Toc174449022)

[EdgeBlock 20](#_Toc174449023)

[BlockPart 20](#_Toc174449024)

[Top 20](#_Toc174449025)

[Wall 20](#_Toc174449026)

[Floor 21](#_Toc174449027)

[UndergroundBase 21](#_Toc174449028)

[UndergroundBase 21](#_Toc174449029)

[RandomPointGetter 21](#_Toc174449030)

[SecurityManager 21](#_Toc174449031)

[HumanResourceManager 21](#_Toc174449032)

[WarehouseManager 22](#_Toc174449033)

[TaskManager 22](#_Toc174449034)

[CanteenManager 22](#_Toc174449035)

[WorkingPlaceManager 22](#_Toc174449036)

[ArmoryManager 23](#_Toc174449037)

[PlacedObjectManager 23](#_Toc174449038)

[Tasks 23](#_Toc174449039)

[Task 23](#_Toc174449040)

[EliminationTask 23](#_Toc174449041)

[RangedEngagePositionDistributor 24](#_Toc174449042)

[MeleeEngagePositionDistributor 24](#_Toc174449043)

[DeliveryTask 24](#_Toc174449044)

[ConstructionTask 25](#_Toc174449045)

[WorkingPlaceTask 25](#_Toc174449046)

[LegalForceTask 25](#_Toc174449047)

[AssassinationTask 25](#_Toc174449048)

[SabotageTask 25](#_Toc174449049)

[Recreational Activities 25](#_Toc174449050)

[RecreationalActivity 25](#_Toc174449051)

[Sleeping 25](#_Toc174449052)

[Mountain Occlusion 26](#_Toc174449053)

[OcclusionHandler 26](#_Toc174449054)

[Selection System 26](#_Toc174449055)

[SelectionSystem 26](#_Toc174449056)

[ActionMenu 27](#_Toc174449057)

[ActionButton 27](#_Toc174449058)

[StateIndicationSystem 27](#_Toc174449059)

[StateIndicationSystem 27](#_Toc174449060)

[StateIndicator 27](#_Toc174449061)

[MinionIndicationManager 27](#_Toc174449062)

[Global Operations 27](#_Toc174449063)

[GlobalOperationSystem 27](#_Toc174449064)

[OperationEconomyHandler 28](#_Toc174449065)

[Region 28](#_Toc174449066)

[RegionData 28](#_Toc174449067)

[RegionalOperationBase 28](#_Toc174449068)

[OperationManager 28](#_Toc174449069)

[OperationGenerator 29](#_Toc174449070)

[GlobalOperation 29](#_Toc174449071)

[GlobalOperationData 29](#_Toc174449072)

[GlobalOperationFactory 29](#_Toc174449073)

[GlobalOperationWindow 29](#_Toc174449074)

[GOWDrawer 30](#_Toc174449075)

[RegionDrawer 30](#_Toc174449076)

[OperationDrawer 30](#_Toc174449077)

[GOWProperty 30](#_Toc174449078)

[User Input 30](#_Toc174449079)

[MouseInputHandler 30](#_Toc174449080)

[KeyboardInputHandler 31](#_Toc174449081)

[Camera Movement 31](#_Toc174449082)

[CameraController 31](#_Toc174449083)

[CameraMover 31](#_Toc174449084)

[CameraRotator 31](#_Toc174449085)

[CameraZoomer 31](#_Toc174449086)

[Current Camera Controls 31](#_Toc174449087)

[CameraShifter 31](#_Toc174449088)

[Player 32](#_Toc174449089)

[Player 32](#_Toc174449090)

[HiringManager 32](#_Toc174449091)

[Resources 32](#_Toc174449092)

[ResourceManager 32](#_Toc174449093)

[ResourceProfile 32](#_Toc174449094)

[ResourceDrawer 33](#_Toc174449095)

[ResourcePanel 33](#_Toc174449096)

[DependencyInjection 33](#_Toc174449097)

[Abstract State Machine 33](#_Toc174449098)

[StateMachine<T> 34](#_Toc174449099)

[SuperState<T> 34](#_Toc174449100)

[State<T> 34](#_Toc174449101)

[Humanoids 35](#_Toc174449102)

[Humanoid 35](#_Toc174449103)

[Minion 35](#_Toc174449104)

[EnemyAgent 35](#_Toc174449105)

[HumanoidData 35](#_Toc174449106)

[MinionData 35](#_Toc174449107)

[EnemyAgentData 35](#_Toc174449108)

[External Behavior Trees 35](#_Toc174449109)

[Abstracts 36](#_Toc174449110)

[SetSharedDependencies 36](#_Toc174449111)

[CustomSharedVariables 36](#_Toc174449112)

[HumanoidAction 36](#_Toc174449113)

[HumanoidConditional 36](#_Toc174449114)

[FixedUpdateDrivenAction 37](#_Toc174449115)

[FixedUpdateDrivenConditional 37](#_Toc174449116)

[Commons 37](#_Toc174449117)

[CustomNodes 37](#_Toc174449118)

[Aspects 37](#_Toc174449119)

[AspectPart 38](#_Toc174449120)

[CombatAspect 38](#_Toc174449121)

[MovementAspect 38](#_Toc174449122)

[EscapeAspect 38](#_Toc174449123)

[TaskHandlingAspect 38](#_Toc174449124)

[TaskHandlingPart 38](#_Toc174449125)

[DeliveryPart 38](#_Toc174449126)

[WorkingShiftPart 38](#_Toc174449127)

[RecreationalAspect 39](#_Toc174449128)

[RecreationalPart 39](#_Toc174449129)

[SleepingPart 39](#_Toc174449130)

[EatingPart 40](#_Toc174449131)

[AssassinatedAspect 40](#_Toc174449132)

[DisguiseAspect 40](#_Toc174449133)

[CarryingPoint 40](#_Toc174449134)

[MinionHiringMenu 40](#_Toc174449135)

[MinionHiringMenu 40](#_Toc174449136)

[AvailableMinionMenu 41](#_Toc174449137)

[CharacteristicPanel 41](#_Toc174449138)

[CountPanel 41](#_Toc174449139)

[MinionCard 41](#_Toc174449140)

[Spawners 41](#_Toc174449141)

[Spawner<T> 41](#_Toc174449142)

[MinionSpawner 41](#_Toc174449143)

[EnemyAgentSpawner 41](#_Toc174449144)

[CarriableObjects 41](#_Toc174449145)

[CarriableObject 41](#_Toc174449146)

[ConstructionBox 42](#_Toc174449147)

[Dynamite 42](#_Toc174449148)

[Beds 42](#_Toc174449149)

[Bed 42](#_Toc174449150)

[BunkBed 42](#_Toc174449151)

[Tables 43](#_Toc174449152)

[Table 43](#_Toc174449153)

[Chair 43](#_Toc174449154)

[Covers 43](#_Toc174449155)

[Cover 43](#_Toc174449156)

[HorizontalCover 43](#_Toc174449157)

[VerticalCover 43](#_Toc174449158)

[Food 43](#_Toc174449159)

[Hamburger 43](#_Toc174449160)

[Weapons 44](#_Toc174449161)

[Weapon 44](#_Toc174449162)

[WeaponData 44](#_Toc174449163)

[RangedWeapon 44](#_Toc174449164)

[RangedWeaponData 44](#_Toc174449165)

[Bullet 44](#_Toc174449166)

[LegalForces 44](#_Toc174449167)

[LegalForceBrain 44](#_Toc174449168)

[TaskGenerator 44](#_Toc174449169)

[DisguiseManager 44](#_Toc174449170)

[Appearance 45](#_Toc174449171)

[Grid 45](#_Toc174449172)

[What if we want to change the cell scale? 45](#_Toc174449173)

[Utilities 45](#_Toc174449174)

[MonoBehaviourPool<T> 45](#_Toc174449175)

[AnimatorParameters 46](#_Toc174449176)

[Serializable Dictionary Scripts 46](#_Toc174449177)

[ReactiveProperty<TValue, TOwner> 47](#_Toc174449178)

[Current Problems 47](#_Toc174449179)

[Paused Moment of Development 48](#_Toc174449180)

# GameManager

## GameManager

Currently GameManager serves for the purpose of destruction of placed PlaceableObjects allowing to avoid errors upon exiting the play mode. GameManager also triggers UIManager to initialize the correct UI system on start. It also has GeneralBalanceSettler that handles all regular resource incomes and expenses

For development purposes, GameManager currently depicts FPS on the screen and handles Application.Quit.

## PlacedObjectLog

PlacedObjectLog registers all placed PlaceableObjects and destroys them on command.

## GeneralBalanceSettler

GeneralBalanceSettler is a class that acts as a head accountant. It has several BalanceSettlers in its dependencies. When it is time to add and take resources from the player, GeneralBalanceSettler tells each BalanceSettler to draw a balance for their respective fields. GeneralBalanceSettler knows about the Player’s ResourceManager and interacts with it regarding regular resource incomes and expenses, as well as asks to draw their projections. In general GeneralBalanceSettler does not handle global operation acceptance and construction costs, as well as global operations’ completion rewards.

By closing a resource balance we mean that the balance - which is a difference between a received amount of the resource and an amount of a resource spent within one time step - is either added or deducted from the stockpile of this resource. Whether it is added or deducted depends on whether the income of the resource is more than the consumption.

If a resource balance is not negative, then GeneralBalanceSettler asks the Player’s ResourceManager to add the result amount. If a resource balance is negative, but the resource stockpile allows closing the balance, then the balance will be deducted from the stockpile. If the resource stockpile doesn’t allow to close the balance, then GeneralBalanceSettler will command to its dependency BuildingBalanceSettler to disable some buildings based on the available resources.

