RTS Toolkit

v2020.2.2

email: chanfort48@gmail.com

https://chanfort.github.io/

28/02/2021

System requirements

Unity 2018, 2019 or 2020.

Compatibility and versioning

RTS Toolkit has been continuously developed over last few years and this gives large number of versions available. The table below shows cross-matching versions for Unity and RTS Toolkit which has been tested. When downloading from Asset Store the RTS Toolkit version will be used to match corresponding Unity's version. I.e. if user has Unity 5.3.4, downloading RTS Toolkit will give RTS Toolkit version 5.0. To install most recent version of RTS Toolkit make sure that Unity has been updated properly.

Unity version	RTS Toolkit version
2018.4.32f1, 2019.4.21f1, 2020.2.6f1	2020.2.2
2018.4.31f1, 2019.4.18f1, 2020.2.2f1	2020.2.1
2020.2.0	2020.2
2019.2.0	2019.2
2018.3.0	2018.3
2018.1.0	2018.1
2017.2.0	2017.5
5.6.1	2017.4
5.6.0	2017.3
5.5.2	2017.2
5.5.0	2017.1
5.5.0	5.3
5.4.1	5.2
5.4.0	5.1
5.3.4	5.0
5.3.1	4.1
5.2.3	4.0
5.2.0	3.3
5.1.0	3.2
5.0.2	3.1
5.0.0	3.0
4.6.4	2.1

1. Getting started

First steps to get started is to download and import RTS Toolkit from Asset Store and to open the default main scene. The best idea is to download it as a new project the first time. Start play to enter the play mode and look around what is in game.

Once you become familiar how to run the main scene, the next step is to look at what things are implemented and what you can do with the toolkit. The project contains 4 major sections for RTS core, terrain system, UI and A* Pathfinding Project extension. Each of them are tightly linked with other sections and needs specific treatment in case if user want to remove one or another component from the project.

RTS Core is the largest part of RTS Toolkit, containing systems for economy, military, diplomacy, units movement, selection, scores and many other areas management.

Terrain system is responsible for creating procedural terrain, which is used to set RTS on it. This also manages environment details, such as clouds, water, etc.

UI is the section, which defines how player interacts with the game. It holds menus, buttons and function calls when buttons are pressed.

A* Pathfinding Project extension is the extension to make RTS Toolkit compatible with http://arongranberg.com/astar/ project and use its functions to manage navigation.

The whole system is made under just a single prefab, named “RTSToolkit”, which can be saved and copied into user project to get developers quickly started. The linking between scripts is managed through singletons. Most of these scripts has just a single copy in the scene and their singletons are being set in Awake() function when the game starts. As a result, scripts can access each other through these singletons and use public functions and public variables where needed.

2. RTS Core

RTS Core splits into multiple components, used to manage different areas of RTS. The key questions could be managing large number of units, character setup, multiplayer.

2.1. Large number of units

Large number of units in RTS Toolkit is treated not as just an optimisation step but as one of the most vital problems RTS genre games are facing. RTS games can be frequently classified how epic they are just by telling how many units/objects the game can handle at playable frame rate counts (FPS). However, same games handling just several tens of units, other ones - several hundreds and some of them can do thousands of units. Problem arises not just with one thing, but in large amount of areas, including rendering, animations, pathfinding, AI calculations, etc.

2.1.1. Rendering

Rendering large number of units becomes a problem when number of polygons rendered in the scene reaches critical values (usually several millions) and the engine simply can’t handle more polygons at playable FPS.

2.1.1.1. Level of details

Level of details can be used as a common technique to reduce number of polygons in the scene by using lower poly-count model versions at larger distance from camera. As units are more further away from camera, high level details becomes invisible and it is reasonable to replace higher resolution models into lower. Usually several LOD versions (3-5) is quite enough to achieve a good balance between quality and performance.

RTS Toolkit has it’s own LOD system designed and adopted for RTS Toolkit needs. LODs are managed by RenderMeshModels.cs script.

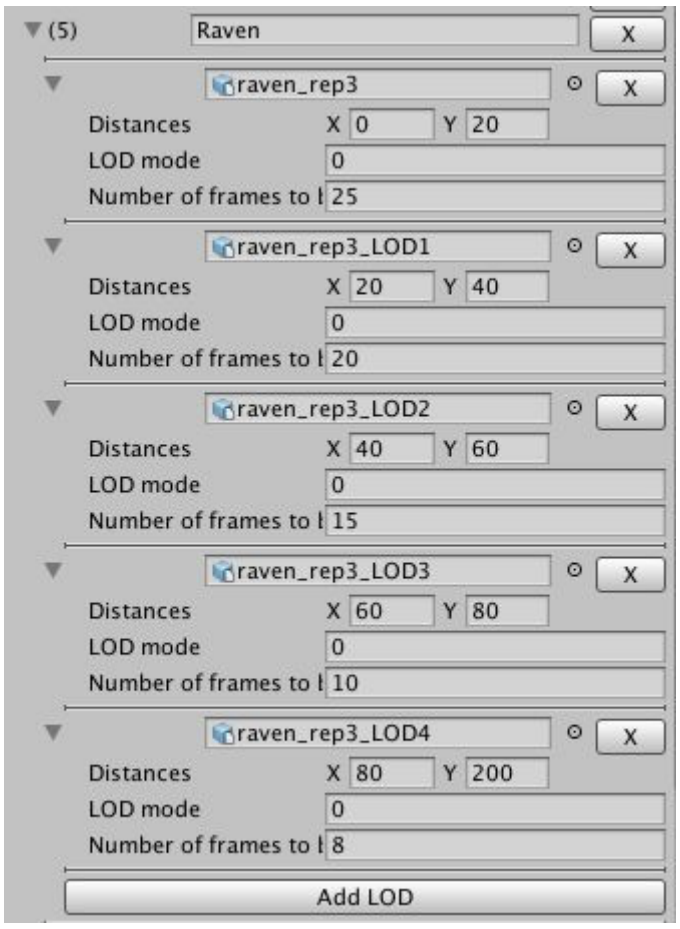


Fig. 1 - LODs setup in RTS Toolkit by using RenderMeshModels.cs

Fig. 1 shows an example of how LODs can be set to use different LODs versions, which are switching based on unit distance from camera. The example model has 6 LOD variants, where the first one has largest number of vertices, while the last one has smallest number of vertices. Simple sanity check can be used to decide what distances to set: lets say Unity can handle 1M vertices at playable FPS. If model has 10k vertices, 1M will be reached just by using **100** models, 1k vertices models will can have **1000** instances in game and if model is 100 vertices only, user can use **10 000** models to reach 1M vertices.

RenderMeshModels.cs has 4 classes representng hierarchical levels of complexity. **RenderMeshModels** is a Monobehaviour class, which has a list of all different models, used in the toolkit. Each model can have several LODs, so list of LODs for a given model is in **RenderMeshLODs** class. Each LOD can have multiple animations, so the list of animations for a given LOD is in the **RenderMeshAnimations** class. Finally each animation has multiple animation frames. Each animation frame data is stored in **RenderMesh** class. SkinnedMeshes are being converted into regular meshes on Start() function and stored in mesh lists, which are being used by Unity's **DrawMesh()** function to render meshes at locations, where units are present in game.

2.1.1.2. Animation

In RTS games there are both type of units - animated and non-animated. When we have non-animated units (usually buildings), they are just a single mesh objects, which is quite simple to render. But there are also in game various units, which are walking, attacking, dying, etc. and playing different animations. So the big question is: how to render large number of units, which could use different LODs and also play their own animations?

As it was described in the previous section, a given LOD variant can have multiple animations. The fastest way to deal with all these different things is to use indexing. Let's take a look at the example when we have 3 models, each of them has 5 LODs, each LOD has 4 different animations and each animation has 25 frames. That would give:

3 models
3 x 5 = 15 LODs
3 x 5 x 4 = 60 Animations
3 x 5 x 4 x 25 = 1500 Animation frames

So 1500 animation frames creates in memory 1500 static meshes, which can be used any time. So our task here becomes as what to display in game at a particular position on a current time? As all 4 different things are placed in all 4 classes, there are no conflicts between them when changing one of the conditions. LOD index is received by using distance from camera to object inside **RenderMeshLODs** class, Animation index is being changed when we ask to switch animation, which is directed to **RenderMeshAnimations** class, and the mesh to render index is obtained by using animation phase index:

```
int animationTimeIndex = ((int)(t * n / length) % (n));
```

where:
t = Time.time-(last reset time)
n - number of frames
length - animation total length

As the mesh index is received, it's being passed directly to DrawMesh() function to display it. As the time passes, t value keeps changing and on each update different meshes are being passed into DrawMesh(), representing a sequence, which is displayed as an animation. Profiling shows that indices are being found with no practical cost on performance and is a very good approach.

Unit animation relevant data is linked with **UnitAnimation.cs** class, which is attached onto each unit instance. Start() function there finds model and animation indices and sets unit instance for rendering in RenderMeshModels.cs classes. Animation can be switched by calling **PlayAnimationCheck()** function in UnitsAnimation. When units die, OnDestroy() function is unsetting unit instances from RenderMeshModels.cs classes.

By default animation parameters are used the same as they are on the original model, i.e. if `AnimationState.wrapMode == WrapMode.Once`, animation is set as non-repeatable; animation length is obtained from `AnimationState.length` and so on.

2.1.1.3. 3dSprites

3dSprites is another way to show units in their positions at a low cost. The idea is to use not the actual model but a quad, facing the camera all the time (billboard). The animation on the quad can be used as a sequence of switching images, which represents different stages of movement. In 3d space model can be viewed from any directions. 3dSprites approximates this by using snapshots of the real model baked into images, which are displayed on billboard. A grid of rotation angles is being used to take snapshots from all possible viewing angles. So 3dSprites uses horizontal and vertical rotations to build a grid. That means that on the top of previously discussed animation system for models, this time it adds 2 additional levels in the hierarchy: **horizontal** and **vertical rotational levels**. Snapshots are taken in the baking scene and saved into images, which later being imported into game and displayed on billboards to represent the unit. Number of rotational levels has to be taken with care, as it can produce very large amount of data, which may not be loaded on memory. Let's take a look at our previous example by saying that we want to use 12 horizontal levels and 6 vertical levels.
3 models
3 x 12 = 36 horizontal rotational levels
3 x 12 x 6 = 216 total rotational levels
3 x 12 x 6 x 4 = 864 animations
3 x 12 x 6 x 4 x 25 = 21 600 animation frames

As we can see, there might be produced 21 600 image files this way. However, 3dSprites writes frame images as a single sprite sheet image.

Fig. 2 - Sprite sheet animation images written on a single image.

Fig. 2 shows how this sprite sheet looks like for one of knight model's one rotational levels. The key point here is to fill all cells in the sheet that there won't be any empty cells left. Resolution of the sheet is dependent on a resolution of one cell. Let's say we want to have 64x64 resolution cells (as it will be the actual resolution of the sprite displayed in the game). This would give w=64x6=384, h=64x4=256, or in other words the image of 384x256 resolution. When animation is running, the billboard zooms in on the center of the required frame to display the unit.

So the main point here is again to find correct indices between models, animations, rotational levels and frames. This is done in a similar way like it was done in RenderMeshModels.cs classes. The difference is rotational levels. They are found by using camera and unit positions and rotations. Once all indices are found, the correct image is being used on billboard to display the unit. `System.Math` functions has been used instead of `Mathf` to reduce computational cost even further. Fig. 3 shows how the overall scene looks like with resolved details on each individual unit when using 3dSprites.

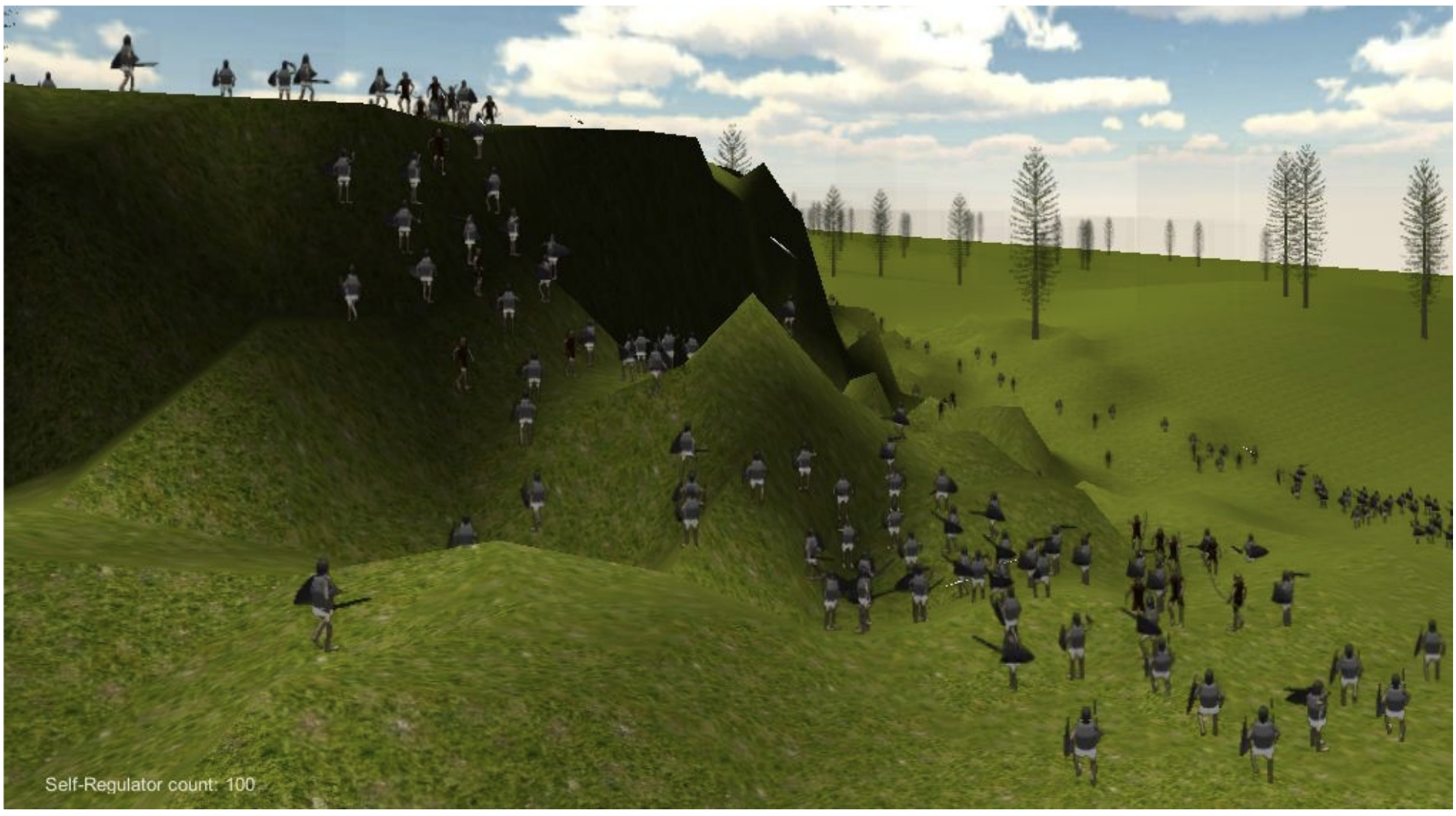


Fig. 3 - 3dSprites uses different models, rotational levels, animations and animation frames to display large number of units and show what they are doing.

3dSprites uses Unity's Shuriken particle system to render 3dSprites as particles. Shuriken is very highly optimised rendering pipeline, being able to handle many thousands or even ten thousands of particles at playable FPS. So the idea is that 3dSprites use unit as a single particle to display. Animation is playing itself though Texture Sheet Animation property as a particle is allowed to evolve over time. Once the particle reaches the end of animation, it's starting time is being reset and animation starts to play again from the beginning. When switching between rotational levels (the switching produces instant jump from one rotational level to another), particle is moved from one particle system to another (it's being killed on the old system and emitted on the new system). Particle system switch is happening only when one of the rotational index is changed comparing with the previous rotational index. It's also moved to another particle system if unit animation is changed.

As for a given example there are created 3x12x6x4=864 particle systems on runtime. It's very expensive to have such a large number of particle systems, so systems, which has no particles in them are being disabled (gameObject is set active to false). Only a small number of particle systems is active at a given time. However, during intensive battles the number of active particle systems can increase (as there is large variety of animations playing) more dramatically, making FPS to drop. Classes in **PSpriteLoader.cs** are the ones, which are controlling all particle systems and switches between them.

One more thing is to take care of switching between 3dSprites and 3d models. This task is taken by RenderMesh class, where is checked distance from camera. If unit is in a correct range of distances, it's being switched to 3dSprite. When switching, animation phase index is remaining the same, as animation start time is written in UnitAnimation.cs, which is used by both RenderMesh and PSpriteLoader. 3dSprites are set through LODs in RenderMeshModels in a similar way like it is done with mesh LODs. The only difference is that user specifies for LOD Modes to be "1" for corresponding LOD, which suppose to be using 3dSprites (see Fig. 1).

3dSprites is a separate asset, available here:
https://www.assetstore.unity3d.com/en/#!/content/50386
Which can be used for any kind of game, not just RTS. In RTS Toolkit user can use 3dSprites but can't create them for new models. To create 3dSprites for new models user needs to get 3dSprites asset from AssetStore, bake models there and when baking finishes, import images (or asset bundles) to RTS Toolkit. 3dSprites parameters are written in text files within Resources/ 3dSprites/config directory, which can be modified manually if needed.

2.1.2. Units movement

Moving large number of units is one of the most expensive parts in RTS. Static units, like building, never move, while dynamic units, like archers, knights, etc. can move. The movement is happening on the terrain, so units should be following the terrain. It can be that the terrain is not smooth and has mountains, lakes, etc. where units should not go through. These problems are solved by using pathfinding. In general pathfinding task is to find shortest path for a unit to reach its destination. If it's a terrain, it's most natural that paths should be leading around mountains, cliffs, lakes and other unwalkable areas. Once paths are being found, units are starting to move along them. The easiest way to understand a path is to keep in mind that it's a sequence of points, through which units has to move. Pathfinding can be used after navigation is being configured. RTS Toolkit by default uses Unity's Navigation, but there is also the extension for A* Pathfinding Project for users who has this package.

Unity Navigation is using **NavMesh**, which needs to be baked in Editor before using pathfinding. Each unit gameObject, which can move use **NavMeshAgent** component to perform movement. The pathfinding and movement is triggered through **NavMeshAgent.SetDestination()** function. This is done only once, when unit starts to move. Later movement is driven automatically by NavMeshAgent. No other type of movement for ground units is made as through SetDestination() function.

While pathfinding problem for large number of units can be solved by searching paths only once, another problem of **local avoidance** is more difficult to solve. Local avoidance ensures that units does not go onto each other. This is quite easy to handle when number of units is small, but with larger numbers, local avoidance becomes one of the most serious problems. To reduce computational cost it requires to search for neighbours, which can be found by using kdtrees, octrees and some other fast algorithms. However, even with them trees has to be rebuild frequently as positions of moving units are changing. Mostly crowded areas (i.e. battle zones) could be changing very rapidly, requiring frequent recalculations. By default RTS Toolkit uses NavMeshAgent which has local avoidance itself. It can easily handle about 1000-2000 units at playable FPS without any additional components. However, tests shown that manually searching paths and moving units without local avoidance at all gives very significant boost in performance, what means that largest computational cost comes just from local avoidance. A* Pathfinding Project allows to control local avoidance much better. However, majority of control comes from reducing frequency of calculations. This does not always gives best solutions, as units starting to miss local avoidance and starting to go on each other in crowded areas. So the problem remains mostly open and is out of scope here in RTS Toolkit. Still, properly chosen configurations gives good results for up to several thousands of units in the scene.

Flying units, such as birds, does not use navigation, as they can move to any direction. Instead of that, they are using manual changes in transform.position in every update. Birds and other animals usually use culling distances, which are used for units being at larger distances than the last LOD outer distance. As a result, these units are being moved into low update frequency coroutines as well. Flying units also uses terrain heights in order to follow terrains on the same height from the surface.

Unit movements can be set by using **UnitsMover.cs** functions, which allows also to switch between idle and move animations if units temporarily waiting on their paths.

2.1.3. Bakeable navigation

From Unity 5.6 it became possible to bake Unity NavMesh on the runtime. The feature has been implemented immediately to work through NavMeshSurface.cs component within Unity's Navigation components API (https://github.com/Unity-Technologies/NavMeshComponents). RTS Toolkit uses UnityNavigation.cs component (Figure 4) in order to set navigation in the way that it would be rebaked on runtime when new terrains are generated (when camera moves between terrain tiles).

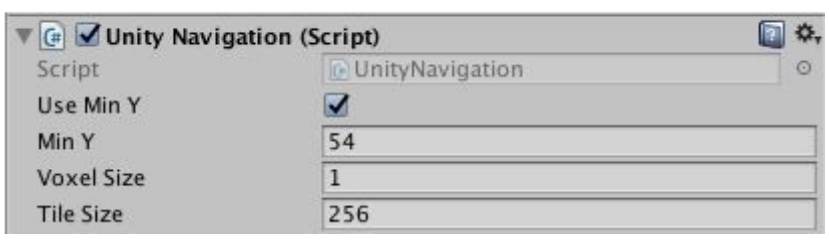


Fig. 4 - UnityNavigation.cs component setup.

useMinY allows us to set if minimum height for navigation is used with the value specified in **minY**. Below minY navigation is not baked. This is very useful when there are lakes, rivers and sea, where below water level is no navigation used. When spawning units, it should be taken care that units would not appear in empty areas with their NavMeshAgents components (this would give warnings and errors). By default, spawning in game is checked: if unit appears in the place where is no navigation, it's not spawned (this can be detected by checking if terrain y value is not bellow minY value specified in UnityNavigation.cs). **voxelSize** allows us to set resolution of the NavMesh for all bakes during the runtime. Lower values gives better resolution, but takes longer to bake. The player does not getting lagged while baking happens as NavMesh baking is running asynchronously. However, larger resolution bakes could take longer and player won't be able to use the terrain for moving units and creating buildings there is navigation is still baking. **tileSize** allows to set tile sizes for all bakes during the runtime.

2.2. Character setup

In depth video tutorials (v2017.2) available on YouTube: https://www.youtube.com/watch?v=JrRMfBEf6rA&list=PLOBSjXd5DmZNO7JWcSuwpYlUTeq2UWGUt

As RTS Toolkit has it's own complex systems for movement, rendering and animations, we will go through the points of how existing characters are set and how new characters can be brought to life. We will start with mesh model characters and then discuss about 3dSprites.

2.2.1. Importing

The first step to do is to make a model of character, which you would like to use in the game. Different software can be used to do modelling and animations (i.e. Blender, 3ds Max, etc). Models, used in RTS Toolkit has been created in Blender and exported as FBX files, which were placed in Assets/RTSToolkit/Models directory. Each model has been placed inside its own subdirectory. Fig. 5 and 6 show how static and animated parameters looks like.

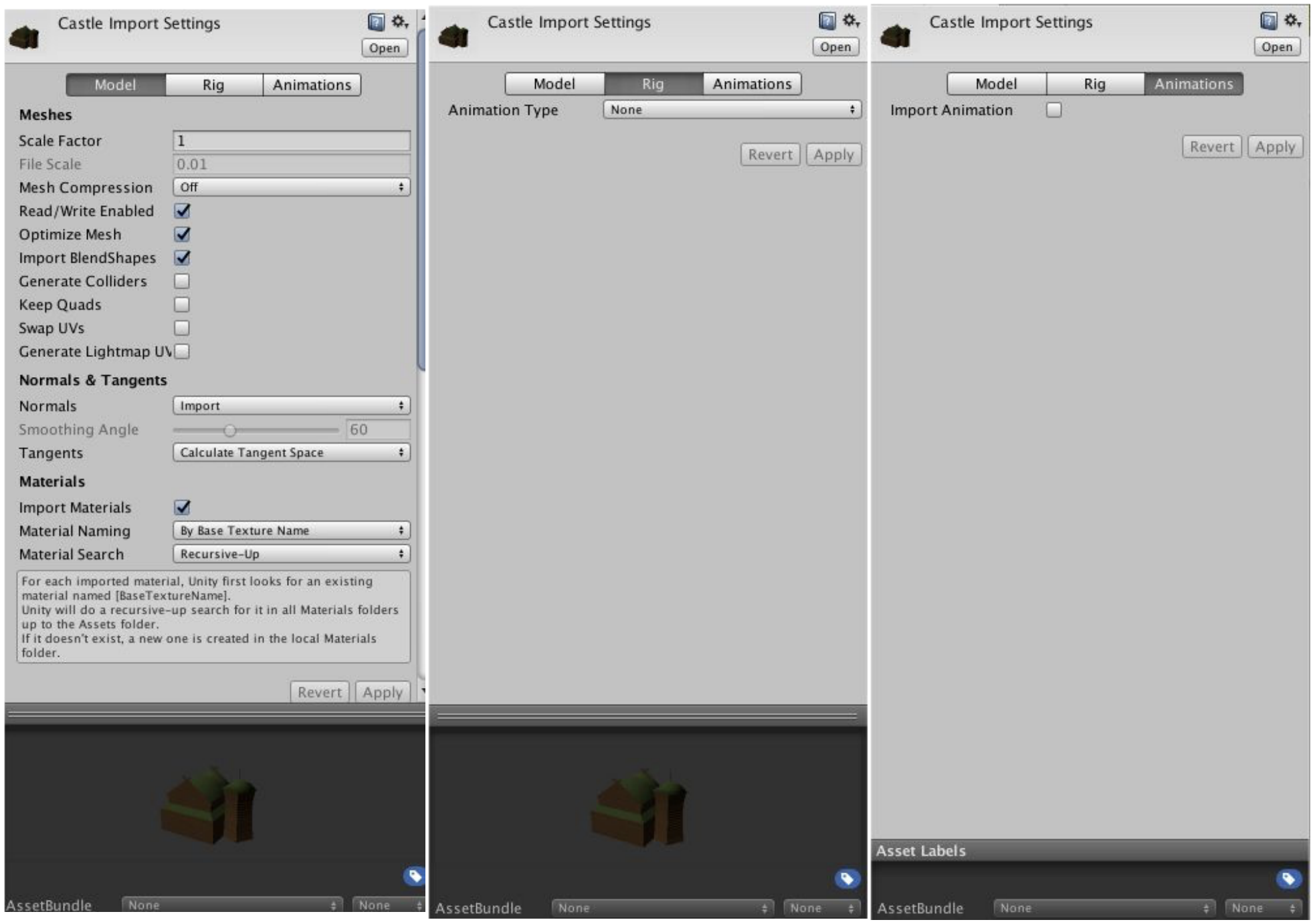


Fig. 5 - Non-animated model import settings.

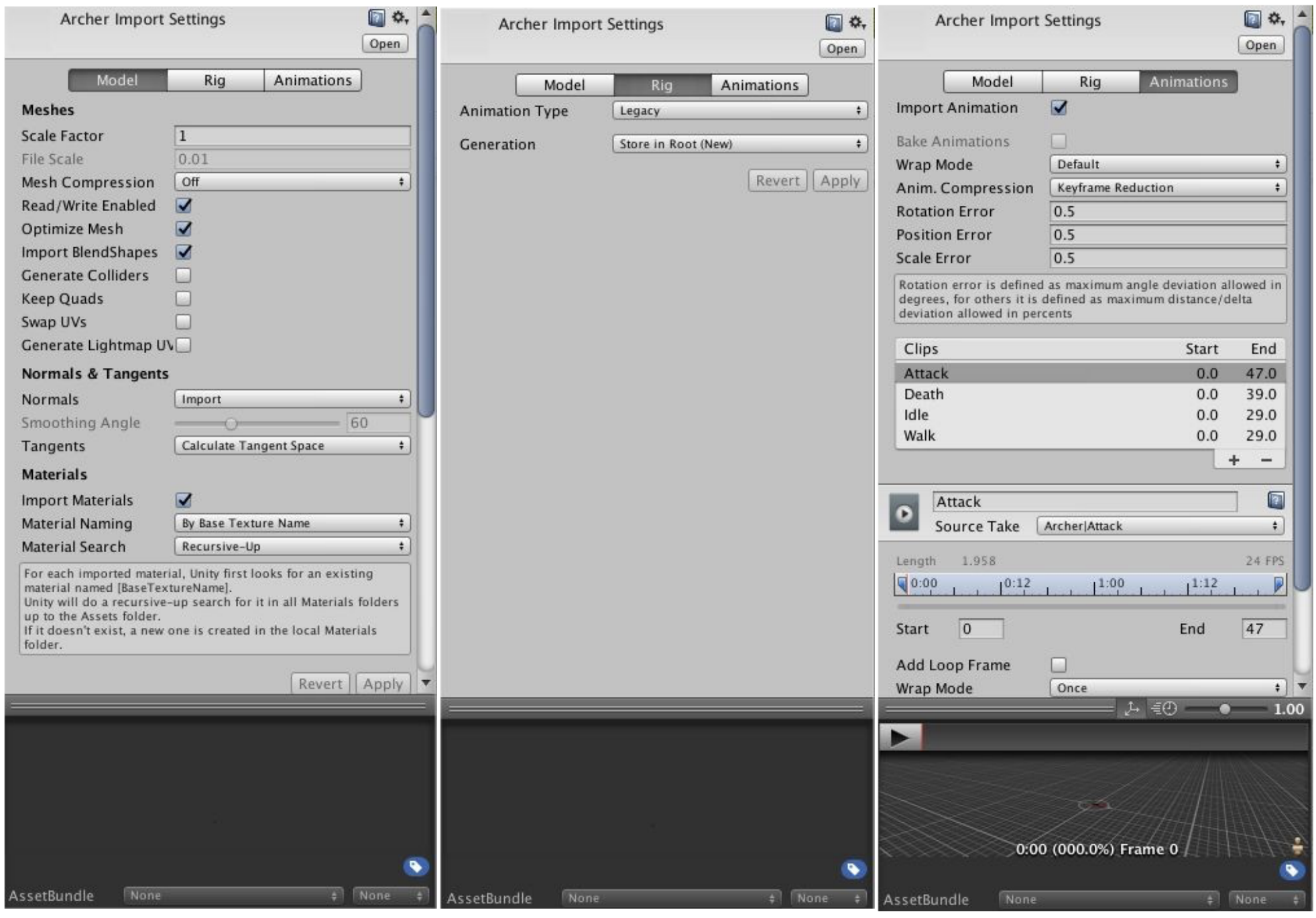


Fig. 6 - Animated model import settings.

The main differences are in Rig and Animations tabs. In Rig tab non animated models RTS Toolkit uses Animation Type as None and in Animations tab Import Animations is disabled. For animated models Animation Type is used as Legacy and Import Animations is activated. There should be also visible a list of all animations model has with their names and durations. This list is being used to set up all animations in RenderMeshModels later. There are 4 animations for the given model in Fig. 6. Some animations, like death and attack has Wrap Mode set as Once, while other (Idle and Walk) as Loop. If it is set to Once, the animation will be not repeatable - it will stop at the end of animation and display the last frame all the time. If it's set to Loop, then animation will be repeated again and again.

The next step to do is to drag and drop model into the scene to check if it is visible. If model is not visible, it could be just because it is very small and may need to scale up. Different software used to create models has different parameters and it is always good to keep in mind that scaling might be needed one way or another.

Once the model displays correctly, find it's gameObject instance in the scene and check for components. The difference for non-animated and animated objects is shown in Fig. 7 and 8. Non-animated object has mesh renderer component, which is used to display its mesh.

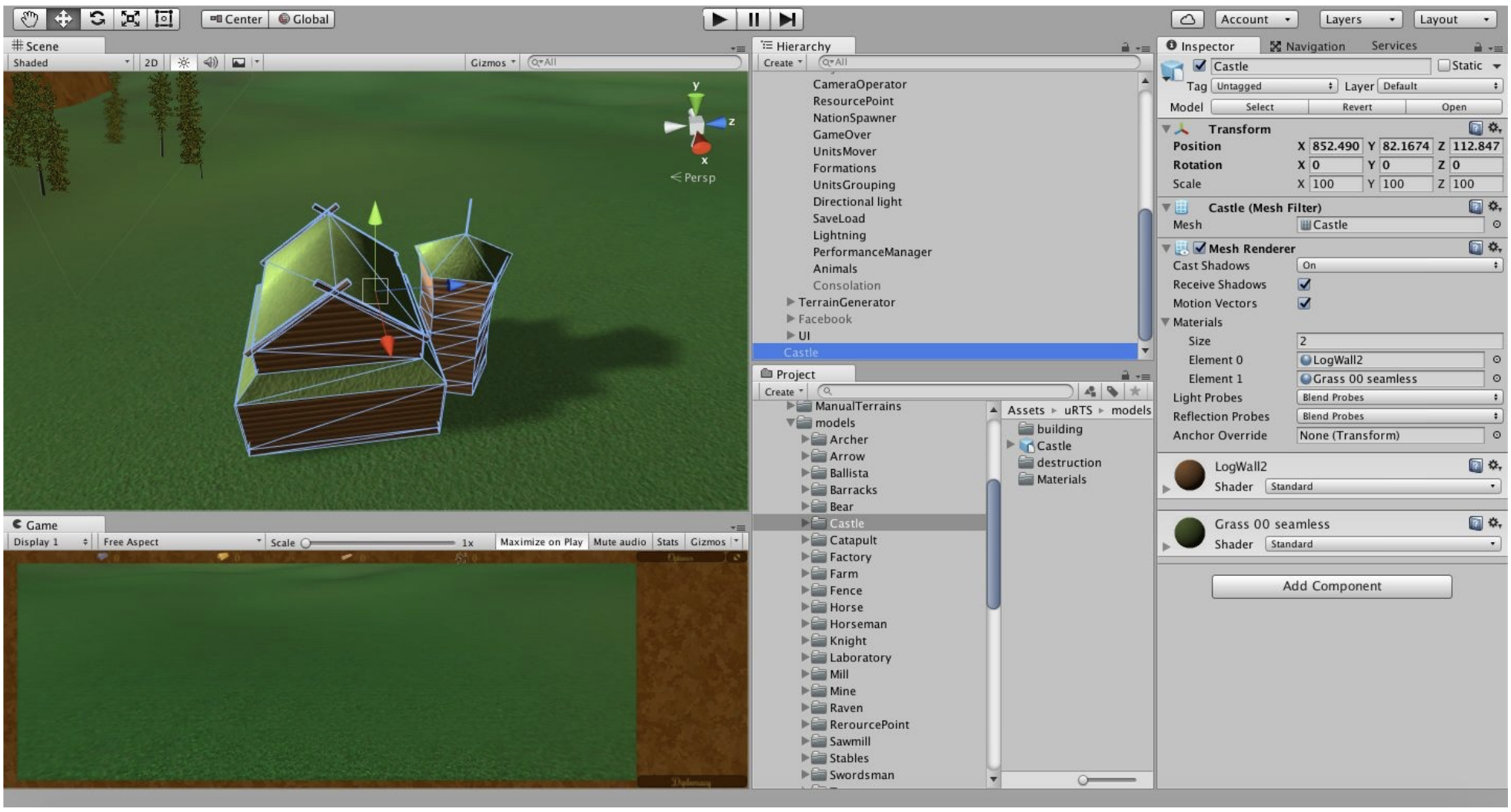


Fig. 7 - Non-animated object components.

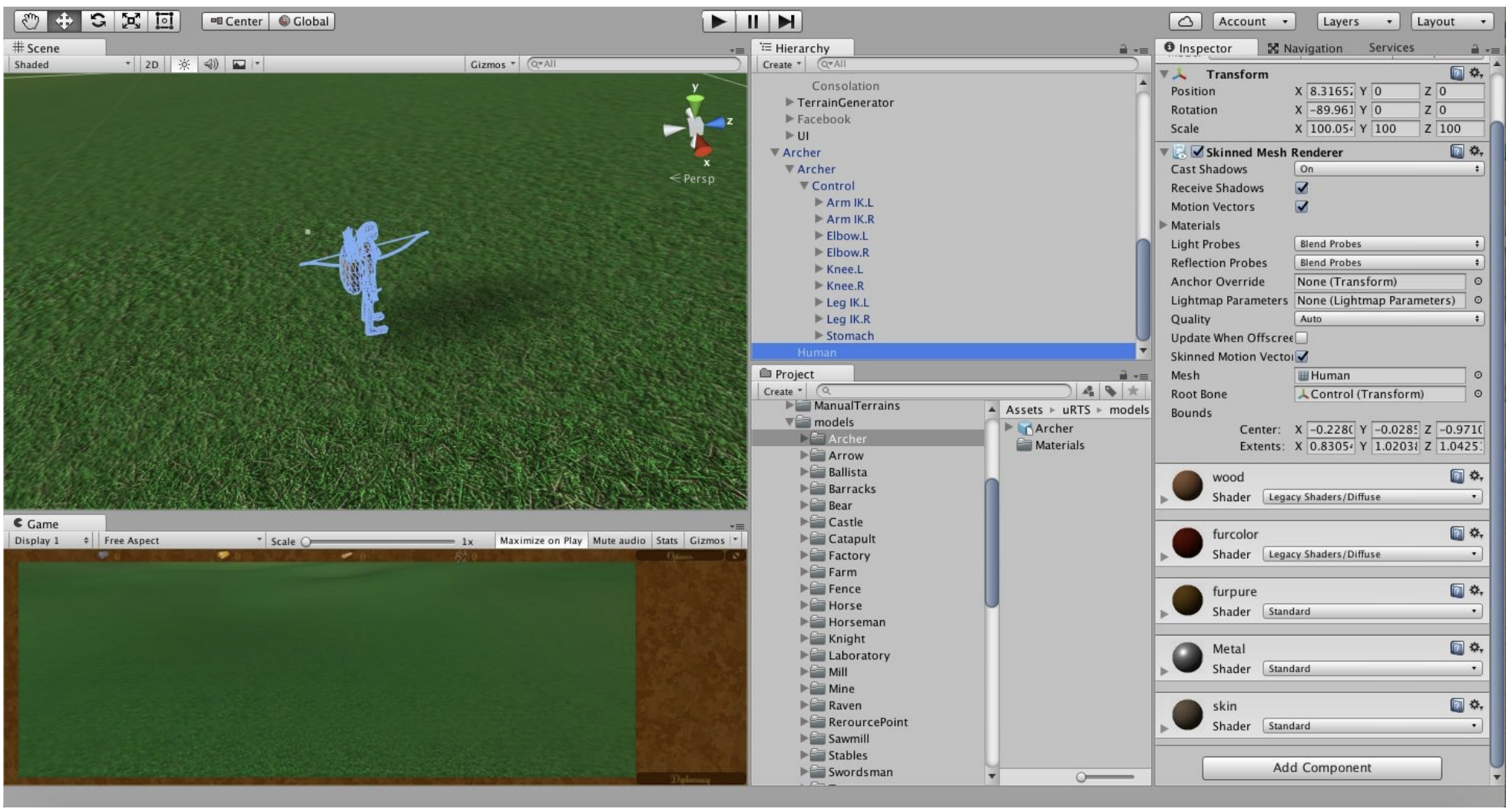


Fig. 8 - Animated object components.

Animated object is more complex - there are several children gameObject below the main gameObject. One of child gameObject has SkinnedMeshRenderer component and materials, used to render it.

RTS Toolkit uses static unit prefabs, which are very similar for static objects like in Figure 2.7, but does not use SkinnedMeshRenderer components for animated units, and instead is set through RenderMeshModels.

2.2.2. Prefab setup

RTS Toolkit used prefabs are stored in Assets/RTSToolkit/Prefabs/RtsUnits directory. It can be noticed that buildings (static units) usually has these components attached to them: **MeshFilter**, **MeshRenderer**, **UnitPars**, **NavMeshObstacle** and some of them **SpawnPoint**. Animated units has these components: **UnitPars**, **UnitAnimation** and **NavMeshAgent**. When creating a new prefab, it could be good to experiment by setting up prefab with exactly the same components. There is also possible to copy components and component values from one prefab to another by using ⚙ icon in the right hand side of each component.

2.2.3. RenderingMeshModels setup

As it can be noticed, units does not have any mesh rendering components. This is because their rendering is being set through RenderMeshModels. UnitAnimation is the one, which adds unit instance in RenderMeshModels. So these prefabs are like “empty” gameObjects, triggering their rendering through RenderMeshModels.

In order to use units through RenderMeshModels, they has to be registered there with their name and the original model (from FBX file). When units are registered, calls from UnitAnimation can be identified and instance rendered.

RenderMeshModels can be used in the scene just once. Parameters of RenderMeshModels are shown in Fig. 9).

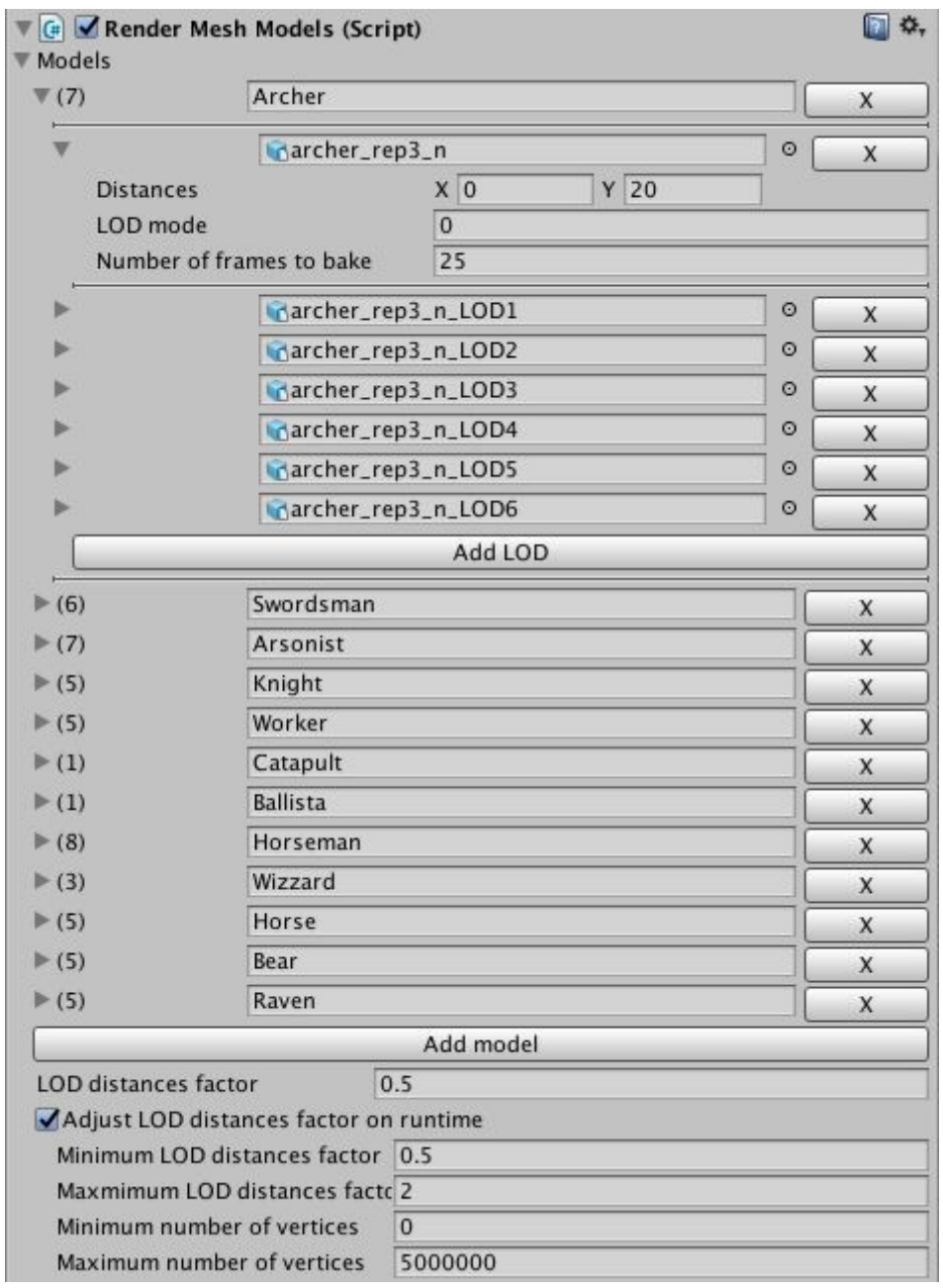


Fig. 9 - RenderMeshModels.

