RTSObject API

Introduction

RTSObject is the base class for all game objects, that is, everything that can be interacted with within the game, such as units, buildings and resources. Concrete objects can be implemented by inheriting from RTSObject. However, as a rule of thumb, you should always inherit from Unit, Resource or Building instead of inheriting directly from RTSObject. Doing so will give your object default implementations for various methods and decrease maintenance costs.

Therefore, the inheritance tree for RTSObject looks like this:

- RTSObject
    - Unit
        - ConcreteUnit1
        - ConcreteUnit2
        - ConcreteUnit3
    - Building
        - ConcreteBuilding1
        - ConcreteBuilding2
        - ConcreteBuilding3
    - Resource
        - ConcreteResource1
        - ConcreteResource2
        - ConcreteResource3

Different objects can have different habilities, such as the hability to move or attack. In order to enable or disable those habilities, some methods have to be overriden (The rationale for having to override methods instead of implementing interfaces is that those habilities may change dynamically at runtime).

Movement

How to use the API: You can check if a object can move by calling the `CanMove()` method. After you have successfully determined that the object can move, call the `MoveTo(Vector3 target)` with the destination position as a parameter. Do not make assumption such as "all units can move" or "a unit which could move before can move now", since those conditions may not hold in the future.
Sample code:

```
var myTarget = new Vector3(123.0f, 123.0f, 123.0f);
if (myObject.CanMove())
{
    myObject.MoveTo(myTarget);
}
```

How to use the default API implementation: You will need to define the `baseMoveSpeed` (speed in meters per second) and to a non-zero value, otherwise the unit won't be able to move. Note that, at the moment, only units have a working movement implementation, so make sure not to set this value for buildings and resources.
How to override the API: You will need to implement (override) the following methods:

- `bool CanMove()`: Here you need to return if the object can move or not. You may change the return value of this function at runtime (e.g. if a object can be "set up" to become fixed such as a trebuchet).
- `float GetMovementSpeed()`: Here you need to return the speed, in m/s, of your object. You may change the return value of this function at runtime (e.g. depending on upgrades).
- `void SetNewPath(Vector3 target)`: Here you need to implement the code that will generate a new path (skipping any obstacles between the position of the object and the target) and begin the movement of the object to the target. Note that the `Unit` class has a generic implementation of this method, so you don't need to implement it.
- `void CancelPath()`: This must cancel a previously generated path using `SetNewPath(Vector3 target)` and stop the unit from moving immediately. Note that the `Unit` class has a generic implementation of this method, so you don't need to implement it.

- `bool HasPath()`: This will return whether the object has a programmed movement path or not (in simpler terms, if the object is moving or not). Note that the `Unit` class has a generic implementation of this method, so you don't need to implement it.

Attack

How to use the API: You can check if a object can move by calling the `CanAttack()` method. You will also need to check if the target object can defend by calling his `CanBeAttacked()` method explained below. After you have successfully determined that the object can move, call the`AttackObject(RTSObject target)` with the target object as a parameter. Do not make assumption such as "all units can attack" or "a unit which could attack before can move now", since those conditions may not hold in the future.
Sample code:

```
var myTarget = GameObject.Find("MyTargetObject").GetComponent<RTSObject>();
if (myObject.CanAttack() && myTarget.CanBeAttacked())
{
    myObject.AttackObject(myTarget);
}
```

How to use the default API implementation: You will need to define the `baseAttackStrength`(number of attack strength points per hit) and `baseAttackSpeed` (number of attacks per second) to a non-zero value, otherwise the object won't be able to attack. Additionally, you can define`baseAttackRange` (meters of attack distance range) to a non-zero value to create a range unit.
How to override the API: You will need to implement (override) the following methods:

- `bool CanAttack()`: Here you need to return if the object can attack or not. You may change the return value of this function at runtime (e.g. if a object can be "set up" to become able to attack such as a trebuchet).
- `int GetAttackStrength()`: Return the number of attack strength points of the unit. The number of actual health points substracted from the target is calculated using the formula explained below. You may change the return value of this function at runtime (e.g. depending on upgrades).

- `float GetAttackSpeed()`: Return the number of attack hits per second that the unit can inflict. You may change the return value of this function at runtime (e.g. depending on upgrades).
- `int GetAttackRange()`: Return the distance at which the unit can attack without moving, in metres. If the unit can't attack at a distance (e.g. an swordsman), return 0. You may change the return value of this function at runtime (e.g. depending on upgrades).

Defense

How to use the API: You can check if a object can defend by calling the `CanBeAttacked()`method. After you have successfully determined that the object can be attacked, you may use the attack API explained above to attack the object.
How to use the default API implementation: You will need to define the `baseDefense` (number of attack defense points per hit) and to a non-null value, otherwise the unit won't be able to defend.
How to override the API: You will need to implement (override) the following methods:

- `bool CanBeAttacked()`: Here you need to return if the object can be attacked or not. You may change the return value of this function at runtime.
- `int GetDefense()`: Return the number of attack defense points of the unit. The number of actual health points substracted from the target is calculated using the formula explained below. You may change the return value of this function at runtime (e.g. depending on upgrades).

Attack formula

The formula for calculating how many health points are substracted from the target is inspired by Age of Empires 2 and it the following:

```
SubstractedHealthPoints = Max(AttackStrength - AttackDefense, 1)
```

Building

How to use the API: You can check if a object can build by calling the `CanBuild()` method. After you have successfully determined that the object build, call the `AssignBuildingProject(Building newProject)` with the project to assign as a parameter. Do not make assumption such as "all civil units can build" or "a unit which could build before can move now", since those conditions may not hold in the future. To deassign the currently assigned project, call `AssignBuildingProject(null)`. To check if a object is building, call the `IsBuilding()` method.
The building you pass to the `AssignBuildingProject` method needs to be a building project. Check the `CreateOnConstructionBuilding()` method of `CivilUnit` for an example of how to set up a building project.
Sample code:

```
var myProject = GameObject.Find("MyBuildingProject").GetComponent<Building>();
if (myProject.CanBuild())
{
    myObject.AssignBuildingProject(myProject);
}
```

How to use the default API implementation: You will need to define the `baseBuildFactor`(construction speed multiplication factor) to a non-zero value (recommended value: 1.0f), otherwise the unit won't be able to build. Additionally, for any buildings you are building, you will need to set the `buildingTime` (time in seconds required to complete the building).
How to override the API: You will need to implement (override) the following methods:

- `bool CanBuild()`: Here you need to return if the object can build or not. You may change the return value of this function at runtime.
- `bool IsBuilding()`: This will return whether the object has a building project assigned or not.
- `void AssignBuildingProject(Building newProject)`: Here you need to set up your object so it starts moving towards to building and calling its `Construct()` method to increment its health points. Make sure that you accept `null` as a parameter.