If all constructed buildings end up disabled and no global operations are conducted, then GeneralBalanceSettler will stop drawing balances and nullifies projections.

After balances for all available resources are closed, GeneralBalanceSettler calculatea a projection for resource consumption for the next time step. In accordance with this projection, GeneralBalanceSettler will command to BuildingBalanceSettler to try enabling some buildings that has previously being disabled due to a resource shortage.

## BuildingBalanceSettler

BuildingBalanceSettler is a class responsible for managing the economy of constructed buildings. It receives notification every time when placement of a new Building is confirmed by the player. This class deals with Buildings, not PlaceableObjects (see the difference between the two in Building paragraph).

BuildingBalanceSettler is a dependency of GeneralBalanceSettler. This class calculates resource balances every fixed time step which is received from GeneralBalanceSettler.

## BuildingDisabler

BuildingDisabler is a dependency of BuildingBalanceSettler. Its primary function is searching for and disabling buildings which the player can not afford to maintain. Currently its search algorithms implement only one pattern: BuildingDisabler prioritizes disabling a Building which consumes a bigger amount of a resource than other buildings and at the same time does not yield anything in return.

## BuildingEnabler

BuildingEnabler is a dependency of BuildingBalanceSettler. Its primary function is trying to search and enable back buildings that are currently registered as disabled. BuildingEnabler currently does not prioritize enabling resource yielding buildings over others. It will enable first buildings in the list of disabled buildings sent to it as long as resource projections allow that.

## OneOffBuildingResourceSettler

OneOffBuildingResourceSettler is a class that does not inherit from BalanceSettler, because it does not need to periodically calculate balances of resources. Its main purpose is invoke events requiring addition and subtraction of certain amount of a resource. This class works with ResourceYieldingBuildings whose yielding type is set to OneOff.

## OperationBalanceSettler

Since the balance for global operations is implied to be always positive (see GlobalOperation section for more detail), OperationBalanceSettler is much simplier that BuildingBalanceSettler. Like its colleague, OperationBalanceSettler draws up balances for its field of responsibility when told be GeneralBalanceSettler.

## MinionBalanceSettler

MinionBalanceSettler is another heir to BalanceSettler. It handles calculating salaries of the minions that the player currently has.

## OneOffMinionResourceSetller

OneOffMinionResourceSettler is similar to OneOffBuildingResourceSettler, it does not calculate regular balances, but deals with adding and reducing minion’s count at the lower panel of the UI in the game.

# General UI System

## UIManager

Currently the UI system of the project is underpinned by class UIManager whose responsibility is to toggle BuildingSystem and SelectionSystem not allowing them to work simultaneously.

## Static Canvas

Static Canvas is a usual canvas renamed to Static. It holds all UI elements that do not move on canvas in runtime.

## DynamicCanvas

Unlike Static Canvas, DynamicCanvas is a class attached to a canvas. This class holds all DynamicUIElements that are supposed to change their positions on the canvas in runtime. The reason for this separation is that a whole canvas is redrawn every time when any detail on it changes.

## DynamicUIElement

DynamicUIElement is a base abstract class for all UI elements that must be placed on DynamicCanvas. Be mindful that a DynamicUIElement must invoke TurnedOff action when you are turning it off so that DynamicCanvas could properly remove it.

## MenuDrawer<T>

MenuDrawer<T> is a base abstract class for menus with construction options. It is used by a specific BuildingSystemManager to identify what exactly the user wants to build next.

MenuDrawer<T> must receive a list of available objects from its BuildingSystemManager, and then it compiles a menu which must be placed on Static Canvas. The menu itself should be a scroll view. When MenuDrawer<T> compiles the menu, it adds Cards<T> to the content of the scroll view. When all the Cards<T> are added, MenuDrawer<T> commands them to create a toggle group so that user could select only one Card<T> at a time.

T must be the type of objects which will be sent by the BuildingSystemManager.

## Card<T>

Card<T> is base abstract class for cards describing building options to the user. Card<T> requires a Toggle component. When selected, Card<T> informs its MenuDrawer<T> of T MainObject. T of Card<T> must be the same T as in MenuDrawer<T>.

Card<T> is also used for MinionCard which let other systems know which minion is currently selected (for more detail, please, refer to MinionHiringMenu section).

## DiscardConfirmationPanel

DiscardConfirmationPanel is a class and a UI Element on Static Canvas. Its purpose is inquiring extra confirmation of the user’s actions. It requires a warning with a clear “yes” or “no” answer and can be used universally.

# DescriptionSystem

## DescriptionSystem

Class DescriptionSystem is meant to show DescriptionWindow when the mouse pointer stays over a describable object. Besides that, DescriptionSystem is used to show detailed info on a built PlaceableObject through ActionMenu. If multiple PlaceableObjects are selected in SelectionSystem, DescriptionWindow will not be compiled, as it is meant to work only for one object at a time.

By default DescriptionSystem does not work in the manual mode, meaning that it will show DescriptionWindow with a delay and with the close button disabled. If isManualMode argument in TryCompileFor method is true, then DescriptionWindow will appear immediately, its MovementType will be set to FixedToWorldPosition and the user will be able to close it by pressing the close button or moving the camera beyond the initial world position of the DescriptionWindow.

## DescriptionWindow

DescriptionWindow is an heir to DynamicUIElement. It shows the description of an object after it has been placed on the Dynamic Canvas. Draw method in DescriptionWindow script must implement realization for specific ScriptableObjects which are sent to it, otherwise, it fires an error. By implementation we mean that it should be defined which properties must be serialized for a specific type of ScriptableObject in the property scroll view of DescriptionWindow.

## DescriptionProperty

DescriptionProperty is a simple panel with two text fields, one of which is for the name of the property and the second for its value. The scroll view located on the DescriptionWindow is populated with DescriptionProperties.

# Building System

## BuildingSystem

Class BuildingSystem separates subscriptions and updates between BuildingSystemManagers which are responsible for more specific tasks. It also reads and relays user inputs.

Besides that, BuidlingSystem handles dependency injection for its managers. It also serves as a bridge to relay requests like replacing a built object or building a room to one of BuildingSystemManagers.

## BuildingSystemManager

Class BuildingSystemManager is abstract and describes common logic for all managers which will be included into BuildingSystem.

## BuildingSystemMenu

Class BuildingSystemMenu links user inputs to BuildingSystem. It allows BuildingSystem to be aware what type of objects the user wants to build next (currently, only two implemented: PlaceableObject and RoomData) and activate the responsible BuildingSystemManager.

## ConstructionPanel

Class ConstructionPanel is responsible for reading confirm and discard user inputs. The panel also has some text fields which depict information on construction costs, however, this information is filled in by responsible BuildingSystemManager.

# PlaceableObjectManager

## PlaceableObjectManager

Class PlaceableObjectManager is an heir to BuildingSystemManager. Upon the user’s selection it updates the new placeable object’s position and calls to assess the placeability of the position. (PlaceableObjectManager does not itself build placeable objects or place them on the grid!). PlaceableObjectManager catches the user’s input related to rotation of a selected placeable object and upon Left MouseKey click commands to try to place the selected placeable object. PlaceableObjectManager holds the list of placeable objects available to the player. It is also responsible for handling replacement of placed placeable objects.

As soon as the player confirms the construction of placed objects, PlaceableObjectManager commands ConstructionTaskIssuer to handle task registration. registers a task with TaskManager for each placed object, if this object is set to be constructed through a task in PlaceableObjectData.

## ConstructionTaskIssuer

ConstructionTaskIssuer is a class and dependency of PlaceableObjectManager. It checks if a placed object is constructed through a task completed by minions or can be constructed right away (this property can be changed in PlaceableObjectData).

Once ConstructionTask for a specific PlaceableObject is completed, ConstructionTaskHandler will finalize the construction of this object. At the same time if the player decides to delete the object before it is constructed, ConstructionTaskIssuer will cancel the ConstructionTask for this object.

## PlacementSystem

PlaceableObjectManager has class PlacementSystem in its dependencies. PlacementSystem is responsible for projecting a placeable object on the grid. The projection is carried out in the way that it guarantees that tile selectors, which actually assess placeability, are always located above the grid. Beside that function, PlacementSystem logs all occupied cells of the grid, therefore PlacementSystem must always be single on the scene which is ensured by Zenject library.

## PlaceableObject

Class PlaceableObject is responsible for trying to place and build itself. PlaceableObject has a SO instance PlaceableObjectData which contains all necessary data for a placeable object such as type by location, construction costs, materials before and after construction and so on. PlaceableObject also implements interfaces required by SelectionSystem such as ISelectable, IDestroyable, IReplaceable, IDescribable.

PlaceableObject relays user and non-user commands to its children, primarily to ConstructionHandler. PlaceableObject has a complex hierarchy and requires multiple child components to work properly - NavMeshObstacle, SelectionIndicator, CarryingPoint, TileSelectors etc.

In order to properly construct PlaceableObject examine prefabs in Prefabs/PlaceableObjects.

## ConstructionHandler

ConstructionHandler is a class responsible for checking constructability of the selected spot, placing the PlaceableObject on the grid and finalizing the construction.

When PlaceableObject selected and called upon by PlaceableObjectManager, PlaceableObject commands to its ConstructionHandler to verify positions of the tile selectors. If all the tile selectors return true and the user presses LeftKey ConstructionHandler will place the PlaceableObject on the spot.