There is a list of models and inside each model are lists of LODs. When creating a new RenderMeshModels object, this list will be empty. New models are added by clicking “Add model” button. Once the model is added, the **modelName** entry appears. In Fig. 9 first entry is for Archer unit and so the model name is “Archer”. This modelName has to match with the **modelName** in **UnitAnimation**, as this is where registered models are being found. There is “X” button which allows to remove unwanted model from the list.

Model contains a list of LODs. When starting new model this list will be empty. New LODs for a model are added by pressing “Add LOD” button and removed by pressing “X” buttons near each element. At the top of the element prefab, which will be representing corresponding LOD, has to be specified. This prefab will be used to get SkinnedMeshRenderer components and set animations. Then there are 3 fields to be filled “**Distances**”, “**LOD mode**” and “**Number of frames to bake**”. Distances specifies the range of distances between which LOD is visible. LOD mode allows to specify if the LOD is actual 3d model (0) or 3dSprite (1). Finally number of frames to bake allow to specify how many static meshes will be baked from model animations at this LOD. Usually high resolution LODs can have

larger number of frames to get smooth animations, while more distant low resolution LODs can have smaller number of frames. As units are using instanced shader, setting number of frames can help to achieve better instancing with lower number of meshes and materials list possible.

Additionally **LOD distances factor** allows to set fractional distances to shift all LOD distances up (above 1) or down (below 1). This helps when designing quality of the game - if the machine has low computing power, it's good idea to shift distances factor down. On the other hand when designing high quality game, it might be better to shift factor a bit up.

"**Adjust LOD distances factor on runtime**" allows to run additional calculations of how many vertices are currently visible in the scene and decide which LOD distances factor is best to use at particular situations. LOD distances factor can be changed between "**Minimum LOD distances factor**" and "**Maximum LOD distances factor**" which is linearly interpolated from "**Minimum number of vertices**" and "**Maximum number of vertices**". E.g. in Fig. 9 LOD distances factor would be assigned to 2 if there are 0 vertices in the scene, and to 0.5 if 5M vertices are reached. Bellow and above these bounds LOD distances factor would stay the same. However, adjusting LOD distances factor on runtime can give many visible flippings between LODs in a short amount of time, what may look not natural.

2.2.4. 3dSprites setup

3dSprites are being set automatically from the directory, where 3dSprites files are being copied.

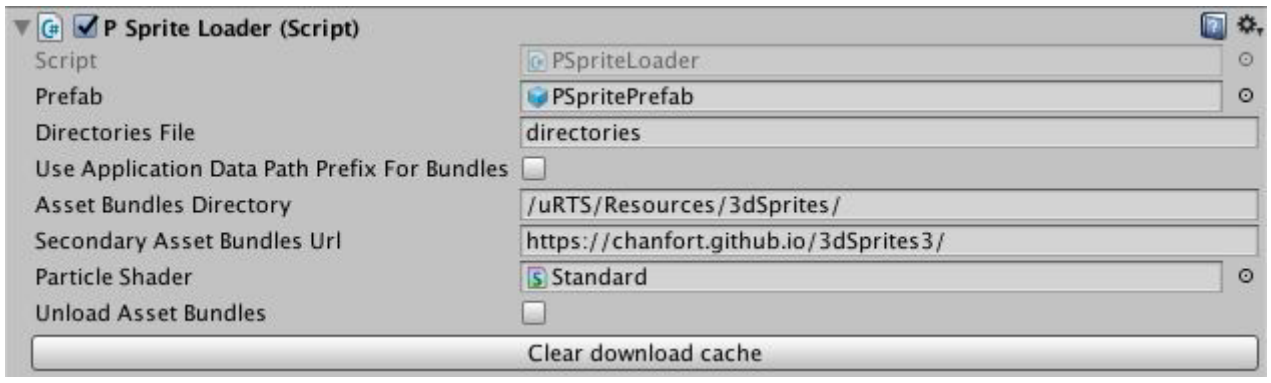


Fig. 10 - PSpriteLoader loads models, which uses 3dSprites.

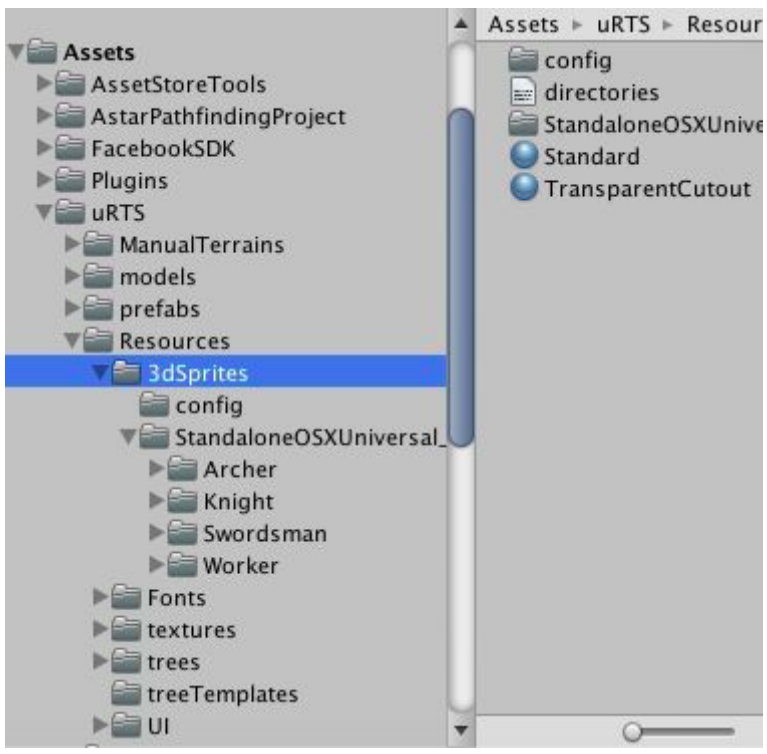


Fig. 11 - Default location of files, used for 3dSprites.

There are 3 different ways to set up 3dSprites. One way (the old one) is to use PNG images as textures, second way is to use asset bundles from local directory and the third way is to use asset bundles from the web (i.e. set up in the website and using public link). PSpriteLoader is the script which handles 3dSprites. In the Fig. 10 is shown how it looks like when used with asset bundles, which are present in the project. In RTS Toolkit there are 4 models, which are using 3dSprites: Archer, Knight, Swordsman and Worker. By default their files are in Assets/RTSToolkit/Resources/3dSprites directory. There is important directory text file, which is used to map all subdirectories for models and their animations. Config directory there contains configuration files how models and animations should be set. All these text files can be edited by hand to change animation settings at any time. For asset bundles 3dSprites saves all models inside directories names as [BuildTarget]_bnd, where BuildTarget here is asset bundle platform (for different platforms asset bundles has to be re-build separately). First time PSpriteLoader attempts to load asset bundles from local machine and if it can't (i.e. no files or files in other directories) at the second attempt files would be downloaded from RTS Toolkit publisher url. If user is preparing to publish the game, make sure to double check if 3dSprites are actually showing in the final build. If not, check in-game console for warnings (i.e. if loading or downloading attempts failing).

Using asset bundles can be good approach as files does not need to be included in the build, they also do not load at the start immediately, but instead downloads by using coroutine from the web. This allows to reduce significantly project size and game startup time (especially on WebGL builds).

In 3dSprites directory can be also found several materials (i.e. Standard and TransparentCutout in Fig. 11), which are used by particle systems. If these materials are not stored in Resources directory, shaders will work only in Editor runs but not in the actual builds, making 3dSprites not visible.

2.2.5. RTSMaster

When prefabs with their components has been created and saved in the project (i.e. default prefabs are saved at Assets/RTSToolkit/Prefabs/RtsUnits), they have to be registered in RTSMaster list rtsUnitTypePrefabs to give the unique identification of RTS unit. The array index at which unit is placed in rtsUnitTypeprefabs has to match with rtsUnitId value written in UnitPars component for the given prefab. Fig. 12 shows how this is set in rtsUnitTypeprefabs list and UnitPars for Archer unit. It can be noticed that Archer prefab is registered as Element 11 in rtsUnitTypeprefabs. But there is also rtsUnitId = 11 on Archer's UnitPars. The same logic applies for all 20 units registered in rtsUnitTypeprefabs. This indexing is being used to directly access the original prefab from many other scripts and also to identify unique unit types used in RTS Toolkit.

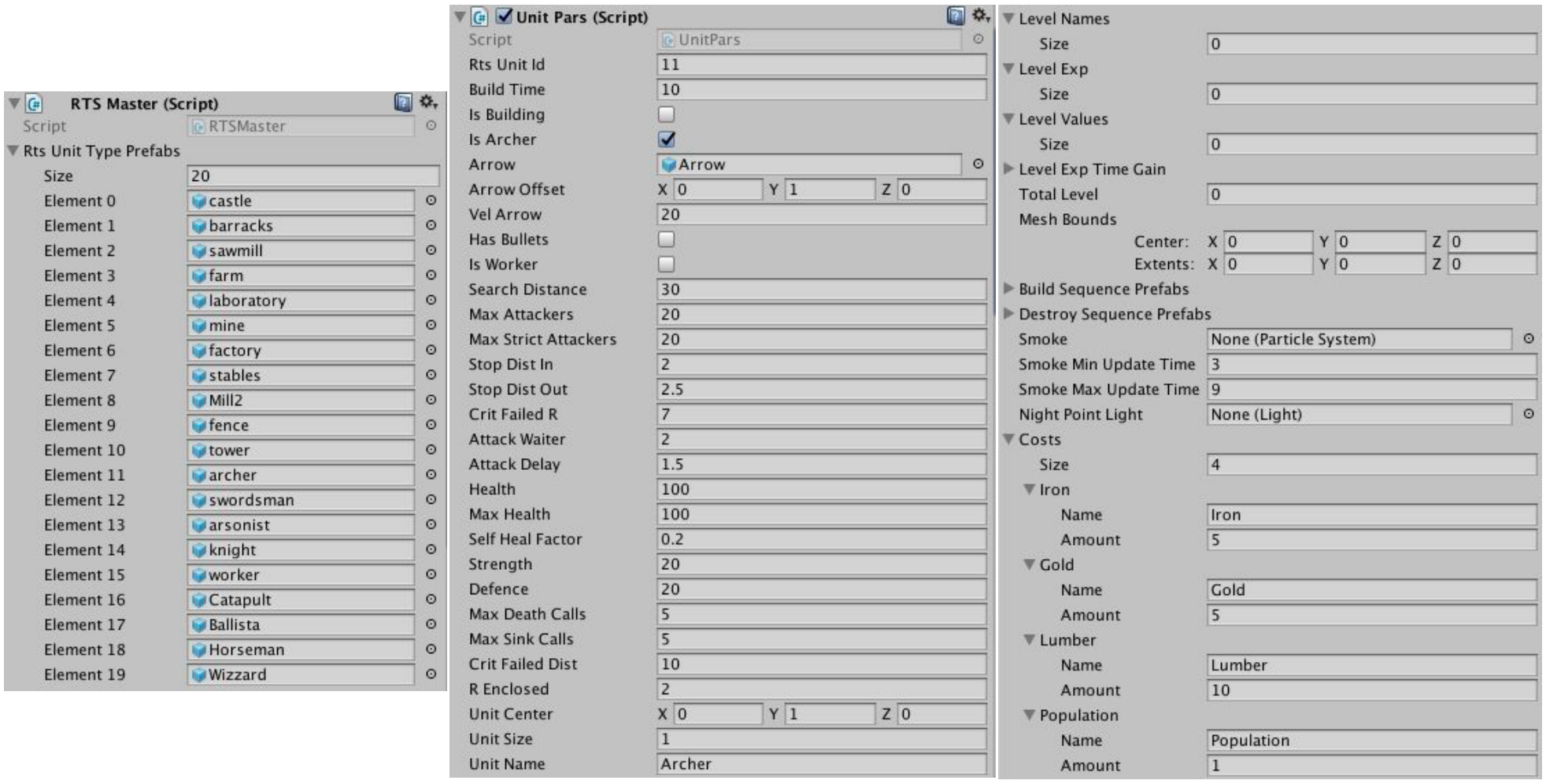


Fig. 12 - RTSMaster and Archer UnitPars with their values.

2.2.6. UnitPars

UnitPars is the main component for each unit and building. It is being used to store data of each unit instance in game. The following public variables are used:

rtsUnitId	Unique RTS unit type id
buildTime	Time in seconds of units to be created
isBuilding	Is the unit building
isArcher	Is the unit archer
arrow	Archer arrow prefab to be used when shooting
arrowOffset	Arrow spawn position relative to archer's pivot point when shot
velArrow	Arrow terminal velocity
hasBullets	Does unit has bullets
isWorker	Is this unit a worker
searchDistance	Distance within which unit can search for its targets
maxAttackers	Maximum number of attackers the unit takes from automatic search (used to distribute attackers over their targets)
maxStrictAttackers	
stopDistIn	Distance from the target at which attacker can start attack their target

stopDistOut	Distance at which attacker should stop attacking the target and come closer to it in order to continue attacking
critFailedR	
attackWaiter	The period of time to wait between attacks. If the time difference from last attack is larger than period attack happens immediately. Otherwise, the unit continues to wait until the full time attackWaiter passes from the most recent attack.
attackDellay	Time delay after which damage is applied to target (used to synchronize damage with animations)
health	Current health of unit
maxHealth	Maximum health unit can have
selfHealFactor	Amount of health to be added to unit health per second. Positive values will do healing, negative will make damage
strength	The strength of the unit
defence	The defence of the unit
maxDeathCalls	Number of calls for unit to spend in Death phase (can be used to increase/decrease the time of body remains visible on the ground)
maxSinkCalls	Number of calls for unit to spend in Sink phase
critFailledDist	
rEnclosed	Enclosed radius of the unit
unitCenter	Position of unit center point relative to the unit pivot
unitSize	Size of the unit
unitName	Name of the unit
maxChopHits	Number of hits worker makes to the tree before it take logs (1 hit is applied every 0.1 seconds)
levelNames	List of level names unit has
levelExp	List of experiences for unit levels
levelValues	List of unit level values
totalLevel	The combined total unit level
meshBounds	Mesh bounds used by building units
buildSequencePrefabs	Sequence of meshes to be played as animation when building is being builded up
destroySequencePrefabs	Sequence of meshes displayed when building is being destroyed
smoke	Particle system, which produces smoke (i.e. smoke rising from chimneys of some buildings)
smokeMinUpdateTime	Minimum update time between enabling/disabling smoke particle emission cycle. The value is chosen between minimum and maximum for each cycle. I.e. if value is chosen 4.0 it means that 4 seconds smoke will be emitted from chimney. If after that random will be chosen as 5.0 it would give that 5 seconds the building won’t emit any smoke. The next random will bring how long it will emit again and so on.
smokeMaxUpdateTime	Maximum update time between enabling/disabling smoke particle emission cycle
costs	The list of costs, which are used in order to spawn building. Each list element is an object which has name and amount properties. name should be referring the name of the resource defined in Economy.cs. amount uses the actual amount of how much unit cost. By default there are 4 types defined: iron, gold, lumber and population. If this list does not contain one or several values, then the unit costs nothing on that particular resource.

2.2.7. SpawnPoint

SpawnPoint is the component, which is being used to create new units in game. As units are being spawned by player using UI, or AI, there are no parameters, which could set through Editor to affect SpawnPoints. Once player or AI triggers SpawnPoint, it starts spawning units (i.e. Barracks spawns Archers) by the amount player asks. AI always spawns units one by one, while it checks decisions between spawns when it is the right time to spawn.

2.2.8. UnitAnimation

UnitAnimation script is attached only to units, which are animated. The two key parameters here are **modelName** and **animName**. modelName allows to identify unit in RenderMeshModels. animName is the name of animation which starts when unit is being spawned.

2.2.9. Dynamic buildings

Dynamic buildings are buildings which has separate moving parts, which movement is controlled from the script. A good example of dynamical building is Windmill building, registered with rtsUnitId = 8. Windmill has its base and rotating blades. Base and blades are made as different models, and blades are used as a child gameObject on the base. There is written custom RotatingPart.cs script, which controls how blades are rotating. Windmill blades rotate based on wind speed and direction. As the wind change its speed and direction, Windmill blades will be rotating faster or slower. As different Windmills are build with different rotation, they all will react to wind differently at the same time. RotatingPart.cs ensures that Windmill blades rotates correctly. This is just one example, but creative developers can make their own dynamical building for any kind of purpose. A good point is that question of dynamical buildings can't be generalized and is a part of art instead. However, artists should keep in mind that epic dynamical buildings/units with large number of moving parts can be computationally expensive and could come with the cost that player won't be able to set large number of such units in game.

2.2.10. Animals and birds

Animals and birds are specific characters, bound to terrain and environment. They are mostly to bring cosmetic effects to the game. Animals are ground units and use navigation while birds are flying over the terrain anywhere and do not require for terrain below to have navigation baked. Both animals and birds are moving randomly. By default in RTS Toolkit there are set two type of animals - bears and horses, and the raven birds.

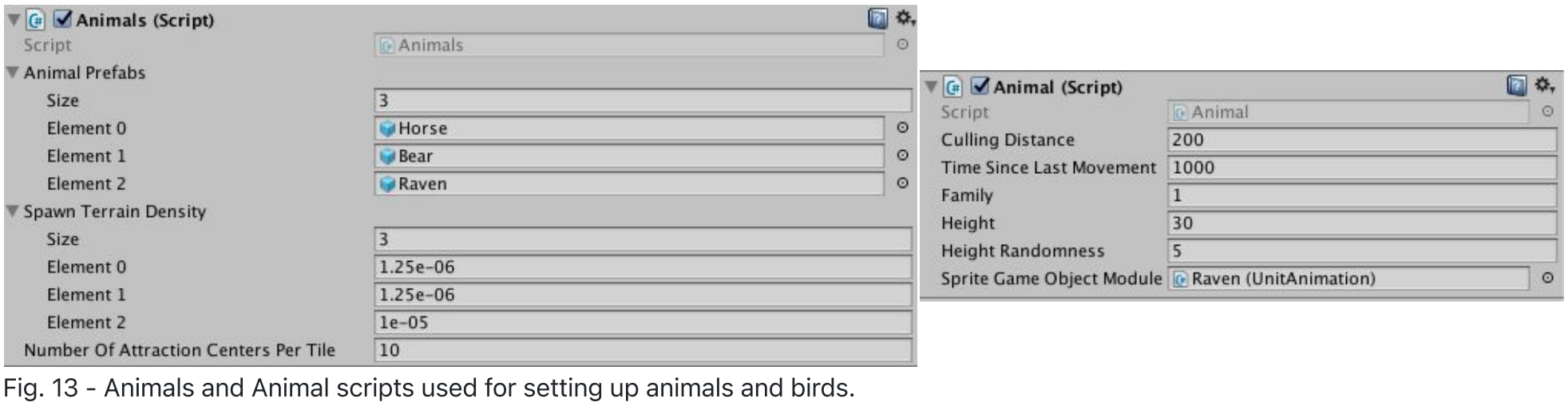


Fig. 13 - Animals and Animal scripts used for setting up animals and birds.

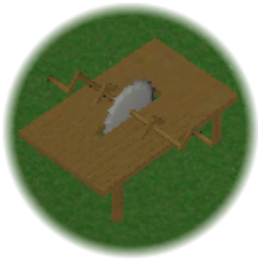
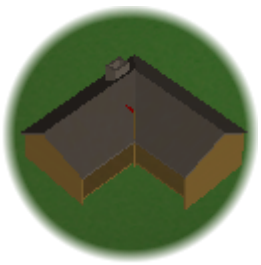
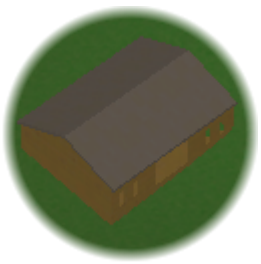
Animals.cs is the one, which controls spawning, movement and unloading animal instances. Animal.cs stores prefab and instance specific data and is attached on each animal prefab (Fig. 13). Animals.cs has a list of **animalPrefabs**, where animal prefabs should be registered. **spawnTerrainDensity** specifies how many animals are instantiated per square unit on the terrain. This number should quite small as on several thousand kilometers are spawned only several animals. Animal.cs (which is attached onto each animal prefab) specifies **family** (0 - for ground animals, 1 - for birds), **height** over the terrain and **heightRandomness** for birds. Animals are being rendered through RenderMeshModels, so their prefabs needs to be registered there as well. UnitAnimation component also needs to be attached on each prefab.

Birds are the ones, which do not follow navigation and does not require pathfinding. Instead they are using attraction points simulation system. There are placed **numberOfAttractionCentersPerTile** on each terrain tile. Each bird flies towards combined direction, derived from distance weighted vectors summation from attraction centers. This allows that birds are flying and spinning in a natural behaviour when following attraction centers. Attraction centers are set to move themselves and that adds nice looking grouping and clustering of birds over the terrain.

2.2.11. RTS Toolkit default characters

Here is the list of RTS Toolkit installed default characters.

	Central Building. This is the main nation building from which town creation is started.

	Barracks. This building allows to spawn light military units, such as Archers, Swordsman, Arsonists and Knights.
	Wood Cutter. Workers brings lumber to Wood Cutter after they cut logs from trees. Can be build anywhere, where is free space.
	House. Increase nation population.
	Research Center.
	Mining Point. Can be build on hotspots where iron or gold is found.
	Factory. Used to create fences, towers and war machines, such as catapults and ballistas
	Stable. Used to create horsemans.
	Windmill. Increases spawn rates for units.
	Fence. Fortify places, units can’t pass through it, unless they break through.
	Tower. This is the only building which performs attack. Shooting arrows into enemies.

	Archer. Light unit, shooting regular arrows.
	Swordsman. Melee unit, attacking from the close distance.
	Arsonist. Similar like Archer, but shooting fire arrows.
	Knight. Similar like Swordsman but stronger.
	Worker. Unit used to collect resources.
	Catapult. War machine, throwing stones with the large speed and causing massive destruction. Moves slowly.
	Ballista. War machine, throwing ballista missiles, which bursts and ignites other units and buildings around.
	Horseman. Fastest moving unit.
	Wizard. This unit can’t be used by player or regular AI nation and are wandering wilderness instead. They can be used to start creating quests. Wizards attack with lightning strikes.

2.3. Nations and diplomacy

Nations defines players in game. There can be two types of nations - player nation and AI nation. Player nation is controlled by player (i.e. players creates new buildings and units themselves), while AI nation develops town automatically.

2.3.1. Nation setup

Nations are being created by using **NationSpawner.cs** script.

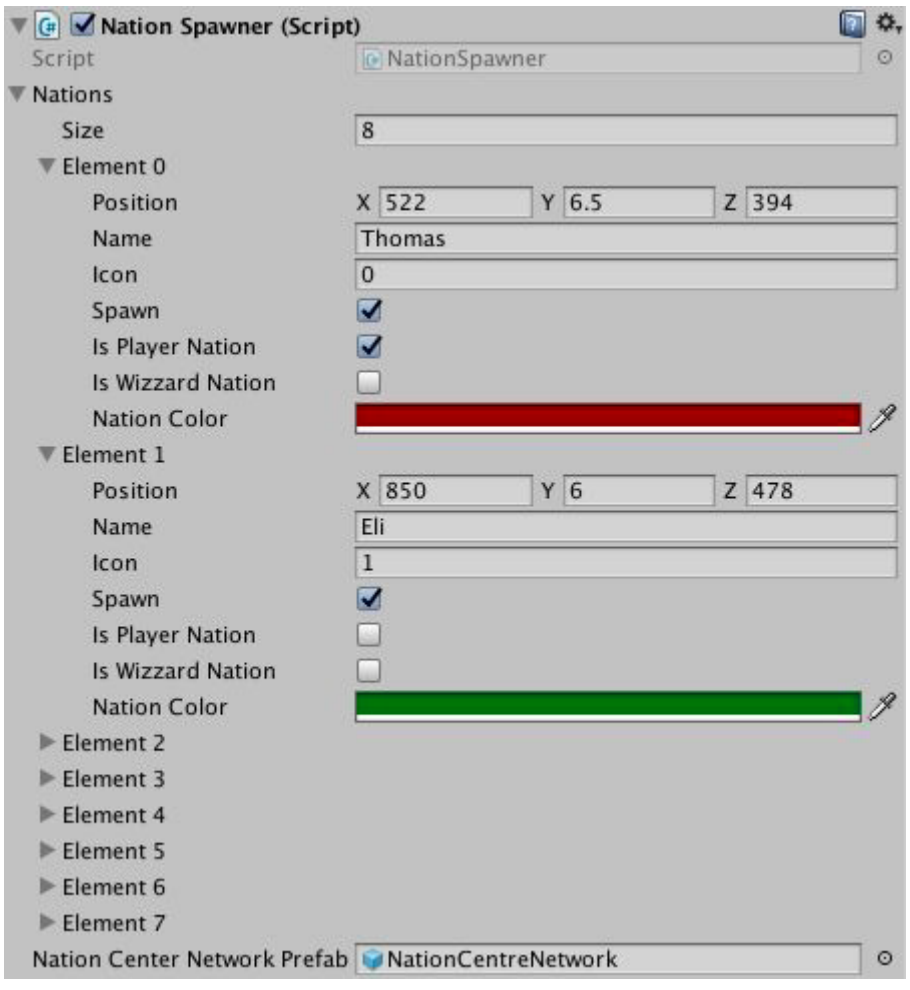


Fig. 14 - NationSpawner component shown for two nations - player and AI.

Fig. 14 shows basic setup of NationSpawner script for spawning player and AI nations. It has a list, which elements corresponds to nations being set. The list is created within Editor. Each nation has several parameters:

position	The position of nation centre to be spawned
name	Unique name of the nation
icon	The index of icon set in **NationListUI.cs** list **nationIcons** used for corresponding nation UI elements
spawn	If unticked, nation is not being spawned
isPlayerNation	If ticked the nation is player's nation. Only one player nation can exist in the game, other nations should be AI
isWizzardNation	If ticked, instead of a regular nation, wizard nation is spawned. The entire nation contains just a single character - the wizard himself.
nationColor	The color which represent the current nation. Unit's which belongs to this nation, have coloured thematic material (i.e. parts of swords, helms and other).

Once in play mode, nations are being spawned on the start of the game.

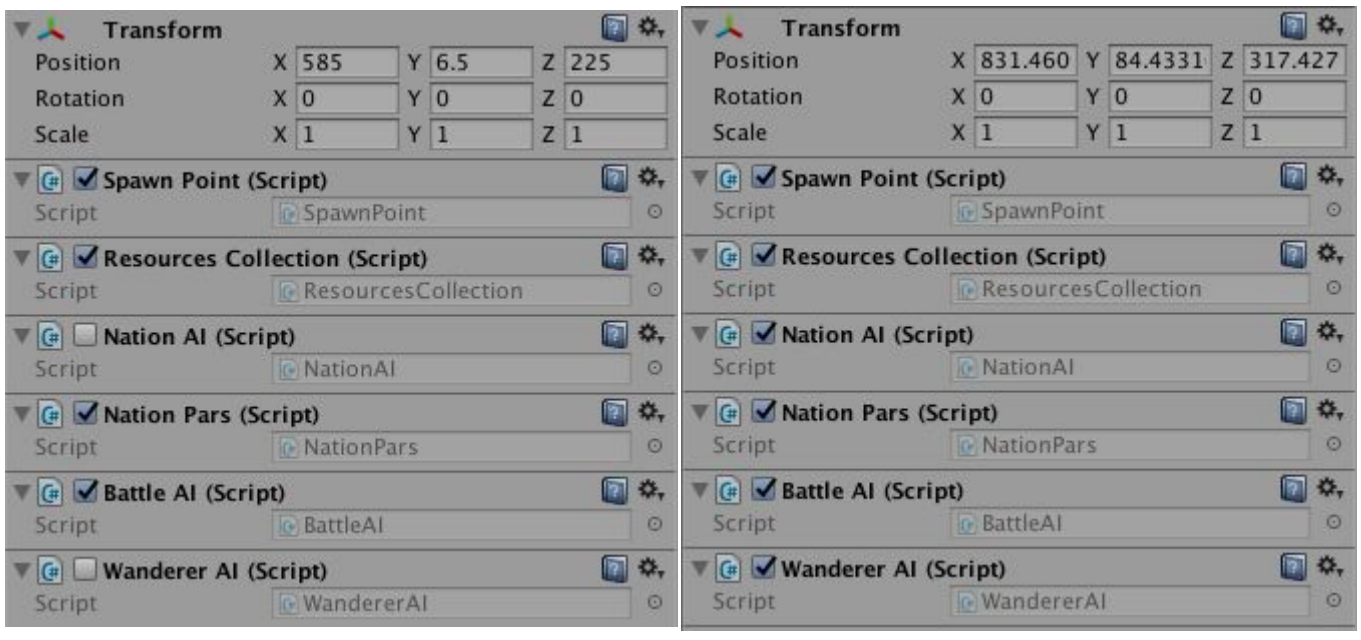


Fig. 15 - Player (left) and AI (right) nations spawned in game.

The nation gameObject is being created and **SpawnPoint**, **ResourcesCollection**, **NationAI**, **NationPars**, **BattleAI** and **WandererAI** components are being added to the nation. NationAI and WandererAI components are disabled on player nations.

SpawnPoint is used to spawn a Central Building and other buildings. On player's nation spawnPoint is triggered from **BuildMark** script, which starts spawning at the second click of player's chosen building position and rotation. For AI nation spawnPoint is triggered via NationAI automatically when town development progresses and AI builds new buildings.

ResourcesCollection is controlling the collection of resources for the nation through workers. Script starts its job when workers are being sent to start collecting resources. For player nation this happens when player clicks on a tree or on the Mining Point. AI workers are being distributed just after spawning on their resource collection tasks automatically from NationAI.

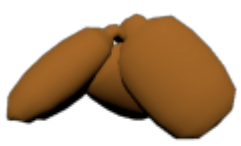
NationAI is the center, where various town development tasks are being performed. It creates new buildings, spawns new workers and military units, sends workers to collect resources, based on available points. It also checks and controls some basic diplomacy decisions. If units of other nations are too close and makes a threat, the nation proposes a war to other nation. If war progresses very long and both sides are winning more or less equally, the one, which loses several dozens of units, proposes alliance. Alliance can break up into war again if one of nations does not intend to leave their military units from another nation town. If one nation is losing heavily, it starts to be mercy. If another nation accept this mercy, the defeated nation pays half of their collected resources to the master nation. This slavery relation can end if enslaved nation restores economy and military power to a level that they could defeat invaders and claim back their freedom. NationAI is only applied to AI nations, as player nation builds, spawns and controls diplomacy manually.

BattleAI controls micromanagement of the battles for both - player's and AI nations. The script searches for each unit if there is no other nation units nearby. If there is and if relation between nations is set as a war, units are assigned other nation close units as targets and the attacking is starting up. BattleAI uses kd-tree for searching nearby units that cross-searches in the crowded fields would be not computationally expensive.

WandererAI is AI nation script responsible for balance between guarding the town and sending units to wander/attack other nations. After military units are spawned, they are distributed at random positions within the town. These units are guards, they stay in these positions all the time. If attack occurs nearby, guards move to the position to defeat enemy units and after that they return back to their assigned positions. When number of guards reaches larger values, about half of town guards are being set as wanderers and being sent to another nearby nation while another half stays in town as guards and keeps it protected.

2.3.2. Diplomacy and relations

The game uses advanced relations system to define what kind of relation is between two nations. Relations are being controlled by Diplomacy.cs. There can be only one **Diplomacy.cs** script set in the scene, which is used for all other nations. Relations data is being stored in Diplomacy.cs **relations** list, which is 2 dimensional integers list. It scales together with number of nations playing in game to make all cross-relations well defined, i.e. if there are 2 nations, there will be 2 relations, if there are 3 nations, it needs 6 relations and so on. Integer values are representing the following relations:

	-1. Undiscovered. If distances are large between two nations, they treat each other as undiscovered. Some of nation units needs to come to another nation to discover it. Undiscovered nations are not visible in diplomacy menu and their relations can't be changed unless nations are discovered.
	0. Peace. When this relation is set, two nations are in peace and neutral to each other. Each of nation units can go free through another nation territory without being attacked.
	1. War. When this relation is set, no other nation units can enter opponents territory. Units attacks each other everywhere, where it will be found nearby enemy units.
	2. Slavery. Nations are in peace, but enslaved nation has to pay half of it's collected resources to the master nation.
	3. Mastery. Nations are in peace. Master nation receives half of enslaved nation collected resources.

	4. Alliance. Relation marks peace and entrust between two nations. Enemies of one nation becomes also enemies of another.

These relations can be changed while playing through the following switches (player nation can attempt doing these actions at any time through diplomacy menu):

Undiscovered > Peace	Nation becomes undiscovered to another nation when one of its units are close enough to another nation to be discovered.
Peace > War	One nation proposes a war to another. Player can do it if it's relation with another nation is set as peace. AI nation comes to this decision when there are troops of more than 6 other nation units are closer to the nation than the radius of the town.
War > Slavery	Player nation can beg mercy for another nation at any time when there is a war. This sends a message and requires response from another nation. If other nation declines this proposal, the war continues. Usually AI nations accept begging mercy. AI nation triggers this proposal when a war becomes a threat to survive, i.e. nation becomes outnumbered and attacking nation does not stop attacking.
War > Mastery	This switch is triggered automatically when begging nation sends beg mercy proposal and the receiving nation accepts it.
Slavery > Peace	Triggered when enslaved nation decides not to pay anymore to the master nation. AI nations triggers this when they restore their military and economy to the level to be able to oppose slaver and claim back the freedom.
Mastery > Peace	Automatically set when enslaved nation leaves slavery.
Mastery > War	Player can trigger this switch any time on any of players enslaved nations. AI nations does not trigger this switch. The action is taken immediately, no response is needed.
War > Alliance	Player nation can ask to make an alliance with another nation while in war. It needs to receive positive answer from another nation in order to set the alliance. If other nation declines the proposal, both nations are remaining in a war condition. AI nations does not accept proposal if player is outnumbered; in that case player should beg mercy. AI nations usually sending this proposal to others if they lose large number of units and see that power of both nations are similar.
Alliance > War	Player can trigger it any time when in alliance with another nation. Works identical to Peace > War.
Alliance > Peace	Player can trigger it any time when in alliance with another nation. AI nations does not trigger this option.

Switches between relations are based on possible reasons, which can look natural to ask at a particular situation.

2.3.3. Setting up new relations

Very exotic option can be for developers to define their own new relations. The main point here is to decide how to interact between two nations, player and AIs. As relation is set as an integer values in relations list, it can be simply switched through Diplomacy.cs function **SetRelation()**.

Integer values below -1 and above 4 are not defined and can be used to define new relations. Important steps can be to define how does AI nation react, what is the meaning of the new relation (i.e. doing something with economies, exchanges, etc.) and extending UI for the player side.

2.4. Combat

Combat is one of the key elements in RTS to increase game playability. RTS Toolkit uses it's own defined from scratch combat system to allow units to enter the combat stage.

2.4.1. Combat phases

Combat can be divided roughly into the following phases: **search**, **approach**, **attack**, **death** and **sink** (**rot**). Search phase is controlled from **BattleAI.cs** while other 4 phases from **BattleSystem.cs**.

Search phase is responsible for finding opponent units to attack. It uses allUnits list from RTSMaster and takes only units from other nations, which are in a state of war. Nearest neighbour from this reduced set of units is being found for each searching unit. If found unit distance from searching unit is smaller than **searchDistance** set in searching unit UnitPars, then found unit is being added as a **targetUP** on a searching unit UnitPars. Attacking unit can have only one target, while target unit can get multiple attackers. If number of attackers is exceeding **maxAttackers**, new attackers will be not added to the corresponding target. If number of attackers becomes greater than **maxStrictAttackers**, the search is stopped immediatelly and restarts from scratch again with this target being excluded. Search phase is accelerated by kdtree for nearest neighbour searches. Searches are running around every 0.5 second.

Approach phase sets units to move towards their targets. When targets are found and assigned in search phase, attackers are set to approach phase. UnitPars uses **militaryMode** integer to specify at which phase unit currently is. If militaryMode == 10, then unit is searching for its targets, if it's 20, then approaching assigned target, and finally if 30, attacking target. As mentioned before, units are not moved from one list to another but instead are switching militaryMode values from one to another, which is fast for large number of units switching between different phases. Once unit is on approach phase, it's being added to UnitsMover.cs. The phase also checks if unit is not too close to its target and if it is, approaching unit is moved to attack phase. Sometimes unit's can't get closer to their targets - this can be cases when units are stuck in the crowd, or targets are running away. For these cases there is additional check, which allows unit to remain on approach phase - the distance between unit and its target should always decrease. If this distance does not decrease, **failedR** counter is increased. Finally, when failedR becomes greater than **critFailedR**, unit is removed from approach phase and brought back to search phase to search for other targets. Usually in such situations approachers loses their bad targets quickly and gets new different targets, which are easier to reach. When units are led manually by player, they are added to approach phase directly, without using search phase. These units also has **onManualControl** == true and they can't be moved to search phase, what means that unit is chasing its target until it reaches it or target dies.

Attack phase is the one, which performs attacking between units. Units enter attack phase after approach phase finds that unit is closer to its target then **stopDistIn**. At this point instead of moving any further, attacker starts the attack. When unit is attacking its target, health of target is being subtracted by the damage attacker possess. This damage is rolled randomly each time and depends on attacker **strength** and target **defence**. Attacker strength increases damage to the target, while target defence decreases it. As a result, the actual damage is in balance between these two. If target dies, attacker is being set back to search phase in order to find new targets. If target moves away further out then **stopDistOut**, attacker switches back to approach phase and continue approaching its target. Attack phase is also running attack animation, which itself is synchronized with the actual attack, i.e. the damage to the target is being applied not at the beginning of the attack, when attacker starts to swing the sword, but slightly later, when sword actually hits the target.

Death phase starts when unit health drops below 0. This phase is irreversible and once dead, unit can't go back to any other phases. As a result, unit is being unset from all its activities and added into deads list. When unit is dead, it's auto healing is also being stopped. Death phase also plays unit fall to the ground animation only once (animation is set as non-repeatable). **maxDeathCalls** allows to set how long unit is visible laying on the ground as dead.

Sink (**rot**) phase follows after death phase. It is the final phase, which destroys unit gameObject that it would not use any computing power in the game. Sink phase also makes body remains to slowly sink into the ground before disappearing. This phase can also be used with remains decay animation, if model has one.

The approach of these 5 phases allows to form dynamical and naturally looking battle front lines (see Fig. 16). As the battle progresses, units in weak areas are being defeated and the winning side units advances. This allows to change the shape of the front line and creates more localized clusters of intensive battle zones.

Fig. 16 - Dynamically forming battle front lines.

2.4.2. Melee combat

Swordsman, Knight and Horseman are units, which perform melee style attacks. When melee attack is performed, unit has to be close to its target. Melee attackers can be attacking inside **stopDistOut** from their targets. Units attacking with swords, playing attack animation and applying random damage to their targets.

Fig. 17 - The battle front line forming from swordsman units, which comes close to their targets to perform the attack.

2.4.3. Archery

Archers, Arsonists, Catapults and Ballistas are the units, which performs archer style attacks. These attacks are performed from a larger range distances when melee attacks. During the attack, unit launches arrow to the target. Arrows fly based on projectile laws and only initial launch conditions are used to calculate the projectile, i.e. shooting direction and the launch velocity of the arrow. Three cases can be considered when calculating the projectile:

1. Shooting on the flat terrain with the static target. This is the simplest approximation when arrow launched by the archer to the specific direction will guarantee to hit the target.

2. Shooting uphill or downhill with the static target. The direction of launched arrow accounts height differences between attacker and target.

3. Shooting uphill or downhill with moving target. On the top of height differences it is used target constant velocity relative to the archer to calculate the position where the arrow should land in order to hit the target. If target will move constantly along its path, arrow will hit them. As a result, to avoid damage from arrows, targets shoulds change their paths. This is happening naturally in intensive battles, as units moves quite randomly here and there, resulting that if unit is more lucky, it can avoid some damage from arrows.

Once launched, arrows are flying on their own and their trajectory is not modified during the flight. This is why archery in RTS Toolkit looks naturally. When arrow passes through the target, damage is being applied if the arrow pass closer to the pivot of the target than **rEnclosed** at its closest point. If the distance is larger than damage is not being applied. Finally, when arrow moves below the terrain, its gameObject is being destroyed.

Each unit uses slightly different type of arrows. Archers use simple arrows, arsonists - fire arrows, catapults - catapult balls and ballistas - ballista missiles. Simple arrows are just flying through the air and applying damage to its assigned target if they hit. Fire arrows has fire set on arrow head. When arrow hits, it applies damage, similar to regular arrow, but there is also possibility to set target on fire. This possibility is random and depends on **chanceOfFire** in **ArrowPars.cs**. Catapult balls works in a similar way like regular arrows but they has much larger initial velocity and carries greater damage to where they hit. Ballista missiles also creates fires, but instead of just a single fire source, they bursts into several burning fragments, which can hit nearby objects and cause secondary fires immediately. As a result, ballistas missiles are more like area attack weapon.

ArrowPars.cs is the script being attached onto each arrow instance and controls these arrows. Parameters of this component are shown in Fig. 18.

Fig. 18 - ArrowPars component, attached on each of the arrow.

buildingDamageModifier	Shows how much damage is used relative to damage of units (i.e. regular arrows can't hit building very well and their damage needs to be reduced, compared to damage cause to units).
fireArrow	If true, the arrow carries fire and could ignite the target.
chanceOfFire	The chance of fire to start when arrow hits the target.
arrowFragment	The gameObject, which is being used as a fragment when the main arrow crashes into the ground. Those fragments work as secondary arrow, moving around the center of crash. Fragment gameObject also has similar setup like arrow, i.e. it can be used as a fire arrow or it can have smaller fragments (third order) themselves. As a result, there is no limit how many orders of fragments can burst.
numberOfFragments	The number of fragments for arrow to break to. If arrow does not have any fragments, use 0. By default only ballista missiles uses fragments. It's also good to keep this number not very high, as when there are several dozens of arrows flying, they will burst into numberOfFragments times more fragments, which may affect FPS rates.
fragmentsFallRadius	The radius around the main crash point, where fragments can fly.
fireLight	The light component from child gameObject if present. Light here is as a point light and lights up the area. By default fireLight is set on fire arrow prefab.
lookToFlyDirection	Arrow is always facing the direction of where it's given velocity vector is pointing at any time.

searchForAllTargets	Search for any target along the projectile. If unselected, only the initial target, to which the arrow was shot, will be used.
arrowRemovalDelay	The time, after which the arrow will be destroyed.
attackAreaDamageRadius	The area within targets can be attacked.
failureErrorScale	The scale of failure to launch the arrow. Arrows are launched not exactly on the target but in the artificial area around the target. The higher is the number, the more arrows will miss the target. If the number is 0, all arrows will directly point to the target.
failureErrorLevelOffset	The offset of the unit level for the error. Ass units are advancing their levels, they are launching arrows more precisely. The final error for arrow's initial velocity is calculated in the following way: failureErrorScale / (failureErrorLevelOffset+attacker.levelValues[1]).

