RTS Framework

--- SINCRESS ---

Welcome to the documentation of the Real Time Strategy Framework for Unity3D. This document describes the example scene, the provided assets and the scripts of the framework.

1. Features

This framework is designed to give your strategy game project a headstart by providing a template for several common features found in RTS games. Here is the complete list of implemented features of this framework:

- Mouse controlled camera with adjustable zoom (height)
- Player and AI units
- Unit selection by clicking or mouse dragging with UI feedback
- Unit selection indicator which displays a white circle around selected units
- Contextual order menu shows common orders of all selected units
- Vehicle and structure type units
- Context depended order issue system
- Unit building system with correct and incorrect placement indication
- Basic unit AI system with move, attack, idle and interact orders
- Unit health system
- User interface for units with a healthbar and a status text which displays current order
- Terrain with mineable zones and props
- Unit pathfinding and collision avoidance using Unity's NavMesh system
- Attacking units shoot projectiles
- Unit properties and settings loading system using scriptableObjects
- Several textured low poly vehicle models: scout, tank, harvester, gun turret, powerplant, armored personnel carrier
- Highly modular script design allowing easy modification and feature implementation
- Map spawn zones for AI unit spawning

- Ingame overlay menu for saving and loading
- Load/save function for units and camera position
- New and refactored codebase!

2. Example Scene

The example scene is in the root of the project folder. It showcases the features of this framework in a minimal example. The *CameraController* script is attached to the **Camera** object, controlling the camera movement and clamping it to stay above the terrain. The **Terrain** object has several child objects; spawn zones for AI units, mineable zones with texture projectors and some visual props (with NavMesh obstacle components). **Mineable Zones** are prefabs which allow the user to interact with the area using units which have the ability to receive Interact orders. It is indicated by a projector which projects a texture onto the terrain. Mineable zone GameObjects are marked by a *Mineable* tag.

The **HUD** object is a full-screen Canvas element with several child objects of interest. The DrawOnTop and DrawOnBottom objects can be used to draw various ingame UI elements or overlaying menus such as an in-game options menu or a pause menu. The **BuildPanel** is the left panel of the user interface which allows the player to place units on the map. It displays icons of units and when clicked allow the user to place the selected unit on the terrain, if the placement is possible. Possible placement is indicated by a green silhouette of the unit, and obstructed placement by a red silhouette. The buttons in the Build Panel are linked to the *BuildUnits* script attached to the **Player** GameObject. Each of the buttons gives the mentioned script a reference to a prefab object which has the mesh of the built unit but changed its material depending on whether placement is possible.

The **UnitPanel** is the right panel of the user interface. It displays available actions depending on selected units. For example, if a scout vehicle is selected, *Idle* and *Move* orders are available. If a tank is selected as well, an additional *Attack* order appears. If the player gives the *Attack* order to both vehicles, only the tank will carry out the order.

The **NPC** GameObject represents an AI player and has two simple scripts attached to it: the *AIUnitSpawner* which spawns a prefab unit at one of the map's spawn zones in a

random interval; and a *AICommander* script which gives AI units random destinations on the map, making them navigate and patrol the terrain.

Finally we have the **Player** GameObject which has the interaction logic of the player. Interaction is divided in two parts, contained in scripts called *BuildUnits* and *SelectUnits*. Unit selection has a reference to the right unit panel which displays available unit actions and handles mouse click and drag selection of player owned units. *BuildUnits* is concerned with placement of built units on the terrain, checking whether they can be placed and replacing the placement model for the actual unit once building is confirmed.

The previous two mentioned GameObjects have unit holder objects as children so that all spawned units can be childed to the appropriate unit holder, keeping the scene hierarchy clean in-game. This sums up the components of the Example Scene which will be the starting point for your work with this framework.

3. Models and assets

Apart from a number of scripts this package contains several art assets as well. There are six low poly SciFi vehicle models with textures included in the project: a Scout, Tank, Harvester and APC vehicle and two structures, a rotating Gun Turret and a Powerplant. These can be easily replaced for any assets of your project and new models and maps can be added with no programming required. The user interface has two panels with buttons with a panel graphic provided in the assets. This design can be easily swapped out for your own, or used in further UI expansions. The provided terrain is built as a Unity3D Terrain asset and has a NavMesh which allows AI and player owned units to navigate, however this can be entirely replaced by a system of your own liking. The modular nature of this framework allows you to easily replace or add any functionality while keeping components separated.

4. Scripts

The RTS Framework contains several modules of separate functionality. Unit selection, unit construction, unit orders and vehicle properties loading from file are some of them. This section provides a script reference for the components of the framework.

UnitController.cs

Abstract class which provides a base to encapsulate access to unit properties and unit orders, together with the implementation of these orders. This is a base class for *VehicleController.cs* and *StructureController.cs*, which offer different implementations of orders and properties. Unit spawning or building and unit destruction are two events which need to be broadcasted because different components rely on knowing which units are presently available. Player and non-player units are differentiated by the tag on their GameObject. The UnitController and its concrete subclasses keep track of the unit's available actions (orders) and the currently given order. They also provide methods which are called when the unit has taken damage (*public void TakeDamage(int damage)*) and when the unit has been selected (*public void SetSelected(bool isSelected)*). All the methods are commented and documented in the source code itself. The *UnitController* script also has a reference to the *UnitProperties* element.

VehicleController.cs and **StructureController.cs**

contain the implementation of orders that can be given to vehicles and structures. All vehicles have a Move and Idle order, and a boolean flag determines whether a unit can Attack or Interact. A target can be set for a unit using the *SetTarget(GameObject target, TargetType type)* method, and the unit will determine whether to ignore it or not based on the target type (structure, enemy unit or mineable zone) and the current order (attack for structures or enemy units, interact for mineable zones, etc). Methods in these two scripts called *ComputeMovement, ComputeAttack* and *ComputeInteraction* perform the unit's given action while considering the units properties such as speed, projectile range, rotation speed and health.

UnitProperties.cs

This script is another abstract base class whose methods are implemented in *VehicleProperties.cs* and *StructureProperties.cs*. It is a scriptable object which contains data relevant to the current unit type.

UnitPlacement.cs

is attached to each of the build prefabs in the /Prefabs/BuildableUnits folder and is primarily tasked with changing materials depending on whether placement is possible. If the build object is intersecting another unit, structure or is in unreachable terrain, the object changes to a red semitransparent material which indicates that placement at this position is impossible. To allow for this, a system of layers is used. Upon confirmation of placement in a possible position, a construct sound is played, the build prefab is destroyed and at its location a unit is spawned. This is controlled by the *BuildUnits* script described below.

UnitDetailsHUD.cs

Another script attached to each unit, tasked with displaying the user interface details of the unit. When ALT is held down, a healthbar and a textual info is shown above the unit. This script instantiates a UI prefab as a child of the *DrawOnBottom* GameObject mentioned in section two. While the details are shown, values are updated automatically. The script is subscribed to the *UnitDestroyed* event and disabled the UI prefab if the destroyed object corresponds to the script's GameObject.

Projectile.cs

This class moves the projectile fired by a unit and controls its interaction with targets or makes it disappear once out of range. The projectiles have a set speed and range and checks collisions with objects in the UnitLayer. The projectile is destroyed either by hitting a target in the UnitLayer or by travelling farther than its range. To fire a projectile, one needs

to be instantiated and a target GameObject has to be set with the *SetTarget(GameObject target)*.

BuildUnits.cs

This is one of the two scripts attached to the Player GameObject, the other being *SelectUnits*. This script works together with *UnitPlacement* and allows the player to place structures and vehicles on the terrain.