Currently PlaceableObject can have one of two types of ConstructionHandlers in its dependencies. The choice is made automatically based upon the indicated type by location in PlaceableObjectData. Though such decision is not great from the architectural point of view, but it allows inheriting Building class from single PlaceableObject class to keep all common functions of future multiple heirs of Building in one place.

## UndergroundConstrcutionHandler

In addition to the main functionality of ConstructionHandler, UndergroundConstructionHandler also digs trenches and cleans them up upon placement and destruction of its placeable object. Currently this script is used only for underground base entrances. UndergroundConstructionHandler alters the heightmap of a terrain to create a trench. The trench must have a form of a quad (otherwise, mathematical calculations will be too heavy). The DigPoint is the center of the quad. The ChangedDiameter is the size of one side of that quad (reflects the width and height of the future trench). The ChangedDiameter must be set as an integer, there will be no point indicating floats, since a heightmap accepts only integers when referring to its points. NewHeight sets the depth of the trench (the more the value the deeper the trench). Since a heightmap accepts integers, its important not to set a low resolution of a heightmap. The recommended value is 2049 x 2049 for the standard 1000 block terrain. Assigning a lower resolution will result that the trench will not fit perfectly the placeable object whatever ChangedDiameter is indicated. The terrain must be 1000x1000 blocks, otherwise the trench will not be dug properly. The terrain also must be placed in 0,0,0 coordinates, otherwise when the script tries to get heights of the heightmap, Unity returns OutOfBound Exception, meaning that we are trying to get heightmap data where there is no heightmap.

## CollisionHandlingZone

CollisionHandlingZone is a MonoBehaviour script which is added to some PlaceableObjects and its heirs in order to prevent certain collisions when target GameObjects are inside this Zone.

For example, when minions are going to take their seats at the dining table, they may collide with each other and with the table which does not allow them to reach the assigned chairs, so this component is used to limit their collisions.

## TileSelector

TileSelector is a class and prefab frame placed under a placeable object prefab to check the grid cell whether it is appropriate for building. TileSelector is a base abstract class for ActualTileSelector and LinkedTileSelector. ActualTileSelector has a list of PlacementConditions according to which it checks placeability. Since many of created placement conditions are based on raycasts which is an expensive operation , in order to raise efficiency we created LinkedTileSelector which does not check cells and just receive bool events from the ActualTileSelector it is linked to. We recommend using LinkedTileSelector for cells which are under the center of a placeable object prefab.

## PlacementCondition

PlacementCondition can be added only to ActualTileSelector. PlacementConditions must be added to the list of ActualTileSelector otherwise they will not be called. PlacementConditions are called in the order they are added to the above-mentioned list and if one condition fails the rest are not checked for optimization purposes. PlacementCondition is a base abstract class for:

* SpecificCellCondition (checks the layer of the surface and height to that surface)
* LandscapeNormalCondition (checks that the surface is not a slope and does not have steep deviations)
* UndergroundPlacementCondition (compares the height of the terrain above the tile selector and compares it with the height of the terrain above a tile selector which is placed on the ground)
* OpenAirPlacementCondition (checks that there is nothing above a game tile selector*. Note that a layer mask must be indicated for this placement condition. Otherwise, the overlapping sphere will collide with the building itself*)

## Placeable Object Prefab Configuration

The root transform of a placeable object prefab must have Building component on it. The root transform also must be located on the same Y level with the tile selectors of the placeable object prefab, since when instantiated by BuildingSystem a new placeable object is projected on the grid by PlacementSystem via its root transform. If placed under the tile selectors or significantly above them the placeable object might be placed incorrectly in the world. A placeable object prefab also must have at least one tile selector and only one selection indicator. Examples of properly prepared placeable object can be found in the folder Prefabs/PlaceableObjects.

## PlaceableObjectMenuDrawer

PlaceableObjectMenuDrawer is an heir to MenuDrawer<T>. It receives the list of available placeable objects from PlaceableObjectManager and compiles a placeable object menu which must be placed on Static Canvas. The building menu is a panel with a scroll view. When PlaceableObjectMenuDrawer compiles the menu, it adds PlaceableObjectCards to the content of the scroll view. When all cards are added, PlaceableObjectMenuDrawer commands them to create a toggle group so that user could select only one PlaceableObjectCard at a time.

## PlaceableObjectCard

Class PlaceableObjectCard is an heir to Card<T>. It holds specific PlaceableObjectData and when selected informs PlaceableObjectMenuDrawer of its PlaceableObjectData.

# Buildings

## Building

Building is an heir of PlaceableObject and a base abstract class for buildings. The main difference between Building and PlaceableObject is that a Building is supposed to do something, for example, yield resources, spawn units, improve infrastructure etc. So if a Building is not engineered to do such things, it must be downgraded to a PlaceableObject. Building has IsWorking property to designate its current status. When a Building is being replaced, this property is set to false.

In addition, since a Building does something, it must consume some resources, which PlaceableObjects do not. This is another vivid aspect which allows distinguishing between Buildings and PlaceableObjects when designing prefabs.

## ResourceYieldingBuidling

ResourceYieldingBuidling class is an heir to Building. All buildings that spawn resources must be of this type. The class has ResourceYieldingData that indicate which type of resources will be yielded and in what amount as well as whether yielding occurs regularly or only once.

## WorkingPlace

WorkingPlace is an heir to ResourceYieldingBuilding. Unlike its parent class which starts spawning resources once it is constructed, WorkingPlace must have an assigned minion to start producing its resource. \_stopWorkingThreshold parameter indicates a time period in seconds after which WorkingPlace deactivates and stops production unless a Humanoid is assigned to continue the work.

## ObjectSpawningBuilding

ObjectSpawningBuilding class is another heir to Building. It is designed to spawn game objects by their types. An order can be placed with ObjectSpawningBuilding, which returns an order number. Objects are not spawned all together, the next order will not be spawned until the previous order is removed from the dispatch zone and cooldown, if any, is passed.

If ObjectSpawningBuilding does not have a prefab with the required component type, then the returned order number is 0.

On Awake ObjectSpawningBuilding creates object pools populated with clones of each prefab it has. ObjectSpawningBuilding returns first matched disabled object or, if not found, spawns a new one from the appropriate prefab. It is implemented for optimization purposes.

## BuildingData

BuildingData is a SO which will contain common base characteristics for all buildings.

## UndergroundEntrance

UndergroundEntrance is an heir of Building. It informs the UCA of the blocks that must be cleared for the initial zone where minions will be able to walk. It also holds Open and Close methods to let minions pass through.

## ClearanceZone

ClearanceZone is a class responsible for detecting blocks to clear in the initial area in a UCA after an entrance has been placed. The zone is drawn in the editor mode by OnDrawGizmos method.

## NoneIdleZone

NonIdleZone is a class required to set a zone around an UndergroundEntrance where minions can not walk idly. If a random point within UndergroundBase generated by RandomPointGetter is within a NonIdleZone of an UndergroundEntrance, then it will be regenerated. This is required to prevent minions from staying idly inside an entrance blocking it from closing its doors.

## UndergroundEntranceDoor

This class opens and closes a specific door in an UndergroundEntrance. This class makes use of DoTween and UniTask.

## UndergroundEntranceFloor

UndergroundEntranceFloor detects humanoids that get inside an UndergroundEntrance and let them know of this Entrance and when they leave its boundaries. This component requires a collider in the trigger mode.

# RoomManager

## RoomManager

Class RoomManager is another heir to BuildingSystemManager. This class is responsible for outlining and building rooms. It has Selector, AccessiblityVerifier and Prebuilder in its dependencies which are responsible for carrying out a specific phase of the room building process. Similarly to PlaceableObjectManager, RoomManager holds the list of all rooms that are available for construction.

## Selector

Class Selector catches the start block of the user’s selection, then draws a rectangular overlapping box between the start block and the last block under the mouse pointer. All ClearableBlocks that get inside the overlapping box get selected. However, if selection starts with a block that has already been selected, all blocks inside the selected zone will be deselected. This allows the user to flexibly change the geometry of the future room. When requested by RoomManager, Selector returns all currently selected ClearableBlocks.

## SelectionCounter

Class SelectionCounter is a dependency of Selector. Its solo responsibility is counting rows and columns of temporary and current selections of Selector. These data are depicted on UI so that the user could see how many blocks are selected and how much construction of the currently outlined room will cost.

## AccessibilityVerifier

AccessibilityVerifier checks whether ClearableBlocks that are sent to it are accessible, meaning that they have a neighbor with a disabled ClearableBlock.Renderer. This guarantees that the player will not build a new room detached from other rooms, corridors or gates. If some blocks are inaccessible, they will be tinted in red by RoomManager.

## Prebuilder

Class Prebuilder prebuilds a room out of blocks provided to it. Prebuilder sets floors and walls of the future room and makes them partly transparent. When the player confirms construction of the outlined room, Prebuilder makes the walls and floor opaque.

## Restorer

Class Restorer is dependency of Prebuilder. This class is responsible for saving information on modified blocks and restoring their appearance to the initial form on command. When a new batch of blocks is sent to Prebuilder, it commands Restorer to registers the blocks and their configuration. If some blocks that were previously sent to Prebuilder are missing in the next batch, Restorer restores their original configuration. The same happens if the player discards changes. Upon confirmation or discard Prebuilder tell Restorer to clear all the information on the previous configurations.

## RoomData

RoomData is a SO class that holds the main information on the room: its name, its type, wall material, floor material etc.

## RoomMenuDrawer

Class RoomMenuDrawer is an heir to MenuDrawer<T>. It is responsible for drawing RoomCards and relaying to RoomManager RoomData of the selected card.