Fig. 19 shows how the battle front looks like when archers and arsonists are launching regular and fire arrows.

Fig. 19 - View of the battle front with regular and fire arrows being used. Fire arrows leaves nice looking smoke trails behind.

2.5. Economy

Updated in depth tutorials for v2017.2 available on YouTube:

https://www.youtube.com/watch?v=fzivpHT-ut4

Economy is used in RTS Toolkit to make units with different cost, to use economic power and to balance the game. Depending on situation, nations can try to achieve a balance between military and economic power. This balance is achieved when nation spends into military power to defend its territory and also collects resources to be able to spawn new military units and buildings. Each nation has its own independent economy. Player nation economy resources are visible at the top of the screen. By default in RTS Toolkit are four type of resources: iron, gold, lumber, and population to be used.

Economy.cs is a script, which definition of all resources (Fig. 20).

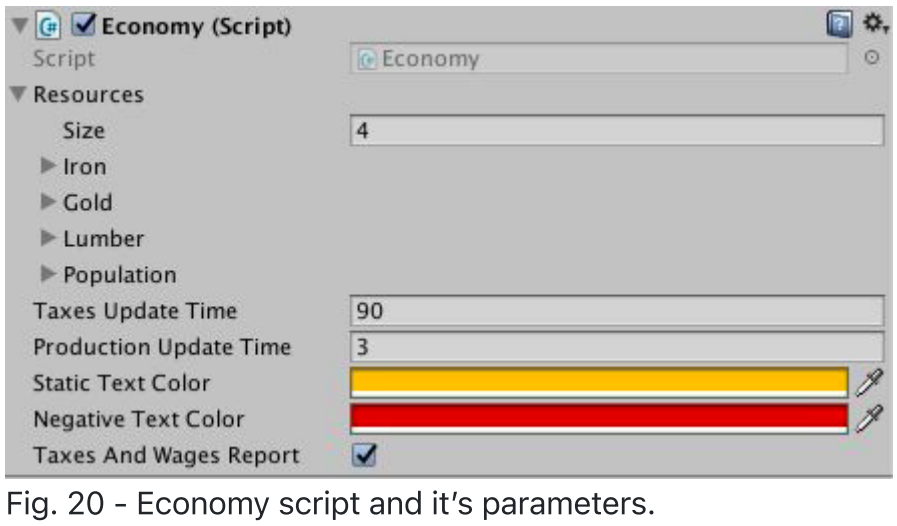


Fig. 20 - Economy script and it's parameters.

The Economy.cs starts with the list of Resources, where resources are actually defined. In the default scene there are these four resource types. The next parameter **taxesUpdateTime** allows to define the period of taxes and wages. **productionUpdateTime** defines how often production is being updated by producers (in the example is how frequently population is increasing if there are extra Houses). **staticTextColor** is the colour of which resources are shown for player in the top UI bar. **negativeTextColor** is applied only to resources which have consumers, and only when consumers exceeds the limit of consumption, when player starts to to lose other resources by paying wages rather than collecting taxes (in the default scene this applies to population, which becomes red once player uses more spawned units when it has in the population budget). **taxesAndWagesReport** allows to enable/disable the report message when taxes are collected and wages paid.

2.5.1. Ground resources

Ground resources in RTS Toolkit are resources which are being placed on the ground of the terrain. This includes iron and gold. The section of how ground resources are being defined in Economy.cs is shown in Fig. 21.

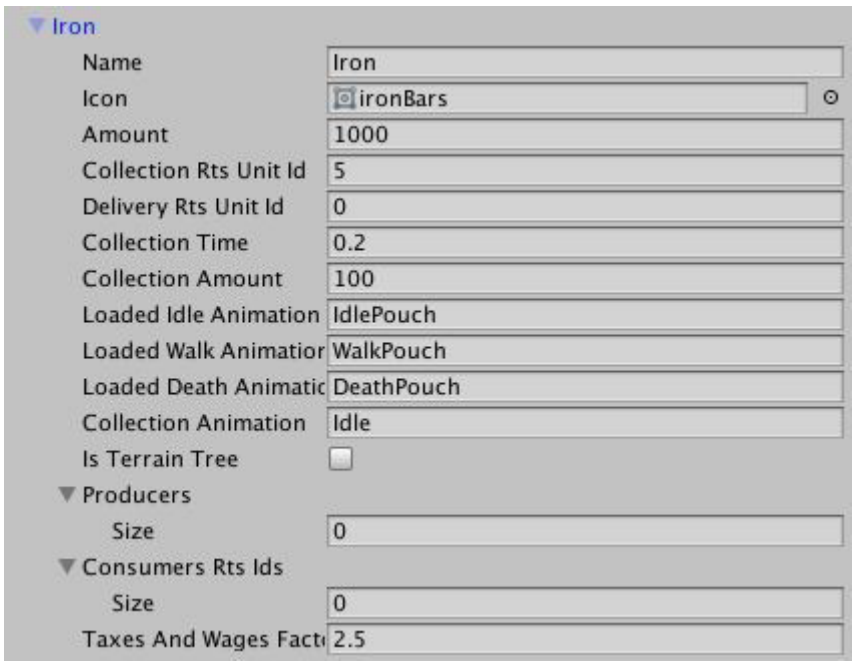


Fig. 21 - Ground resource definition and settings in Economy.cs

The first field here is resource **name**. The **icon** defines how resource should be displayed in UI elements. **amount** is used for the initial amount of resource each nation gets at the start of the game. **collectionRtsUnitId** references the index of unit, which is needed in order to collect the resource. For ground resources it's used to be the index of Mining Point building. If collectionRtsUnitId is specified as positive, nation has to build that unit first on the resource point before resource can be collected by workers. If collectionRtsUnitId = -1 then resource can be collected without creating a building. **deliveryRtsUnitId** is used in the same way as collectionRtsUnitId, to refer a unit, where resource should be delivered (which is Central Building for ground resources). **collectionTime** allows to define how long workers will be collecting the resource once they approach the resource point. For ground resources value is set to be low, which appears like worker is collecting resource immediately and starts to move towards delivery building. While resource is being collected through collectionTime, worker plays **collectionAnimation**. **collectionAmount** allows to define how much given resource worker can carry in a single go. Once the worker is set to collect particular resource, it "ping-pongs" between collection and delivery building. Each cycle worker takes out the resource from collection building until there is no resource left there. Once that happens, worker is assigned to find a new collection point and continues its work in collecting resources. When unit moves from collection to delivery point, it carry resource and "Loaded" animations are used to display how worker appears while carying this type of resource. **LoadedIdleAnimation** is used when worker stands and not moves while carrying the resource. **LoadedWalkAnimation** is shown when worker walks with the resource. **LoadedDeathAnimation** plays if by some reason unit dies and falls on the ground together with the resource it carries. By default when collecting ground resources (iron and gold) worker plays IdlePouch, WalkPouch and DeathPouch animations.

2.5.2. Terrain tree resources

The terrain tree resources are the ones, which can be collected from Unity terrain trees. By default in RTS Toolkit terrain tree resource is defined as Lumber (Fig. 22).

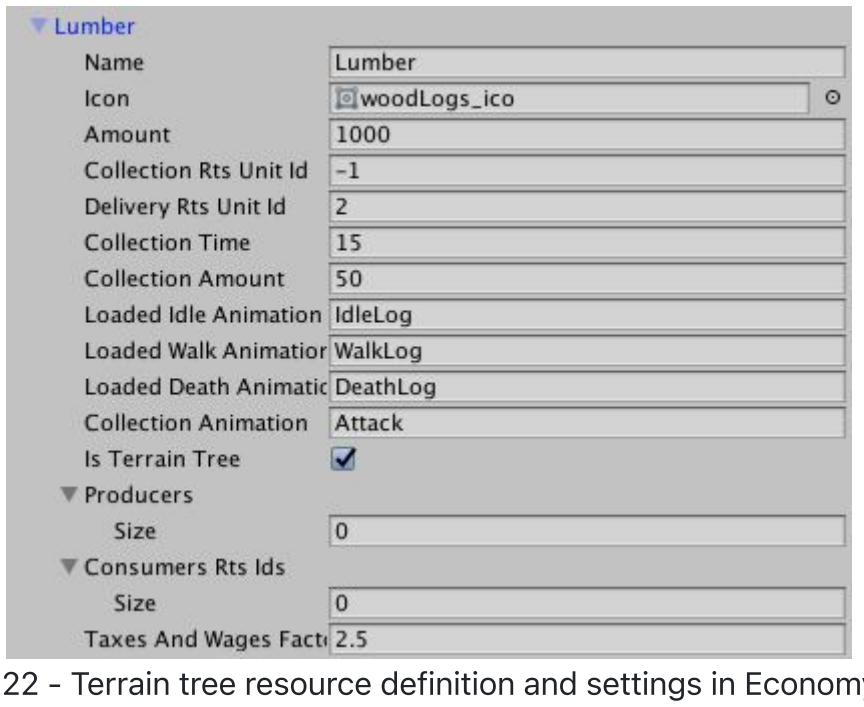


Fig. 22 - Terrain tree resource definition and settings in Economy.cs

Some parameters for terrain trees are identical to ground resources, however, there are differences as well. **isTerrainTree** needs to be set to true in order to use the resource from terrain trees. **collectionRtsUnitId** this time is set to -1, which means that there is no building needed in order to collect this resource. **deliveryRtsUnitId** is set to 2 for Lumber, which means Wood Cutter building. **collectionTime** is set to larger value, so that worker could play swinging its hatchet animation (Attack) while chopping lumber. Due to large number of trees around, **collectionAmount** is set to lower value (50), which balances Lumber collection workflow in economy. The “Loaded” animations are set to IdleLog, WalkLog and DeathLog, which appears as worker carries a log on its shoulder while bringing this resource. Once the tree resource is depleted, the tree is removed from terrain, and workers, which are assigned to that tree, are automatically re-assigned to chop neighbour tree.

2.5.3. Population

Population is very special type of resource (Fig. 23).

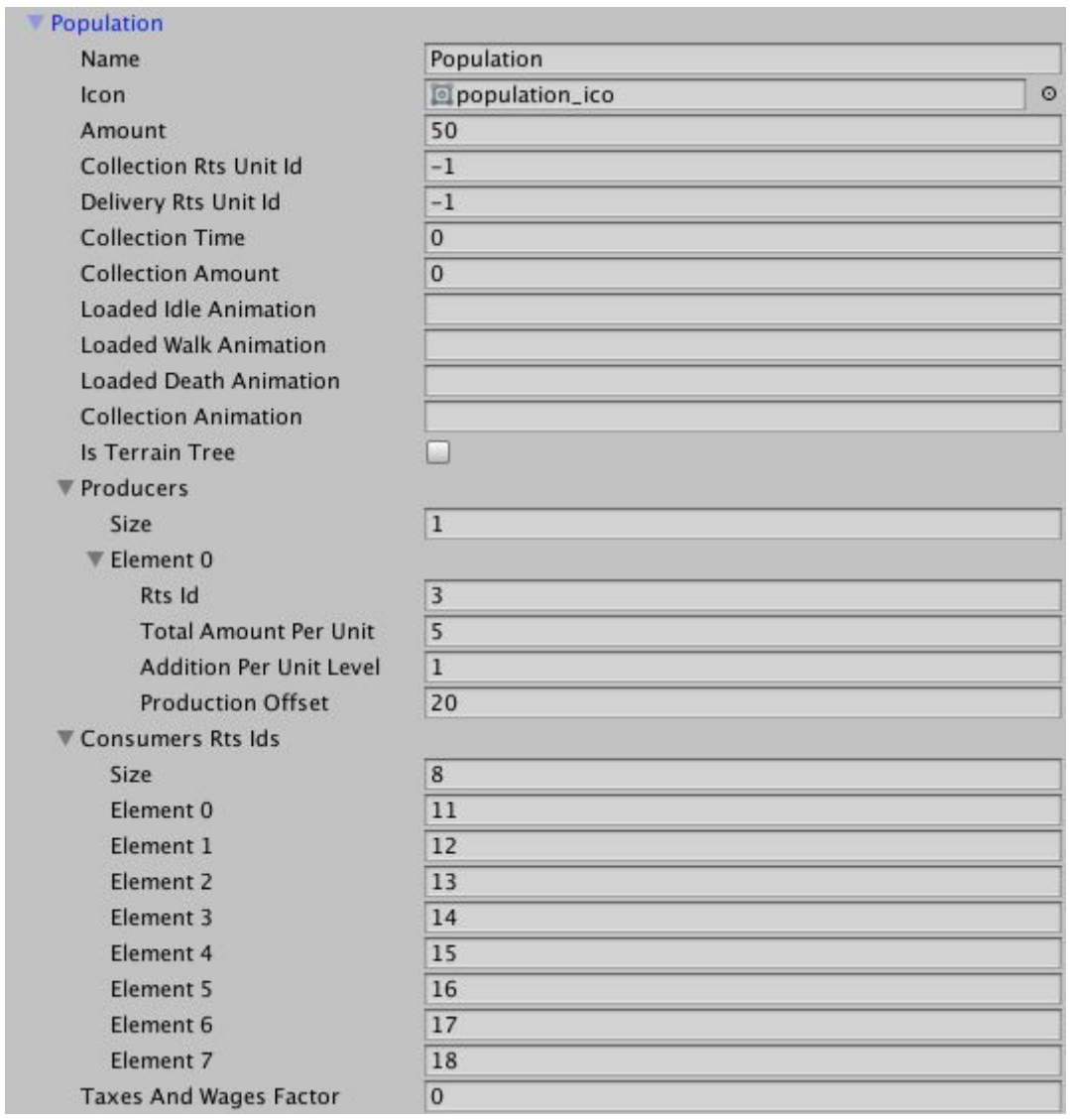


Fig. 23 - Population example in Economy.cs

Population is set not to be able to collect or deliver. The initial **amount** is set to 50 which is much smaller than any other resources. But the main difference is that population has filled **producers** and **consumersRtsIds** lists. producers list has one element with **rtsId** set to 3, which means a House building. Once a nation creates a new House, it eventually starts increasing population with a total amount set on **totalAmountPerUnit** (5). This gives that every House adds 5 population units to the nation budget. **additionPerUnitLevel** allows to set the additional amount of population added for each increased unit level. **productionOffset** defines the minimum population amount from which nation starts to gain its population. Let’s say nation has 20 population initially. Once it creates a House building, it will start increase its population every **productionUpdateTime** (3) by 1 until it reaches 25 population units (it would happen within 3*5 = 15 seconds).

consumersRtsIds allows to define units, to which wages are being paid. The list contains only integer ids, used for all units (not buildings). Once one of these units is spawned, nation starts to pay wage during every **taxesUpdateTime**. On the contrary, the current amount of population serves as a tax collector. As a result, the difference between consuming units and amount of population is used to find if nation gains resources or loses them. The amount of the difference is being multiplied by **taxesAndWagesFactor** (defined for each resource individually) and the result is being added or subtracted in all resources, which have taxesAdWagesFactor to be non-zero (population itself has taxesAdWagesFactor = 0 as it shouldn't be gained from itself). If there is more unused population when units, nation gets resource, otherwise it looses it. This brings a balance in economy system as well as in the battle system: if nation spawns many units, it will not have free population and starts losing its resources by paying wages to units. So nation has to always think how to balance between military power and the resources in order not to lose all resources and being unable to spawn anything.

2.5.4. Ground resource points

In order to be able to gather ground resources, they have to be placed and available on the terrain. If resources are defined in Economy.cs but not placed on the terrain, they will appear in game but nobody will be able to collect them as there will be no collection points around. For ground resources **ResourcePoint.cs** is used to place them on the terrain (Fig. 24).

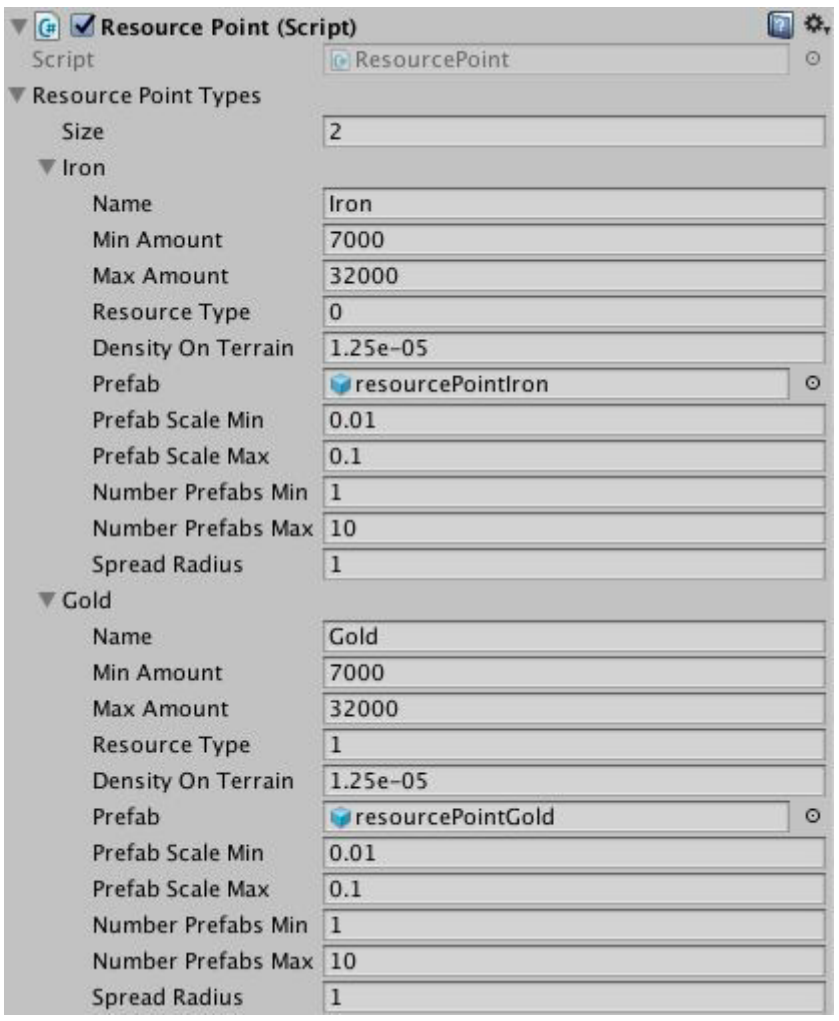


Fig. 24 - Ground resources placement settings in ResourcePoint.cs

Each of the resource type, which is being placed can be defined in **resourcePointTypes**. By default iron and gold resources are defined in ground resources. Each of type has different settings. There should be only one ResourcePoint.cs used in the scene. Parameters are explained in the table below.

name	The name of the resource
minAmount	Minimum amount of resources possible in the point.
maxAmount	Maximum amount of resources in the point.
resourceType	The reference to the resource type in Economy.cs
densityOnTerrain	The number of nodes to be spawned per square unit on the terrain.
prefab	Grain (droplet) model prefab to be used for nodes. Both, iron and gold prefabs are used as a low resolution icospheres with different materials and saved in Assets/RTSToolkit/Prefabs/Nature folder
prefabScaleMin	Minimum size of the grain in the node.
prefabScaleMax	Maximum size of the grain in the node. As seen in Fig. 24, grains in the cluster are slightly different size. They are being distributed randomly between min and max values specified here.
numberPrefabsMin	Minimum number of grains in the cluster. Corresponds to number of grains for minAmount.
numberPrefabsMax	Maximum number of grains in the cluster. Corresponds to number of grains for maxAmount. As nodes obtain random number of resource, they also make random number of grains visible in the node. In other words, more grains visible in the node, the richer node is in resources.

spreadRadius	The radius, within resource nodes are spawned.

Ground resources are being placed randomly on the terrain. The default iron and gold resources are set that they would appear as tiny grains in little clusters on the ground, with grey colour for iron and yellow for gold (Fig. 25). When resource point is being created, all grains are being dropped on the terrain and once in the correct positions, they are merged into the single mesh using CombineMesh() function for each cluster.

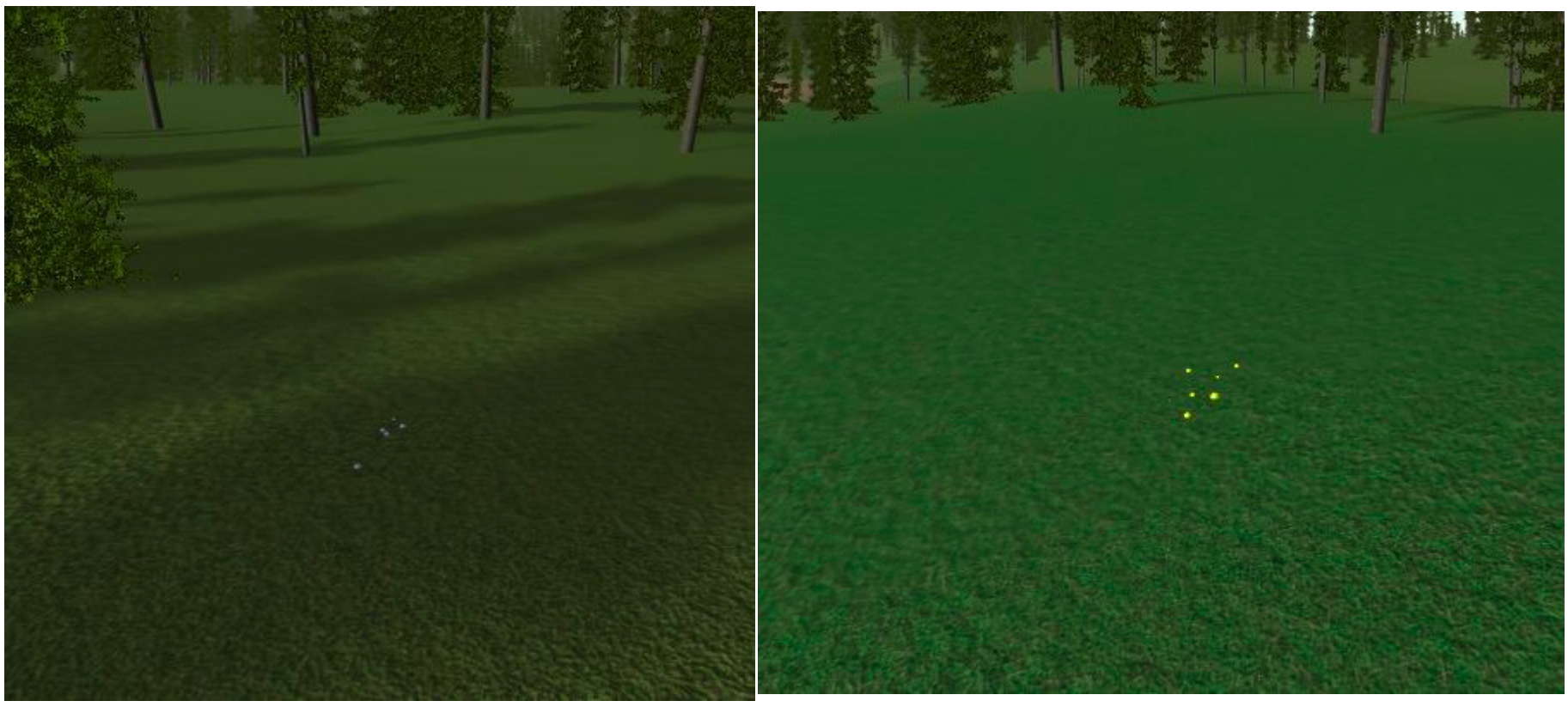


Fig. 25 - Locations where Mining Points can be build has tiny droplets visible on the ground. Iron appears are grey-blue (left), while gold as yellow (right) droplets.

Once player finds these places where resources are being placed, he can build the Mining Point there. It might be challenging to find some locations, so player needs to do some exploration to find where to collect these resources. Once location is found, nation need to build a Mining Point. Finally, when Mining Point is built, nation needs to send some workers, which brings these resources from Mining Point to Central Building. So it is always good to find resource locations and build Mining Points as close to the Central Building as possible, because resource collection rate directly depends on the distance between the Central Building and the Mining Point. There are limited amount of resources, which can be collected from the Mining Point. When resources are depleted, Mining Point starts to collapse and nation needs to find other spots where to collect resources. When player builds a Mining Point, workers can be sent to Mining Point by using  button or simply by right clicking on Mining Point with workers selected.

2.5.5. Terrain tree resource points

Terrain tree resource points are being placed while spawning trees via Forest.cs. For each tree type **resourceType** and **resourceAmount** in Forest.cs are the values which controls what resource type the tree represents and how much resource is available on each tree.

By default lumber can be collected by chopping down trees. Firstly nation needs to build a Wood Cutter, where chopped lumber can be delivered. Then workers are being send to chop some trees. Usually it takes several seconds to collect a log. Several logs can be obtained from a single tree before the tree finally falls down. Trees, similar like iron and gold are non-renewable resources, so once nation chops down forest, it needs to advance further or to move into a new location, where forests exist. Player can send workers to chop lumber by using  button or simply by right clicking on the tree when workers are selected.

2.5.6. Building and unit costs

Each of building and unit can have its own costs, which are set through UnitPars. There inside **costs** list can be added individual resource costs (these only needs to specify resource name and amount of the cost). Unit does not need to have all costs defined - if there is no defined a particular resource, when unit will cost nothing on that resource. Default prices for RTS Toolkit buildings and units are written in the table below.

Building/unit	Cost iron	Cost gold	Cost lumber	Cost population
Central Building	-	-	-	-
Barracks	50	50	80	2
Wood Cutter	30	30	50	2
House	-	20	30	-

Research Center	30	30	50	3
Mining Point	50	50	50	4
Factory	100	90	90	5
Stable	50	80	100	6
Windmill	100	100	200	10
Fence	-	20	30	-
Tower	30	80	180	4
Archer	5	5	10	1
Swordsman	10	5	-	1
Arsonist	5	20	10	1
Knight	40	20	-	1
Worker	5	5	-	1
Catapult	20	20	50	1
Ballista	20	20	50	1
Horseman	50	30	-	1

If nation resources are below the cost of the building/unit, then nation can’t create this building or unit and has to wait until enough resources are collected.

2.6. Multiplayer

Multiplayer mode in RTS Toolkit allows players not just to play the game themselves but also to play with their friends. Currently RTS Toolkit multiplayer is UNET based, where one player starts a host and other players are joining host as clients.

2.6.1. Main components

RTSMultiplayer.cs is representing player gameObject in the scene. The prefab named as MainPlayer, which is saved in Assets/RTSToolkit/Prefabs/Network contains this components. This prefab is registered onto **RTSNetworkManager.cs** script as PlayerPrefab. RTSNetworkManager.cs is overridden script for NetworkManager.cs and uses several sligtly modified overridden functions on connections and disconnections to handle units loading and unloading. When host or client starts multiplayer mode, MainPlayer gameObject is instantiated, which corresponds to that player. RTSMultiplayer has many Cmd and Rpc functions, used to communicate between server and clients.

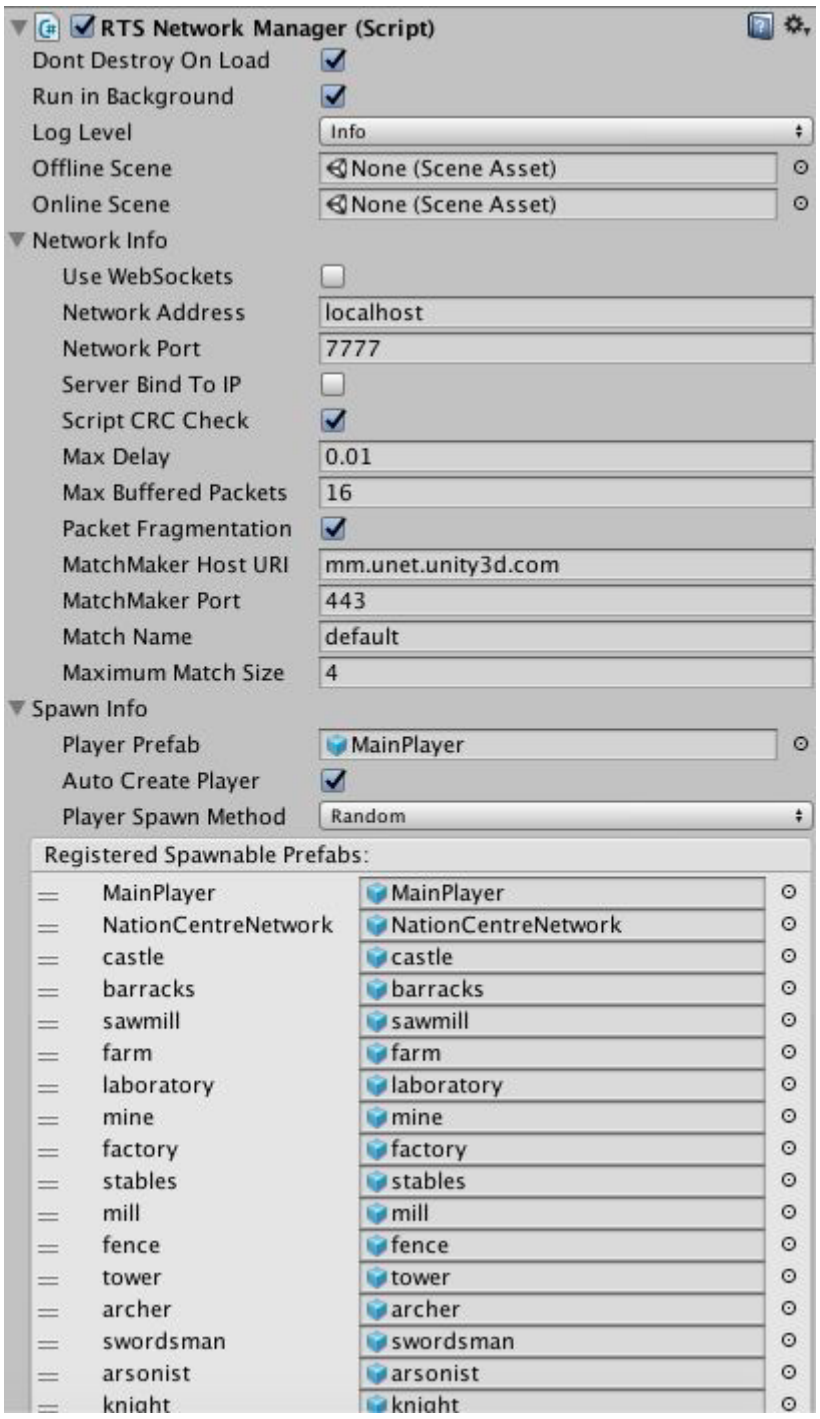


Fig. 26 - RTSNetworkManager component attached to the main RTSToolkit gameObject in the scene.

RTSNetworkManager is attached to RTSToolkit gameObject with the configuration shown in Fig. 26. There is also visible large list of Registered Spawnable Prefabs. In this list are registered all units and buildings prefabs used for multiplayer mode and saved in Assets/RTSToolkit/Prefabs/RtsUnitsNetwork directory. These prefabs are different from non-multiplayer mode used prefabs as they have attached **NetworkIdentity**, **NetworkTransform** and **NetworkUnique** components to them. This ensures that when unit or building is spawned, it is spawned as a network object and appears to all players, not just to one, which spawns it. Network versions of units are also registered in RTSMaster.cs list **rtsUnitTypePrefabsNetwork** and when player runs in multiplayer mode (isMultiplayer == true in RTSMaster), all other scripts, which access unit and building prefabs use multiplayer versions.

2.6.2. Synchronizing position and rotation

NetworkTransform.cs component attached on each unit instance ensures that unit position and rotation are synchronized automatically over the network.

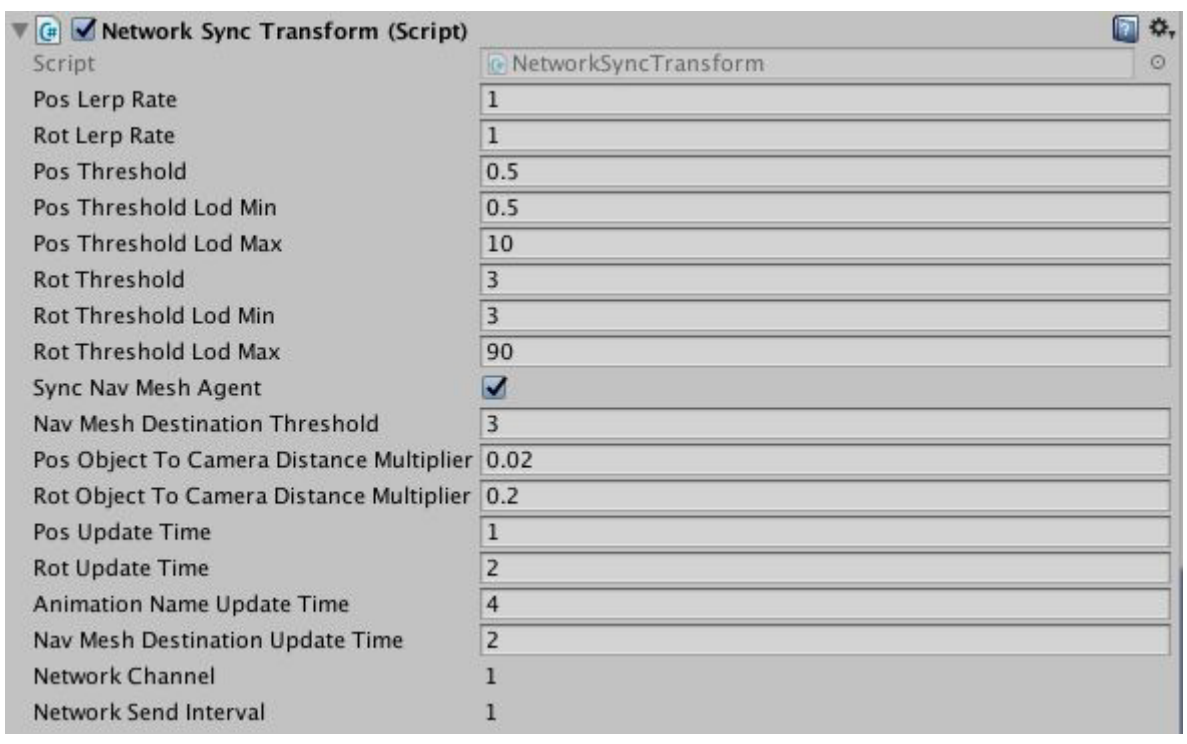


Fig. 27 - NetworkSyncTransform component attached on one of the unit prefabs.

NetworkSyncTransform parameters are set in a way, shown in Fig. 27. The script is syncronising position, rotation, navMeshAgent destination and unit animation. Position and rotation are synchronized only if navMeshAgent is not. Otherwise, only navMeshDestination is checked if changed and walking along paths is done in each client locally without using network traffic. Lerp

rates here are used to define how frequently position and rotation should be interpolated between synchronization steps. The bigger is the number, the faster it approaches final synchronized position or rotation. Lower values moves units slower but more smoothly. Thresholds are used to check when position and rotation needs to be updated. Threshold lod parameters allows to define custom thresholds of when to synchronize positions and rotations. If units are far away from any player cameras, then synchronization is done only is unit moves larger distances or rotates by larger angles. objectToCameraDistanceMultiplier's are coefficients for closest player camera distance. updateTime's allows to further control position and rotation synchronization frequency by the time. If **syncNavMeshAgent** enabled, then navMeshAgent destinations are synchronized instead of positions and rotation.

navMeshDestinationThreshold specifies of how far the new destination should be set in order to send it to the network. **navMeshDestinationUpdateTime** is the minimum time between the sequential updates for a given unit. **animationUpdateTime** allows to define how frequently animation name can be synchronized through the network when changed for a given unit.

2.6.3. Sending other data through network

Positions and rotations are not the only properties of units to be transferred through the network. Other properties, such as damage, animations, creating/removing units and nations, setting relations between nations should also be synchronized properly. Most of these properties are being sent through Cmd and Rpc functions in RTSMultiplayer.cs.

Health in RTS Toolkit is being changed through **UpdateHealth()** function in UnitPars. If the game runs in non-multiplayer mode (i.e. isMultiplayer == false in RTSMaster.cs) then new health is simply overwriting the old health. If multiplayer mode is on, then health would not be synchronized via the network and other devices would see no changes at all. This is where NetworkUnique.cs do a work. Firstly unit gameObject together with the new health is being sent to Cmd_UpdateUnitHealth() function in RTSMultiplayer. On the server it gets NetworkUnique component for the unit, which has its own **health** variable. This health variable is also set as SyncVar with hook function HealthChanged() being called every time health variable changes. Finally HealthChanged() access health variable on UnitPars and overwrites it with the new value.

Similarly also works Cmd_DestroyUnit() and Rpc_DestroyUnit to destroy unit on all devices, Cmd_SetRelation() and Rpc_SetRelation() to change diplomacy relations and several other ones. This way commands are only being sent to the network when calls are triggered and in most cases should be negligible compared to amount of data synchronized through NetworkSyncTransform.

In multiplayer mode are also used "cosmetic" units, which are not real units, but instead used just to display various objects. For example, each unit uses navigation and pathfinding only on the device where it belongs. On other devices when objects are instantiated, pathfinding components (i.e. NavMeshAgent and NavMeshObstacle when using Unity navigation) are disabled if synchronization for the navMeshAgent destination is not used. This ensures that transform is only being changed by original device, where pathfinding components are on.

When attacking and playing attack animation, damage is only applied from the original device to which attacking unit belongs and on other devices only showing animation. Arrows are also only real on a device where they belongs to - other players only see cosmetic arrows, as they do not carry any damage themselves (damage is carried by the one, which is shot on original device). Setup that corresponding clients control their own object and sends to server only what is needed allows instant control of units on all clients. Cosmetic units also can reduce significantly computing resources, as most of pathfinding and local avoidance calculations are distributed on clients: the setup is as follows - each player nation units are simulated on their own machines and AI nations and units are simulated on server (host). For example let's say there are playing 3 nations - one nation is host player nation with 500 units, another nation is AI nation with 500 units and the third nation is second player (client) nation with 500 units. As a result the first device (host) will have 1000 units to simulate (first player and AI) and the second device (client) will only simulate 500 units for second player. So in multiplayer there will be 1000 and 500 units simulated on two computers, while with a single player mode similar setup all 1500 units would be simulated on a single device.

2.6.4. Diplomacy dialogs

Diplomacy dialogs in RTS Toolkit multiplayer unveils it's true meaning by allowing players to have direct diplomatic dialogs, which can shape the fate of their nations. Dialog system between player and AI in multiplayer mode works exactly the same as described in 2.3.2 section. It is very similar when dialog is sent between two player nations. In this case the only difference is that a message, which requires other side decision is staying on that side until the side decides and sends back the answer. Let's say there are two player P1 and P2. P1 proposes a war to P2 and both player are in a war. After some time P1 decides to stop the war and has only 2 options: to propose alliance offer or to beg for mercy. Let's assume P1 decides to make alliance with P2 and sends alliance proposal. If P2 would be AI nation, it would automatically decide and send back the answer to P1 if alliance is created or not. However, in this case P2 receives P1 proposal and can answer to P1 if yes or no. Let's say P2 decides to accept the alliance. At this time when P2 accepts, alliance is automatically created and relation switched from war to alliance. P1 receives confirmation message that P2 accepted P1 proposal.

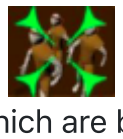
So diplomacy decisions here are taking place as a real decisions between players how to shape their matches.

2.7. Units grouping and formations

RTS Toolkit supports units grouping and creation of formations in game. Groups are used just to easier select a group of player's chosen units, while formations are representing more physical group, where units are also remaining close to each other.

2.7.1. Groups

Groups are handled by **UnitsGrouping.cs** component. There should be only one instance of UnitsGrouping.cs in the game. Managing unit groups includes group creation, destruction and merging multiple groups into one. Groups are defined through **UnitGroup** objects, which when created are added to the **unitsGroups** list in UnitsGrouping.cs. UnitGroup object is also assigned to each unit on their UnitPars variable **unitsGroup** and UnitsPars refrenece is added to **members** list in UnitGroup object. This cross-referencing allows easy accessing of various types of objects.

Player can add all selected units into a group by using  button. Once clicked, a new UnitGroup object is created and all selected units are added into its members list. If some units, which are being added on the new group has been previously in other groups, they are removed from them as unit can be only on one group. If by removal the old group becomes empty, it is being collapsed and removed from groups list. This grouping is done by **GroupSelected()** function.

Player can remove all its groups from the list. The function **CleanUpGroups()** is called, where **unitGroup** reference on each unit is being set as null and after all units from the group nullifies their references, the group itself is being removed from the **unitGroups** list. This destroys all groups player has created.

When player click on a group button, it selects all units within the group. Selection happens as **SelectGroup()** function is triggered, where all units from the group are found and non-selected units are being selected.

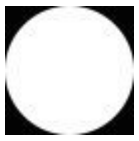
Groups can be very handy to manually put selected units onto them and later easily select all these units, which has been grouped.