## RoomCard

Class RoomCard is an heir to Card<T>. It holds specific RoomData and, when selected, informs RoomMenuDrawer of its RoomData.

# UndergroundConstructionArea (UCA)

## UndergroundConstructionArea

UndergroundConstructionArea is a class meant to automatically generate a buildable area for a base under any mountain. IntialYPosition must equal the height of the terrain, otherwise overlapping spheres are likely not to hit anything. DownOffset is the offset by which UCA will go down when generated. DownOffset must be an integer since placeable objects will be placed on a grid which accepts only integers. It is required to perfectly align the UCA with the bottom of the underground entrance. UCA depends on MainAreaGenerator, EdgeGenerator and BottomGenerator. UCA is stored as a prefab, just place it under a mountain and click Create button to generate a UCA. DeleteAll button will delete all child objects of the UCA.

In runtime UCA registers placed UndergroundGates and commands the building system to establish first room of corridor type, so that the player could expand the base.

## MainAreaGenerator

MainAreaGenerator creates the main buildable zone of UCA. First, it generates a central block, then attaches to it additional blocks from each side on the global Z-axis (builds two half rows). MaxHalfRowLength tells how many blocks should be placed within each side of a central block. If you see that rows do not reach edges of the mountain on the global Z-axis, then you should increase this value. When the first row is finished, MainAreaGenerator will divide it in sections and build columns to align the zone with the geometry of the mountain. If you notice, that some areas on the global X-axis are not reached by the generated zone, then you should increase MainColumnCount. It must be an odd number!

## EdgeGenerator

EdgeGenertor will find edges of the zone generated by MainAreaGenerator. It can also add extra rows of edges. But extra rows will be always displaced downward so that their top would almost align with the bottom of the UCA.

///currently unrequired since humanoids move on Floors of ClearableBlocks

## BottomGenerator

BottomGenerator generates a plane, then uses spherecasts to find edges of the mountain and rebuilds the mesh of the plane, so that it would align with the mountain. If you want to get more precise geometry of the bottom, you can decrease AngleStep and DistanceStep. If the bottom does not reach the edges of the mountain, then you should increase MaxDistance.

# UCA Blocks

## Block

Block is a base abstract class for building blocks of UCA. It holds a bunch of appearance change methods that are used by Prebuilder to customize a room as well as references to Walls.

## ClearableBlock

ClearableBlock is the main block of UCA which can be turned off during room building process. In addition to Wall references, it also must have a Top and a Floor.

In order for a clearable block to be able to work as an obstacle for Humanoids, it must have NavMeshObstacle component on it.

## EdgeBlock

EdgeBlock can not be selected by Selector and turned off in room construction and must outline the entire UCA.

## BlockPart

Class BlockPart is a base abstract class for Wall, Floor and Top. Like Block, it has some common methods for appearance customization. All heirs to BlockPart (Top, Wall, Floor) must have a transparent material on the gameobjects!

## Top

Class Top acts as a selection indicator for ClearableBlocks. Its color is also changed by RoomManager depending on feedback provided by AccessibilityVerifier.

## Wall

Class Wall marks walls of a ClearableBlock that will be customized to match a particular room type. *Be mindful that wall gameobjects must be attached to a Block prefab in accordance with the local transform directions (the front wall must be on the side to where transform.forward points and so on).*

## Floor

Similarly to Wall, class Floor is required to customize a room. It also holds information on what type of the room it belongs to. This is necessary for creating another PlacementCondition for TileSelector.

The layer of Floor must be included into layers from which NavMeshSurface is baked. In this case when a room is built, Humanoids will be able to walk freely inside the base.

# UndergroundBase

## UndergroundBase

UndergroundBase is a main class which holds links to all managers that will carry out specific aspects of control over the base. UndergroundBase will register all changes with the base and inform its dependencies of them. Currently it reads events of UCA when a new room or an UndergroundEntrance is constructed or demolished.

## RandomPointGetter

This class is a dependency of UndegroundBase and receives all built rooms and entrances from it. The main task of this class is finding a random point inside the base or in close proximity to one of its entrances. This class gives minions positions when they have no task at hand and idle walk around the base.

## SecurityManager

SecurityManager is another dependency of UndergroundBase. Currently it controls access to UndergroundEntrances. Any minion that intends to walk inside or outside must file a PassApplication (struct). At the moment all minions are let through. But additional verification logic can be added in future.

## HumanResourceManager

HumanResourceManager is another dependency of UndergroundBase. It is responsible for keeping track of all minions that the player has under his command. Its primary function is to return IReadOnlyList of all minions to other scripts. HumanResourceManager is not supposed to select specific minions for a specific Task.

HumanResourceManager is also notified of all placed beds. Every frame HumanResourceManager loops through all the registered minions and, if finds anyone without a bed, assign a vacant one.

## WarehouseManager

WarehouseManager is responsible for keeping track of all ObjectSpawningBuildings that the player places and removes.

It also has a public method that takes the position of a minion and returns the most suitable ObjectSpawningBuilding. Any logic can be implemented in this method. Currently it determines the best warehouse not only from the perspective of distance to cover, but also from the perspective of order counts that the registered warehouses have. If no warehouses are built, then the method returns null.

## TaskManager

TaskManager handles all the Tasks issued by the player or scripts. It has a public method that allows registering a new Task. Using UniTask, every frame Task Manager checks if there are any pending Tasks – Tasks that have been registered but have no minions to execute them. If such a Task is found, TaskManager requests the list of available minions from HumanResourceManager and chooses the most appropriate ones to assign the Task to them.

If TaskManager finds a Task whose minions are unable to complete it (they can be dead or running away), then TaskManager will assign new minions to vacant positions of the Task.

In general TaskManger tries to find idling minions that are not engaged in a Task or a RecreationalActivity, but, if a Task’s priority equals or is higher than \_absolutePriority, TaskManager can interrupt RecreationalActivities or other less important Tasks.

## CanteenManager

CanteenManager is another dependency of UndergroundBase. It keeps track of all spawned Tables where Humanoids can take their meals. It assigns Tables and Chairs to Humanoids upon request and must be notified when a Humanoid finishes its meal.

CanteenManager also contains prefabs of all available food and distributes its instances upon request.

## WorkingPlaceManager

WorkingPlaceManager is a class that registers all spawned WorkingPlaces and issues WorkingPlaceTasks to fill these Places with minions. It is important to remember that WorkingPlaceManager only issues the tasks and places them with TaskManager, but the latter is the one which finds specific minions for execution.

After WorkingPlaceManager issues a Task for a specific WorkingPlace it keeps asking TaskManager whether a Humanoid has been assigned to execute it. Once it is assigned, WorkingPlaceManager registers this Humanoid with WorkingPlace itself. Without that a WorkingPlace does not know that some is working on it and does not spawn resources.

## ArmoryManager

ArmoryManager is a class that holds all weapons available for minions and returns one upon a request.

## PlacedObjectManager

PlacedObjectManager is UndergoundBase’s dependency that keeps track of PlaceableObjects that the player places on the map. It can also return the list of these objects.

# Tasks

## Task

Task is a specific operation that is carried out by minions, like target elimination, delivering an object etc. Task is an abstract class. After it has been created, every frame Task checks whether its completion conditions have been met. These conditions are set for each heir to this class separately.

Every Task also holds information on the minions that carry it out. Task is supposed to receive this information from TaskManager when it assigns minions. This feature, though being a doubtful architectural solution, allows simplifying the way to determine which minions exactly are engaged with the particular Task.

The Transform dependency of the Task is designed to be a specific point where the Task must be executed. For example, for DeliveryTask Transform is the point to where the delivery object must be brought.

Each heir of Task must have a separate behavior tree implemented for a minion or an enemy to perform some actions to complete the Task. For each heir of Task there is also must be a specific number of FatiguePoints which are deducted from Humanoid.RecreationalAspect.CurrentActionPoints, once the Task is completed.

## EliminationTask

EliminationTask is an heir to Task that implies elimination of a specific Humanoid. EliminationTask has a method that issue engage indices for minions who are trying to assume it. An engage index is required to determine the minion’s position around the target. If after requesting an engage index a minion receives 0, it means that the maximum number of executors has already been achieved.

EliminationTask keeps track of executors, and if an executor dies, then the Task makes its engage index vacant. But if an executor starts running away from the fight, then it will be removed and its position is reassigned to another minion by TaskManager.

EliminationTask ends when the target is eliminated or has escaped.

## RangedEngagePositionDistributor

RangedEngagePositionDistributor is a dependency class of EliminationTask. Minions that have a ranged weapon call this class to get an engage position.

When searching for a proper engage position, this class makes sure that a Minion will stand at the best distance from the target, that there are no other Minions, buildings or covers in the fire line. It also samples the potential position to the NavMesh to guarantee that a Minion will be able to reach this position and helps to find engage positions in narrow spaces.

If the class fails to find an appropriate engage position, it returns Vector3.zero.

## MeleeEngagePositionDistributor

MeleeEngagePositionDistributor is another dependency of EliminationTask which is almost similar to RangedEngagePositionDistributor, but contains a simpler logic for position calculation due to a simpler nature of close combat fighting.

This class helps to determine a proper fighting position in narrow spaces and makes sure that a Minion will not attack the target through a cover, if the target has one.

If the class fails to find an appropriate engage position, it returns Vector3.zero.

## DeliveryTask

DeliveryTask is another heir to Task. Its completion condition is delivering an object of the specified type to its Transform. The delivery object itself is unknown when DeliveryTask is created, so it must be assigned later on when a minion receives an object of that type from an ObjectSpawningBuilding. DeliveryTask ends when the delivery object is brought to the designated position and used (currently only CarriablleObjects are used to issue a DeliveryTask).

## ConstructionTask

ConstructionTask is a class which is issued when a PlaceableObject must be constructed to its final form by minions. However, the important part is that ConstructionTask doesn’t finalize construction of PlaceableObjects, which is handled by ConstructionTaskIssuer. However ConstructionTask represents conditions which must be met for the construction finalization to happen.

Currently ConstructionTask is almost identical to DeliveryTask, but ends with using the delivered CarriableObject.

## WorkingPlaceTask

WorkingPlaceTask is an heir of Task which is issued for WorkingPlaces by WorkingPlaceManager. Besides inherited parameters, its constructor requires a reference to a WorkingPlace.

WorkingPlaceTask is meant to be an imitation of a working shift, so it is successfully over when the shift time is over.

## LegalForceTask

LegalForceTask is an abstract heir to Task that is specifically created for Tasks that are assigned to EnemyAgents and meant to harm the player.

## AssassinationTask

AssassinationTask is a LegalForceTask for killing a minion.

## SabotageTask

SabotageTask is a LegalForceTask for causing a malfunction on a PlaceableObject.

# Recreational Activities

## RecreationalActivity

RecreationalActivity is a abstract class that holds general information on all RecreationalActivities. Currently it has only abstract property that returns the number of points which must be restored while this activity is executed.

## Sleeping

Sleeping is an heir to RecreationalAcititvity which is executed by a humanoid when its CurrentActionPoints are below a designated threshold. All of this is determined and handled by RecreationalAspect.

# Mountain Occlusion

## OcclusionHandler

OcclusionHandler is a class that finds all OcclusionZones in the scene. When the CameraContoller invokes the CameraMoved event OcclusionHandler loops through all the found zones and detects the closest one to the camera. For the sake of smooth transitions between different occlusion zones OcclusionHandler interpolates between the previous cutoff position and the new occlusion zone. To influence this interpolation you can change the LerpRate parameter. In late update OcclusionHandler updates occlusion based on the parameters of the closest occlusion zone.

*OcclusionZone*

OcclusionZone is a class that holds parameters of occlusion for the surrounding region. CutoutMaterial assigned to a zone must have CutoutShader on it, otherwise occlusion will not work. CutoutMaterial accepts a world position, finds pixels around it in a designated radius defined by CutoutSize and nullifies their alpha. FalloffSize allows smoothing the cutout by leaving some edge pixels visible. This value must be increased to make edges of the cutout opaquer. CutoutDistance represents a benchmark how close the camera must approach the occlusion zone for the cutout effect to start working. Based on this value CutoutSize is recalculated every time when OcclusionHandler updates occlusion. Basically, in order to change the size of occlusion and adjust the distance when the player will be able to see the occlusion effect, altering CutoutDistance will suffice.

# Selection System

## SelectionSystem

SelectionSystem allows the user to click on an object, select it and see possible further actions. SelectionSystem is primarily active when BuildingSystem is off, however, BuildingSystem fires a bool event that is listened to by UIManager. According to this event, UIManager toggles SelectionSystem to work simultaneously with BuildingSystem.

In general when user hits the left mouse key SelectionSystem tries to find ISelectable object on the mouse position. If the object is found successfully, SelectionSystem sends a draw command and activates ActionMenu.

## ActionMenu

ActionMenu receives the object from SelectionSystem and tries to find every implemented interface (IDestroyable, IReplaceable, IDescribable etc.) on the object. If an interface is detected, ActionMenu instantiates the prefab of ActionButton which is among its dependencies, and sends it a UnityAction to perfom. ActionMenu also keeps track of all created ActionButtons and clears them when an action is performed or the selected object is deselected.

## ActionButton

ActionButton requires the Button component and performs the action assigned by the ActionMenu on the user’s click. When the action is performed, ActionButton invokes an event that makes ActionMenu destroy all the instantiated ActionButtons and close itself until a new object is selected.

# StateIndicationSystem

## StateIndicationSystem

StateIndicationSystem is another UI system. Its main purpose is visualizing states of buildings and units by placing a StateIndicator over them on DynamicCanvas. This script should contain all sprites for all states that maybe assigned to indicated objects, so it makes use of SerializableDictionary (see Utilities section for more detail). Placed StateIndicators can be turned off though a toggle.

## StateIndicator

StateIndicator is an heir to DynamicUIElement and a prefab which consists only of UI image.

## MinionIndicationManager

MinionIndicationManager is a class that tells StateIndicationSystem to assign StateIndicators over minion and chooses a StateType (enum) for them.

# Global Operations

By global operations we mean operations that spawn on the GlobalOperationPlatform. The player can accept these tasks to receive resources or other benefits. Global Operations are closely connected with the player’s renown with Legal Forces.

## GlobalOperationSystem

GlobalOperationSystem is the backbone of global operations. It is responsible for determining a gameobject that the player clicks on, and gives respective commands to GlobalOperationWindow. In future GlobalOperationSystem will also interact with the player’s ResourceManager to deduct and add resources earned by or required for a GlobalOperation

## OperationEconomyHandler

OperationEconomyHandler is a dependency of GlobalOperationSystem. Its main purpose is to handle one-off resource incomes and expenses related to GlobalOperations. It also contains methods that check whether the Player has enough resources to unlock a selected region or to start the selected GlobalOperation. Therefore, this class knows about the Player’s Resource Manager and sends relevant requests to it.

Be mindful that OperationEconomyHandler does not handle regular incomes incurred by GlobalOperations. This aspect is handled by GeneralBalanceSettler and its dependencies.

## Region

Region is a class and a game object that is located on the map of GlobalOperationPlatform. The mesh of the region should outline a target area where GlobalOperations and RegionalOperationBases will spawn. Region is responsible for generating random points on its mesh for different objects to be placed on. All functions related to a Region will be executed only if the Region is unlocked by the player.

Region holds information on all **inactive** GlobalOperations within it. Once a GlobalOperation is accepted, it is registered with RegionalOperationBase and should be accessed from there.

## RegionData

RegionData is an heir to ScriptableObject which contains a Region’s name, description, difficulty, unlock requirements and other characteristics.

## RegionalOperationBase

RegionalOperationBase is a MonoBehavior class that is required to place a RegionalOperationBasePrefab. This class also holds information on all **active** GlobalOperations and sends notifications when a GlobalOperation is accepted and completed.

## OperationManager

OperationManager is a dependency of Region. It handles all GlobalOperations spawned within the Region.

## OperationGenerator

OperationGenerator is a dependency of OperationManager. It chooses sites and preferred types of GlobalOperations and spawns them within the Region. OperationGenerator makes sure that GlobalOperations are not located to close to each other or the RegionalOperationBase.

As for the preferred types, OperationGenerator takes into account current and previous GlobalOperations and tries not to spawn GlobalOperations of the same type. OperationGenerator regenerates all GlobalOperations, that are not accepted by the player, after set amount of time.

## GlobalOperation

GlobalOperation is a class and a prefab that gets spawned within a Region. It can be selected by the user. Its main component is ROGlobalOperationData. RO stands for “readonly”. ROGlobalOperationData class allows reading information on characteristics of the GlobalOperation.

Similarly to buildings, global operations periodically yield resources. But unlike buildings, global operations do not incur regular resource expenses. Therefore, the balance for global operations can not be negative by default. If when designing an operation you expect it to constantly take some resources from the Player, then relevant parameters such as GoldIntervalIncome must be made negative when adding such an operation to GlobalOperationFactory.

## GlobalOperationData

GlobalOperationData is a ScriptableObject that holds information on the characteristics of a GlobalOperation such as type, description, rewards and expenses etc. GlobalOperationData is immutable after generation.

## GlobalOperationFactory

GlobalOperationFactory is a class responsible for generating GlobalOperationData. All generated GlobalOperationData get wrapped into ROGlobalOperationData and returned to the OperationGenerator which has requested it.

GlobalOperationFactory holds all descriptions for all operation types as well as reward and expenses base parameters and modifiers. Any new operation type must be first implemented in GlobalOperationFactory. Otherwise, an error will be fired.

## GlobalOperationWindow

GlobalOperationWindow is a UI class that allows reacting to user commands. GlobalOperationWindow has CompileFor method which is currently overloaded to accept Regions or GlobalOperations. Depending on which object is sent to it, GlobalOperationWindow tells its relevant dependency GOWDrawer to draw a UI panel with information on that object and possible actions.

GlobalOperationWindow consists of NameField, DesciptionField, ScrollViewPanel and ButtonPanel. The ScrollViewPanel gets tuned off if no GOWProperties are required for the current object.

## GOWDrawer

GOWDrawer is an abstract base class for any drawer supervised by GlobalOperationWindow. A GOWDrawer contains common methods and variables for all drawers. A separate GOWDrawer must be created when creating another overload for CompileFor method in the GlobalOperationWindow script. Draw method of a GOWDrawer is the place where the configuration of GlobalOperationWindow for a specific object must be defined.

## RegionDrawer

RegionDrawer is an heir to GOWDrawer and defines GlobalOperationWindow’s configuration for any region sent to it.

## OperationDrawer

OperationDrawer is another heir to GOWDrawer and defines GlobalOperationWindow’s configuration for any GlobalOperation sent to it.

## GOWProperty

GOWProperty is a class and a UI panel that holds three text fields and a button and fills in the relevant information into these fields. GOW stands for GlobalOperationWindow.

Since, when GlobalOperationWindow is configured for a region, the user must have an opportunity to go to any GlobalOperation within the region, GOWProperty must have a reference to GlobalOperationSystem which is sent by it by Zenject.