2.7.2. Formations

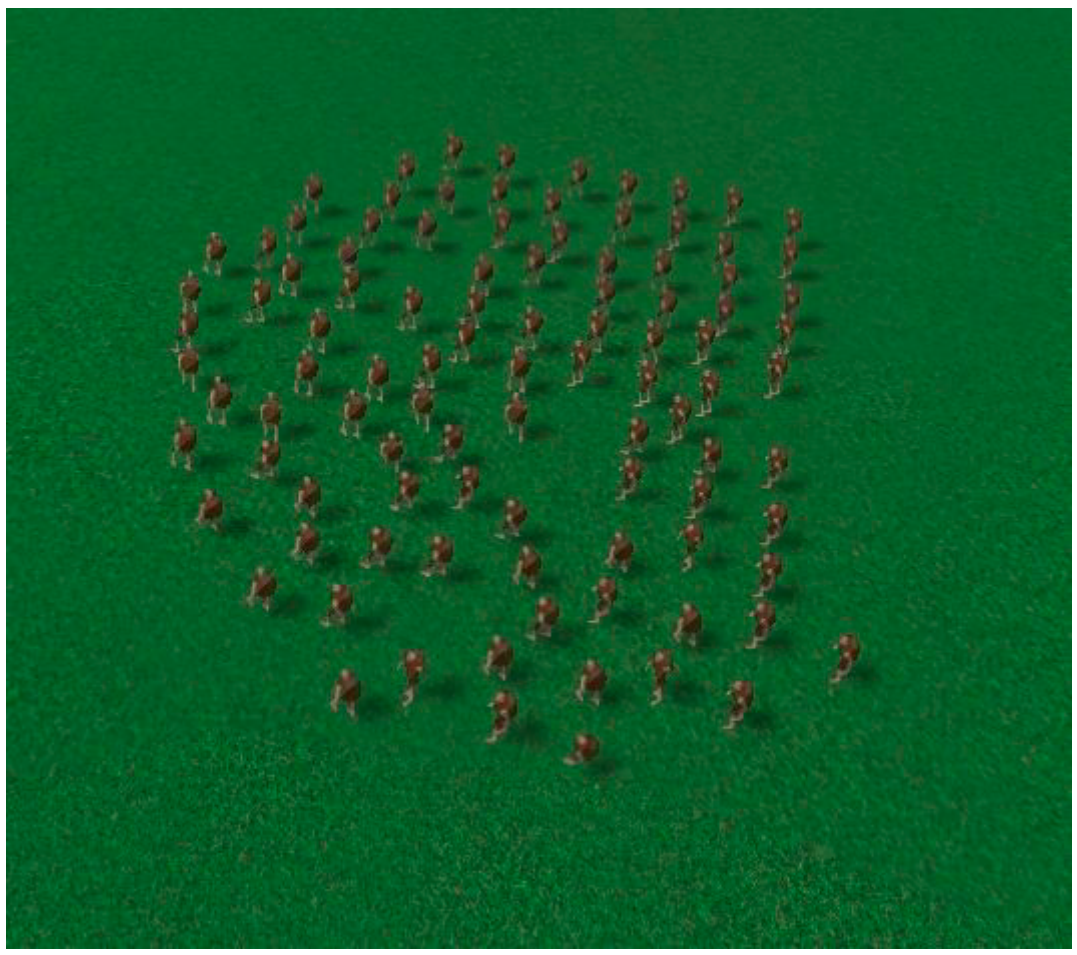
Formations are handled by **Formations.cs** component, which should be added to the scene not more than once. Formations are different from groups as they represents a group of units as a single unit. When attempting to select one unit from formation, all units in that formation are selected automatically. When selected formation is set to move, at the end of formation units form a shape of formation, defined by **formationMask** texture.

Texture for formationMask is a black and white image, where white areas are showing places where units can stand, while black areas gives avoidance zones. There are 5 different formationMask textures created and included in RTS Toolkit, which can be found in Assets/RTSToolkit/Textures/Formations directory.

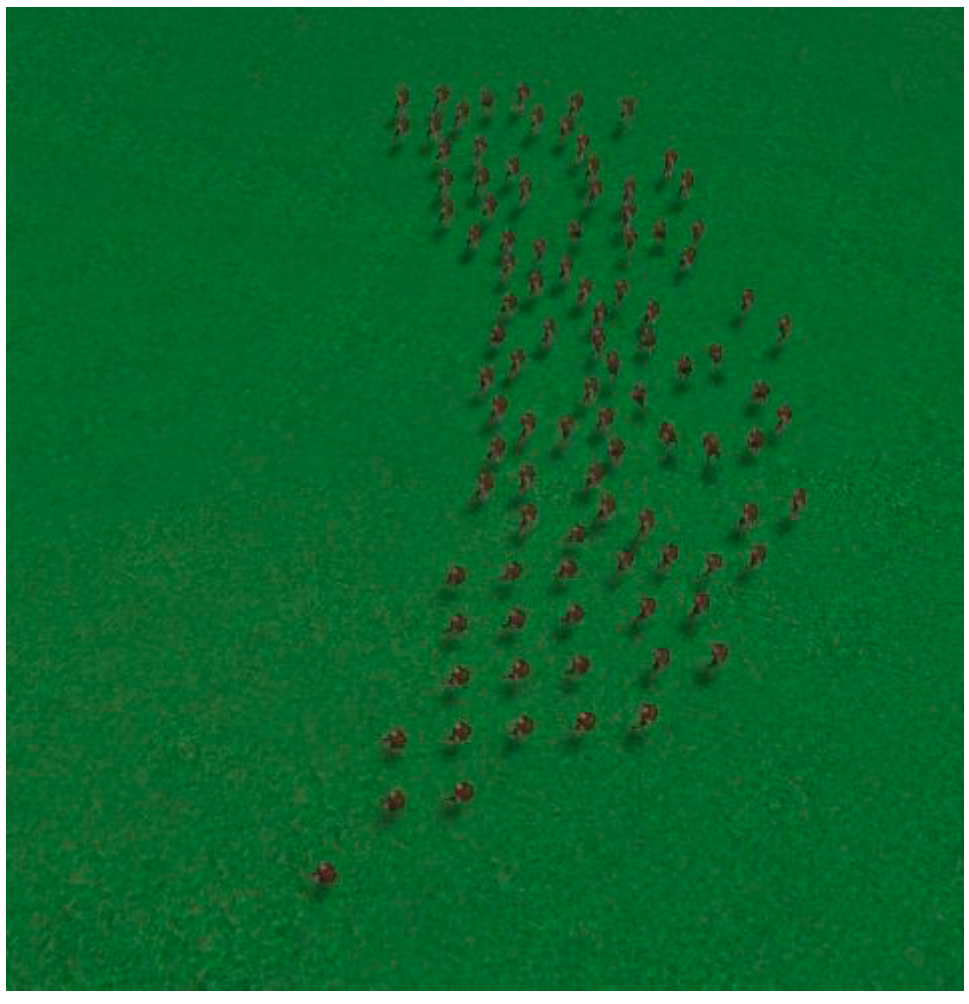
 **Bicircle**	

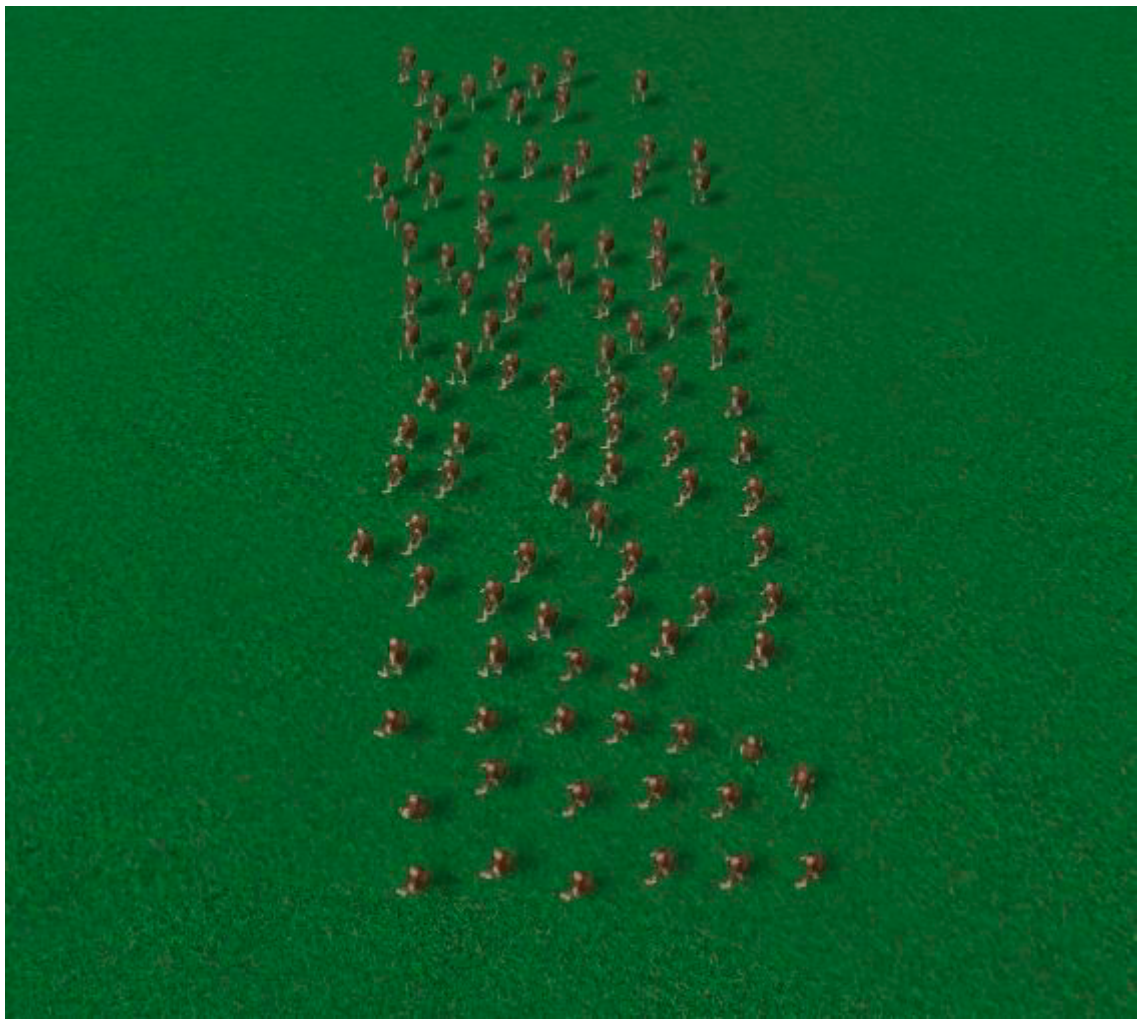
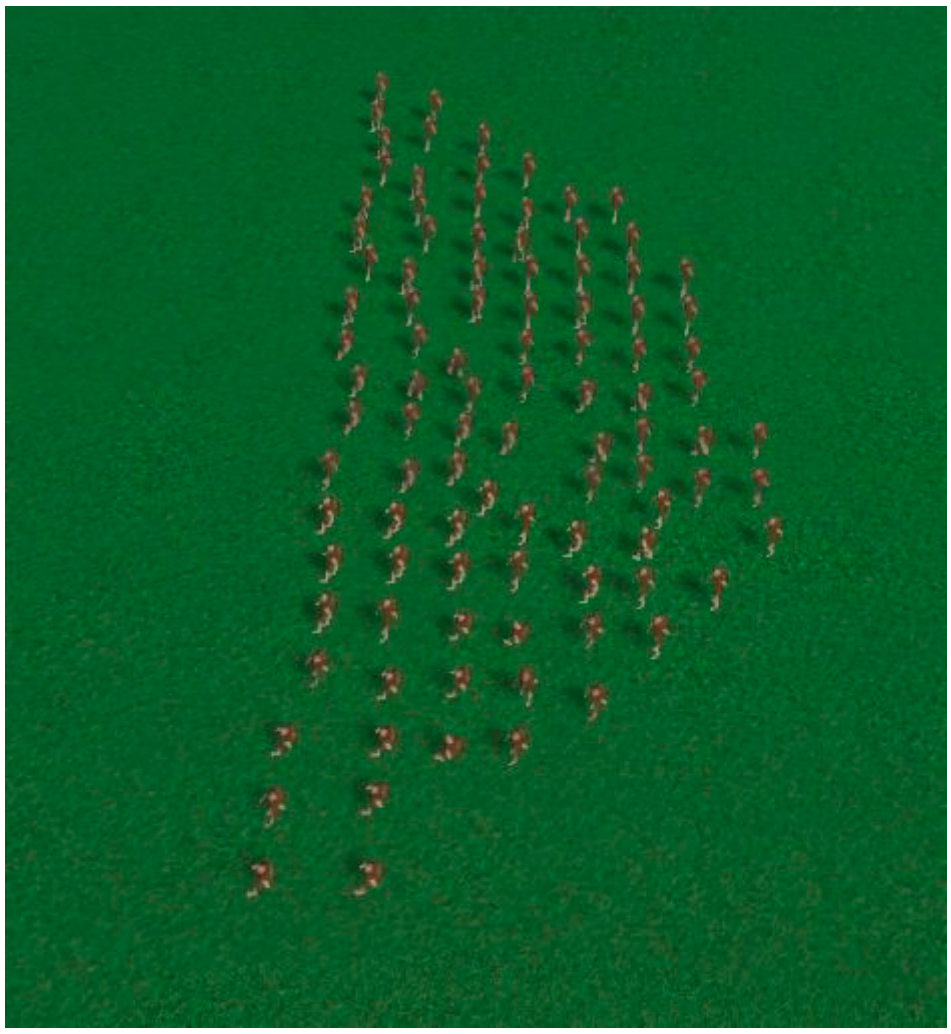


Circle



Crescent



Rectangle	
Triangle	

As it can be seen, units distribution at the final position in the formation replicates assigned formationMask texture. If texture is not assigned, formation will replicate circle version. Developers can create their own texture masks for formations (make sure to enable read/write in texture importer as pixels are being read with GetPixel() inside GetPointBrightness() function).

As it can be seen, units distribution at the final position in the formation replicates assigned formationMask texture. If texture is not assigned, formation will replicate circle version. Developers can create their own texture masks for formations (make sure to enable read/write in texture importer as pixels are being read with GetPixel() inside **GetPointBrightness()** function).

When formation is created and player clicks to move, the location where player clicked is used for the final center of the formation. Then current center of the formation is being calculated by calling **CurrentMassCentre()** function in the Formation class, which sums and averages all units positions to find the centroid. The direction of the formation is calculated from the difference vector between final and current centroid positions, i.e. formation will point its direction from current position to where it will move. Then individual units positions are calculated by putting units on the grid, which centre is in the final position. The formationTexture is applied when checking available grid cells. If number of grid cells is becoming lower than the number of units, grid is expanded by one row and one column until it is found that all units fit in the white area of the formation. This automatically scales the size of the formation based on number of units and amount of white area on the texture. Note, that you should be carefully when creating a texture - if you will submit a black texture without white areas, while loops will become infinite and the game will crash! So make sure you account this in your textures properly. It will also make formation bigger for larger number of units. Sometimes it may happen that formation is not fully filled just because by expanding the grid by one row and column gives to many slots, when there is not enough units to fill. Distances

between formation units in the grid are defined from the largest NavMeshAgent radius of all formation units (calculated by **GetMaxUnitSize()** in Formation class). When final positions are found, each unit is set on its own path with destination of these final positions and movement begins.

Formations can be created from any selected units (but not buildings, as buildings can be selected only by one). It's the same procedure as creating group, but player should enable the tick for "Formation mode" at the bottom of grouping menu. The group button will be also created, which are brown for formation mode (for non-formation mode groups buttons are grey). As a result the whole formation can be selected by clicking on the group button, by selecting one unit from the formation (will select all formation units) or by dragging rectangle selection on some or all of formation units. Formations can be destroyed by cleaning groups list.

2.8. Fires

RTS Toolkit has its own setup for fires as well. Units and buildings can be set on fire from attacks of fire arrows, used by arsonists, ballista missiles and wizard lightning. Fires usually start from a small spot but soon can engulf and destroy large buildings. In RTS Toolkit fire is based on particle systems. The default prefab for fire is saved in Assets/RTSToolkit/Prefabs/Nature folder as BuildingFlame. This prefab is also set as reference in RTSMaster for **buildingFirePrefab** in order to instantiate fire instances. BuildingFlame prefab has two child gameObjects - one is Smoke with particle system which produces dark fumes, and another is Light, which has a Light component attached and creates point source of light inside the fire. The main (parent) gameObject has particle system for flames and **FireScaler.cs** script attached (Fig. 28), which controls how fires behave.

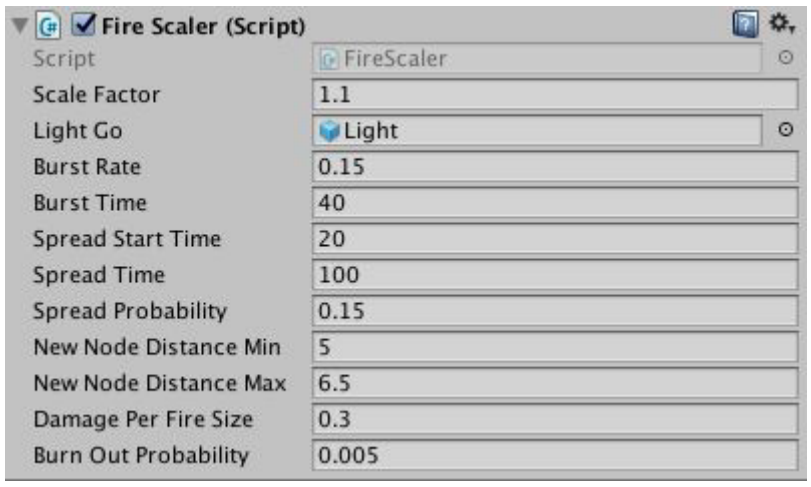


Fig. 28 - FireScaler.cs component on BuildingFlame prefab with its default values.

FireScaler has the following parameters set:

scaleFactor	Initial scale factor for the fire.
lightGo	Reference to children light gameObject.
burstRate	The flame eventually grows in size (bursts), burst rate shows how fast it grows, i.e. in Figure 15 flame grows 15% per second.
burstTime	The total time in seconds for the flame to burst. After burst time passes, flame does not grow in size anymore.
spreadStartTime	Each flame can start making other flames around itself after spreadStartTime.
spreadTime	The time during which new fire nodes can appear around the old fire.
spreadProbability	The probability to create a new fire source if random fire distributor finds the right place to start.
newNodeDistanceMin	The minimum distance at which new fire node can appear from the old one.
newNodeDistanceMax	The maximum distance at which new fire node can appear from the old one.
damagePerFireSize	The amount of damage (in health units) per current size of fire, per second. I.e. if damagePerFireSize = 0.3 and currentSize = 10*original size, the actual damage to unit or building will be 3 health points per second.
burnOutProbability	The probability for fire to burn out (extinguish).

Fires are starting from the single fire node, which itself slowly increases over time. At spreadStartTime fire is big enough to give a birth of other fire nodes, which will go on themselves and can give further spreads. This makes fires as a chain reaction on buildings. When spread is starting, MeshCollider is created on the building to find where are building walls, roof or floor. Two random point fare generated within the range of newNodeDistanceMin and newNodeDistanceMax from the original point. The ray is casted from one point to another which is used to check if it crosses the surface of MeshCollider. Crossing point is additionally checked if the distance to any other existing fire nodes is larger than newNodeDistanceMin (without original node from which spread is initiating, there can be other nodes nearby, which has to be checked; checks are accelerated by nearest neighbour search in kdtree). If so, on the top of that spreadProbability is applied. If probability check passes, the new fire node is created. Parameters in Fig. 28 gives a good balance

between rates, such that fires would not spread very fast or too slow, that burnout would not exceed spreading trends if building is not being extinguished and that individual blazes would not become unrealistically large. Fires are spreading only on buildings, while on units there are only appearing a single fire node - at the origin point of unit.

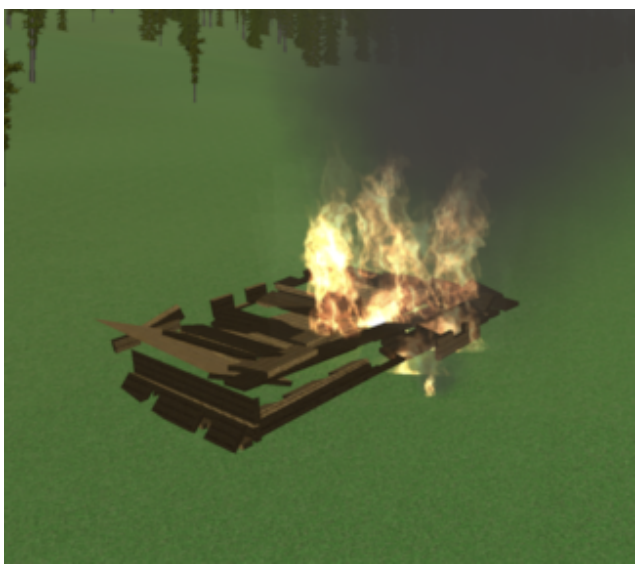
a)

b)

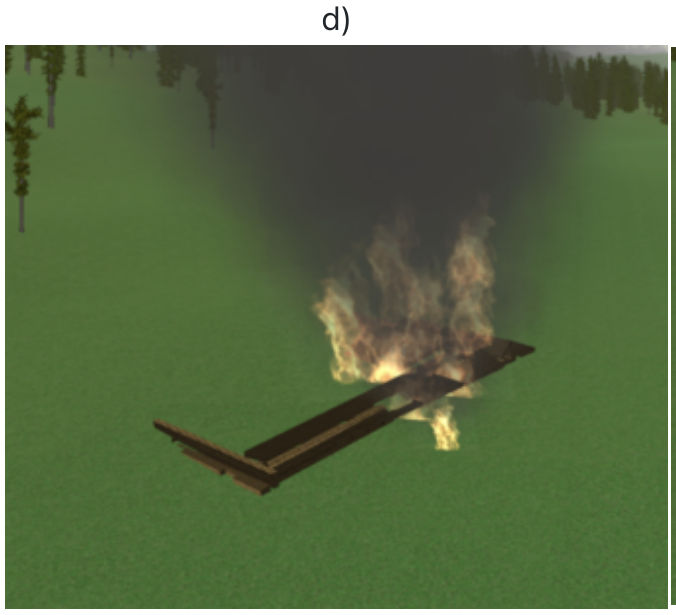
c)

d)

e)

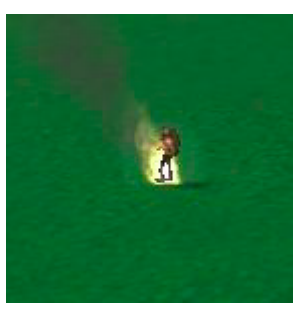


f)



g)

h)



i)

Fig. 29 - Progress of fire on buildings and unit.

Fig. 29 shows the progress of fire on a building (a - h panels) and the fire set on unit. Single fire node with a small flame starts burning on the Central Building wall as arsonist shot fire arrow which ignites it (a). The flame becomes larger and creates the second flame (b). Third fire are coming out soon and fires becomes larger in size (c). Several more fire sources coming out rapidly engulfing entire Central Building (d). Central Building starting to collapse (e) and (f). Finally when there are no health left, fires are burning out (g). At this point building material colour is changed to black and when fumes clears out, there are only some building remains left visible (h), which sinks into the ground (sink phase) and disappears. The burnout cycle (a-h) happens rapidly just in a few minutes, so player needs to keep an eye if its nation buildings are not on fire (click repair on damaged buildings). When unit is set on fire, only single fire node appears. Units has much less health than buildings and dies quickly if fire is not getting out. When ballista missiles crashes, it can ignite multiple units and buildings. In a war multiple town buildings are usually set on fires, when significant amount of town can be lost (Fig. 30).

Fig. 30 - Multiple buildings burning in the fire at the same time during the war.

3. Environment

Environment used in RTS Toolkit is where the entire game is embedded. All parts of environment needs to be consistent with each other and in balance between quality and computing speed. Environment is based on the terrain with meadows and forests, the sky has changing bright clouds, following the wind direction, deep terrain pits are filled with water to make lakes, landscape is made from soft hills, where in one or another place cliffs surfaces, some wild animals are walking through fields and forests, flying birds just over the trees and meadows, various bird sounds, and everything continues endlessly to any direction player goes (there are no physical boundaries or map sizes - the environment is free to explore, just like in the real world). This brings well balanced and robust environment system, where developers can start using their own models, textures and tune up some parameters to find out which artistic pace fits the best to their games.

3.1. Terrain

Terrain is the base of the RTS Toolkit environment. RTS Toolkit creator thanks

http://code-phi.com/infinite-terrain-generation-in-unity-3d/

tutorial, which classes has been adopted in setting up proceduraly generated terrain.

So the terrain is based on Unity Terrain system. Firstly Perlin noise is used to generate heightmap in float[,] array. This heightmap is used to set terrain heights with SetHeights() function. As a result, this creates standard perlin terrain. The procedure is repeated for several tiles around the camera. Once camera in game moves, new terrain tiles are being generated, while others unloaded. This allows to move camera to any direction infinitely with new landscapes. This creates illusion that world is endless without any bounds. Terrain is generated in both - editor and play modes and generation of perlin maps is also accelerated by multithreading in both modes.

GenerateTerrain.cs is the script, which manages terrain generation. It's default parameters are shown in Fig. 31.

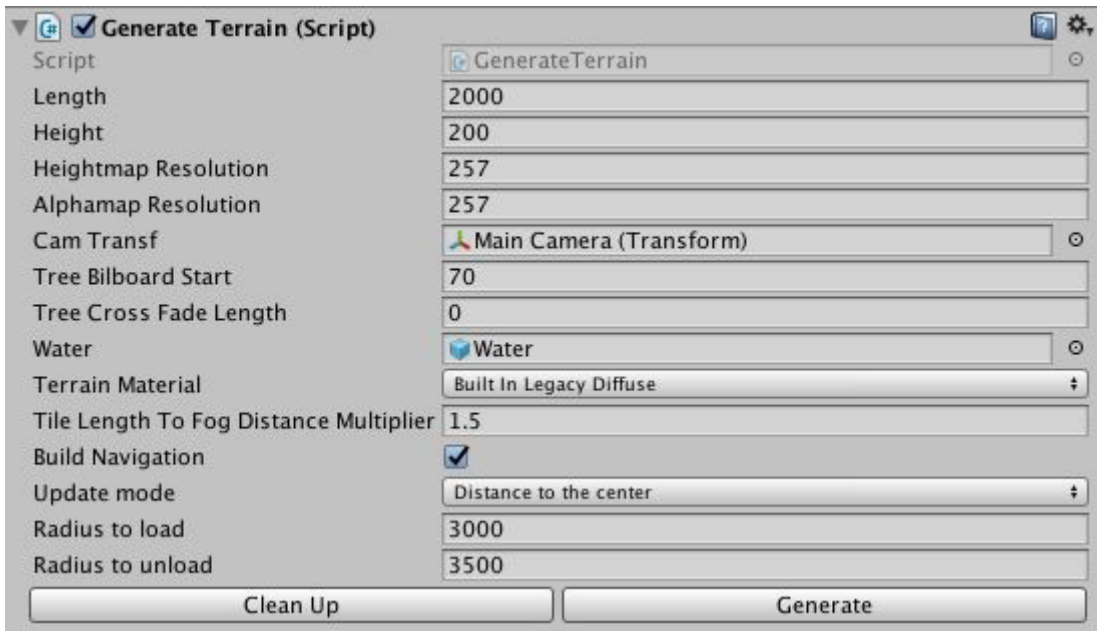


Fig. 31 - GenerateTerrain parameters for terrain generation.

length	The length in world units of the terrain edge. 2000 gives 2 km size terrain tiles.
height	The maximum height terrain could rise to.
heightMapResolution	Resolution used for the heightmap.
alphaMapResolution	Resolution used for alpha map.

camTransf	Camera transform reference, which should be followed to create terrains around.
treeBilboardStart	The distance from camera at which trees starts to be rendered as billboards.
treeCrossFadeLength	Terrain option set on all generated unity terrains. It specifies at what distance from the camera tree meshes starts to lean towards billboards for a smooth transition.
water	Water gameObject from the scene to be used for making lakes.
terrainMaterial	The material to be used for generated terrains (use “Build In Standard”, “Build In Legacy Diffuse” or “Build In Legacy Specular”).
tileLengthToFogDistanceMultiplier	Used to calculate the linear fog distance. In the example if the length is 2000 and tileLengthToFogDistanceMultiplier is 1.5 then the linear fog in lighting settings will be set to start at 0 and end at 3000.
buildNavigation	Builds unity navigation, which is set on all terrains and rebuild on runtime automatically. By default this option is enabled. If unset, then navigation is not built (can be useful for quickly generating and exploring empty worlds or bake large scale maps).
updateMode	how terrains should be generated. “Do not update” will keep set currently existing terrain tiles, “Distance to the center” will trigger to spawn new terrains within.
radiusToLoad	Distance from the camera and unloads terrains within radiusToUnload from the camera. “Number of tiles” simply generates the number of terrains around the camera (2 gives one central terrain below the camera and 8 terrain tiles around it, making 9 terrain tiles in total; 3 gives 5x5 = 25 tiles in total) and terrain tiles are loaded and unloaded at the same time. Scan allows to use external terrain generators to find already generated terrains and set the RTS Toolkit to work on them (tested with MapMagic).
“Clean Up“ button	Used to clean previously set terrains, clean lists and reset GenerateTerrain.
“Generate” button	Set up the terrain system and generates new terrain tiles.

Multiple gameObjects with GenerateTerrain.cs component can be present in the scene but only one of them has to be enabled and others disabled. The enabled one will be used for terrain generation while disabled ones will do nothing. This setup allows to have many terrain presets at the same time and quickly switch between different options.

Terrain size and resolution can be also changed in game by player via the graphics settings.

3.1.1. Heightmap

Terrain heightmap is used to generate the pattern of the terrain. Here are how TerrainHeightmap component looks like.

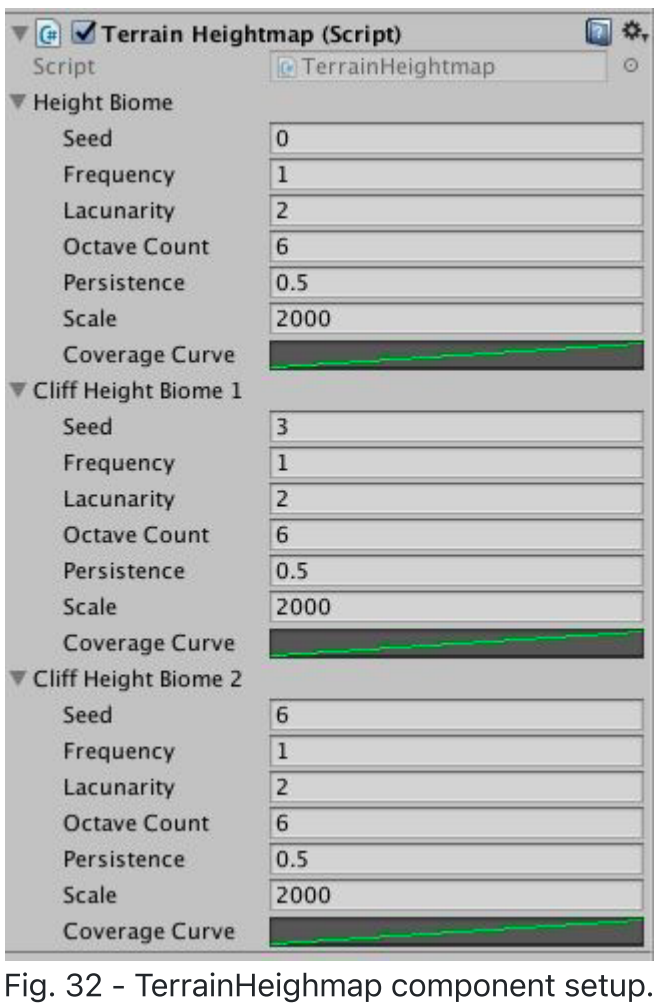


Fig. 32 - TerrainHeighmap component setup.

There are three biomes, which are defining the final heightmap:

heightBiome	The biome, which settings are used to calculate the main heightmap.
cliffHeightBiome1	The biome, which defines at which elevations to create cliffs.
cliffHeightBiome2	The biome which defines how large cliffs can be.

Each of the biome has the following parameters:

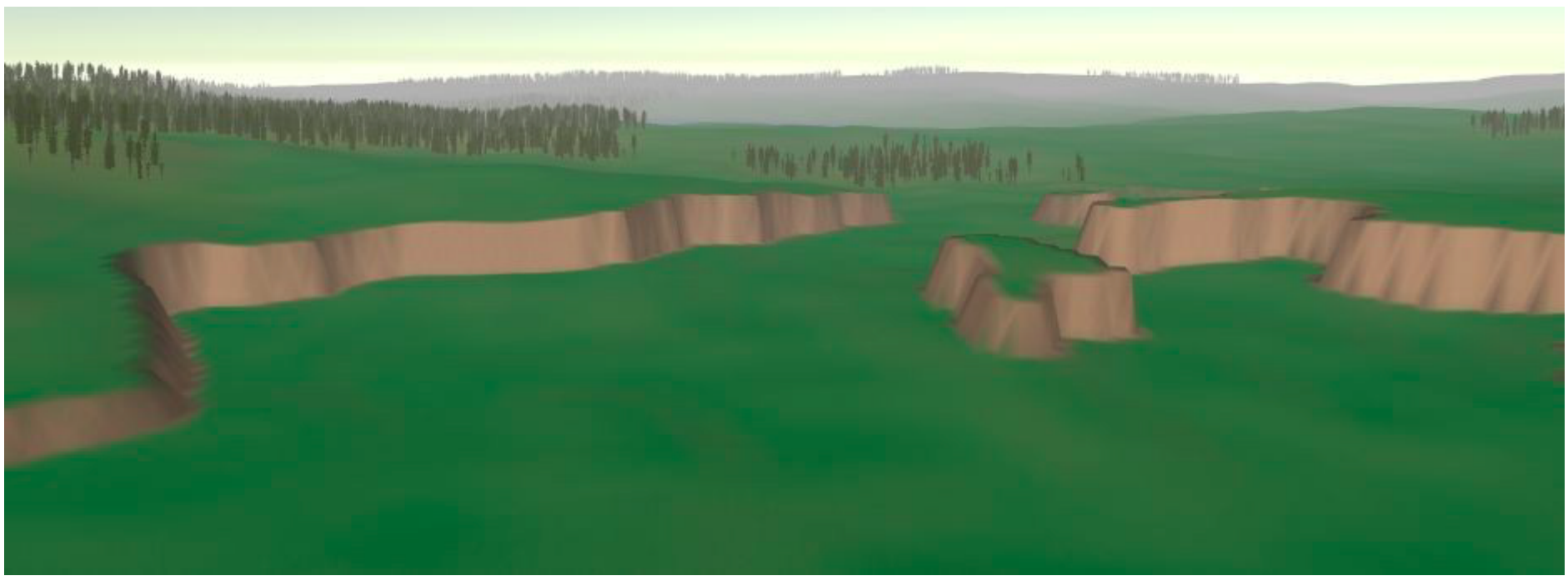
seed	The seed, used for perlin noise.
frequency	Perlin noise frequency.
lacunarity	Perlin noise lacunarity.
octaveCount	The number of octaves used in perlin noise
persistance	Perlin noise persistance.
scale	Perlin noise positions are multiplied by this value.
coverageCurve	Defines mapping of perlin values.

3.1.2. Cliffs

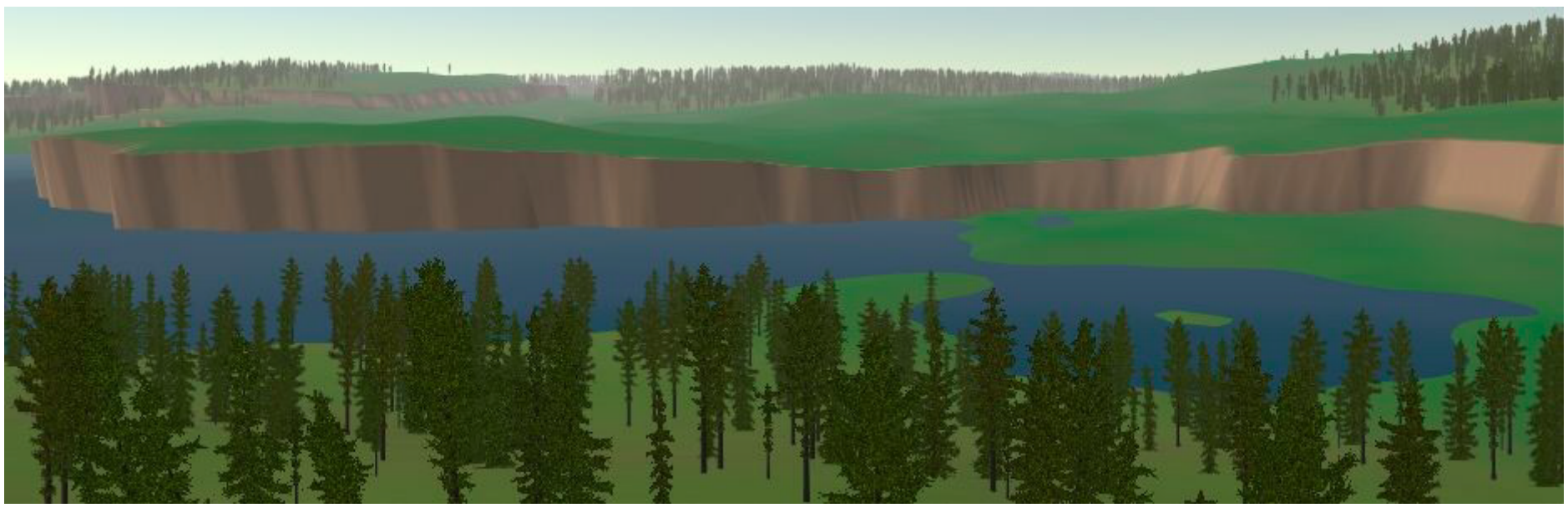
Cliffs are used on the terrain to add more variety to the landscape. For making cliffs it was used two additional perlin noise maps together with the main heightmap. That makes terrain generation to run 3x slower (each new noise adds additional cost). But as all 3 noises are running in multiple threads, calculations are still running fast enough. All 3 perlin maps has different seeds in order to bring randomness between noises. “Seeds” here are being used as the 3rd coordinate on 3d Perlin noise, so their values can be not necessarily integer numbers.

The combined height from all 3 noises is calculated in GetHeightJumpy() function in TerrainChunk class. The idea of calculations is that if we have second noise with it’s values between 0 and 1, there will be isolines, where values will be 0.5. In one side of isoline the value will be more than 0.5, while in another side it will be less than 0.5. Now we can use this to modify the main heightmap (from the first noise) - if the value of second noise is below 0.5, we reduce the height and if it’s above 0.5 - we increase it. This makes a sharp drop in the terrain heights where isolines passes, making cliffs in these locations. However, these cliffs would be everywhere exactly the same. This is where we need third noise to tell by how much terrain should be lifted. The third noise is also subtracted with 0.5, i.e. its values are between -0.5 and 5. We can use positive values for the height difference of how much second noise should be lifting the main heightmap. Where are negative values, we can use them as not to generate cliffs at all there (lifting would be 0, meaning no modification of the heightmap). This produces landscapes with various sizes cliff zones and cliff-free zones as well. Cliff width is also dependent on terrain heightmap resolution - higher resolution terrains allows to have more dramatic falloff, more detailed cliff edges and defines the minimum height of the cliffs.

a)



b)



c)

Fig. 33 - Several cliff examples generated on the terrain: a) short extending cliff with blending edges, b) cliffs pillars appearing in random locations, c) cliffy shores of the lake.

Fig. 33 shows just a few variations of how cliffs can look like in various locations. Cliffs can be blending in one side and another with flat terrain (a), there can be forming small pillars, where second noise suddenly jumps a bit above 0.5 (b), or it can be making steep lake shores. As the terrain is endless, player can find unique procedurally generated landscapes with these cliffs.

3.1.3. Rocks

Rocks (boulders, stones, etc.) can be spawned on RTS Toolkit terrain. RockPlacer.cs is the component, which distributes rocks on terrains (Fig. 34). The component has a list of rock species, which can be placed on the terrain. **prefab** specifies the prefab, which is used to spawn number clusters of rocks on each terrain tile, which is calculated from **density** (i.e. number = density*size.x*size.z). **steepnessMin** and **steepnessMax** allows to restrict spawn locations based on terrain steepness. Individual rocks are being spawned in circular clusters, which sizes and number of rocks can be random. The number of rocks inside a cluster is between **nClusterMin** and

nClusterMax. Cluster sizes in distance units are between **clusterSizeMin** and **clusterSizeMax**. Sizes of individual rocks are specified by power law s = **sizeMultiplier** * (Random ^ **sizeExponent**) + **sizeShift**. This allows to spawn more natural distribution of rocks, where smaller stones are more common than larger ones.

RockPlacer also perform a simple sliding simulation, where rocks are being moved towards “water flow” directions in their spawning positions. There are random number of iterations (between **slideIterationsMin** and **slideIterationsMax**) being performed for each individual rock. **slideVelocityMin** and **slideVelocityMax** specifies random velocity boundaries during each iteration. If velocity is positive, rocks are being pulled downhills, while negative velocities move rocks uphill. Finally **useThis** allows to easily deselect prefabs, which we don’t want to spawn. **useDrawMesh** allows to use Graphics.DrawMesh() function from the script instead of using MeshRenderer on each instance. As there are no LODs on these instances, **drawMeshViewAngle** is used to set that if object is viewed from camera with smaller angle, that it would not be displayed.

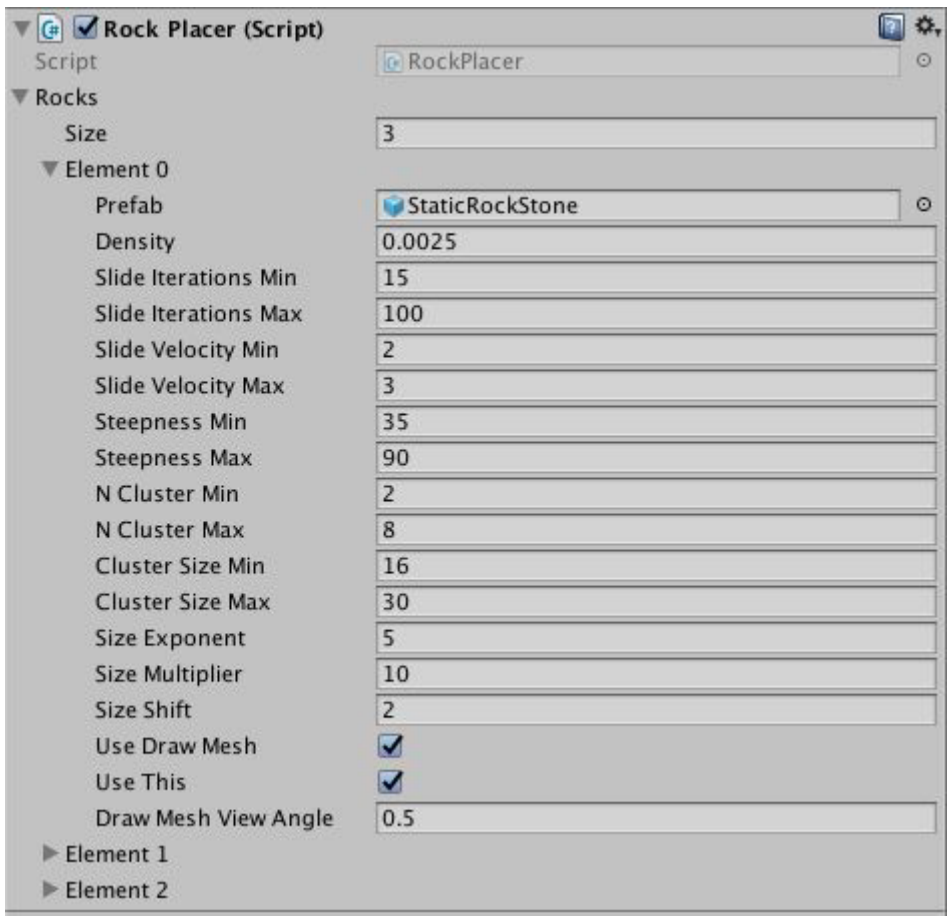


Fig. 34 - RockPlacer component and its setup.

Rocks placed with high steepness and some downhill simulations allows to get naturally looking rubble stones in the bottom side of the cliff (Fig. 35).