# User Input

## MouseInputHandler

MouseInputHandler deals with all mouse inputs performed by the user. It also contains methods required by both BuildingSystem and SelectionSystem when they need to receive the mouse position or do a raycast from the mouse position. This class implements interface Zenject.ITickable to avoid unnessary inheritance from MonoBehaviour.

## KeyboardInputHandler

KeyboardInputHandler detects the following keys

E – right rotation;

Q – left rotation;

Esc – go back / cancel;

It also implements Zenject.ITickable for the same reason as MouseInputHandler.

# Camera Movement

## CameraController

CameraController is a class responsible for camera movements. CameraController depends on CameraMover, CameraRotator and CameraZoomer. CameraController registers user inputs and asks its dependencis to try change the camera’s position and rotation in LateUpdate. If either has changed, CameraController invokes CameraMoved event.

## CameraMover

CameraMove moves camera on local axes depending on the current direction input.

## CameraRotator

CameraRotator applies Y rotation to the camera and freeze rotations on X and Z axes on the indicated values.

## CameraZoomer

CameraZoomer moves the camera on Y axis allowing the user to zoom in or out on game objects.

## Current Camera Controls

WASD – move on axis,

Hold LShift + WASD – move on axis with increased speed,

Hold RightMouseKey – rotate on Y axis,

MouseScroll – zoom in / zoom out

## CameraShifter

CameraShifter is a class responsible for toggling different camera groups in a scene. Currently it changes the view only between the main camera and the global operation camera. Before changing a camera group CameraShifter enables a transition panel which blocks the view for the user for a fraction of a second.

CameraShifter also toggles canvases related to camera groups. If multiple canvases which are rendered by different camera groups are left active at the same time, Unity EventSystem can behave in an unexpected manner.

# Player

## Player

Player is a class that allows other systems to interact with Player’s dependencies.

## HiringManager

HiringManager is a class that hires minions on behalf of the player. It has HumanResourceManager and ResourceManager in its dependencies, as well as a MinionSpawner. This class listens to events from MinionHiringMenu, checks whether the Player can hire designated number of minions of the selected type in terms of available resources. HiringManager also caps the maximum number of minions the Player can hire.

When hiring is confirmed by the Player and if all requirements are met, HiringManager tells the MinionSpawner to spawn the Minions and registers them with the HumanResourceManager.

# Resources

## ResourceManager

ResourceManager is a class and a dependency of Player. It acts as a warehouse which turns in incoming resources, returns current stockpiles and tries to deduct requested amounts of resources. All resources that are sent to ResourceManager must be previously registered with it. It is necessary due to the composition of a resource struct and close cooperation between ResourceManager and ResourceDrawer which is responsible for sheer depiction of available resources and projections (how much will be added or consumed during the next temp step by BuildingBalanceSettler).

## ResourceProfile

ResourceProfile is a scriptable object which holds the main information on a resource: its type, description, UI image etc. ResourceProfile is one of the two fields in Resource struct. The second field is of int type called Value and it holds information on the available amount of the resource.

## ResourceDrawer

ResourceDrawer is a UI class which is responsible for adding visual information on the player’s resources. It has separate methods to draw the main Value of a resource and its projection. If a projection equals 0, then it will not be drawn. Otherwise, a projection looks like (-3) or (+5) added after the main value of a resource in UI.

## ResourcePanel

ResourcePanel is a UI class and a dependency of ResourceDrawer. At the same time it is a prefab, which represents a simple panel with a UI image and a text field. A new ResourcePanel is instantiated every time when a new resource type is registered with ResourceManager.

ResourcePanel requires a ResourceProfile to be created. It is necessary for DescriptionSystem which will show a info on the resource when the player hovers over its ResourcePanel.

# DependencyInjection

Many of the scripts depend on Zenject library to receive required dependencies. Construct methods with attributes [Inject] must not be deleted despite the fact that Unity marks them as unused. The object SceneContext on the scene also must not be deleted and must contain BuildingSystemInstaller which should be added as a component and placed to Mono Installers.

///AbstarctStateMachine is currently removed from the project and substituted with ///BehaviorDesigner. But it can be reimplemented for creating less complicated ///behaviors.

# Abstract State Machine

Abstract State Machine includes 3 scripts that are located in the folder named “AbstractStateMachine”. This is a custom state machine developed to sustain a complex behavior of gameobjects. This state machine is underpinned by UniTask which calls its Update and FixedUpdate methods. Using UniTask allows not to inherit the state machine from MonoBehaviour and eliminates its dependence on being ticked by the object it controls.

The main idea of this state machine is that a gameobject must always be in a specific state, until it is disposed of.

## StateMachine<T>

StateMachine<T> is the main body of the state machine. Its main logic is setting a SuperState<T> in the Update method, and then calling updates on CurrentSuperState.CurrentState. SuperState updates correspond to Unity’s Update method, while State updates correspond to Unity’s FixedUpdate. However, they both can be easily called in any update.

## SuperState<T>

SuperState<T> is a class which is meant to hold inside itself a few States<T> which must be enacted only if some specific conditions are met. SuperState<T> must have at least one State<T> which it will set as DefaultState and CurrentState on launch. When creating a SuperState<T> , a programmer will have to implement the abstract TryChangeState method in which the logic of state change must be implemented and which always! must return a State<T> (returning null is not allowed).

A common example of a SuperState<T> is GroundedState. When the StateMachine<T>, where T is a gameobject which this StateMachine controls, see that T is grounded, it sets GroundedState as CurrentSuperState. After that the StateMachine<T> asks CurrentSuperState to set a new State<T> as CurrentState. At this point GroundedState will check its own state transition conditions implemented in the TryChangeState method and will return a new State. For example, if T’s velocity is small, it will return WalkState. If T’s velocity is above a certain threshold, it will return RunState. If T’s collider hit a ceiling, it will return CrouchState. All these three states have one thing in common – in order to transit to them T must stand on the ground, and they will never be called if T is airborne.

This separation between SuperState<T> and State<T> allows to simplify transition logic between states.

## State<T>

State<T> is a class meant to hold specific information on what T’s components must do when T is in this state. Since T can have many components which are modified in this state and since all States<T> are created inside SuperState<T>, it is easier to let State<T> know about T then sending to it only required components in the constructor.

# Humanoids

## Humanoid

Humanoid is an abstract class that relays external changes and interactions to behavior trees. Humanoid also requires Unity’s NavMeshAgent component and Unity’s ThirdPersonCharacter. NavMeshAgent allows humanoids’ navigation on terrain and inside the UCA. ThirdPersonCharacter enables humanoids to move around and decently blends walk/run animations.

Humanoid and its heirs make use of Behavior Designer which is an imported asset allowing creation of complex AI. The documentation and tutorial videos are available online <https://opsive.com/support/documentation/behavior-designer/overview/> , <https://www.youtube.com/watch?v=PuLuwzgYB4g&list=PLCPL5IYZm1NLVVxJx3-CFTLuHvLbCgmzk&ab_channel=opsive>

## Minion

Minion is an heir to Humanoid. It holds all methods specific to common minion.

## EnemyAgent

EnemyAgent is an heir to Humanoid. It contains specific methods that allow EnemyAgent to interact with the environment.

## HumanoidData

HumanoidData is an abstract scriptable object that holds parameters common for all Humanoids.

## MinionData

MinionData is an heir to HumanoidData that includes parameters shared by all Minions.

## EnemyAgentData

EnemyAgentData is an heir to HumanoidData that includes parameters shared by all EnemyAgents.

## External Behavior Trees

An external behavior tree is a number of nodes that are exported from the parent tree and can be reused in other trees by adding BehaviorTreeReference nodes.

The project widely exploits these external trees to avoid duplication. For example, if both a minion and an enemy agent must defend themselves if aggressively approached by a humanoid, then this behavior can be created for either of them and then exported to the other.

All behavior trees are stored in \_Scripts/BehaviorTrees and in \_Scripts/BehaviorTrees/ExternalTrees.

## Abstracts

Abstracts is a folder in \_Scripts/BehaviorTrees. It is comprised from abstract classes, utility classes and Actions which are too general to place them elsewhere.

## SetSharedDependencies

SetSharedDependencies is a class which inherits from Action class of Behavior Designer. Since many custom nodes, created specifically for the current project, require a reference to a Humanoid or its heir, the main purpose of this class is to find and set a reference to the Humanoid and its variations.

## CustomSharedVariables

Behavior Designer allows sharing dependencies among multiple nodes and behavior trees. However, in order to do this such variable must be created as a shared one. Common data types are converted to shared variables by default. However, in order to share a Humanoid reference among multiple nodes, a respective heir of SharedVariable class must first be created.

CustomSharedVariable is a declaration script for all custom SharedVariables that are defined in the project.

## HumanoidAction

HumanoidAction is an heir to Action class of BehaviorDesigner. It is main purpose is preventing duplication of the code related to receiving a reference to the Humanoid in custom Actions. HumanoidAction also contains references to all aspects Humanoid has, so these aspects can be referenced directly omitting Humanoid itself.

Multiple heirs to this class can be created, each of them can contain a reference to the target class inheriting from Humanoid, such as MinionAction, EnemyAction etc.

## HumanoidConditional

HumanoidConditional is an heir to Conditional class of BehaviorDesigner. It is main purpose is preventing duplication of the code related to receiving a reference to the Humanoid in custom Conditional. HumanoidConditional also contains references to all aspects Humanoid has, so these aspects can be referenced directly omitting Humanoid itself.

Multiple heirs to this class can be created, each of them can contain a reference to the target class inheriting from Humanoid, such as MinionConditional, EnemyConditional etc.

## FixedUpdateDrivenAction

FixedUpdateDrivenAction is an heir to HumanoidAction. It contains common logic for custom Actions which must be executed in FixedUpdate (mostly because they work with Unity physics).

## FixedUpdateDrivenConditional

FixedUpdateDrivenConditional is an heir to HumanoidConditional. It contains common logic for custom Conditionals which must be executed in FixedUpdate (mostly because they work with Unity physics).

## Commons

Commons is a folder which is comprised of HumaniodActions and HumanoidConditionals that are general both for minions and enemies.

Minion and enemy specific Actions and Conditionals are stored in their respective folders in \_Scripts/BehaviorTrees.

## CustomNodes

CustomsNodes are all Actions and Conditionals that are created specifically for this project. In the selection menu of Behavior Designer they all are united in the subcategory named of after the root namespace of the project (MasterOfMayhem). Most nodes are pretty self-explanatory and straightforward and does not contain complicated code or logic. Fields that might be found confusing are commented with ToolTip Attribute.

## Aspects

Aspect is an abstract class designed to aggregate all methods related to a specific behavior in one script. Otherwise, all these methods would have to be placed inside Humanoid script overloading it.

Methods placed in an Aspect are mostly used by CustomNodes. If a custom node is an heir to HumanoidAction or HumanoidConditional, it can reference Aspects directly avoiding Humanoid.

Currently there are 5 Aspects implemented – CombatAspect, MovementAspect, EscapeAspect, TaskHandlingAspect, RecreationalAspect.

## AspectPart

AspectPart is an abstract class that is meant to be created for a specific activity of a Humanoid. AspectPart must have references to Humanoid and ParentAspect.

## CombatAspect

CombatAspect holds all methods related to combat, such as TakeDamage. CombatAspect keeps track of all humanoids how have dealt damage to the Humanoid and discards them if they are out of combat (either dead or running away).

CombatAspect has CloseCombatPart and RangedCombatPart in its dependencies. Each of them handles corresponding combat types.

## MovementAspect

MovementAspect holds information on the entrance, if any, which the Humanoid must go through and inquires a pass permission on command.

## EscapeAspect

EscapeAspect holds methods which are required when the Humanoid is stopped for a check and calculates the exact health level when the Humanoid can run away from combat.

## TaskHandlingAspect

TaskHandlingAspect is responsible for receiving Tasks and nullifying the CurrentTask when it is completed. Currently TaskHandlingAspect also holds quite a few methods which solely relate to DeliveryTask. Most likely, in future when there are more kinds of tasks implemented TaskHandlingAspect must have separate dependencies to handle different tasks.

## TaskHandlingPart

TaskHandlingPart is an abstract heir to AspectPart which is meant to be created for every Task which a humanoid must execute. In general, TaskHandlingPart and its heirs are added to unload TaskHandlingAspect. However, some Tasks such as EliminationTask can be handled by separate Aspects which might not be divided into AspectParts.

## DeliveryPart

DeliveryPart is a class that holds all methods and variables related to delivery of CarriableObjects.

## WorkingShiftPart

WorkingShiftPart is a class that holds all methods and variables related to carrying out working shifts.

## RecreationalAspect

RecreationalAspect is more complicated that the other Aspects since besides being a sheer container for activity specific methods, it also acts on its own to monitor the Humanoid’s needs and sets appropriate RecreationalActivities.

The most important parameter of RecreationalAspect is CurrentActionPoints. Normally TaskManager picks minions to execute a new task from minions who have enough CurrentActionPoints to cover Task.FatiguePoints. Once CurrentActionPoints are below the level indicated as SleepingThreshold, RecreationalAspect will assign Sleeping as CurrentRecreationalActivity triggering the corresponding behavior in BehaviorTree. RecreationalAspect also takes into account how swiftly the Humanoid has carried out the Task and what its outcome is and corrects the deducted amount of FatiguePoints.

However, CurrentActionPoints are deducted not only when the Humanoid completes a Task. There is an automatic tire rate which deducts the defined number of ActionPoints in the defined time interval while the Humanoid is not engaged in a RecreationalActivity or a Task.

Unlike CurrentActionPoints, RecreationalAspect constantly decreases EatingPart.SatietyPoints and, when they equal 0, assigns Eating as CurrentRecreationalActivity.

If a RecreationalActivity is set, then RecreationalAspect restores parameters corresponding to this activity, such as CurrentActionPoints if Sleeping or EatingPart.SatietyPoints if Eating.

## RecreationalPart

RecreationalPart is an abstract heir to AspectPart. It is meant to include methods and variables related to one specific RecreationalActivity. RecreationalPart needs to know about RecreationalAspect since it needs to change some of its parameters during the game. In general, RecreationalPart and its children were added to unload RecreationalAspect.

## SleepingPart

SleepingPart is an heir to RecreationalPart. It contains SleepingThreshold which is used by RecreationalAspect to determine when Humanoid must go to bed, as well as some methods related to sleeping behavior which are used by BehaviorDesigner.

## EatingPart

EatingPart is another heir to RecreationalPart. Like SleepingPart, it contains methods for BehaviorDesigner and some constant values which are used by RecreationalAspect to properly assign Eating as CurrentRecreationalActivity.

## AssassinatedAspect

AssassinatedAspect is a common Aspect that every Humanoid has. It contains methods and variable which define Humanoid’s behavior when he is being assassinated by another Humanoid.

## DisguiseAspect

DisguiseAspect is an EnemyAgent specific Aspect. It contains information on the original Appearance of EnemyAgent and his fake Appearance. It also contains some methods for BehaviorTrees and variables to determine whether the agent is currently disguised.

## CarryingPoint

CarryingPoint is a MonoBehaviour script and an empty object inside Humanoid’s hierarchy that serves as the marking point to where a carriable object must be attached when carrying is started.

When Humanoid is carrying something, CarryingPoint updates its position relatively to the center of Humanoid’s mesh and the designated offset. There are currently two offsets – one is static and applied when Humanoid is standing in one place, the other is dynamic and applied when Humanoid is running with a CarriableObject. These offsets are added to improve the visual part of carrying objects.

# MinionHiringMenu

## MinionHiringMenu

MinionHiringMenu is a MonoBehaviour UI class that reads user input for hiring minions. It has CountPanel and AvailableMinionMenu in its dependencies. When the parameters of those two scripts change, MinionHiringMenu invokes ParameterChangeDelegate that MUST have a method of HiringManager in its subscriptions. This target method should check whether the Player can currently hire the desired number of minions.

If he can not, MinionHiringMenu blocks the Hire button and shows a notification with the reason for not hiring. If he can, MinionHiringMenu leaves the button interactable and invokes the corresponding event on a button click.

## AvailableMinionMenu

AvailableMinionMenu is a MonoBehaviour UI class responsible for drawing hiring options for the Player and returning the selected option to other scripts. This script makes use of MinionCards that inherit from Card<T> and assigns a toggle group to these cards.

On Enable this script also checks whether any new minions for hire have been added or may be removed, and if they have, it redraws the UI.

## CharacteristicPanel

CharacteristicPanel is a class which draws the selected minion’s stats for the Player. It automatically redraws the UI when a new hiring option is selected. This script makes use of DescriptionProperties.

## CountPanel

CountPanel is a class responsible for reading and relaying the count input for minion hiring. It blocks negative integers as well as caps the maximum input to 99. Other scripts can read CurrentCount.

## MinionCard

MinionCard is an heir to Card<T> where T is a Minion. It is actively used by AvailableMinionMenu.

# Spawners

## Spawner<T>

Spawner<T> is a base abstract class for all spawners which spawn humanoids. T must be a humanoid or its heir. Spawner<T> makes use of MonoBehaviourPool.

## MinionSpawner

MinionSpawner is a Spawner<T> that spawns simple minions.

## EnemyAgentSpawner

EnemyAgentSpawner is a Spawner<T> that spawns enemy agents.

# CarriableObjects

## CarriableObject

CarriableObject is a base abstract class for game objects that are lifted, moved and put down by minions. These objects are supposed to be relatively big, like a box (weapons and firearms are most likely to be implemented through another class).

All CarriableObjects must have a lift position designated by a separate Transform. Minion who is lifting the object first approaches the lift position and then starts assigned animations.

CarriableObjects are meant to be used in some way. Therefore, they have Activate method where all logic related to visual effects etc. must be placed. After CarriableObject was activated, it will turn off. In Enable method we can apply some logic to reset the state of a CarriableObject so that it can be properly used again.

## ConstructionBox

ConstructionBox is an heir to CarriableObject. It is used as the key item in ConstructionTasks. A minion must deliver a ConstructionBox to the DeliveryPoint of a constructed PlaceableObject. Once ConstructionBox has been activated, PlaceableObjectManager finalizes the construction of the PlaceableObject.

## Dynamite

Dynamite is another heir to CarriableObject. Currently it is used for SabotageTasks.

# Beds

## Bed

Bed is an heir to PlacableObject where Humanoids can sleep. It is assumed that one Bed can be used by one Humanoid only (even if the owner currently is not sleeping in it).

Bed has Occupy() method which returns the int corresponding to a sleeping level. Bed has one sleeping level. If a Bed has already been occupied, it returns 0.

Every Bed must have a sitting point which is a transform indicating from where the Humanoid must start a sitting animation.

## BunkBed

BunkBed is an heir to Bed, but unlike it, BunkBed has two levels meaning that two humanoids can sleep in one BunkBed. BunkBed is considered occupied only when all the levels are assigned to humanoids.