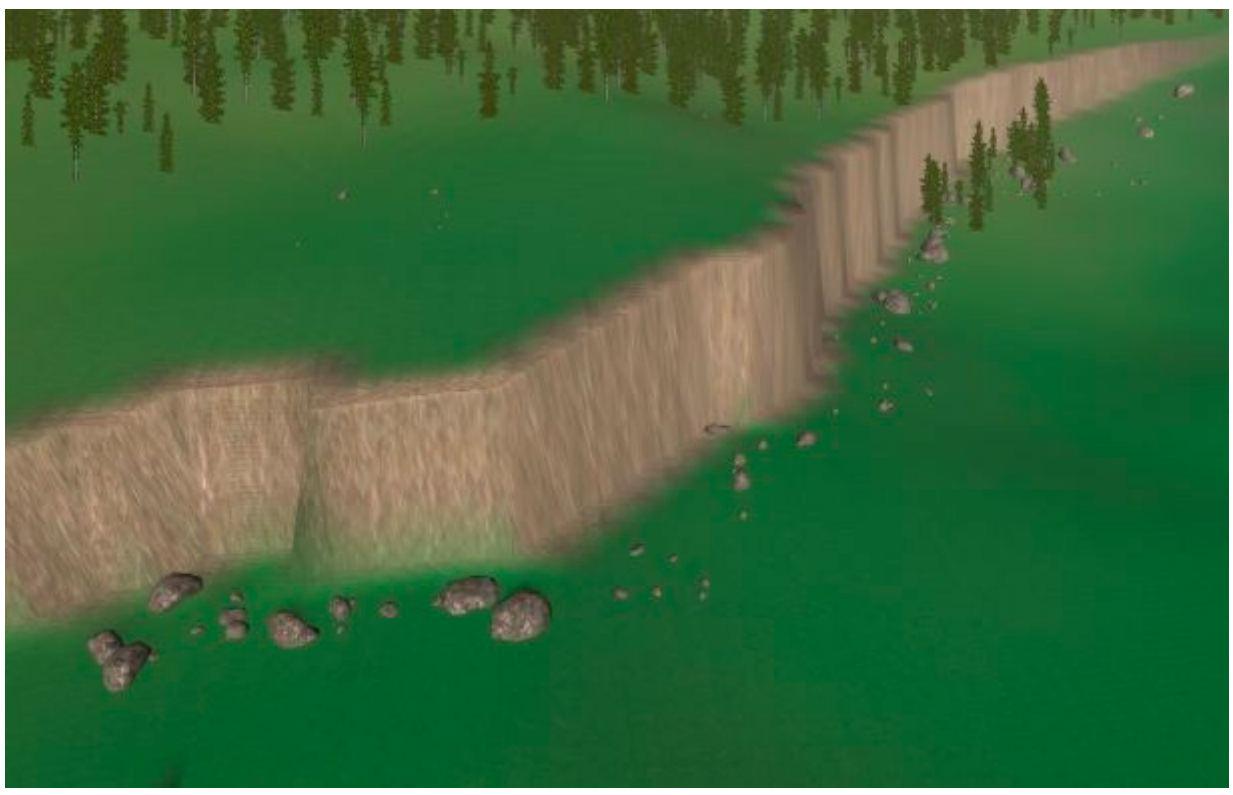


Fig. 35 - Rubble rocks placed at the bottom of cliffs.

3.1.3.1. Falling rocks

RTS Toolkit includes exotic feature - falling rocks from cliffs. Simulation uses two types of prefabs - **FallingRockSpawner** and **FallingRockStone**, which both can be found in Assets/RTSToolkit/Prefabs/Nature directory (Fig. 36).

FallingRockSpawner is empty gameObject, which is placed on locations where cliffs can be found and is responsible for spawning falling rocks via FallingRockSpawner.cs component. FallingRockSpawner instances are being spawned by RockPlacer.cs component while running 0 slide iterations on steep terrain locations. This places FallingRockSpawner instances directly on cliff sides.

rockToSpawn in FallingRockSpawner.cs is a prefab used to spawn rocks. Rock instances are being spawned with their random sizes between **sizeRangeMin** and **sizeRangeMax** within random time intervals between **spawnTimeMin** and **spawnTimeMax**. shapeVariation allows to get more size variations in different axis, e.g. it allows to create sometimes flat and long shape stones. When rock is spawned, it gets random velocity, defined by **velocityVariation**, what allows to move rocks to slightly random directions. FallingRockSpawner.cs checks if there are some units nearby and falling rocks starts to fall when somebody gets their units closer to cliffs than **spawnTriggerDistance**.

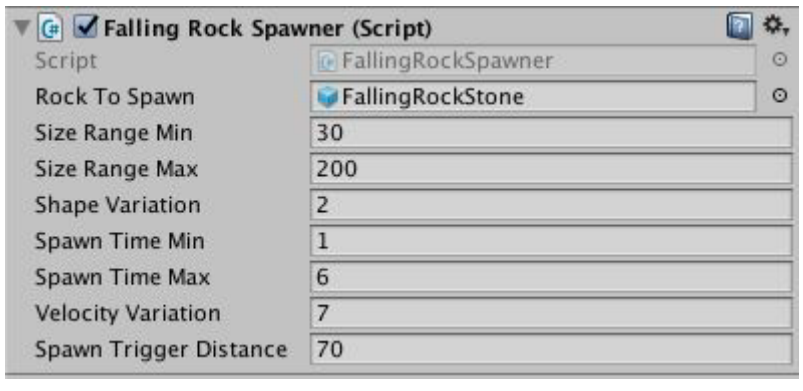


Fig. 36 - FallingRockSpawner setup.

When rock is created, it starts to fall, as fall as rock gameObject has Rigidbody component with gravity attached. Box Collider is used to detect when falling rocks are hitting the terrain. In that case rocks are bouncing back from the terrain before they stop moving. The FallingRockStone gameObject has arrowPars script attached, which checks for the damage on units, walking on the path where stones are falling.

3.1.4. Textures and splatmaps

Terrain textures adds more realism to the surface of the ground. They are managed by **TerrainTextures.cs** component (Fig. 37). Textures can be adjusted by using 3 types of biomes: height, slope and perlin. All biomes are managed by adjusting through animation curves. The higher value on animation curve means larger weight for corresponding biome. For example **slopeCurve** sets arguments between 0 and 1, which corresponds to slope angles between 0 and 90 degrees. Similar rules applies for **heightCurve**, where argument limits between 0 and 1 corresponds to minimum and maximum terrain height. For **perlinBiomes** it is used **coverageCurve** for this, where curve argument means perlin noise value. In addition, perlinBiomes allows to use perlin **seed**, **frequency**, **lacunarity**, **octaveCount** and **persistence** to customise perlin noise behaviour. **useBiome** allows to set which perlin biomes in the list can be used.

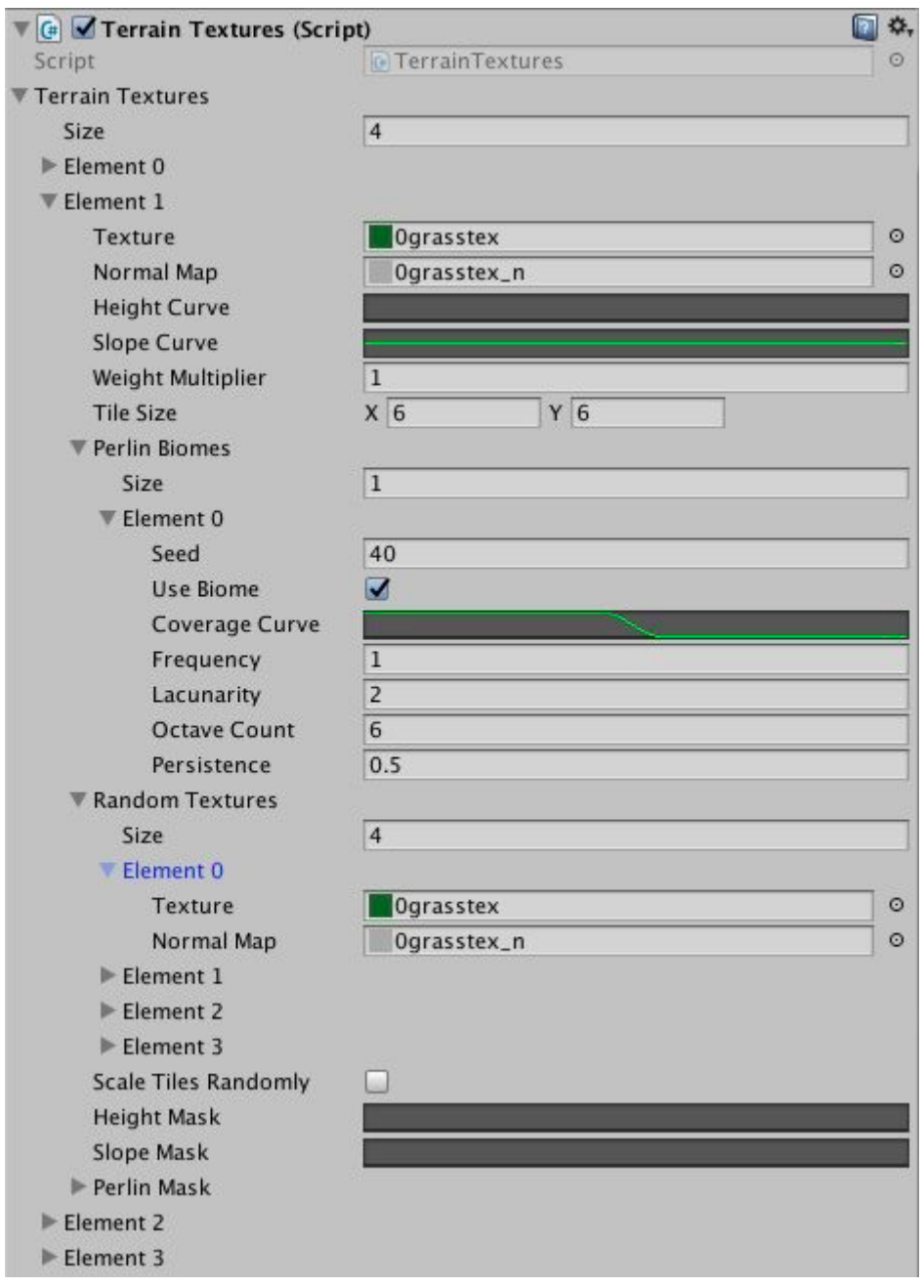
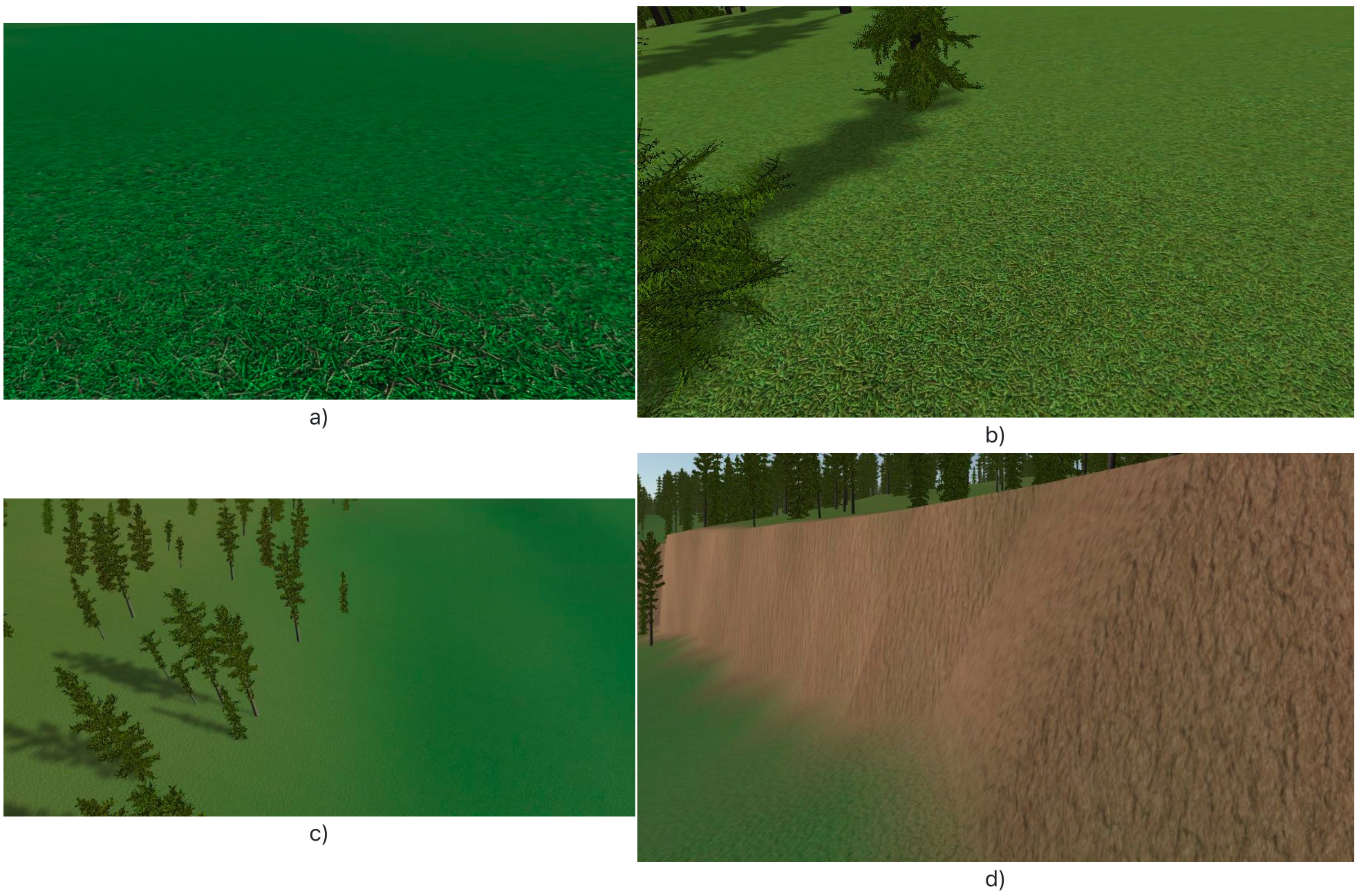


Fig. 37 - TerrainTextures component with its settings.

terrainTextures elements has main **texture** and **normalMap** fields where are set main textures for each element. weightMultiplier can be used to give higher weight for one texture than another. And texture **tileSize** can be adjusted as well. However, it would fill entire biome with a single texture, meaning that tiling artifacts could be visible. But TerrainTextures allows to solve tiling problem by mixing

multiple **randomTextures** and no need of any external terrain shader. On a given splatmap pixel is randomly picked up one of the textures from **randomTextures** list and it's weight is being set as maximum at this pixel, while other textures from this list gets 0 weight. As on each pixel is randomly chosen different textures, this appears as a mixture within a biome and reduces tiling artifacts. By default this is applied for grass and moss textures in the package. Another way is to vary tile sizes on each of random textures: this can be done by ticking **scaleTilesRandomly**. In this case all random textures are getting different tile sizes. By default scaleTilesRandomly is applied to cliff textures.

heightMask, **slopeMask** and **perlinMask** allows to define masks, where corresponding textures should be avoided to appear. As an example sand texture is used with height biome slightly above water level to create sandy beaches. But perlinMask with forest seed is used to avoid sand to appear near water in areas which are overgrown with trees.



a) b) c) d)

Fig. 38 - Textured zones on the terrain.

Fig. 38 shows differently textured zones on the terrain. The areas, where are no forests has grass texture (a), forest areas are textured with moss (b), the border between forest and meadow can be seen from distance as a change of colour between yellow-green and dark-green respectively (c), cliffs texture is painted on steep areas of the terrain (d).

Grass and moss textures has been generated by randomly spawning individual weed or moss leaves on a flat surface and taking screenshots of it, while cliff texture by drawing "random walk" tracks on the texture. There are made 4 variations of each texture for grass and moss, which are saved in Assets/RTSToolkit/Textures directory within grassGroundTextures and mossGroundTextures subdirectories respectively. One cliff texture is used from Assets/RTSToolkit/Textures/Resources/textureGen/cliff directory. All textures has their corresponding normal maps.

3.2. Trees and forest

Forests in RTS Toolkit are managed by **Forest** component (Fig. 39) and spawned procedurally.

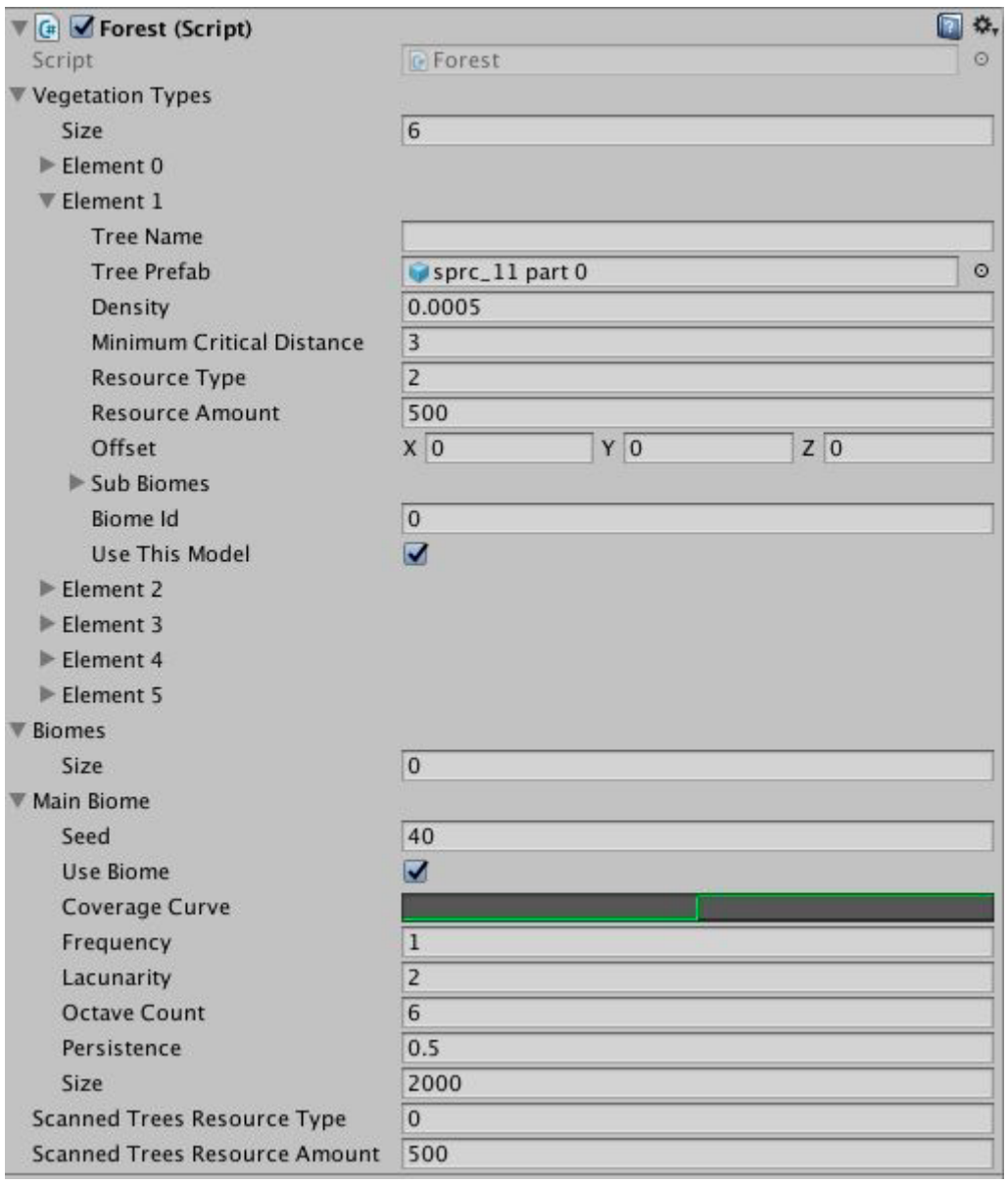


Fig. 39 - Forest component, where all trees are being set.

Each tree can be defined in **vegetationTypes** list. **mainBiome** is a perlin biome which defines the boundaries of overall forest. However, each trees can use **biomes** list to subdivide forests by tree types (i.e. broadleafs, spruces, etc.). On each vegetation type **biomeId** can be used to specify to which perlin biome the tree belongs to.

Each vegetationType contains **treePrefab**, which is the prefab used to spawn treen on terrains (trees generated with TreeGen, which contains multiple parts can be set by using **treeName** instead - note that treeName is not used in generator if treePrefab is not null and can be used just for labeling). **density** specifies how dense will the specie be placed on the terrain (density is defined as number of objects per square unit). **minimumCriticalDistance** controls removal of tree neighbours spawned closer than specified value. **resourceType** is used to refer which resource in Economy.cs this tree type corresponds to, and is used when collecting from the tree. **resourceAmount** is the total amount of the resource, which tree contains. **offset** allows to adjust the tree position if it is not well centered. Finally useThisModel allows not to spawn tree types without removing them from vegetationTypes list.

ForestPlacer class is being used to create tree instances and store them. ForestPlacer object is created for each terrain tile separately and has only trees, which belongs to that tile. Each tree has **TerrainTree** object created as well, where it contains **treeHealth** variable, which is used for resource collection when trees are being chopped by workers. Once treeHealth drops below 0, the tree is removed from the terrain. When player moves with camera between tiles, some tiles are unloaded together with trees. If player comes back, tiles are loaded again and as trees are spawned always from the same noise, they will be spawned in the same positions. So the system uses tree indices and is also mapping which trees are being chopped down. These indices are being used to save in a memory chopped trees after terrain tile is unloaded with the purpose that when player comes back, there would be always the same trees chopped down (not regrowing). The number of chopped trees is usually much smaller than the number of all trees on the terrain, so it's more reasonably to keep chopped tree indices in memory than alive tree indices. This way if player just travels around and no trees are chopped, nothing is saved in memory and player can explore the world indefinitely without worrying that memory could fill up.

Thanks to Unity's billboarding feature, thousands of trees can be spawned on the terrain without significant reduction of FPS. By default setting there can be several thousands of trees (up to 12000 if fully covered with forests) on a single terrain tile. On all nine tiles this number reach several ten thousands easily. However, as billboard system is fast, there is very little cost when rendering these billboards. Tests has been done with larger numbers, however, tree densities in forest becomes high and there is difficult to see warriors through these trees. **treeBilboardStart** in GenerateTerrain is the distance from camera at which trees on all terrains will be switched to billboards.

Tree models in RTS Toolkit has been generated by TreeGen. In the default project there are imported 6 spruce models from TreeGen grid: sprc_1, sprc_11, sprc_46, sprc_47, sprc_54, sprc_55. They can be found in Assets/RTSToolkit/Resources/trees directory under corresponding subdirectories. There are leaf front and back, and bark materials, tree mesh, tree gameObject prefab and config file (which has the number of how many meshes tree contains). As Unity uses 65535 vertices limit, TreeGen splits meshes which has more vertices than that number into 2 meshes and so on. This allows to use unlimited number of vertices per tree. If the tree has more than one mesh, it will be automatically imported and set through tree prototypes in order to use them. Tree health and removal when trees

are chopped also works with multi-mesh trees. This is why GenerateTerrain has a list of strings for treeName rather than the prefab reference in order to find and load all meshes per tree. Tree model textures are saved in Assets/RTSToolkit/Resources/trees/textures directory. Spruce models use grey bark texture “bark3” and branches use “spruce_branch512” texture.

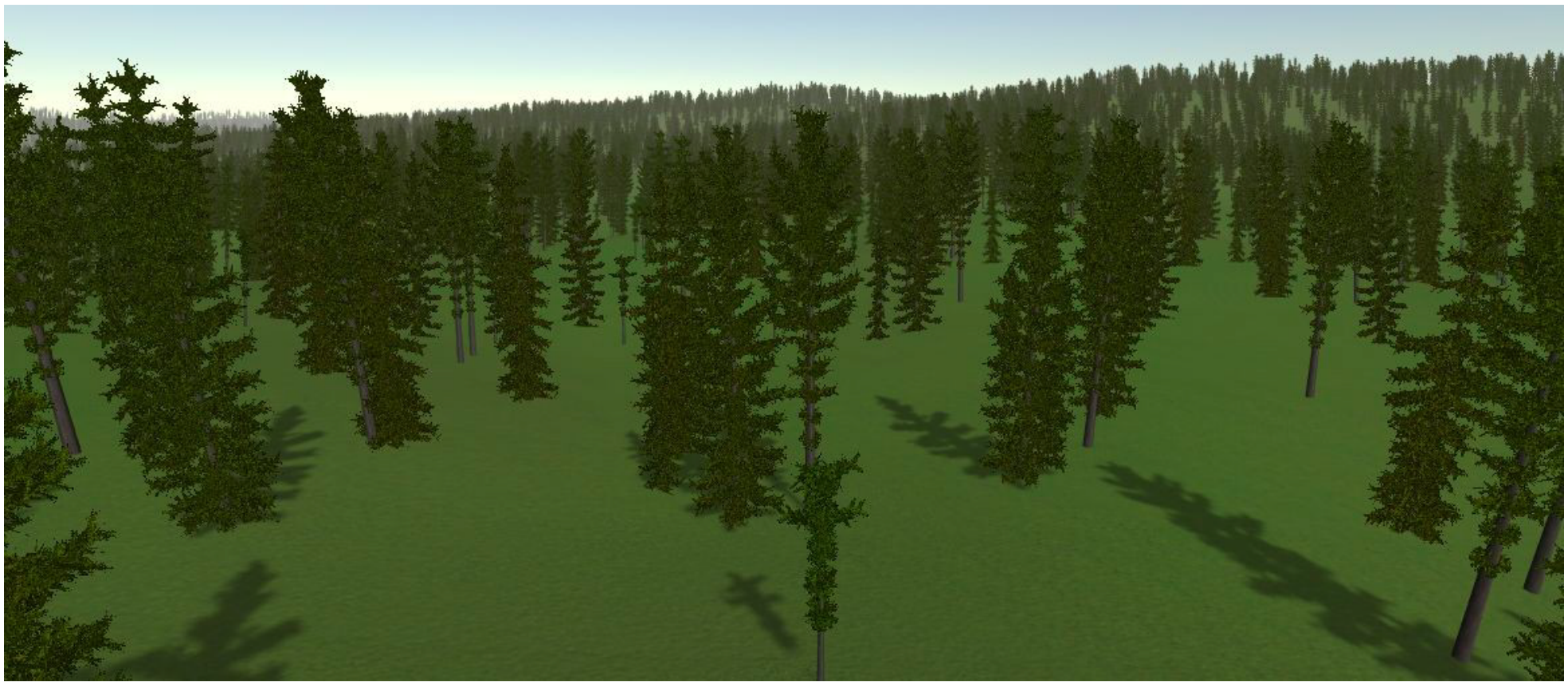


Fig. 40 - Forest with various spruce tree models.

In Fig. 40 can be seen a forest, which contains large number of trees all the way to the horizon. There are also visible that different tree models are being used in different places, making a rich variety of trees inside the forest.

3.2.1. Nature Package forest setup

Nature Package (https://www.assetstore.unity3d.com/en/#!/content/42225) has one of the most realistic tree models ever designed. Its biomes has been set and configured in RTS Toolkit. However, in order to benefit from Unity tree billboarding tree models should be slightly changed to use only one LOD instead of multiple and switch to soft occlusion shaders. To fix pivot issue models with a single mesh can be re-exported using Blender or any other modeling software. In Assets/RTSToolkit/Prefabs/Nature is Forest_NaturePackage prefab, where prefab fields are left empty for users who wish to use Nature Package. Model names used for each vegetationType entry are entered in treeName fields. After setting up treePrefabs user can put Forest_NaturePackage gameObject into the scene instead of another gameObject with Forest component (note that multiple gameObjects with Forest component could exist in the scene but there has to be only one of them set as active). Fig. 41 show how forest looks like when using Nature Package models.

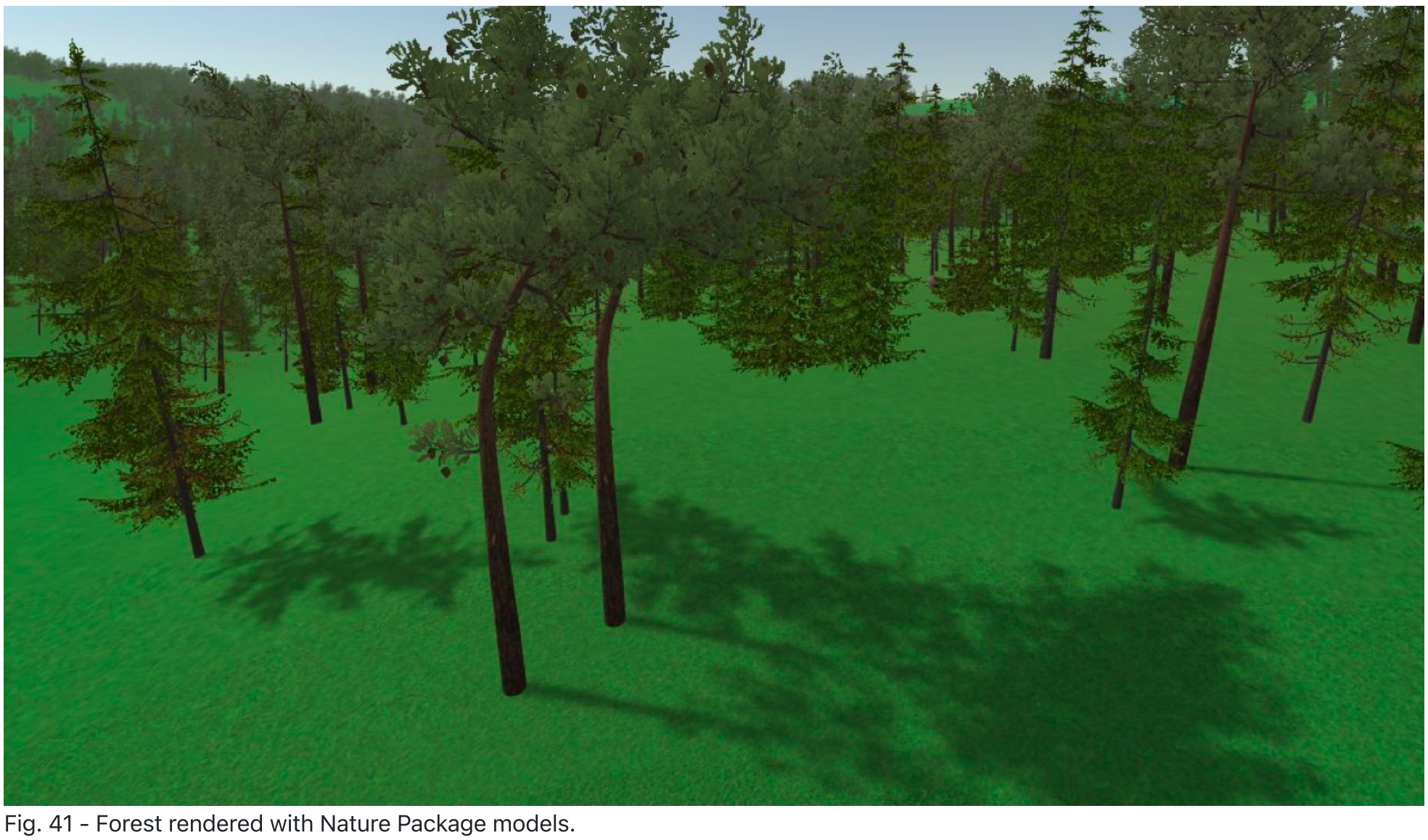


Fig. 41 - Forest rendered with Nature Package models.

3.3. Water and lakes

RTS Toolkit has water gameObject set in order to make lakes (Fig. 42).

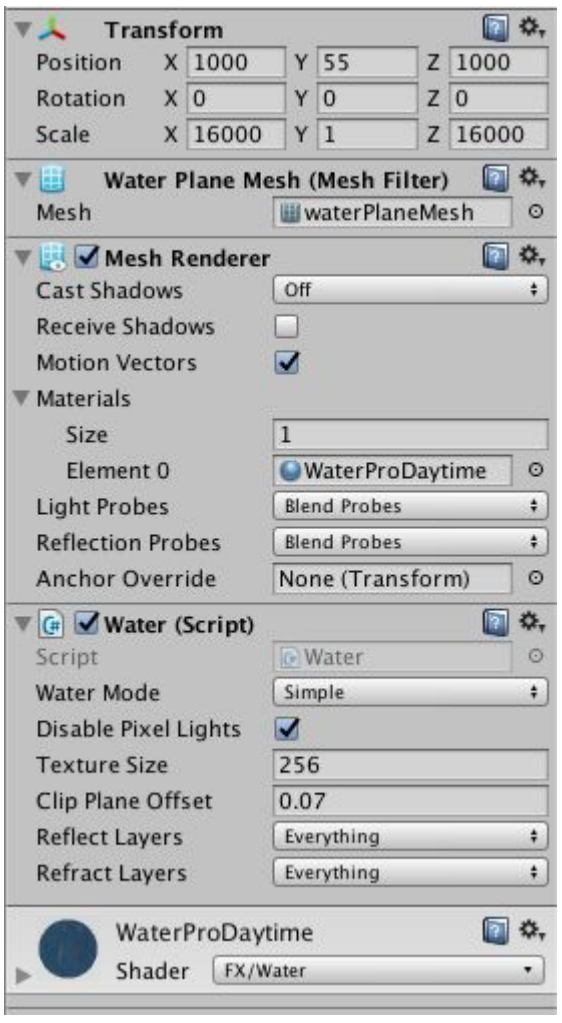


Fig. 42 - Water gameObject components in the scene.

The gameObject uses default water prefab from Standard Assets (Standard Assets/ Environment/Water/Water/Prefabs/WaterProDaytime), which is scaled up to be behind far clipping plane of the camera. For example if camera's far clipping plane is 6000, setting scale >12000 units for water gameObject makes it behind the camera's horizon. So that gives a nice looking horizon in all directions without any sharp edges. The quad is initially placed on a middle of zeroth terrain tile, i.e. if terrain tile size is 2000x2000, the middle point of zeroth tile would be at x=1000, z=1000. y coordinate gives the height level of the water, i.e. if y=55 then water heigh in the game will be set at 55. GenerateTerrain uses water gameObject reference from the scene to avoid spawning trees below water level. It also triggers shifting water gameObject position to the central tile when camera moves between terrain tiles and new tiles are generated.

Water gameObject also uses Water component which moves (animates) water surface waves for waterBasicDaytime material. This material uses FX/Water shader and Simple waterMode. waterMode can be changed to Reflective or Refractive to get higher quality water, but these modes are computationally expensive.

The places, where terrain height values drops below water level are underwater. In these locations lakes are being created (Fig. 43). If developer decides to change water level, it needs to clean terrain tiles by pressing "Clean" button in GenerateTerrain and generate new terrain with the new water level being set.

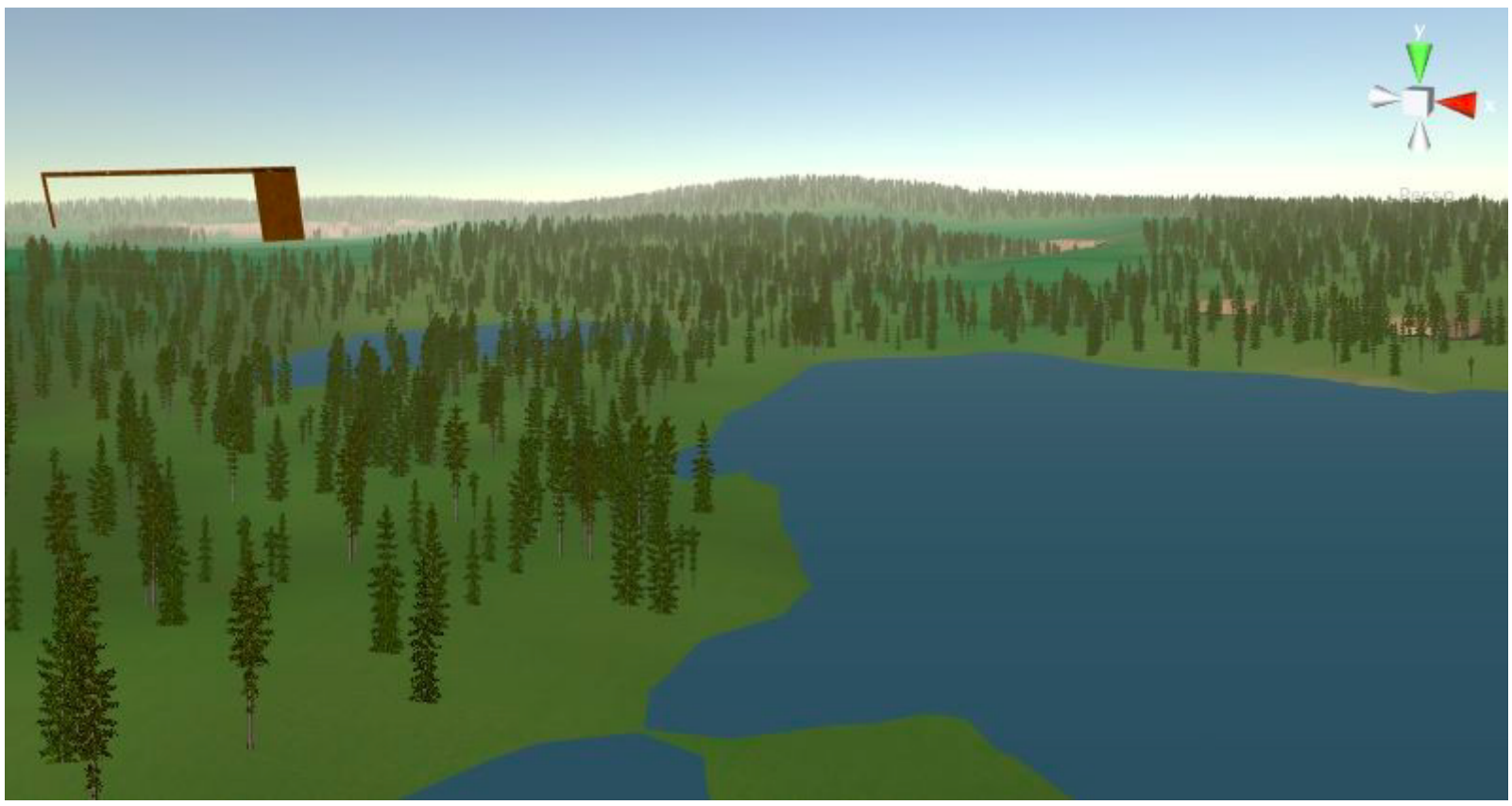


Fig. 43 - Places where terrain height is below water level appears as lakes. TerrainGenerator uses water y position to avoid spawning trees in the water.

3.4. Rivers

RTS Toolkit use procedural river valley carving in the terrain. Diffusion limited aggregation is used to create wiggling river paths with sub-merging branches of smaller rivers. The middle part of the path drops below water level and this is where river appears.

Rivers gameObject has **Rivers** component attached to it (Fig. 44). The parameters are:

pt	Particle spawning point for diffusion limited aggregation (DLA) - the default values are slightly above 0 to allow particle to randomly travel to reach 0.
origins	Initial attraction points, where particles are sinking to. These becomes locations where rivers ends.
randomSpeed	The random speed component at which particles are travelling.
flowSpeed	The directional speed component towards attraction points.
maxMovementSteps	Maximum number of steps to prevent particle random travelling from being near infinite (there is no guarantee that each particle will reach attraction point). If particle makes more movement steps than maxMovementSteps, it will be discarded and instead a new DLA particle would be spawned.
numberParticles	Number of particles to proceed over the whole DLA simulation.
numberSubSplits	the number of times to sub-split each river segment between two river points. It also adds random path between these two river points, i.e. can be seen as increment of resolution. numberSubSplits has to be a power of 2.
worldOffset	The offset of the final rivers map in the world positions.
worldRotation	The rotation of final rivers map in degrees.
minimumTerrainHeight	The minimum height of the terrain in the deepest parts of the river.
nShorePixels	the number of terrain pixels where river valleys are interpolated between actual terrain height and minimumTerrainHeight. Larger values of nShorePixels gives wider valleys of the river.
randomHeightAmount	The amount of random height along the river path.
randomHeighPosibility	The possibility to add random height along the river when generating. These could help when creating shallow points through which units can cross the river.
shoreProfile	The curve of how the river shore should be carved.

So DLA algorithm firstly spawns a particle, which travels randomly in space based on randomSpeed and flowSpeed until it reaches one of the attraction points. At that moment DLA particle itself becomes attraction point connected with the previous attraction point. After that new particle is spawned and process repeats. As this continues, fractal-like structure builds up until all particles are being used. The list of attraction points defines river segments which are subdivided into smaller segments in order to obtain final paths of these rivers. As subdivision happens, one point is inserted between two previously defined points, which results as splitting segment into 2 sub-segments. Then subdivision completes, final list of points is used to carve down the terrain heightmap, which comes out as rivers (Fig. 44).

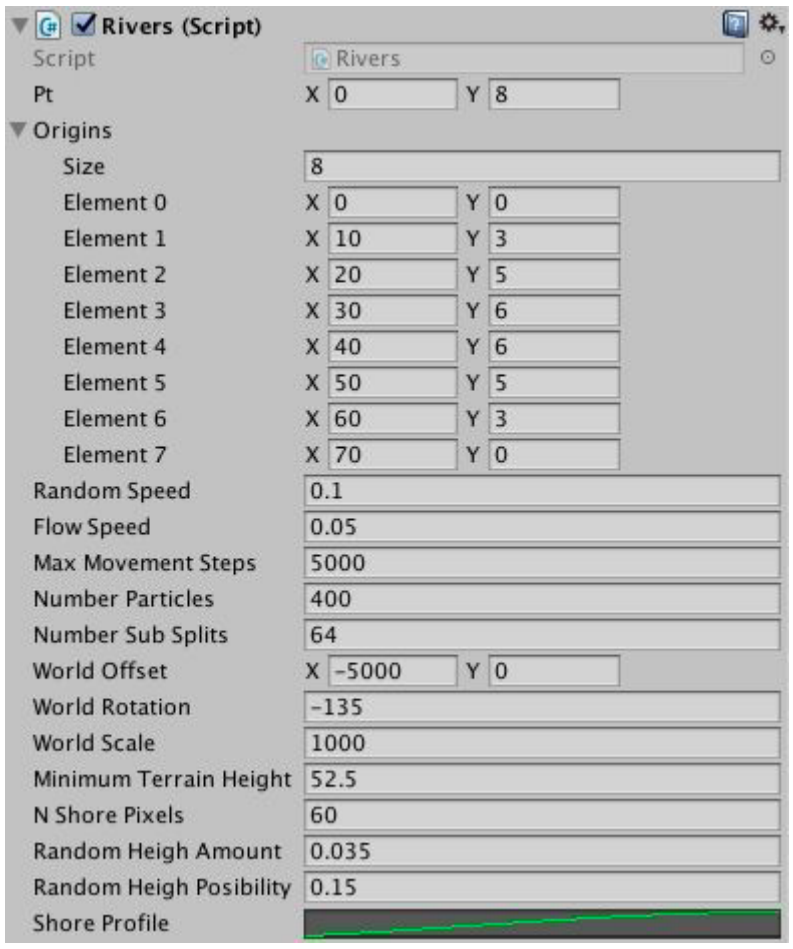


Fig. 44 - Rivers component setup.

River paths made in this way goes up to multiple dozens of kilometers, just like in real world. In some places rivers are passing through lakes, other places, where DLA fractals merges are also merging point of two rivers (Fig. 45). River valleys and zigzag trajectories also looks natural. River paths are never becoming randomly placed when coming back to the same map as random numbers are always running from the same seeds. As a result, this procedural rivers generation approach allows to bring naturally looking rivers to the endless terrain.