Despite having two sleeping levels, BunkBed still has one sitting point, that’s why there is a need for WaitingDistance which defines how far from the sitting point a waiting humanoid must stand while his neighbor is getting on or off his bunk.

# Tables

## Table

Table is an heir to PlaceableObject. Its main purpose is tracking its Chairs and returning whether it is occupied or not. A Table is considered occupied when all of its Chairs are occupied.

## Chair

Chair is a MonoBehaviour script. It has IsOccupied property and SittingPoint which shows the exact coordinate where a Humanoid must start the sitting animation.

# Covers

## Cover

Cover is an abstract MonoBehaviour class that indicates a place where a humanoid can take cover from gunfire. It has IsOccupied property that prevents multiple minions from taking the same cover.

## HorizontalCover

HorizontalCover is an heir to Cover that indicates a place where a humanoid must crouch to hide and stand up to shoot.

## VerticalCover

VerticalCover is a Cover type where a humanoid does not crouch and moves to the side of the Cover to shoot.

Food

## Food

Food is an abstract MonoBehaviour class which also implements IMonoBehaviourPoolElement interface in order to allow CanteenManager to spawn its heirs more efficiently.

When a Humanoid eats Food, Food is parented to the Humanoid’s CarryingPoint, and in order for the food product to look natural when a Humanoid is eating it, its position and rotation must be adjusted. Therefore, Food has serialized fields PositionOffset and EulerRotation.

## Hamburger

Hamburger is an heir to Food with no new logic.

# Weapons

## Weapon

Weapon is an abstract MonoBehaviour class. It contains a few abstract methods which are common for all weapons in the project as well as implements I MonoBehaviourPoolElement so that all weapons can be pooled.

## WeaponData

WeaponData is an abstract heir to ScriptableObject. It contains fields that are common to all weapons.

## RangedWeapon

RangedWeapon is an heir to Weapon. All firearms must derive from this class. It has a pool of Bullets and when a minion wants to fire, RangedWeapon finds an idling bullet and sends it flying with a correct direction and rotation.

## RangedWeaponData

RangedWeaponData is an heir to WeaponData. It contains extra fields that are necessary for RangedWeapons.

## Bullet

Bullet is a MonoBehaviour class that also implements IMonoBehaviourPoolElement. Its main purpose is to set and preserve bullet’s velocity once it is fired.

# LegalForces

## LegalForceBrain

LegalForceBrain is a MonoBehavior script that takes decisions on when and how to attack the player and his UndergroundBase. It has an EnemyAgentSpawner, TaskGenerator and DisguiseManager in its dependencies. In regular intervals it tells TaskGenerator to return the best task, and then spawns EnemyAgents and sends them to execute it.

## TaskGenerator

TaskGenerator is a script that periodically checks the development of the UndergroundBase (it has access to all buildings and minions) and issues potential Tasks. When commanded by LegalForceBrain, it returns the most pressing Task.

## DisguiseManager

DisguiseManager is a class that holds information on all available fake Appearances and returns one for a newly spawned EnemyAgent.

## Appearance

Appearance is a MonoBehavior script that holds mesh renderers and rigs for a specific humanoid models as well as a reference to an avatar for those rigs. When a fake Appearance is applied to an EnemyAgent, its avatar gets substituted with the avatar from the applied Appearance. Otherwise, animations will not work.

# Grid

## What if we want to change the cell scale?

Currently the grid is set for cells 2x2. We highly recommend against changes to the scale of cells for a few reasons:

1. If cells are made 1x1, it will be necessary to make more tile selectors for each placeable object which will result in consuming additional computing power;
2. Cells 2x2 seem nice and make possible to place small outdoor objects in one cells which will look great on a standard terrain of 1000x1000.
3. Currently PlacementSystem is optimized for the set cell scale. If the scale is changed, the const field MinDistance in PlacementSystem will have to be changed too to strike a balance between optimization and snappy reaction to mouse movement. We recommend this distance to be 0.5f smaller than the set cell scale.
4. UndergroundConstructionHandler is also set for cells 2x2. If the scale is altered, the const field TileScaleFactor will have to be adjusted as well. Currently it is 4 because we need to offset the center of the trench by half of the changed diameter multiplied by the cell scale. So if the scale is set to 1x1, the field should equal 2.
5. Bear in mind that many scripts (such as PlaceableObjectManager, SelectionSystem etc.) which are dependent on mouse movements, utilize simple formulas to check the distance between the current and previous mouse positions. It is implemented for performance purposes, so that extra calculations will not run if the difference in the positions is too small. Scripts that use these checks have private const float MinDistance field at the top, it is recommended to adjust this constant which is currently set between 1f and 1.5f. MinDistance should be approximately half of the cell scale.

Taking into account these points will help to optimize and adjust the building system to a new cell scale.

# Utilities

## MonoBehaviourPool<T>

MonoBehaviourPool is a utility class that simplifies controls over MonoBehaviours that get spawned and turned off regularly, eliminates code duplication among different classes. In most cases UI classes such as DescriptionWindow and StateIndicationSystem make use off of MonoBehaviourPools.

In order to be initialized inside the MonoBehaviourPool, T must implement IMonoBehaviourPoolElement that requires implementation of public GameObject {get} property, since game objects in a MonoBehaviourPool<T> are manipulated through their activation and deactivation.

When instantiated, a gameObject is supposed to call its Awake() method automatically. However, there were some errors when GameObject remained null since Awake() was not called after instantiation in a pool. To fix this issue, IMonoBehaviourPoolElement must also implement public Awake() method which will be called by a pool if GameObject of its new T element is null. This guarantees that any object in a pool will be constructed properly.

## AnimatorParameters

AnimatorParameters is a static class that holds all hashed references to parameters of animators. Currently these parameters are related to only one animator which controls Humanoids.

Making a static class with all animator parameters allows avoiding multiple hashing of the same parameters by different States and ThirdPersonCharacters. This class must not contain any logic!

## Serializable Dictionary Scripts

Serializable Dictionary scripts are all the scripts added to the folder Utilities/SerializableDictionary. They are used to solve the problem of dictionary serialization in Unity.

SerializableDictionary<TKey,TValue> is a generic class, that holds all the logic related dictionary with some custom modifications from the author. The link to the source: <https://forum.unity.com/threads/finally-a-serializable-dictionary-for-unity-extracted-from-system-collections-generic.335797/> . According to the author, he took the standard dictionary script out of System.Collections and removed/added some lines of code.

DictionaryDrawer<TK,TV> is a generic class responsible for visualization of a SerializableDictionary<TKey, TValue> in UnityEditor. The author says that DictionaryDrawer makes impossible setting reference types as keys without prior modifications to the script. However, I have managed to solve this problem by making the script create new instances of TK if it is of a reference type.

Class CustomDictionaries is a general C# script where specific instances of the two previous types must be created for a specific project. There are also some comments in it containing useful advice on how exactly custom SerializableDictionary heirs must be created and worked with.

## ReactiveProperty<TValue, TOwner>

ReactiveProperty<TValue, TOwner> is a generic class created to fire events automatically when TValue changes. TOwner is added mostly, because in our experience subscribers need to know what script exactly TValue belongs. The set accessor for TValue property is left public for no other choice, but if another script tries to change TValue, then it can be easily tracked from TOwner script which declares its custom ReactiveProperty.

# Current Problems

1. Optimization concerns regarding the number of gameobjects.

The current size of the mountain relative to the standard terrain of 1000x1000 results the UCA under it must consist of about 20 000 blocks (15 000 are ClearableBlocks and the rest are EdgeBlocks). Since each ClearableBlock has 4 Walls, Top and Floor attached to it, it makes the total count of the ClearableBlock related gameobjects to equal about 15 000 x 6 + 15 000 = 105 000 units. EdgeBlock has only 4 walls as its child objects, accounting for additional 5 000 x 4 + 5 000 = 25 000 gameobjects. The total count of gameobjects in the UCA is currently 105 000 + 25 000 = 130 000 units.

Such a big amount of gameobjects doesn’t currently affect the performance, since all the blocks do not move in runtime. However, it significantly slows down entering and exiting the Unity play mode, making it impossible to continue development on the main scene. There is also some ground to assume that such a big count of gameobjects prevent the main Integration scene (Assets/\_NewIsland/Scene/Integration.unity) from being built. Unity freezes on BuildingResource forlder when we try to make a build out of that scene.

**Recommendation**: A 3D graphics designer must develop a UCA block model with submeshes to avoid spawning block parts as separate child gameobjects. This will allow decreasing the gameobject count 5-6 fold which will make the game start faster.

1. Currently Building is not an abstract class. It must be made abstract for code architecture purposes.
2. BehaviorDesigner which is widely used in the project, is not a free asset. Currently a pirate version is used. Before the release it is recommended to be the asset officially to avoid legal claims (its price is below 100 USD and it can be purchased in the Unity Asset Store).
3. Redundant scripts (must be deleted if remain unused):

* Folder \_Scripts/AbstractStateMachine;
* BottomGenerator, Bottom

# Paused Moment of Development

1. Create a state indication system for enemy agents. It must be toggled between their actual tasks (assassination, sabotage etc.) and fake ones, so that the player would not be able to find infiltrated agents only by enabling icons and developers could track what agents are doing;
2. Implement health bars for humanoids as floating points that change color from green to red depending on the remaining health amount;
3. Implement for minions an ability to expose enemy agents when they see agents carrying out harmful tasks.
4. DynamicCanvas receives an old position for DynamicUIElements when it reenables them if they got disabled when out of screen boundaries. The position is updated only when a humanoid starts moving. Fix this bug.