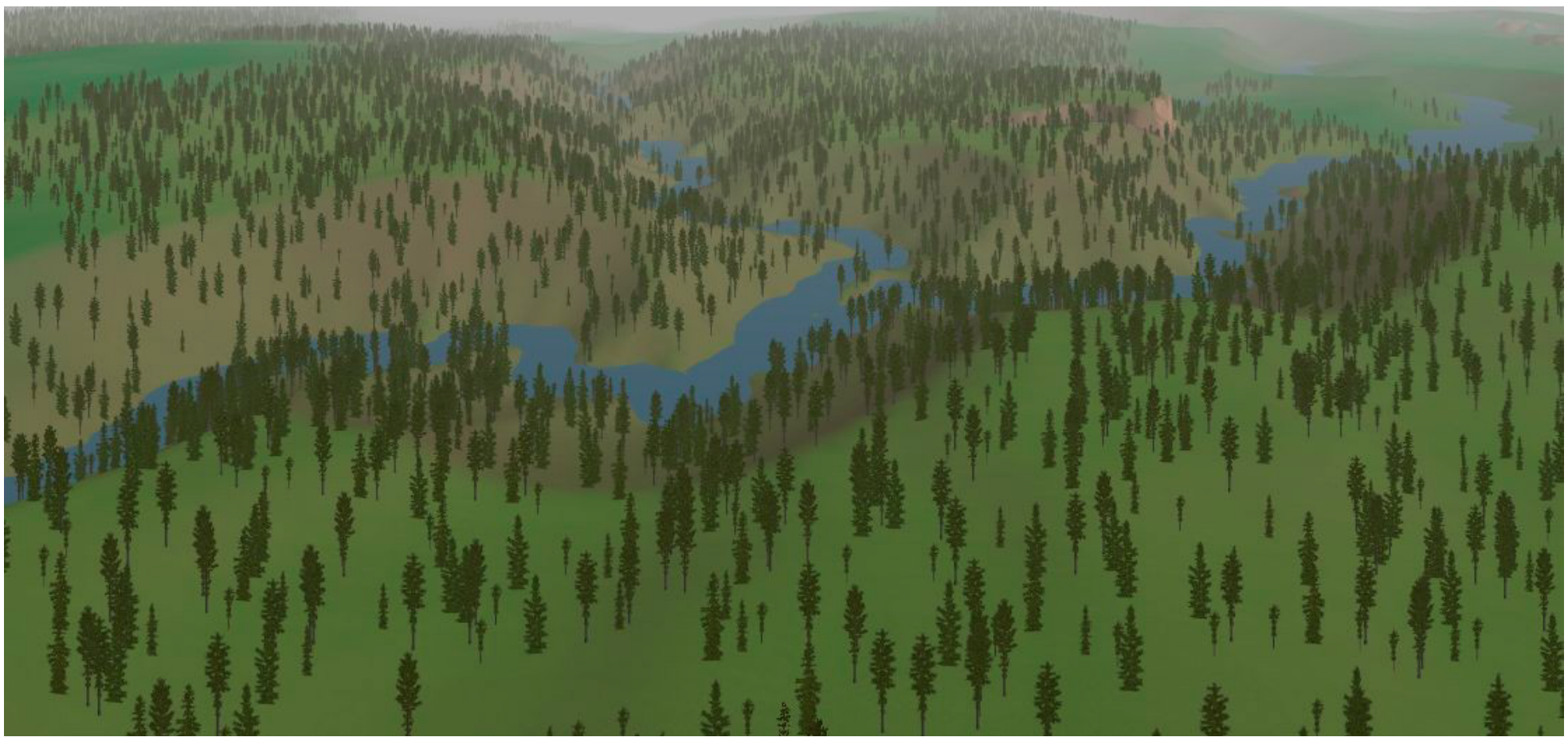


Fig. 45 - Rivers as they look in game. In the picture is visible merging point of two rivers.

3.5. The sea

As there are rivers, they always ends into the sea. RTS Toolkit uses very simple sea generation. The circle with radius of up to several hundred kilometers is used to drop down terrain heights everywhere inside this circle below water level.

There is used **GenerateSea** component in order to create a sea (Fig. 46). **seaCenter** defines where is the circle center of the sea. **seaRadius** encloses maximum depth area, where terrain heights are dropped by **dropTerrainAmount** value. **outerRadius** defines where the coastline finishes and sea does not affect terrain heights. In the area between seaRadius and outerRadius is a coastline area, where heights are interpolated. **worldRotation** allows to rotate circle center around 0,0 position and **worldOffset** allows to offset sea center position after it's being rotated.

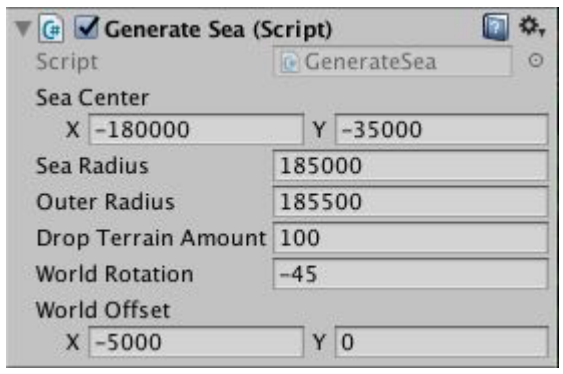


Fig. 46 - GenerateSea componentsetup.

Fig. 47 shows the coastline area where one of the rivers enters the sea. The coastline is not smooth as sea height drop is applied on the top of generated heightmap. Where terrain heights are larger, these areas are likely to be above water level further into the sea. This makes natural looking coastline areas with smooth lands and cliffs submerging into the sea. There may also appear some islands in the sea in some areas as well.

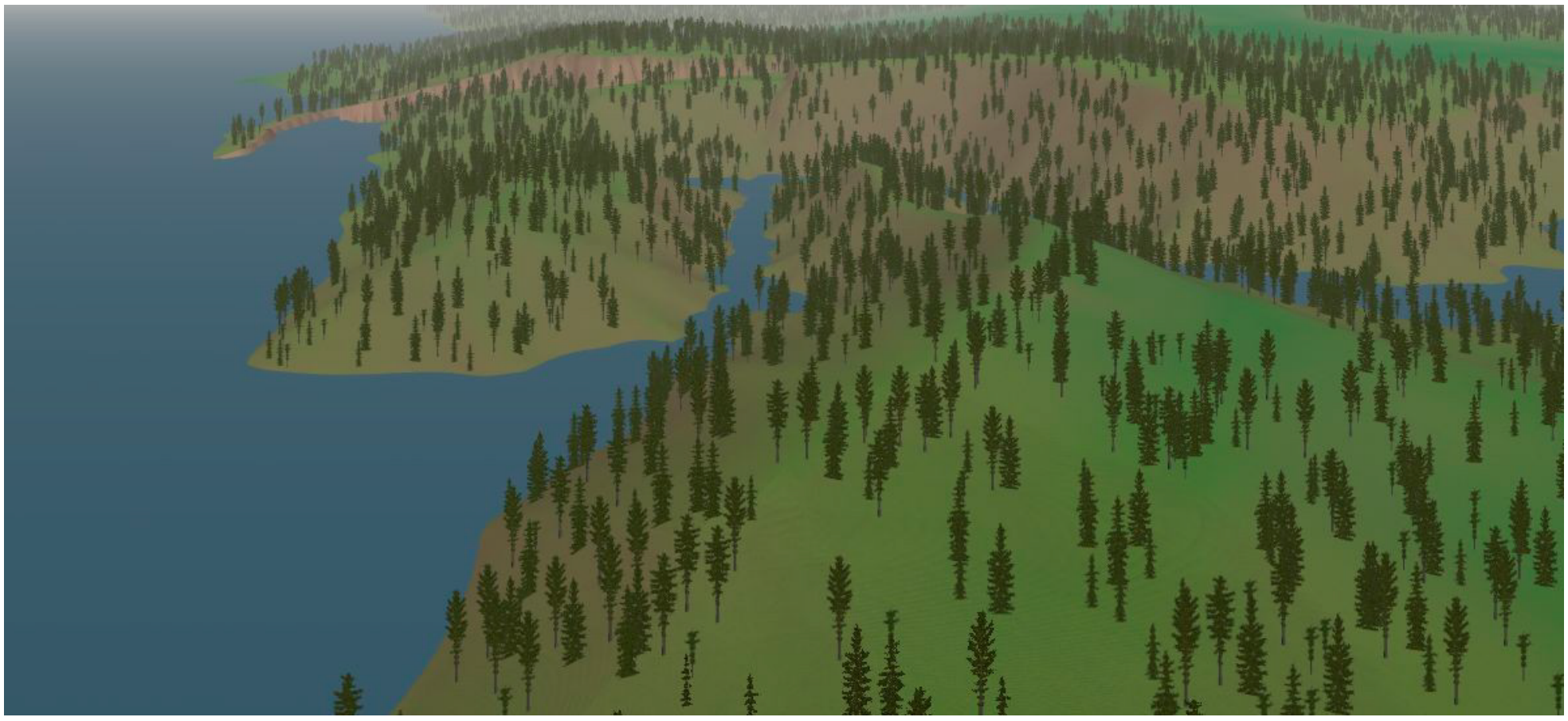


Fig. 47 - The sea coastline area and the river entrance point.

3.6. Clouds

RTS Toolkit uses non-expensive particle system to make near-realistic clouds in the sky (Fig. 48).

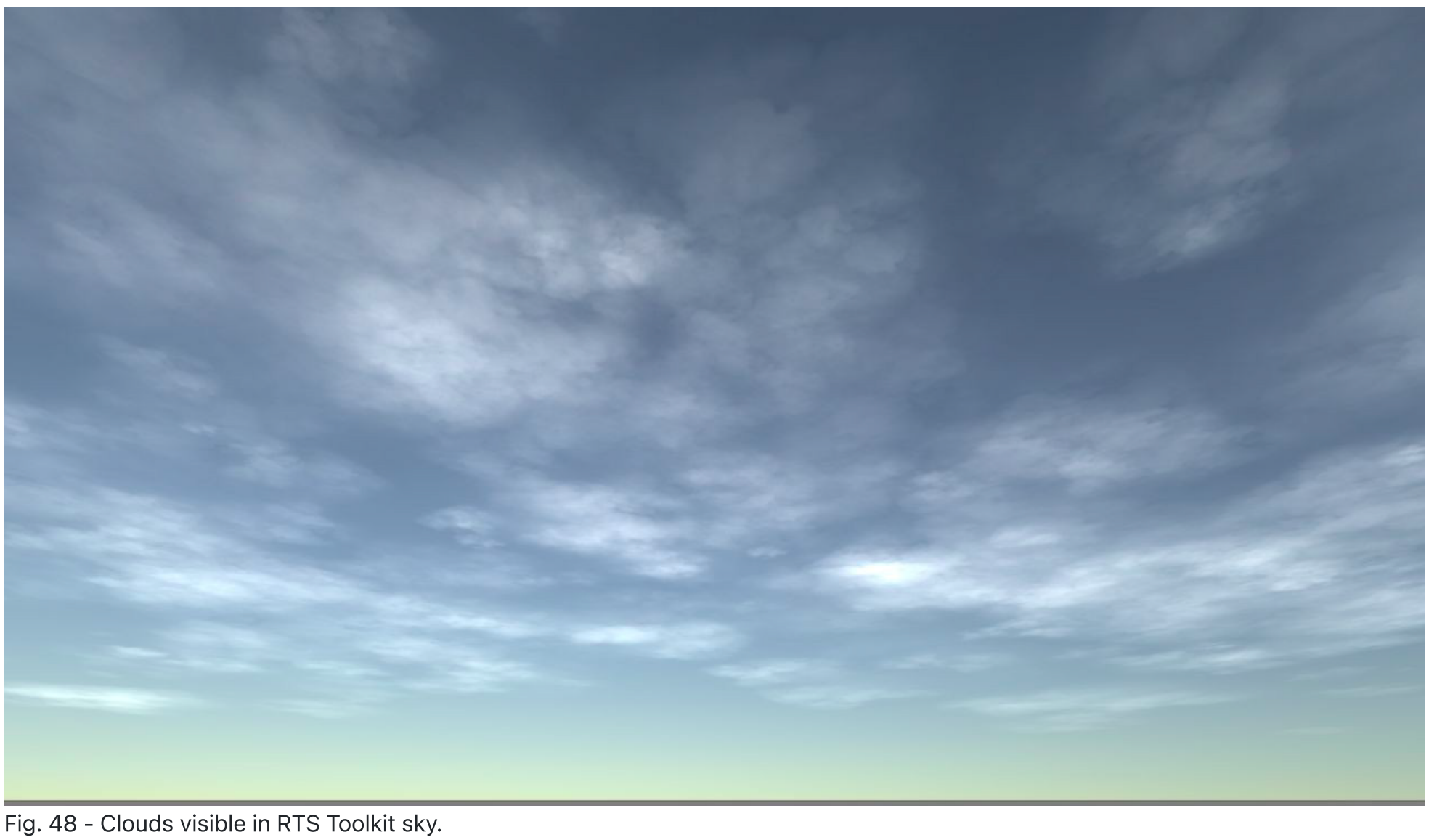


Fig. 48 - Clouds visible in RTS Toolkit sky.

The cloud particle system has been set from “Dust Storm” particle system, included in Standard Assets (see tutorial: https://www.youtube.com/watch?v=saGyjU5wHWw). Fig. 49 shows the setup of clouds gameObject.

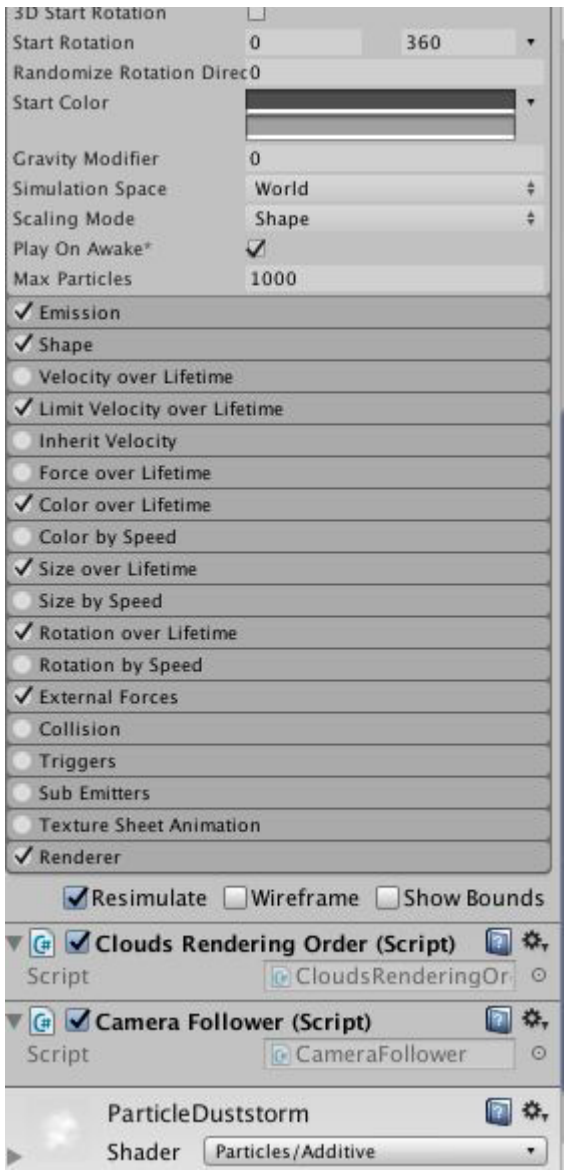


Fig. 49 - Clouds gameObject setup.

Particle system has been scaled 100 times and moved to height of 900. To reduce cloud evolution speed to more natural rates **duration** and **startLifetime** was set to 100. Cloud sizes are increased by setting **startSize** to 1500 - 2500. This gives one cloud represented as one big particle, which slowly moves and rotates. This gives that the whole sky can be filled with clouds by using just ~1000 particles. To keep clouds in their position when gameObject is moved (this happens when camera moves), **simulationSpace** is set to World. By enabling **externalForces** we allow clouds to follow the wind direction. So this setup gives naturally looking clouds.

Clouds material use Particles/Additive shader which gives bright-white colour. The texture is used as ParticleCloudWhite. This setup gives feathers-cumulus like bright and thin clouds in sunny day. More distant clouds, visible towards the horizon appears more like feather clouds. Developers can very easily create a completely different clouds by just changing this texture. It’s also noticeable, that different types of clouds (i.e. non-feather ones) can be achieved by setting **renderMode** in Render different than Horizontal Billboard (i.e. Billboard will make clouds to always face camera and won’t give feather clouds close to horizon), or a combination of several cloud gameObjects.

CloudsRenderingOrder.cs is attached component on clouds gameObject, which corrects that clouds would be rendered behind tree billboards, as by default in Unity, tree billboards are displayed always behind any particles, independently on actual distances to camera. The correction is done by setting clouds material **renderQueue** to a high value in Start() function.

CameraFollower.cs allows to follow clouds gameObject the moving camera. This allows that when player moves camera, that cloud gameObject would move together with it. When clouds gameObject moves to a new position, previously generated cloud particles remains in their positions due to simulationSpace=World. New particles are being spawned in the new boundaries of particle system and when old clouds dies out, all remaining clouds becomes visible within new boundaries. Cloud particles are also created in 3d space, so when camera moves around, player can see that it also moves relative to these clouds, what can’t be achieved by using static skyboxes.

3.7. Wind

Wind is not visible in the game, but it allows to have more dynamical and alive environment by moving clouds, smoke from chimneys of various building, fire and smoke created by burning places, rotating Windmill blades, and bending trees. To make all these movements to look natural and good, wind parameters needs a careful setup. Wind gameObject setup is shown in Fig. 50.

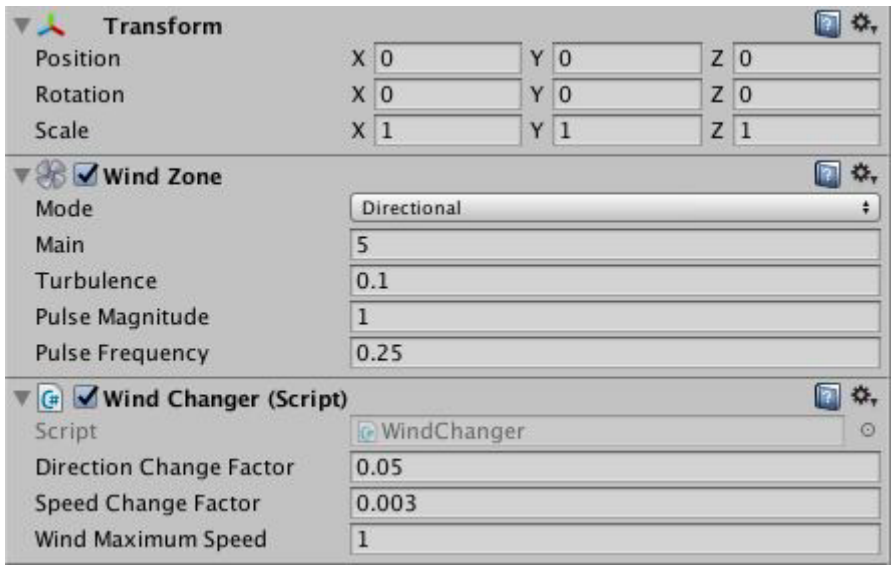


Fig. 50 - Wind gameObject with its components.

WindZone is the main component, which automatically sets that trees and particles with external force would react to the wind. The direction of the wind is defined by wind gameObject rotation. The default 0 values in rotation gives that wind is blowing along z axis to positive direction. **WindChanger** is the script, which changes wind's direction and it's strength. **directionChangeFactor** controls how fast wind direction is changing. It's value is in terms of maximum possible angle per frame, i.e. 0.05 means that in a frame wind direction can change no more than 0.05 degrees. Direction is changed based on random walk approach - on each new frame random number is rolled between directionChangeFactor and -directionChangeFactor, which is added to the direction vector. Only y component of wind gameObject rotation is modified, as wind blows horizontally. **speedChangeFactor** controls wind speed in the same way as directionChangeFactor controls the direction. Speed is defined through windMain variable in WindZone ("Main" in Editor). As random walk approach can move speed values to unrealistically high, **windMaximumSpeed** parameter is used to cap maximum value of the wind speed. In that case if random value tries to increase speed above windMaximumSpeed, the actual speed remains at windMaximumSpeed. The values of windMaximumSpeed gives the maximum possible value for windMain.

3.8. Environment lighting and the fog

RTS Toolkit uses very basic setup of environment lighting without any precomputed or baked GI, as these lighting approaches are very computationally expensive. The main light source in the game is the sun. However, it produces very hard directional light (areas in shadows and environment at the sunset or sunrise becomes unrealistically dark) and in addition to the sun there is used ambient single colour light, set through Lighting window of the scene (environment lighting and fog parameters can't be set through components, but are set instead through Lighting window, which can be found in Window > Lighting).

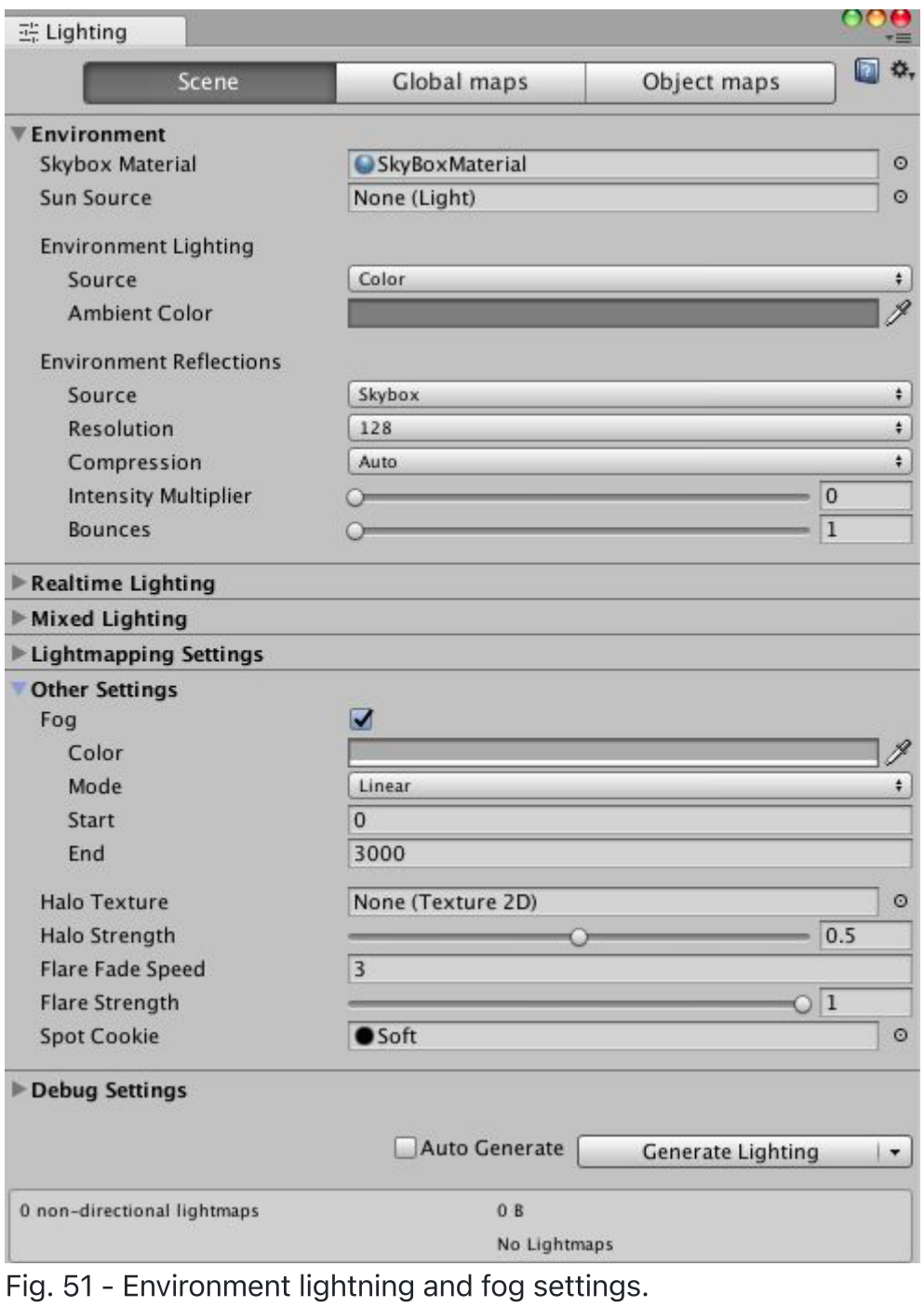


Fig. 51 - Environment lightning and fog settings.

Default scene lighting settings are shown in Fig. 51. There are set fog parameters for the scene. Fog is being used to blur more distant objects (rough and not computationally expensive approximation for atmospheric scattering). Linear for mode is used to blur objects between 0 and 3000 distance units. It's also good to keep fog colour similar to the sky colour near the horizon. Fig. 52 shows how distant trees appear more grey than the close ones when fog is used.

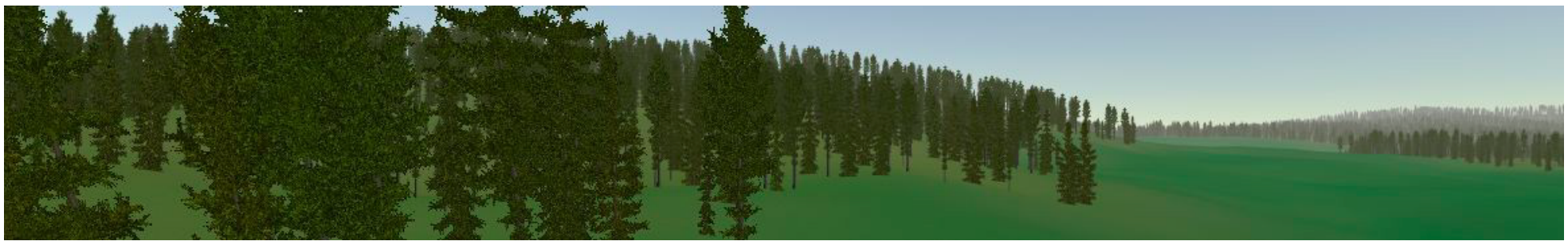


Fig. 52 - Fog makes more distant landscapes and trees to appear blurred.

The skybox uses default Unity's procedural skybox setup with several changed parameters. That was done by creating Skybox/Procedural material and saving it in Assets/RTSToolkit/Prefabs directory as SkyBoxMaterial (Fig. 53).

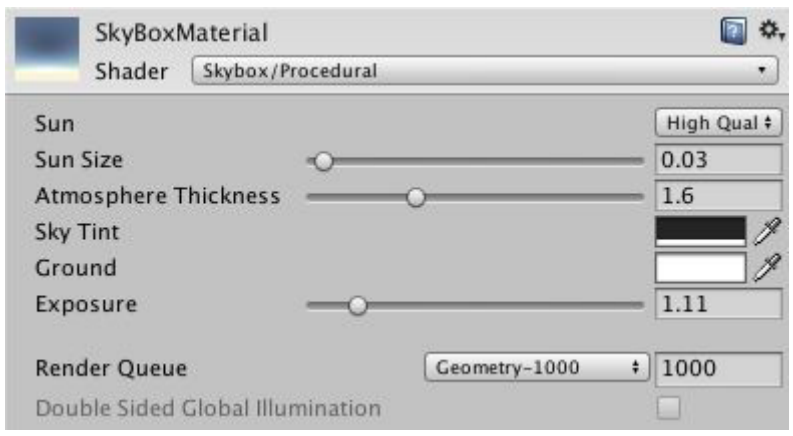


Fig. 53 - Modified skybox material.

The main reason of creating this material is to set sun to the High Quality and smaller size than in the default scene, as it brings much more realistic looking sun compared to the default one (Fig. 54).

a)

b)

Fig. 54 - The sun visible in the mid-day (a) and before the sunset (b).

3.9. Time of the day

RTS Toolkit use sun's rotation to create day and night cycles. This is done from TimeOfDay component attached to the Sun gameObject (Fig. 55).

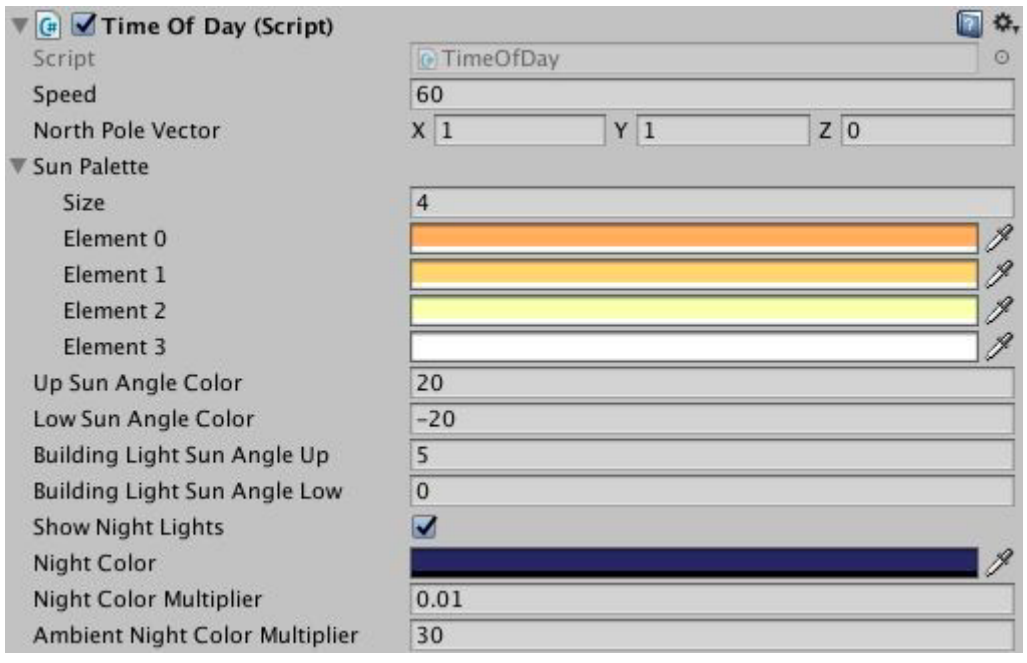


Fig. 55 - Sun gameObject components and setup.

The component rotates the sun around earth axis in order to reproduce realistic sun movement over the sky. Earth axis in TimeOfDay is defined through **northPoleVector**, which shows the pointing direction of Earth's North pole. For example northPoleVector with (1,1,0) gives sun movement in the sky for 45 degrees northern latitudes with the direction of the North to be towards positive direction of x axis. northPoleVector with (-1,1,0) would give 45 degrees southern latitudes, etc. The sun's gameObject is being rotated by using RotAround() function, which rotates one vector around another (in this case sun would be rotated around earth axis).

Speed is used to control how fast sun rotates relative to the real period of the day (24 hours). The speed of 60 means that 1 hour in the game world will take 1 minute and that the length of the day is 24 minutes in the game (i.e. between sunsets or sunrises).

sunPalette list is used to determine the colour of the sun's directional light. The first colour (solid orange) gives the colour of the sunlight at the closest point to the horizon. The last colour (white) is used for the sunlight at high heights. Colour between are used as a palette, i.e. to define more intermediate colours, such as yellow rather than just interpolate between white and orange. Fog colour is also changed based on the sunlight colour. To detect when to change the colour, sun's height above horizon is calculated as the angle between the full vector to the sun and the same vector XZ projection. The sun's colour starts to change at **upSunAngleColor** and ends changing at **lowSunAngleColor** angles (i.e. 20 and -20 means that sun's colour will start to go from white towards yellowish when the sun's height above horizon starts to drop below 20 degrees, and reaches the most orange colour when it reaches -20 degrees). Sunlight intensity is also changed based on its height from horizon - it drops to 0 when sun is below -20 degrees. Ambient light is also changed by using sun's angle. **nightColor** allows to set the color of the night, which is being used for ambient light. **nightColorMultiplier** multiplies the nightColor as nightColor can't be set at extremely dark colors. After that it's also multiplied by **ambientNightColorMultiplier**.

TimeOfDay supports the usage of "Time Of Day System Free" asset (https://www.assetstore.unity3d.com/en/#!/content/71422) skybox material. This material is named "TOD_SYSTEM_FREE_SKY" and can be set through Window > Lighting > Settings > Scene > Environment > Skybox Material. Once the material is assigned, several more variables will automatically appear in inspector onto TimeOfDay component. This includes **moonTexture**, **starsTexture** and **starsNoiseTexture**, which will be set into shader in order to use sky dome not just with sun, but also with moon and stars at night. Moon and stars also rotate in the same way as the sun.

TimeOfDay script also controls building lights, which are being turned on at night time, when it becomes dark in order that player would see where are its buildings and units. Building light are set by using child gameObjects on building prefabs, which uses point lights (Fig. 56).

Fig. 56 - Building light setup on a child gameObject of building prefab.

buildingLight component on that prefab is the one, which registers building light in TimeOfDay when building gameObject is being instantiated, and removes when building is destroyed. buildingLight enables Light component at nighttime and disables it at daytime. The sun height angles, used for enabling and disabling building lights are defined by **buildingLightSunAngleUp** and **buildingLightSunAngleLow** in TimeOfDay script (see Fig. 55). Random values between these limits are assigned for each building. This gives that buildings turn on their lights not instantly but one by one just before the sunset and also turns of their light one by one after the sunrise. Only some buildings has set their lights, as bringing large number of point lights is computationally expensive. These buildings can be important town buildings, such as Central Building, Barracks, Factory, Research Center, Stable, while other buildings, such as Wood Cutter, House, Mining Point, Windmill, Tower has no lights and are dark at night time (these buildings can have child gameObject with Light and BuildingLight components attached, but BuildingLight components should be disabled in order not to use the light).

Fig. 57 - Building lights illuminates building interior and the area around the building, which allows to see units at a night time.

Fig. 57 shows how building are illuminating areas around them in the town. Player can easily see units, which are close to light sources, while more distant units are harder to see. This makes possible game play in the realistic night environment for player. Night time can be a game changer for players who has different experiences in making night time battles and adds additional dimension of playability in RTS games. It can be sometimes expensive to have many point lights in the game, so there is possible to disable them by deselecting **showNightLights** tick in TimeOfDay script (Fig. 55).

3.10. Ambient sounds

The RTS Toolkit terrain is also filled with ambient bird sounds. These sounds are 3D sounds and are dependent on player camera position. When player camera moves on the terrain, sounds are also changing. This is also working with continuous endless terrain - as player camera moves and new terrain tiles are generated, sounds are also placed on new terrain tiles.

Ambient bird sounds are managed by SoundManager component (Fig. 58).

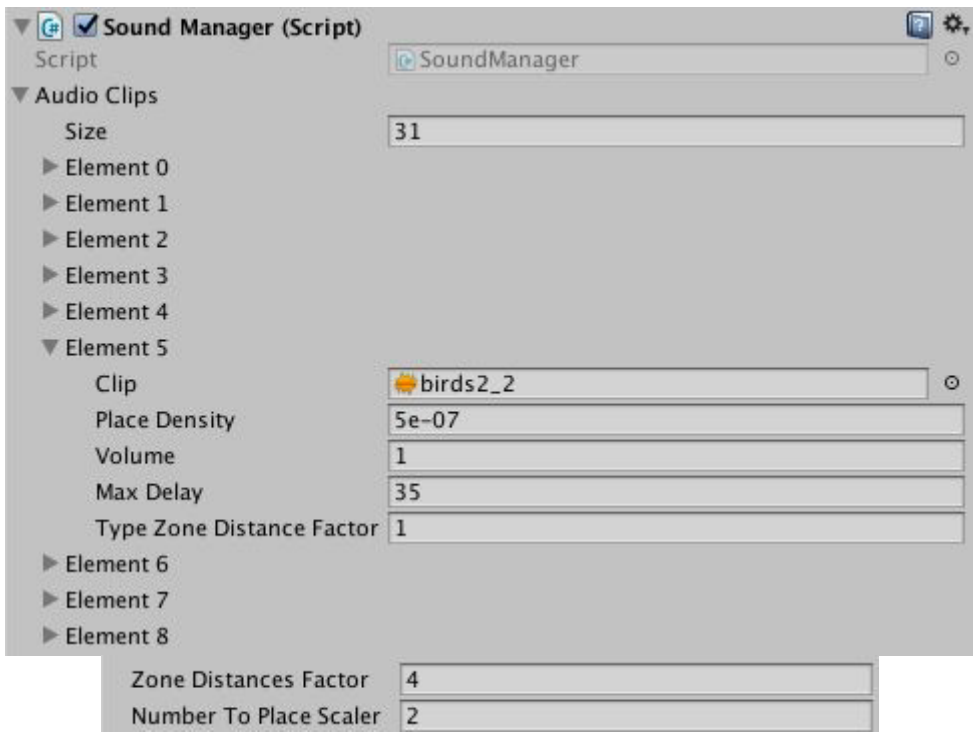


Fig. 58 - SoundManager component which is responsible for placement and playing ambient sounds on RTS Toolkit terrain.

SoundManager component has a simple setup. Firstly, **audioClips** is the list of all environment sounds, which are placed on the terrain. Each element in the list has several parameters, which describes how the sound should be distributed on the terrain and played. **clip** is the reference to the audio file. **placeDensity** defines how many clips should be created per every square unit on the terrain. The number is quite low on default terrain with size of 2000. **volume** specifies how loud the sound should be playing. **maxDelay** is the maximum delay to wait for the sound to repeat again. **typeZoneDistanceFactor** is the multiplier for minimum and maximum distances from camera where the sound is set to play. Each of these are properties for individual sound types.

There are also two other properties, which are applied to all sounds. zoneDistancesFactor is the same as **typeZoneDistanceFactor**. The difference is that it is applied to all sound instances over all types of sounds, on the top of typeZoneDistanceFactor factor. **numberToPlaceScaler** is the multiplier applied to the number of sounds being spawned based on placeDensity.

Once player starts the game, sounds are being instantiated based on their density. Only sounds, close to camera are played while more distant sounds are kept as disabled. In other words this technique to play sounds only close to camera can be called as sounds LOD. Users, who change sounds playing distances, should be aware that large distances can make very large number of sounds being played, which can degrade gameplay performance.

4. UI

UI is defining the way how player interacts with the game. It includes camera interactions, menus, buttons, selection marks, etc. This involves usage of Unity 4.6 GUI system, older OnGUI() calls and custom scripts, where built-in systems can't handle one or another issue.

4.1. RTS Camera

Many RTS game developers starts to create RTS games from setting up camera. There are also rumors that "the main thing to set in RTS games is RTS camera". However, RTS camera is nothing more than just a small part of UI system, allowing player to control camera. On the other hand, proper setup of camera is a good step as usually camera controls are the first thing, which player see when starting the game.

The basic concept of RTS camera is that the camera is not bound to any unit and moves in free space. Camera should be controlled in a way, that player could easily see it's all units, buildings and select them when needed. This approach may look simple, but different style of RTS games uses different variations of RTS camera, i.e. 2D game is likely to have only movement of orthographic camera without rotation. 3D games may include rotation and zooming operations as well as perspective view of the camera. As there is large number of possible configurations, the RTS camera setup depends on the actual game and is the art of setup.

It is usually easier to have most complex and most realistic model of the camera setup and later simplify it to the needs rather than to have simple model when one or another feature would be needed to implement from scratch. We can call it as top-down setup, when developer takes realistic setup and deletes what is not needed. RTS Toolkit uses RTS camera based on this top-down approach: it has features to move, rotate, zoom, follow terrain, etc. Developers can simply comment out unwanted parts for one or another operation in the code and easily adapt to what the game needs.

The setup of RTS camera gameObject in RTS Toolkit is shown in Fig. 59.

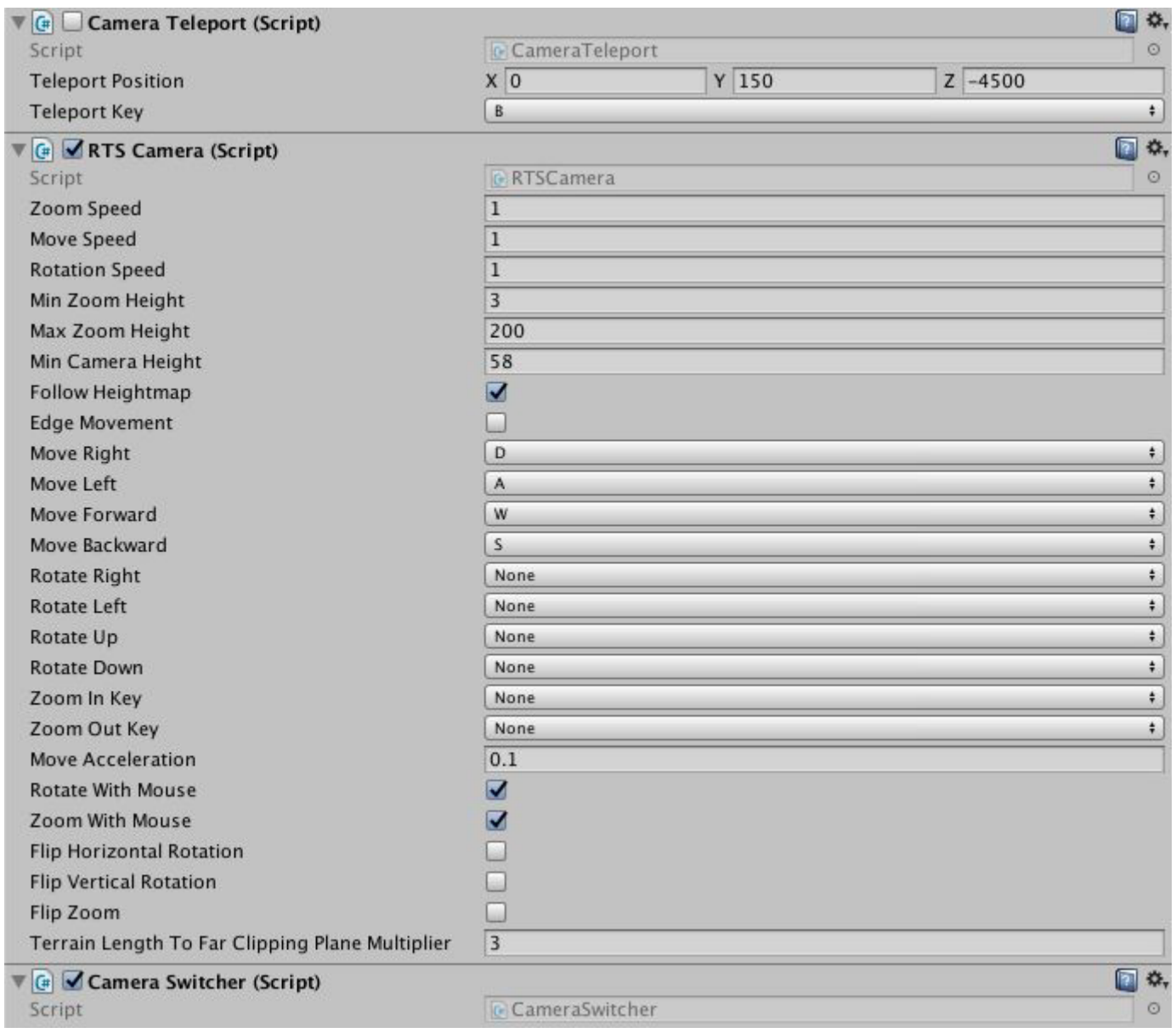


Fig. 59 - Setup of RTS camera gameObject components.

First four components (Camera, GUI Layer, Flare Layer and Audio Listener) has been set at the beginning when creating a new camera in the scene. The last four components (CameraTeleport, RTSCamera, CameraSwitcher and RPGCamera) are custom scripts, which controls the camera.

The main component **Camera** has been slightly modified compared to the default camera. Far clipping plane is set to 6000, what means that player can view as far as 6000 distance units in the game. This distance is high enough to see everything properly towards horizon with 2000x2000 size terrain tiles. The horizon is created by water, which is scaled to size of 16000 units, what gives, that water quad extends up to 8000 units from camera and being clipped at 6000 units by far clipping plane, setting smooth horizon. Viewport Rect is also slightly modified. X, Y, W and H values are set to make camera visibility only on the active area of the game, to avoid rendering pixels behind always visible GUI frame with menus and buttons.

RTSCamera is the main component, which controls movement of RTS camera. The movement of camera is defined by changing its position. It's set by key codes through the inspector as **moveRight**, **moveLeft**, **moveForward** and **moveBackward**. By default their values are set as D, A, W, S. When player press these keys, camera moves to the corresponding direction, which is relative to the camera's current viewing plane. Movement can be softened (made more incremental) by assigning higher values of **moveAcceleration**.

Camera's rotation is made by holding down right mouse button and moving mouse around if **rotateWithMouse** is set to true. Camera can rotate free all 360 degrees horizontally, but is restricted to only 180 degrees vertically. There can be also set camera control keys (**rotateRight**, **rotateLeft**, **rotateUp**, **rotateDown**). Both horizontal and vertical camera rotations can be flipped by setting **flipHorizontalRotation** and **flipVerticalRotation** ticks.

Player can zoom in and out by scrolling mouse wheel (if **zoomWithMouse** is enabled), what makes camera height to change between **minZoomHeight** and **maxZoomHeight** distance units from the terrain. **zoomInKey** and **zoomOutKey** can be also set in order to control zooming with keyboard. **minCameraHeight** allows to specify minimum height below which camera should never go, e.g. water level. Zoom direction can be changed by setting **flipZoom** tick.

moveSpeed, **rotationSpeed** and **zoomSpeed** allows us to set how fast camera can move, rotate and zoom in arbitrary units. Actual move and zoom speeds depends on camera height from the terrain as well. If the height is low, movement and zooming speeds are reduced, if it is high - increased. This makes much more natural way to move and zoom when playing on different scales.

terrainLengthToFarClippingPlaneMultiplier allows to set the multiplier of how far clipping plane of the camera should be automatically scaled to the terrain tile length defined by GenerateTerrain.

CameraTeleport is a quick test tool, which allows to quickly teleport game camera to **teleportPosition** by pressing **teleportKey** in game. If the component is disabled (like in Fig. 59), teleport won't be used. If player clicks the key second time, camera returns to the original position, from which it has been teleported. CameraTeleport can be used to quickly explore more distant lands, which would require long time to move, as well as checking how loading-unloading various objects works when switching between terrains.

4.2. RPG Camera

Various RTS games may have some content, which could be played by one character (i.e. heroes, kings, quest content, etc.). As a result, RTS Toolkit provides simple approach of how to switch between RTS and RPG camera both ways.

RPG mode is switched when one of the units (not building) is selected by clicking  button. If the button is clicked second time, camera switches back to RTS mode.

CameraSwitcher is the component, attached on main camera, which controls switches between RTS and RPG modes. The script saves RTS camera position and rotation before switching to RPG mode that when player switches back to RTS mode, camera would appear in the same position and rotation as it was before switching. When RTS mode is active, only RTSCamera component is enabled, which RPGCamera component is disabled. When RPG mode is enabled, only RPGCamera is enabled and RTSCamera disabled. This gives that only one camera mode input controls are running (RTS or RPG) and not the both simultaneously. CameraSwitcher component is enabled all the time in order to detect when player clicks the button to make a switch. The same game camera is used for both RTS and RPG modes.

RPGCamera controls game camera in RPG mode. Its components setting are shown in Fig. 60.

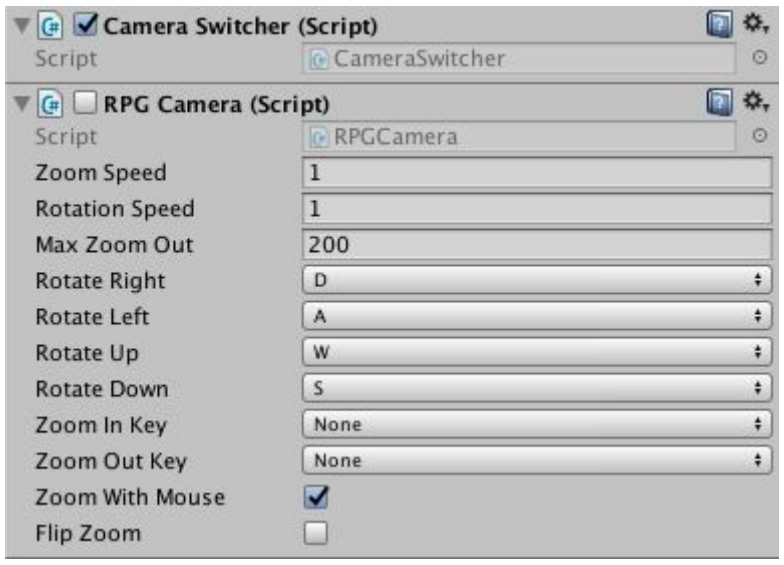


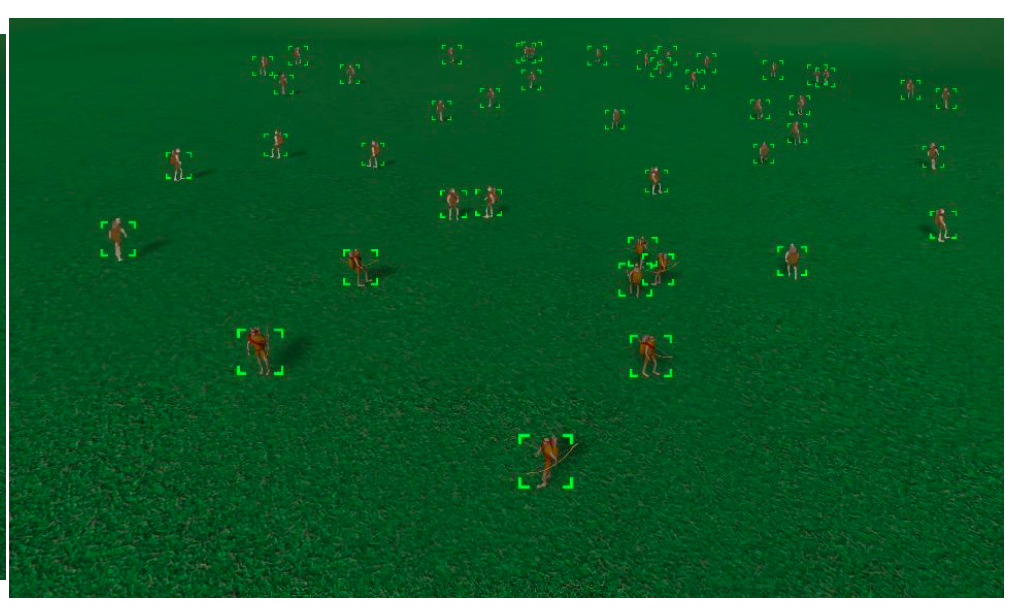
Fig. 60 - RPG camera settings.

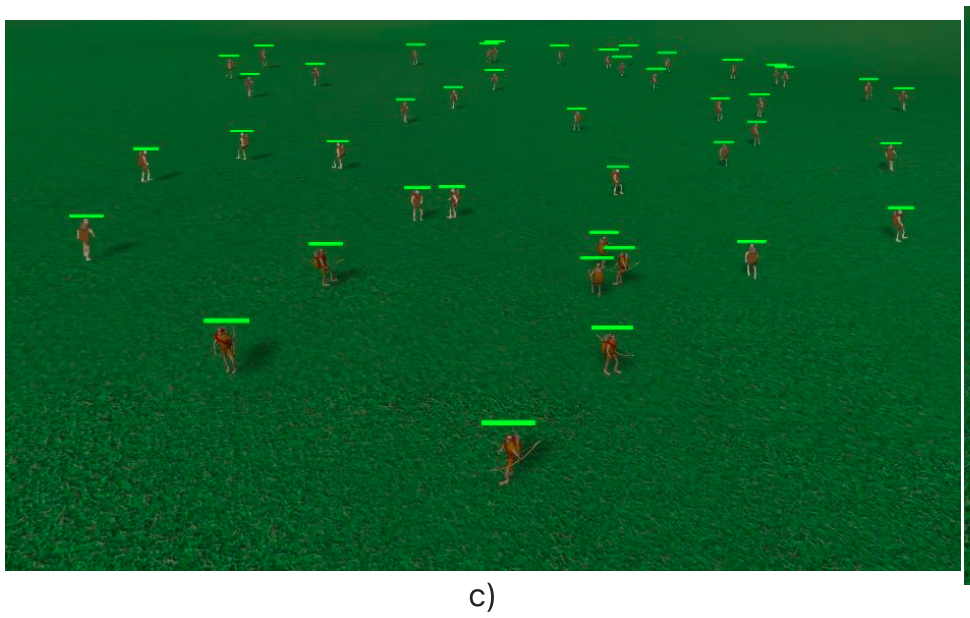
RPG Camera works in a similar way like RTS Camera, but it has no free movement over terrain and only moves together with unit, on which RPG camera is set. It also looks into this unit all the time. Player can zoom in between inner distance, set by unit NavMeshAgent radius and the outer distance set by **maxZoomOut** parameter. Player can also rotate around the unit by using **rotateRight**, **rotateLeft**, **rotateUp** and **rotateDown** keys. By default it is set in a similar way with "DAWS" like in RTS camera mode. Zoom keys and flipping can be set by using **zoomInKey**, **zoomOutKey** and **flipZoom** tick correspondingly. RPG camera allows nice and smooth movement when following units. It can be used to explore areas as well.

4.3. Unit selection

In RTS Toolkit all units can be selected, which are marked by selection marks and health bars. Unit selection can be done in two ways - clicking on the unit and dragging rectangle over the screen when all units within this rectangle are selected. Click method selects only one unit. Any unit can be selected by click. Rectangle method is done by clicking and holding left mouse button and dragging the mouse over the screen. Blurred rectangle appears in the area, where units will be selected. Finally all units are selected within rectangle when player releases left mouse button. Only player nation units (not building) can be selected with rectangle, while buildings and non-player nation units can be selected only one by one with a click.


a)


b)


c)

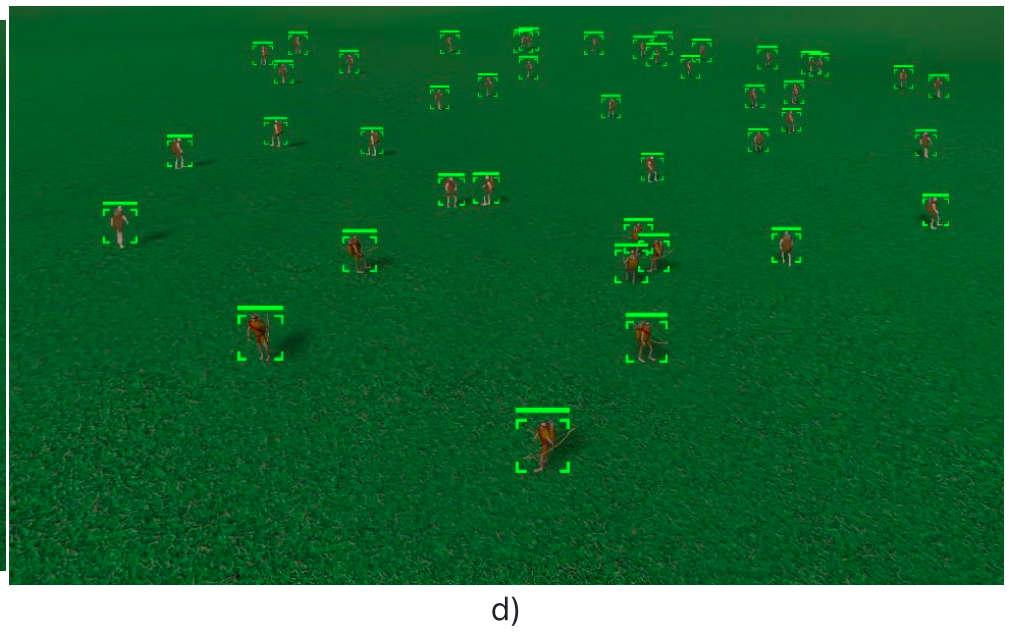
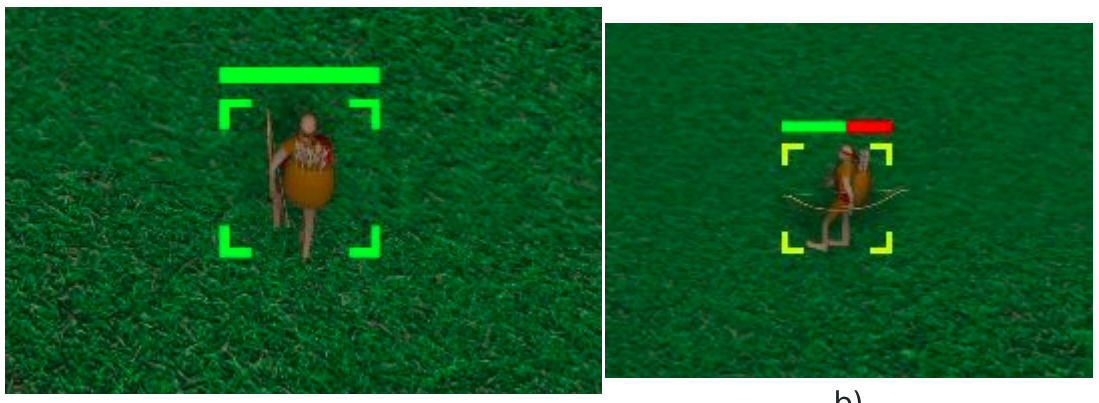

d)

Fig. 61 - Differences between selection marks and health bars. a) units are not selected or selection selection indicators are disabled; b) selected units marked with selection marks; c) selected units marked with health bars; d) selected units marked with selection marks and health bars together.

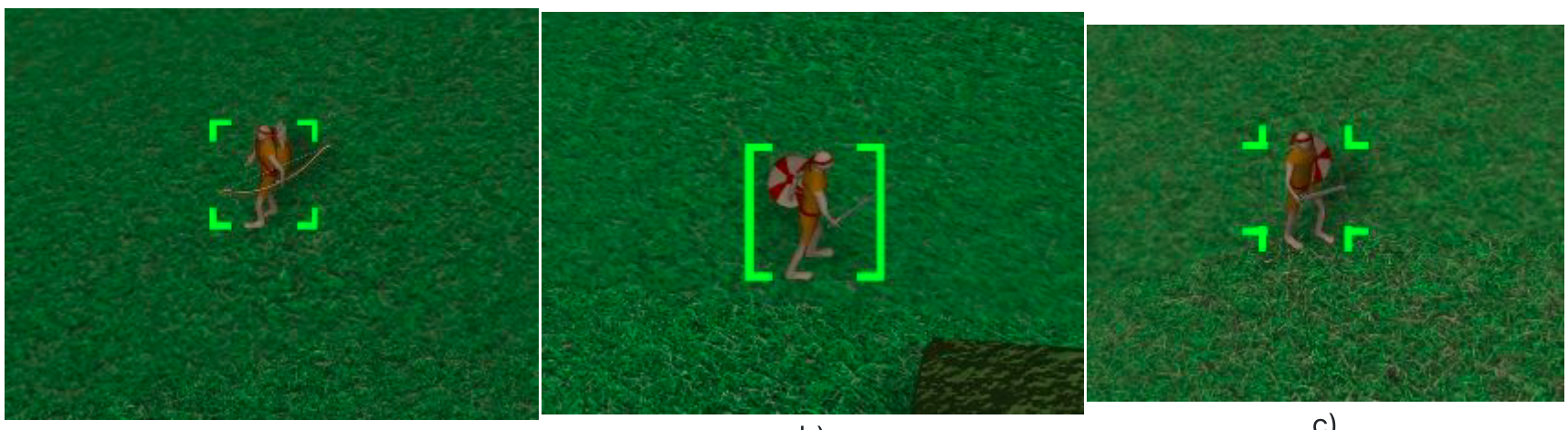
Fig. 61 shows the differences between selection marks and health bars. Selection marks appear as a corners around each unit, which indicates which units are selected. Health bars appears above each unit. They indicate how much health unit has left. When unit is damaged (injured) only part of health bar is green, while another part is red.



a) b)

Fig. 62 - Selection marks and health bars of healthy (a) and injured (b) units.

Fig. 62 shows the difference between healthy and injured units. As unit gets more damaged, its green part of health bar drops and red part increases. Selection mark is changing its colour based on how many health unit has left - healthy units are marked with green, slightly damaged with yellow, badly damaged with orange and almost dead ones with red selection marks.



a) b) c)

Fig. 63 - Selection marks for player nation unit (a); unit of another nation, which is not in a war with player nation (b); and unit of another nation, which is currently in a war with player nation (c).

Selection marks has one more distinct feature - their shape shows if unit belongs to a player nation (Fig. 63 (a)) or to another nation, which player does not control (Fig. 63 (b) and (c)). If unit does not belong to a player nation, then selection mark shows if the nation, to which selected unit belongs is in a war with player nation (c) or not (b).

Unit selection is controlled just by two scripts SelectionManager and SelectionMark. Both of them does not have parameters and should be used only once per scene.

SelectionManager is the one, which checks for input and detects selection. Its concept is relatively simple. When player clicks left mouse button on the screen, the ray is sent and all selectable units are being checked of what is the smallest distance between the ray and the unit pivot. If one of the units distance to the ray is closer than rEnclosed of the unit, then unit is selected. If no unit is selected and player holds left mouse button, rectangle mode is enabled. When in rectangle mode, selection rectangle is changing based on how player moves the mouse - on each update new rectangle is overwriting previous one. At the end, when player releases left mouse button all selectable gameObject positions are passed into screen coordinates by using **WorldToScreenPoint()** function. Units, with screen coordinates, which are in selection rectangle are finally selected. Deselection happens when player clicks LMB in empty place, where are no units. As each time before selection always runs deselection, all previously selected units are automatically deselected.

SelectionMark stores a list of all selected units (selectedGoPars), which is managed from SelectionManager. When unit is selected, then it is added to this list and removed when deselected. SelectionMark main functionality is to display selection marks and health bars. Both are displayed by using **GUI.DrawTexture()**. For selection marks 3 texture lists are used for all three Fig. 63 cases (player and non-player nation selection marks). Each list is filled with several different colour textures to represent differently damaged units. Health bars are also using list of baked textures, but it needs only one list, as all units has the same shape of health bar. Texture baking is done when the game starts. As baked textures are in the lists, they are fast to use, as only list indices are calculated on runtime. **OnGUI()** function draws all textures for selected units. For each unit in selectedGoPars list WorldToScreenPoint() function is used to find what are unit pivot positions in screen coordinates (note, that selectedGoPars list is usually shorter than allUnits list, as usually only some units are selected). These coordinates are used to center the texture on a unit. Texture size is calculated from the distance between camera and unit. Each unit can have one, two or no textures. If selection marks mode is enabled in Options menu, then just one of three textures is used for selection marks based on unitPars nation. If health bars mode is enabled, then health bars are displayed as well.

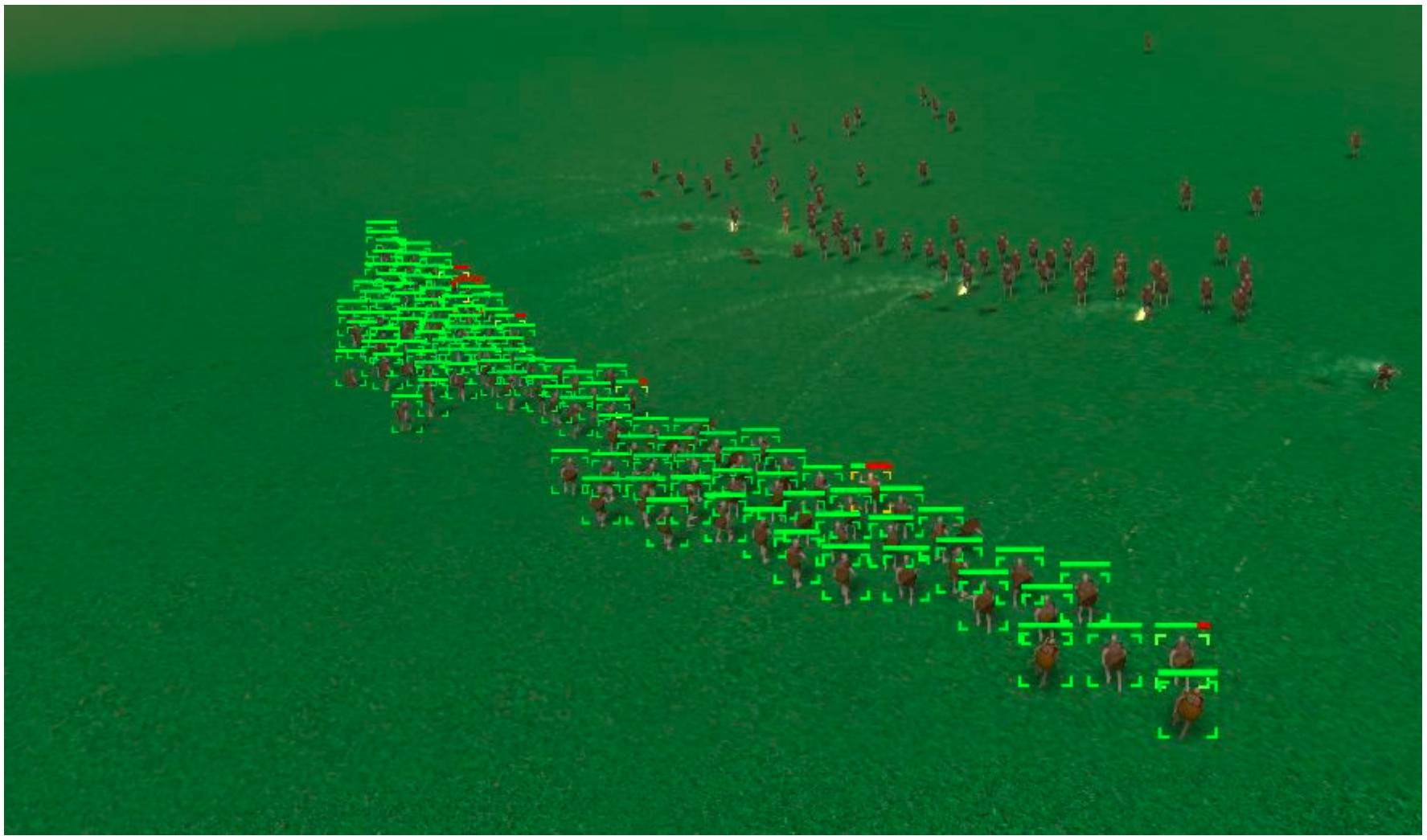


Fig. 64 - Multiple player units are selected and marked by selection marks and health bars in a war frontline.

This texture setup is optimised to use large number of selection marks and health bars (Fig. 64). Player can select hundreds or even thousands of units without big impact of performance, as textures are rendered through OnGUI, do not create any additional gameObjects in the scene and only array indices are being calculated.

4.3.1. Shuriken selection marks and health bars

RTS Toolkit has also implemented Shuriken based selection marks and health bars, where each unit is marked with a Shuriken particle. This method allows to have massive amount of selection marks and health bars with almost no impact on performance.

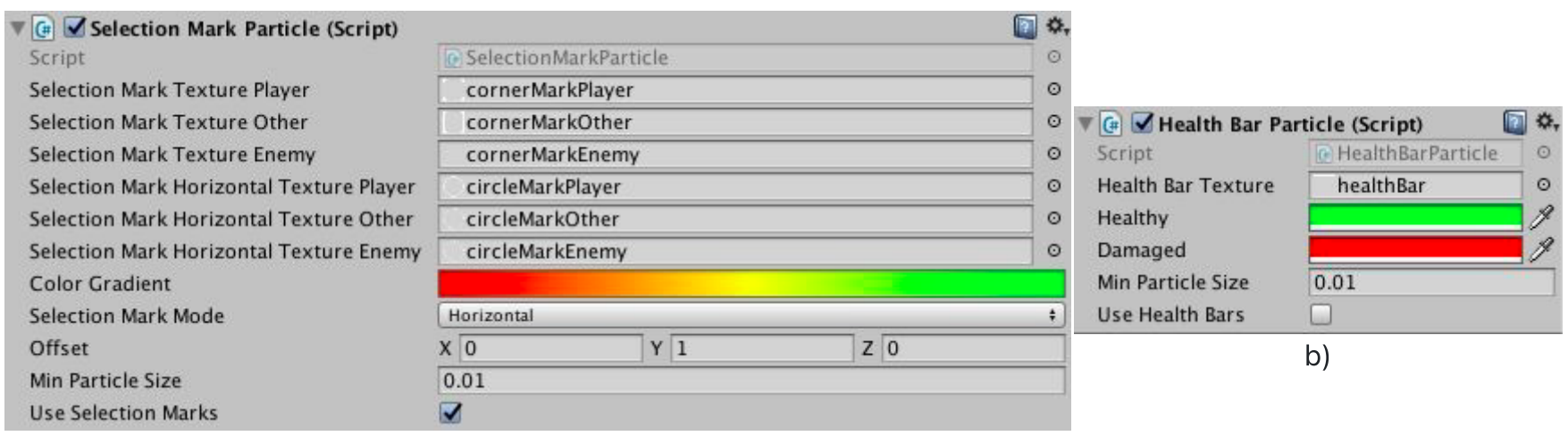


Fig. 65 - Shuriken based selection marks (a) and health bars (b) setup.

Fig. 65 show component setups for selection marks (a) and health bars (b). Selection marks can have 3 different textures marking units in the same way as it is done in Fig. 63. These textures are set through **selectionMarkTexturePlayer**, **selectionMarkTextureOther** and **selectionMarkTextureEnemy**. **colorGradient** allows to set the gradient for unit health to show how much health unit still has. **minParticleSize** defines how small particles can when camera is moving out from the player. If minParticleSize is reached, particle sizes will not scale down anymore when camera moves further away from the unit. useSelectionMarks allows to disable selection mark usage.

Selection marks also allows to set horizontal billboard based selection marks, i.e. when unit is selected by drawing a circle around it (Fig. 66). Textures for horizontal selection marks can be set via **selectionMarkHorizontalTexturePlayer**, **selectionMarkHorizontalTextureOther** and **selectionMarkHorizontalTextureEnemy**. Horizontal selection marks also accounts for a terrain slope, so that billboards would be always facing normal direction from the terrain, preventing some parts of the textures to be hidden. **selectionMarkMode** allows to chose between **CameraFacing** and **Horizontal** modes.

Fig. 66 - Units marked with Shuriken based selection marks and health bars.

Health bars are also based on Shuriken particle system (Fig. 65 b). But instead of just one particle, there are used 2 particles for each unit health bar: one particle is displaying the green part of the health bar and another the red part. Each particle positions and horizontal sizes are adjusted to represent the correct state of health bar. **healthBarTexture** is used to display health bar. Only the top part of the texture is coloured as white, while the rest of the texture is transparent. **healthy** and **damaged** colours allows to set colour of corresponding health bar parts. **minParticleSize** controls minimum size of the health bar, while **useHealthBars** allow us to enable or disable health bars usage on selected units.

4.3.2. Comparison between Shuriken, 4.6 UI and OnGUI models

These 3 models allows to chose which option could be the best to apply in the game. However, each of them has their own pros and cons.

OnGUI is the oldest model to mark selected units in RTS Toolkit and might be best fit for older or retro style games. It also allows to set nice looking textures with dedicated size in pixels to avoid blurred edges. However, if there are Unity 4.6 UI graphics elements in the game, OnGUI marks will be draw on the top of them all, leading that marks will be visible on menus, buttons, labels and other areas where they should not to appear. OnGUI also is memory-expensive, as for each color it bakes separate textures and stores them into lists on runtime.

4.6 UI is more like a standard way of marking units with 4.6 Image components. There is quite a lot of control for this approach. However, it creates large number of gameObjects and destroys them on deselection. This cause lags especially if there are large number of units in game being selected. 4.6 UI marks are slightly more performance friendly, but they also takes large amount of performance if many units are selected.

Shuriken is almost performance free method for selection marks and health bars - many thousands of units can be selected and deselected almost with no performance impact at all. However, horizontal selection marks can be not visible if units are behind trees or other objects. Camera facing selection marks and health bars can appear sometimes visible while other ways behind trees, as Shuriken particles are placed halfway between camera and selected unit.

4.4. Main GUI frame

The main GUI frame is stretched around the edges of the screen and contains various game menus and buttons. It is the main frame through which player interacts with the game most of the time. Main frame GUI contains top bar with player resources, bottom bar with info and unit costs, and right bar with options menu, buildings menu, diplomacy button and full screen button. Main frame is based on Unity's 4.6 GUI system (Fig. 67), where most of UI elements were manually designed by adjusting UI element boundaries and positioning anchors. They can be easily redesigned by simply re-adjusting UI elements and anchors and changing textures. Multiple short scripts, which are used to control UI elements can be found in Assets/RTSToolkit/Scripts/UI directory.

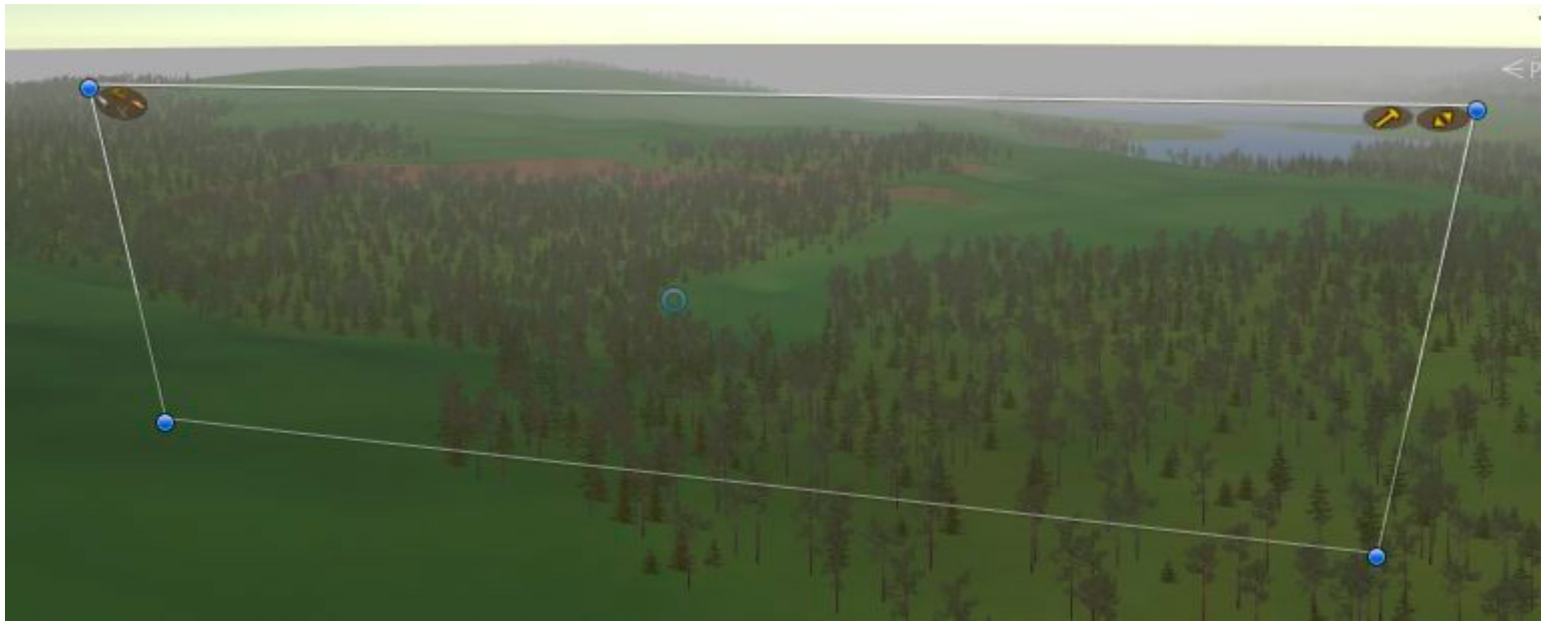
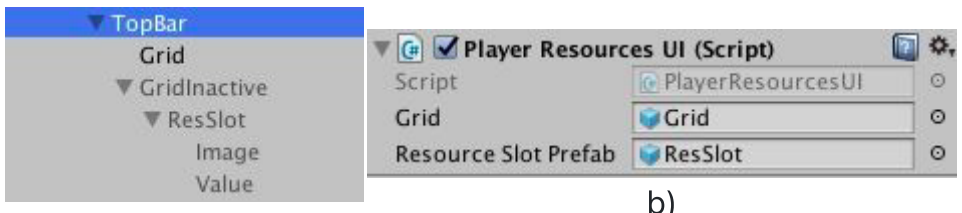


Fig. 67 - Main frame GUI with its elements.

4.4.1. Top bar and player resources

Top bar appears on the top side of the screen with 4 player resources: iron, gold, lumber and population. Top bar gameObject uses grid into which resource slots are being added (Fig. 68 a). “Grilnactiveˮ is the same grid as “Gridˮ but instead is set as inactive. The top bar has **PlayerResourcesUI** component attached onto it. This component has the Grid and resourceSlotPrefab gameObjects set (Fig. 68 b). From **Economy** number of resources is being used to instantiate resourceSlotPrefab and assign it to grid gameObject. This is how all 4 default resource labels are being placed on the top bar grid when game starts. As there is only one PlayerResourcesUI object in the scene, its singleton is used to update resource values from Economy script.

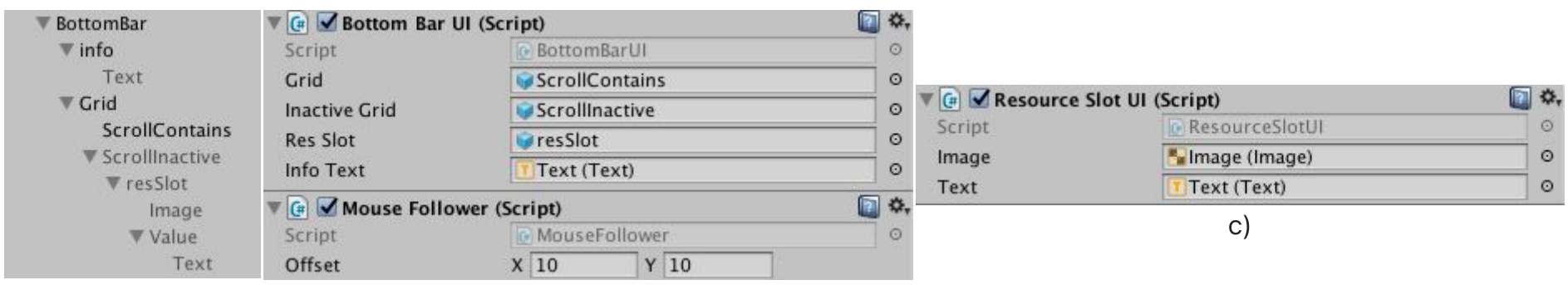


a) b)

Fig. 68 - Top bar structure (a) and PlayerResourcesUI component parameters (b).

4.4.2. Bottom bar and units info

Bottom bar appears at the pointer when player places it over the object of interest, and displays information about costs when player is choosing to build some units. There is object info text, which appears above the resources in the game. This object info is used to display unit or building name, which player want to build.



a) b) c)

Fig. 69 - Bottom bar hierarchy (a), BottomBarUI and MouseFollower components (b), and ResourceSlotUI component (c) setup.

Bottom bar setup is shown in Fig. 69. The grid is used in order to place all resource elements inside. The ScrollInactive contains a resSlot, which is instantiated and assigned as a child gameObject on ScrolContains when displaying costs.

BottomBarUI is the component, attached on the root BottomBar gameObject. References of active and inactive grids are set, which are used for setting parent when resSlot object is instantiated upon the displaying of costs. resSlot itself is a universal resource object, which is instantiated on a runtime when cursor is placed on the build button. The number of resSlot’s is the same as the number of defined resources in the UnitPars. When the cursor is moved away from the build button, costs are disappearing by destroying resSlot instances.

Each resSlot has **resourceSlotUI** component, through which the resource icon (image) and the amount (text) is shown.

Finally, bottom bar has assigned the **MouseFollower** component, which allows the UI to move with the cursor in a way that resources would appear just over the button of interest.

4.4.3. Building buttons grid

When player selects the building, building buttons grid is used to display repair and destroy buttons (Figure 70 (a)).

a) b)

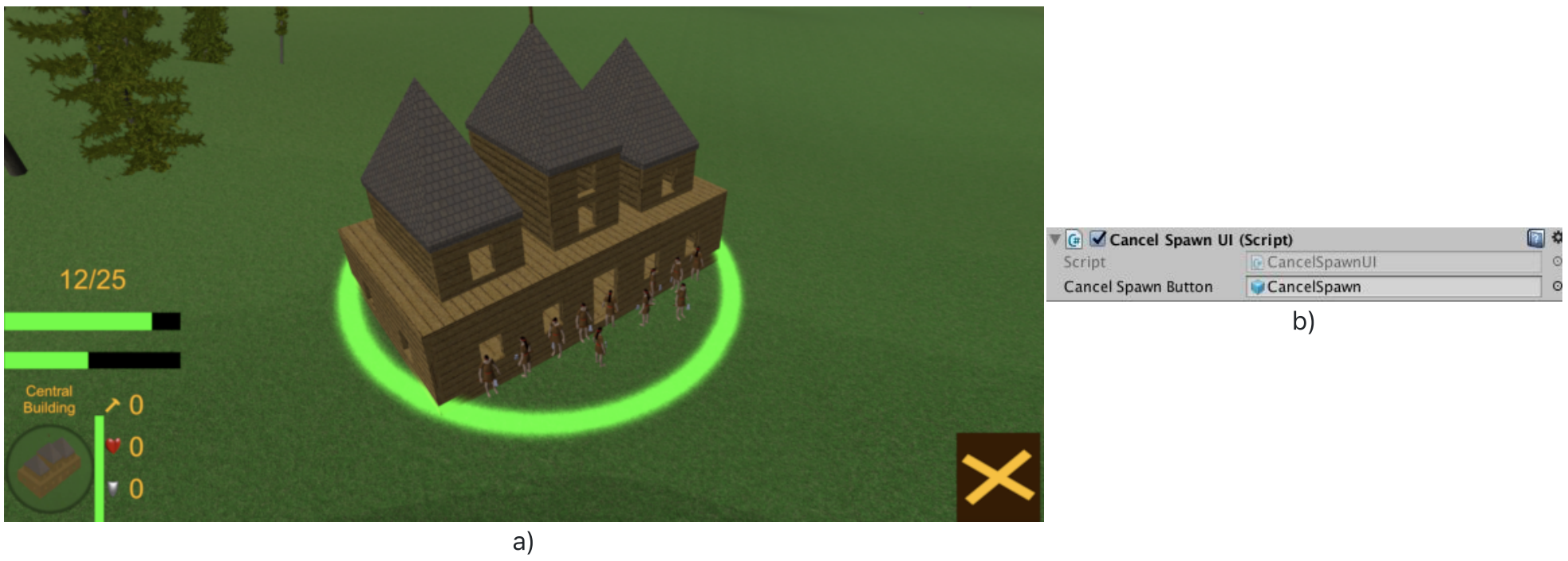
Fig. 70 - Building buttons grid with “repair” and “destroy” buttons activated in the grid near the minimap (a) and grid hierarchy (b).

The grid setup is shown in Fig. 70 (b). The first slot is a repair building button. This repair button is hidden if building has full health and becomes visible if building gets damaged. Repair button is the only way for player to restore buildings health as buildings do not self-heal. Restoring also consumes half of resources per health unit compared to the fresh build price. Repair button triggers SpawnGridUI function RestoreBuilding(), which itself triggers restoring coroutine in selected gameObject UnitPars to start restoration. Restoration rates are the same as fresh building rates.

The second slot has destruction button, which is used to immediately destroy selected building. These buttons only appear on player nation selected buildings and not on other nation buildings, as player can’t control another nation units and buildings.

4.4.4. Spawning building grid

Spawning building grid is the grid, which replaces building buttons grid when building is spawning units. Its setup is identical to building buttons grid, but it only has cancel button to stop spawning units. Fig. 71 (a) shows how selected building menu looks like when spawning units. Progress bars are visible in the bottom left side of the screen, just above the unit icon. Cancel spawn icon appears in the right bottom side of the screen, near the minimap. When cancel button is pressed, in **CancelSpawnUI** (attached to root gameObject of SpawningBuildingGrid, Fig. 71 (b)) CancelSpawn() function is called, which cancels unit spawning and switches back to building buttons grid mode.



a) b)

Fig. 71 - Spawn bars as it is shown in game when the Central Building is selected together with the cancel spawn button (a). CancelSpawnUI component is responsible to cancelling the spawning queue and hide/show the cancel spawn button (b).

4.4.5. Building icon

Building icon is visible in the bottom left corner of the game screen when building or unit is selected (Fig. 72 (a)). The icon also has building or unit name and its health bar.

a) b) c)

Fig. 72 - Building icon with the name of building/unit and its health bar (a), its gameObject hierarchy (b) and SelectedIconUI component (c).

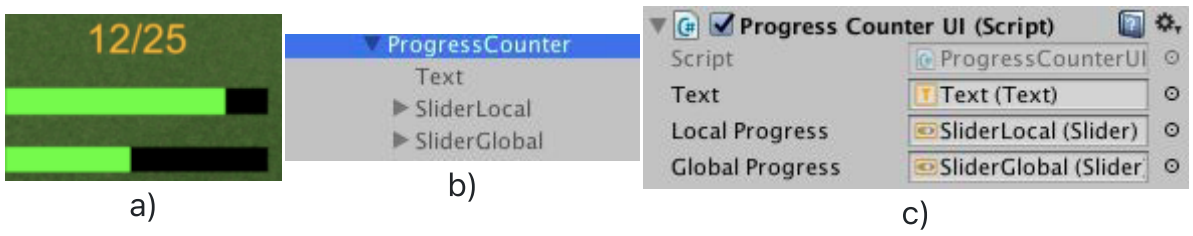
All three elements are represented by child gameObjects named as “Image”, “Text” and “HealthRed” (Fig. 72 (b)). **SelectedIconUI** component, attached on the root gameObject controls behaviour of these 3 elements. All units and buildings in game shares the same BuildingIcon, when only image, text and health bar values are changed when player selects different buildings or units. SelectedIconUI has these three references and directly modifies corresponding elements. Health bar uses two sliders - one is red and another is green. Green health bar is attached as a child on the red health bar. Red health bar has Slider component, which references green health bar RectTransform. As slider values are changed (between 0 and 1), different fractions puts slider at different positions. The ratio of health/maxHealth is used to set the health bar. SelectedIconUI enables all three components when unit or building is selected and disables them when it’s deselected.

4.4.6. Numbers and labels

These are similar UI elements like building buttons but they don’t have direct input from the player.

4.4.6.1. Progress counter

Progress counter shows the progress of the spawning queue when selected building spawns units. It appears just above the building icon (Fig. 73 (a)) and has 3 parts. The most upper one is text, which displays how many units are already spawned from the total amount to be spawned (i.e. in the example 5 units are already spawned out of 25). Below the text there are two bars - the top one shows the progress of the currently spawning unit, while the bottom one shows the overall (global) progress. Progress counter gameObject has all these three elements set on child gameObjects (Fig. 73 (b)). Both sliders are set in a similar way like the health bar slider for building icon. **ProgressCounterUI** is the component, responsible for management of text and sliders for both bars and can be found on root ProgressCounter gameObject.

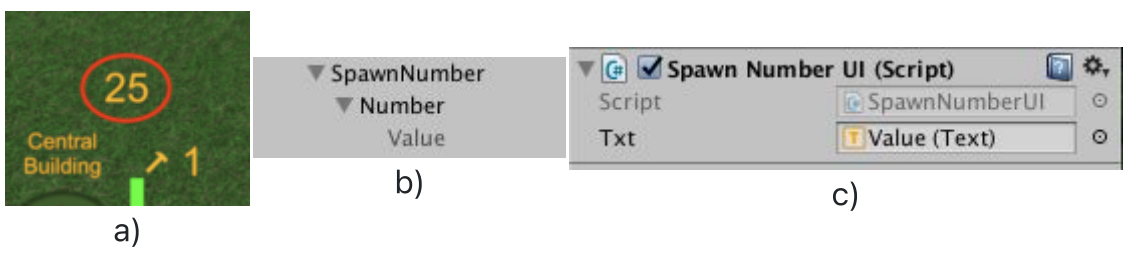


a) b) c)

Fig. 73 - Progress counter in the game (a), gameObject hierarchy (b) and ProgressCounterUI component (c).

4.4.6.2. Spawn number

Spawn number is the number to display of how many units player want to spawn. It appears as a single number just above the building icon (Fig. 74 (a)). Spawn number is used only when spawning units, not creating buildings. The gameObject of spawn number is relatively simple (Fig. 74 (b)), where is used only text component in order to display the number. **SpawnNumberUI** is the script, used to control the text, visible for spawn number (Fig. 74 (c)). When player clicks on unit, spawn number appears and by scrolling mouse wheel the number of units to spawn can be changed. When player adjust the number, it clicks second time on the unit icon, spawn number disappears and units start spawning.



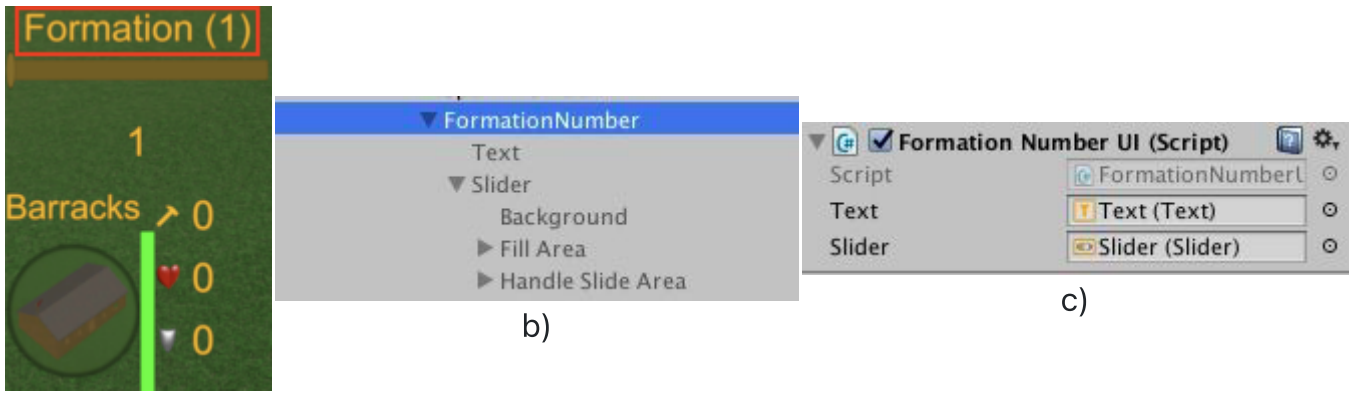
a) b) c)

Fig. 74 - Spawn number used to set the number of units to spawn.

4.4.6.3. Formation number

Formation number is used to put spawned units directly on the formation. By default it is set to 1, what means that each unit is spawned individually. Formation number appears only on units which can be added to the formation (military units), while some other units, like workers can’t be added to a formation and only has spawn number available before spawning (Fig. 75 (a)). So it has two elements - the slider and text above the slider. Formations up to 20 units in a formation can be created. When player adjusts formation number, it also changes spawn number in order to reproduce fully filled formations (i.e. if formation number is set to 20, player can spawn units in 20, 40, 60, 80...). When player starts to spawn units, they won’t spawn individually, but would be spawned all together. Formation spawning rates are defined by using individual unit spawning rates and scaling them to the formation size. For example if it takes 1

second to spawn one unit and player spawns 20 units, it takes 20 seconds to spawn them all but each unit appears outside the building every 1 second. If player spawns these 20 units with formation size of 10, then 2 formations will be created. All 10 units from the first formation will appear after first 10 seconds and another 10 units from the second formation will appear in the next 10 seconds. As a result, in both ways, player can spawn all these 20 units in 20 seconds.



a) b) c)

Fig. 75 - Formation number with its slider in game (a), its gameObject hierarchy (b) and FormationNumberUI script (c), which manages text and slider.

Formation gameObject (Fig. 75 (b)) has corresponding text and slider children gameObjects. The slider gameObject here is designed by using default Unity slider template. **FormationNumberUI** is the script, which controls the slider and updates text values.

4.4.6.4. Mining Point label

Mining Point label is used to display how many iron or gold is remaining in the Mining Point. This info is shown just above the building icon when Mining Point is selected (Fig. 76 (a)).

a) b) c)

Fig. 76 - Selected Mining Point with visible Mining Point label (a) telling that in the Mining Point is 11794 gold available. Mining Point label gameObject hierarchy is shown in (b) and MiningPointLabelUI component in (c).

Mining Point label contains two elements - the icon and value, which are represented by two gameObjects (Fig. 76 (b)). MiningPointLabelUI script controls these values (Fig. 76 (c)). As label is shared between any Mining Points, its icon can be changing between iron and gold. For this reason iron and gold sprite images are referenced as last two parameters in MiningPointLabelUI. Activate() function is triggered from SelectionManager, which also sends resource type and amount from selected UnitPars. Based on that, icon is updated and the value is set on the text.

4.4.7. Unit buttons

Unit buttons show available options for selected units. There are slightly different menus for military units (Fig. 77 (a)) empty workers (Fig. 77 (b)) and loaded workers (Fig. 77 (c)). Differences arise as workers have the option to chop lumber and collect resources from Mining Points, while military units can't. When workers are loaded with a resource, they can't attack and the attack button is hidden, while the new return resources button is shown.

Fig. 77 - Unit buttons for military unit (a), empty worker (b), loaded worker (c), gameObject hierarchy (d), and UnitUI script setup (e).

unitButtons gameObject has the grid with all buttons for all units inside (Fig. 77 (d)). When the unit or a group of units is selected, **UnitUI** script (Fig. 77 (e)) knows what type of units are selected and displays corresponding controls for them. UnitUI script has all individual button gameObjects specified within its public variables and as a result it knows what to do with them.

Each button has assigned OnClick events, which calls corresponding functions in UnitUI, i.e. the first Group gameObject has Button component attached and OnClick calls UnitUI.Group() function.

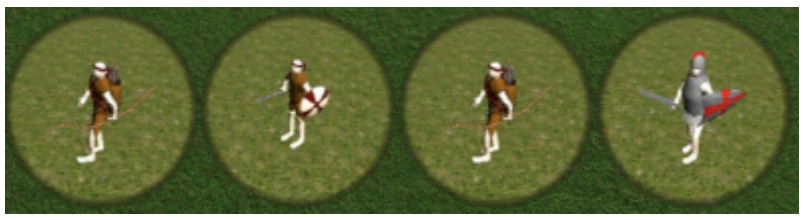
The most distant in the right button is Group, which allows player to put selected units on a group. If formation mode is enabled, then selected units are added to a new formation. “RPG” button switches camera to RPG mode by centering it on a selected unit. Shield button (3rd slot from the right) stops unit from movement or attack. 4th button from the right (arrows) is used to set unit to move. When button is clicked, cursorMode in SelectionManager is changed to 1 and points arrows appear for player to chose a location where to move units. When clicked second time, units starts to move. 5th button from the right is attack button. The attack works in a similar double mode way like move button, except that this time player has to click on a target rather on the empty place. If player clicks on empty place, nothing happens. On attack mode, the game keeps checking if cursor is over the unit and if so, the cursor changes colour. Player can always exit from 2nd mode by right clicking anywhere on the screen.

6th button is for workers only and is used to send workers to the Mining Point to collect resources. It also has this double mode way and changes colour when cursor is over the mouse, showing that worker can go there to collect resources. Similarly 8th button is used to collect lumber and while in 2nd click mode, it appears as active only when cursor is over the tree. These double mode button (4, 5, 6 and 7th buttons) actions can be shortened by right clicking on a corresponding object when unit(s) is (are) selected. The 7th and 9th buttons are hidden when the worker has no carried resources. Once the worker is loaded with Mining Point resources or lumber, standard collect buttons are hidden and return resource button is shown. As in such conditions when carried a bunch of resources worker can’t fight, the attack button is also hidden until worker deposit carried resources. After that the worker gets back to the regular round and has collect and attack buttons are displayed again but return resource button is hidden.

4.4.8. Spawn line

Spawn line (building menu) is the line of buildings and units available to spawn. It appears at the bottom side of the screen when player selects the building.

a)



b)

c)

d)

Fig. 78 - Spawn lines for Central Building (a), Barracks (b), Factory (c) and Stable (d).

By default there are 4 buildings which has spawn lines available (Fig. 78): Central Building, Barracks, Factory and Stable. Central Building allows player to create Barracks, Wood Cutter, House, Research Center, Mining Point, Factory, Stable, Windmill and Worker (Fig. 78 (a)). Barracks are used to create military units: Archer, Swordsman, Arsonist and Knight (Fig. 78 (b)). Factory creates Fence, Tower, Catapult and Ballista (Fig. 78 (c)). And Stable is used to spawn horseman (Fig. 78 (c)).

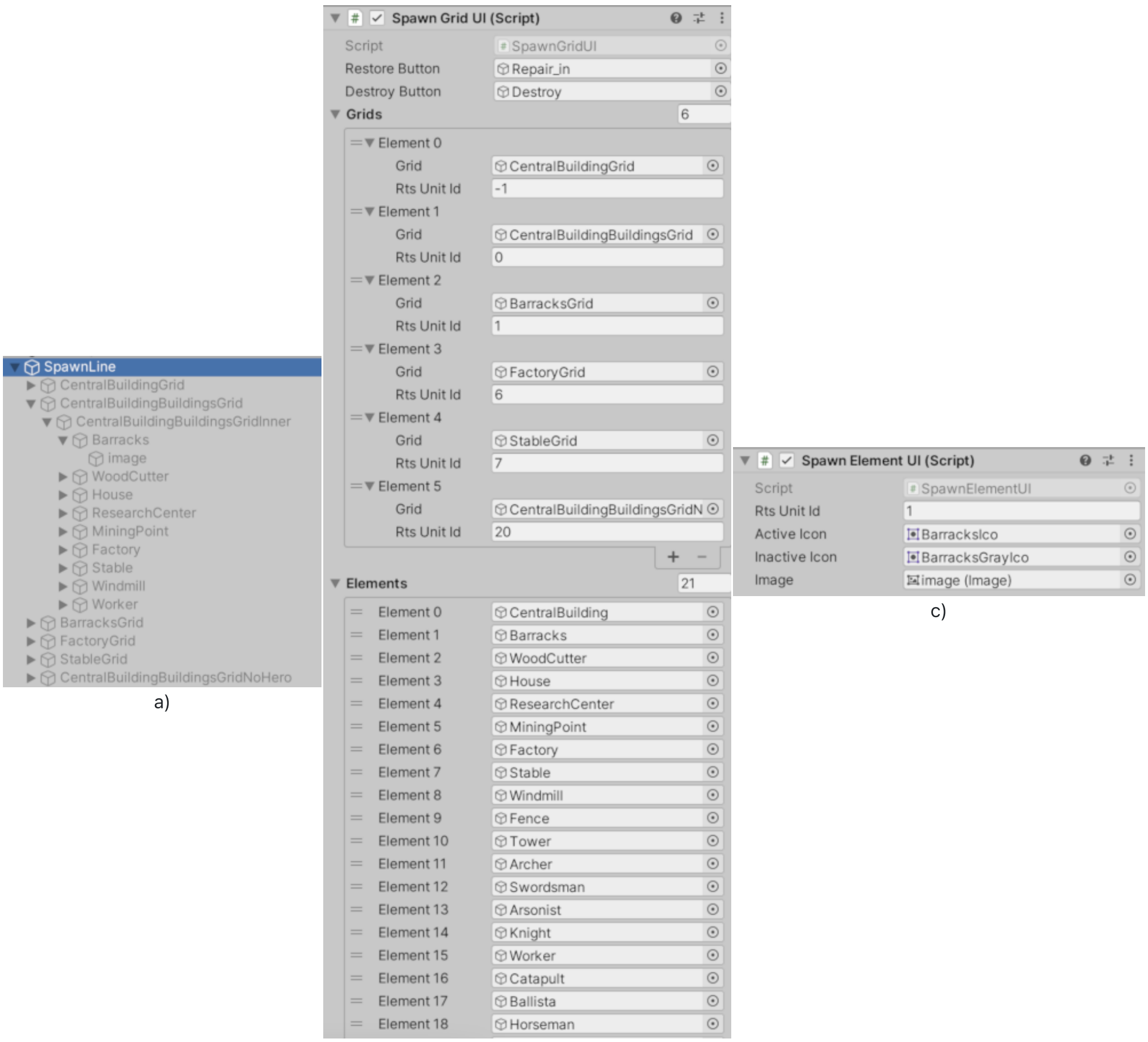


Fig. 79 - Spawn line gameObject hierarchy (a), SpawnGridUI setup (b) and SpawnElementUI attached on each spawnable element (c).

All 4 buildings with spawning lines have their child game objects set (Fig. 79 (a)). Root gameObject has **SpawnGridUI** component attached, which controls all spawn lines (Fig. 79 (b)). Each spawn element on spawn lines has **SpawnElementUI** component attached, which store rtsUnitId used to identify which RTS Toolkit unit the element represents (Fig. 79 (c)). SpawnGridUI has a list of **grids** references to all 4 buildings with rtsUnitId to know to which building corresponding spawn line belongs. If spawn lines are not open, their gameObjects are deactivated. When player opens one of the spawn lines, the corresponding grid gameObject set in grids list is activated and the spawn line appears. Another list is **elements**, which lists individual spawnable elements from each grid. These elements are used for unlocking new buildings and units as player progresses in the game. For example, by default Arsonist element is disabled in Barracks grid. When player opens up barracks spawn line, it does not see arsonists available to spawn. After some time by spawning many other units, player unlocks arsonist units. The unlock is happening through elements list, where elements are added in a way that element index in this list would match with corresponding rtsUnitId. So, from anywhere in other scripts can be triggered to unlock new unit by simply activating gameObject in the elements list, which can be accessed through SpawnGridUI.active singleton.

SpawnElementUI is used to trigger the spawn of the corresponding unit or building. There are set EventTrigger components on each gameObject, holding SpawnElementUI. This allows us to trace when the mouse is placed over the button. In such case, it displays units or building name and its cost in the cursor popup menu. When button is clicked, TriggerSpawn() function is called, which uses rtsUnitId to set build mode, when player can choose where on the terrain to place the building. When building is deselected, the spawn line is closed. When spawning unit, it first time enters scroll mode, then player can set number of units to spawn and the size of formation, and on a second click start spawning is triggered.

4.4.9. Active screen

Active screen is a simple panel, which detects if player's mouse is currently on the main game screen and not over other UI elements. It appears on the main area, where player see the gameplay (Fig. 80).

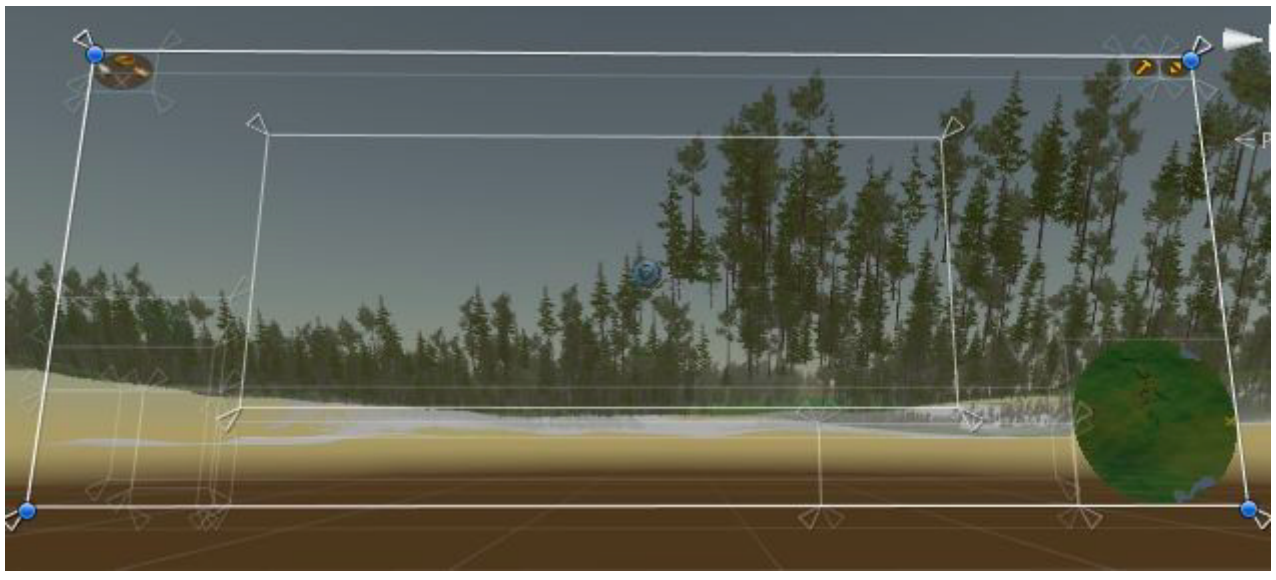


Fig. 80 - The active screen position as it appears in the editor.

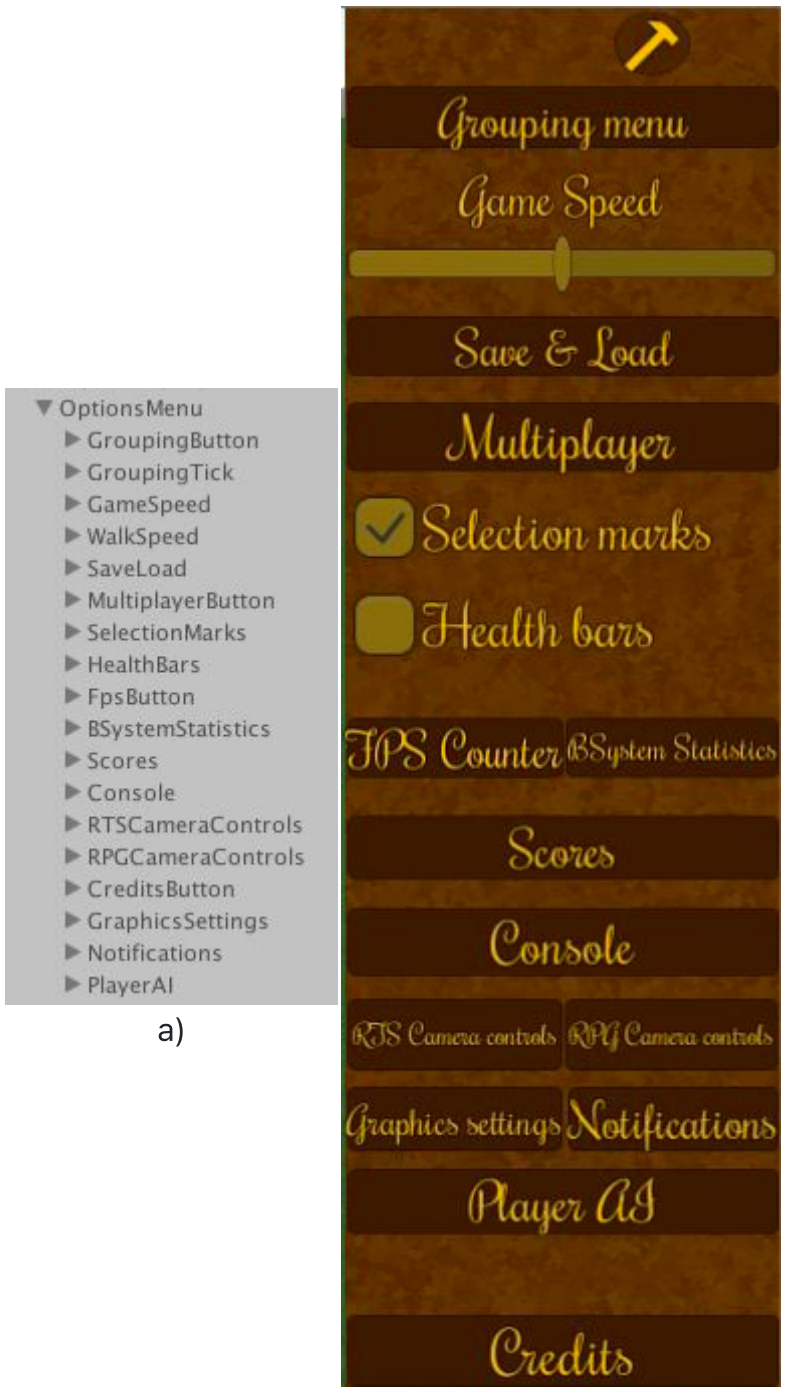
Active screen has **ActiveScreenPEC** component attached on its root gameObject. There are also EventTrigger component set with Pointer Enter and Pointer Exit events. They are used to detect when player enters or exits the active screen area. As other GUI elements appears on the top of active screen area, mouse is identified as being outside of the active screen when it is actually on another GUI element, which overlays on the top of active screen. EventTrigger calls PointerEnte() and PointerExit() functions in ActiveScreenPEC, which itself triggers ActiveScreenTrue() or ActiveScreenFalse() in SelectionManager, and sets bool isMouseOnActiveScreen values there. If isMouseOnActiveScreen is false then no mouse input is traced in SelectionManager, as mouse is not on active screen and player is doing something else than operations on active screen. This allows to avoid any conflicts for input detected through active screen.

4.4.10. Full screen button

Full screen button can be found in the top right corner of the game screen. The button can be recognized as  icon. When player presses this button, game enters the full screen mode. If the game is already in the full screen mode, then button exits the full screen. The button can be used for web games, i.e. WebGL or WebPlayer, where player by default see the game in browser window frame. The main FullScreenButton gameObject has **FullScreen** component attached, which contains SwitchFullScreen() function. That function is triggered OnClick and Screen.fullScreen = !Screen.fullScreen trick is used to switch between full and non-full screen modes.

4.4.11. Options menu

Options menu allows player to access various more advanced game settings and controls. Player can open options menu by using options button. The menu allows player to access grouping menu, to change game speed, to open save and load menu, set multiplayer, switch selection marks and health bars, turn on FPS counter and BattleSystem statistics, scores, access controls, player AI and credits window (see Figure 4.24).



a) b)

Fig. 81 - Child gameObjects of options menu (a), which shows sub-menus and other options available for player (b).

When options menu is activated, it overlays on all elements in the right hand side of the screen, including the minimap. When options menu is turned off, the minimap and other UI elements behind re-appears.

4.4.11.1. Options button

Options button is visible at the upper right side of the screen with  icon. The button is used to open options menu and to close it.

Options button gameObject has a basic setup with only one child gameObject which has Image component on itself to show the background and another image component on one child gameObject to show the hammer icon. On the root gameObject Toggle component is attached, which uses two phases: IsOn is set to true or false. OnValueChanged has added OptionsMenu gameObject reference with GameObject.SetActive function. This gives that when player presses the button, it sets IsOn to true and enables options menu gameObject. If IsOn is true, then it sets it to false and closes menu. So when player keeps clicking on the button it works like a flipping button, which enables or disables options menu. Options button is not a child gameObject of OptionsMenu, as the button itself is always enabled, while the menu is not.

4.4.11.2. Grouping button and menu

Grouping menu is used in game to group player's units, to put them on the formation, or to start the journey. By default when player starts the game, grouping menu is disabled, but it can be easily enabled by clicking Options > Grouping menu button while in game. Grouping button and menu are separate gameObjects in a similar way like Options button and menu are: grouping button only enables grouping menu, while grouping menu itself is not a child gameObject of OptionsMenu.

Grouping button setup is very simple. It's just a single gameObject with one child gameObject, which has text component. Root GroupingButton gameObject has OnClick() action to enable GroupingMenu gameObject.

a) b) c)

Fig. 82 - Grouping menu appearance in game (a), its gameObject setup (b) and GroupingMenuUI component attached to the root gameObject (c).

Grouping menu appears in the left hand side of the game screen (Figure 4.25 (a)). At the top-right corner of the menu is close button, which hides the menu. Below the close button is the list of all player's created groups, formations and journeys. If there are no groups, then the list is empty. At the bottom of grouping menu are two ticks: one for formation and another for journeys mode. There is also the Clean button. If formation mode is turned on, when creating new groups, player also puts units on a formation. If journey mode is enabled, when grouping, Journey window opens to plan the journey. As groups, formations and journeys appears in the list of the grouping menu, they are also different colour - groups are marked as grey, formations as brown and journeys as green. In the example of Figure 4.25 (a) there are set three groups: "1" represents a group of units, "2" units are added to the formation, "3" units are on the journey. Clean button allows to clean all existing groups in the list.

GroupingMenu gameObject contains 5 child gameObjects for Clean button, formation mode tick, journey mode tick, groups grid (list), and close button. It also has **GroupingMenuUI** component attached to it. Clean button gameObject has a simple button with OnClick() action registered to call Clean() function in GroupingMenuUI script. Once clicked, it destroys grid elements, representing the list of groups and formations and sets all grouped units as ungrouped.

FMode tick gameObject is a simple tick with Toggle component and the text. When the tick is changed, it changes isOn values in Toggle component. As this Toggle is referenced on GroupingMenuUI as **formationToggle** variable, toggle is directly accessed from GroupingMenuUI, which allows to check is formation mode is enabled or disabled. In the same setup is implemented the tick for Journey Mode and **journeyToggle**.

GroupsGrid is a standard grid, where new grid elements can be added. GroupingMenuUI has **cellPrefab** being set, which is instantiated and the instance gameObject is set as a child on a **grid** gameObject.

The prefab used to represent group buttons in the list can be found as GroupsGridCell set on the grid as inactive element. The cell prefab gameObject has just a button and child gameObject with Text component, where is displayed the group index. There is also attached **SelectGroupActionUI** component on the root gameObject (Fig. 83), which has SelectGroup() function. This function is triggered when cell button is pressed and selects group units through **UnitsGrouping**. SelectGroupActionUI also has SetFormationColor() and SetJourneyColor() functions, which sets colours of the button specified through **nonFormationColor**, **formationColor** and **journeyColor** via Editor.

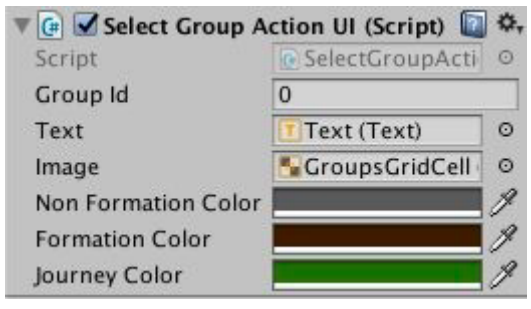
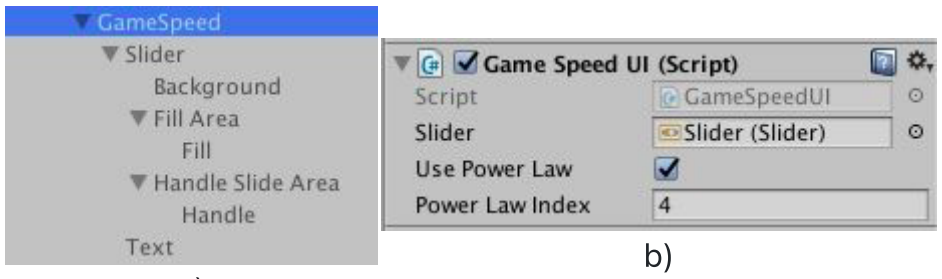


Fig. 83 - SelectGroupActionUI component attached on GroupGridCell prefab.

Close button is a regular button with one child gameObject, which has set Image component with cross icon. Button has OnClick() action, which disables the main GroupingMenu gameObject or in other words, closes grouping menu.

4.4.11.3. Game speed

Game speed can be adjusted by using game speed slider. The slider can be found between “Grouping menu” and “Save & Load” buttons. Sliding to left gives lower game speed, while sliding right increases the game speed. The default values are between 0 and 2, but can be set to other values on Slider component. Here 0 means that game is paused and 2 that game is running two times faster than on a regular rate.



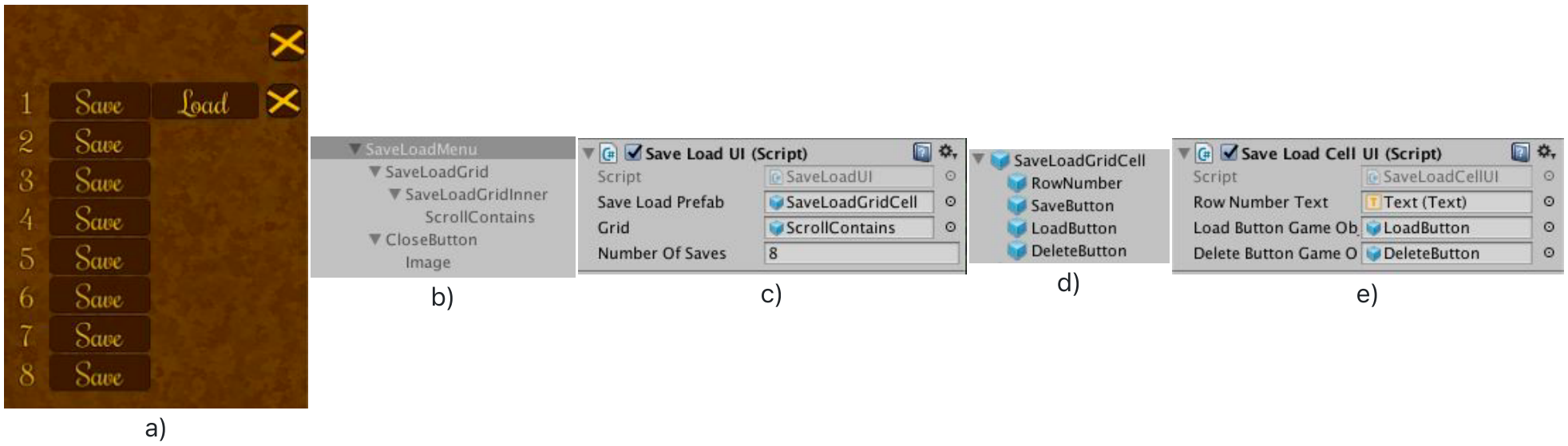
a) b)

Fig. 84 - Game speed gameObject setup (a) and GameSpeedUI component.

GameSpeed gameObject setup (Fig. 84) is simple - it contains default Unity’s slider setup with different colours (to match medieval style) and the Text gameObject, which displays “Game Speed” label just above the slider. Root gameObject has **GameSpeedUI** component attached, which has slider reference attached to it. Slider itself has OnValueChanged action registered to call ChangeGameSpeed() function in GameSpeedUI. As the function can’t recognize from where the call is coming, it uses Slider reference to check what are the new values and adjusts game speed via **Time.timeScale**. Developer can also select usePowerLaw and specify powerLawIndex. In this case, instead of actual values on slider, there is used the law slider.value^powerLawIndex in order to calculate the time scale.

4.4.11.4. Save and load button and menu

Save and load menu is used to save and load the game state of player’s progress. In game it appears in the place of Options menu after the Save & Load button is pressed (Fig. 85 (a)). By default it has 8 slots for games to be saved. If game is not saved in the slot, only save button is visible. If game is already saved in the slot, there are also Load and X buttons to the right of Save button. Save button allows to save a new game. If slot is already taken, pressing Save button will overwrite previously saved game. Load button allows us to load game. And X button clears the slot. The X button is visible above all slots in the top right corner of the menu and allows us to close save and load menu, and return back to Options menu.



a) b) c) d) e)

Fig. 85 - Save and load menu appearance in the game (a), it’s gameObject setup (b), SaveLoadUI component (c), SaveLoadGridCell prefab gameObject (d) and SaveLoadCellUI component (e).

The setup of save and load button and menu is similar to grouping button and menu setup. **Save & Load button** is visible in Options menu just below Game Speed slider and above Multiplayer button, and allows player to open save and load menu. The button setup is identical to the grouping button and enables SaveLoadMenu gameObject, which is not a part of Options menu gameObject.

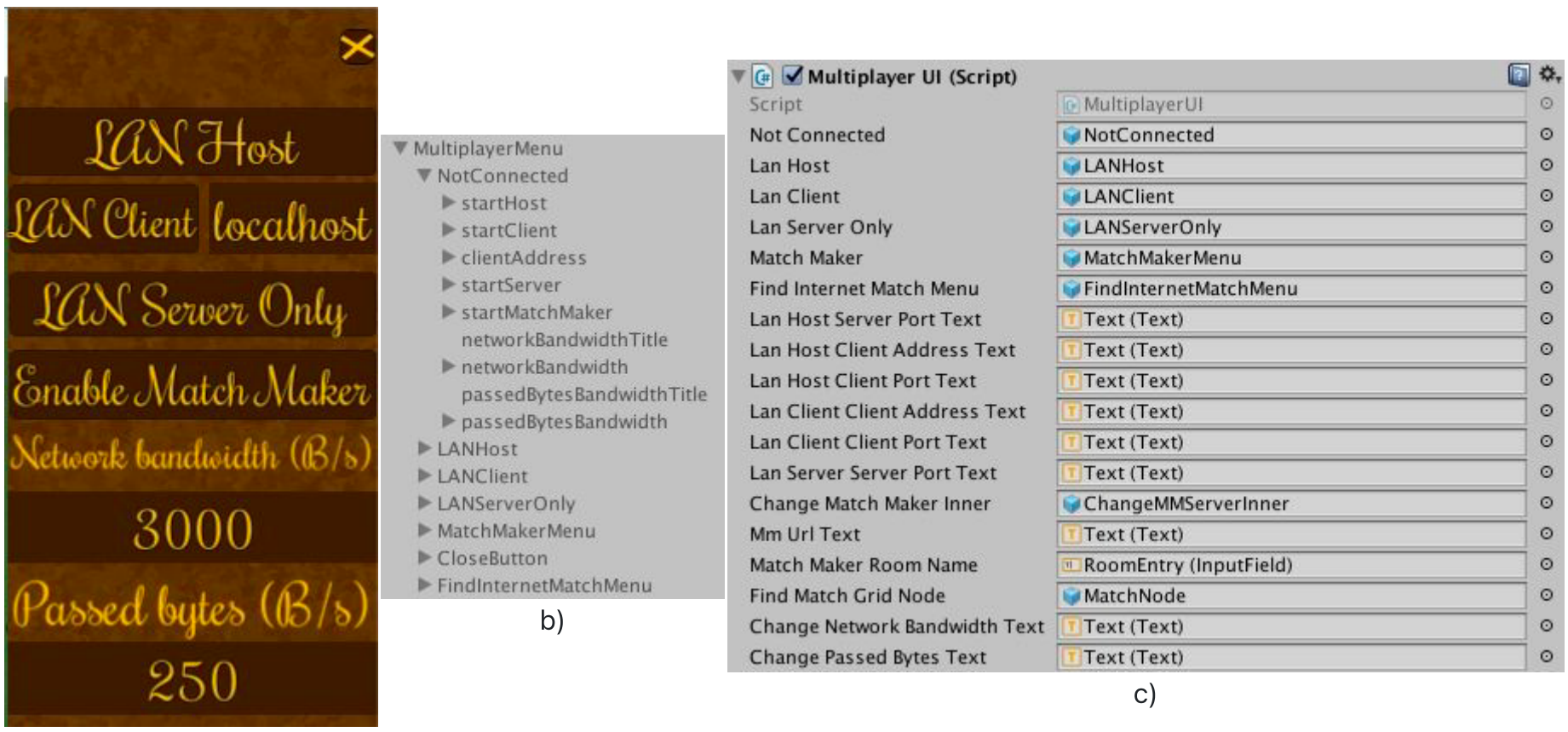
Save and load menu has relatively simple setup - its main element is the grid, which holds slot elements in the list and the close button (Fig. 85 (b)). The root gameObject has **SaveLoadUI** component attached (Fig. 85 (c)). SaveLoadUI has **saveLoadPrefab** reference which is instantiated and instances are added as child gameObjects to **grid** gameObject. **numberOfSaves** is used to determine how many save and load slots are available for player.

saveLoadPrefab in SaveLoadUI by default has a reference to SaveLoadGridCell prefab, which is set as inactive grid element. The prefab gameObject has 4 child gameObjects (Fig. 85 (d)). RowNumber is a simple label to hold the id of the slot. SaveButton, LoadButton and DeleteButton has Button components with OnClick actions to call Save(), Load() and Delete() functions in SaveLoadCellUI respectively. **SaveLoadCellUI** itself (Fig. 85 (e)) has loadButtonGameObject and deleteButtonGameObject references, which are used to disable these two buttons when slot has no saved state. This is checked by using File.Exists() function inside Application.persistentDataPath directory, where files are saved as 1.sav, 2.sav ..., and number is assigned from slot id.

Each time player opens save and load menu, Refresh() function is called in SaveLoadUI, which reloads all slot instances and cleans old ones. So if developer changes numberOfSaves, it always applies automatically when player opens save and load menu. Close button in SaveLoadMenu works identically to the one in grouping menu and simply disables SaveLoadMenu gameObject.

4.4.11.5. Multiplayer button and menu

Multiplayer menu allows player to enter the game in multiplayer mode. Multiplayer menu is opened by pressing multiplayer button, which can be found below “Save & Load” button and above “Selection Marks” tick. Once pressed, it opens a multiplayer menu which overlays and hides Options menu (Fig. 86 (a)).



a) b) c)

Fig. 86 - Multiplayer menu as seen in game (a), its gameObject setup (b) and MultiplayerUI component (c).

As it can be seen from Fig. 86, multiplayer menu is set in the similar way like Unity’s NetworkManagerHUD component. The difference is that in RTS Toolkit it is based on Unity’s 4.6 new GUI system, while NetworkManagerHUD is based on the older system. The new GUI system allowed to easily adopt the layout to RTS Toolkit medieval style. It’s also more flexible as designer can easily adjust various elements if needed. Various functions were also based on new UI source code https://bitbucket.org/Unity-Technologies/networking with main usage of Runtime/ NetworkManagerHUD.cs script. In other words, RTS Toolkit multiplayer menu is a remade version of NetworkManagerHUD into modern Unity 4.6 GUI style.

Multiplayer menu is opened and closed through multiplayer button in the same way as Save and Load menu. However, the menu itself is quite different - it has no grids or list, but instead use manually set buttons and switches between states. There are several phases of multiplayer, which includes not connected, connected through local multiplayer as LAN Host, LAN Client or LAN Server only, and connected through matchmaker. When on different phases, player see different menus with different buttons (Fig. 87).

a) b) c) d) e)

Fig. 87 - Different phases of multiplayer menu: not connected (a), connected as host (b), connected as client (c), connected as server only (d), matchmaker menu (e).

For such reason, child gameObjects for MultiplayerMenu are set in a similar way (Fig. 86 (b)) in order to easier flip between different multiplayer states by enabling or disabling one or another child gameObject. **MultiplayerUI** component, attached to the root gameObject stores these references for different phases in order to enable or disable particular gameObjects. It also has functions, which are triggered from clicking on particular buttons to start changing the phase. However, the phase in multiplayer is changed not immediately, but after some time. For example the phase from not connected to connected as host is changed at the end of the process, when player starts by clicking “LAN Host” button, it attempts to connect as a host and when finally connected, the phase is

changed. This is when menu for player would change from Fig. 87 (a) to (b). So when LAN Host button is pressed, RTSNetworkManager is being used to call StartHost() function. When it connects, OnStartHost() function automatically triggers. This function is overridden, and switches to the menu, which represents connected as a host.

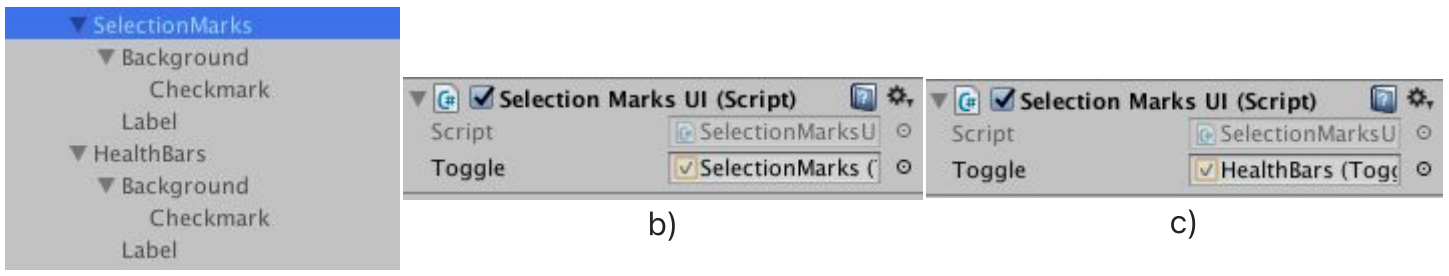
Not connected menu has 4 buttons, one text field for client address, which by default is “localhost”, and two input fields: one for network bandwidth and another for passed bytes (Fig. 87 (a)). “LAN Host” button triggers StartHost() function in MultiplayerUI, which itself triggers StartHost() in RTSNetworkManager. Similarly “LAN Client” button triggers StartClient() functions in both scripts. But client can be started when another player on LAN has created host already. When connecting, the address typed in the text box is used, i.e. “localhost”. In the case of connecting to server only, StartServerOnly() function is called in MultiplayerUI, which itself calls StartServer() in RTSNetworkManager. After connected, one of the menus in Figure 4.30 (b-d) appears instead of not connected menu. They have text labels, showing server and client info. “Disconnect” button calls one of the functions HostDisconnect(), ClientDisconnect() or ServerDisconnect() respectively, which are firstly unsetting and removing units and buildings, and then disconnecting.

Multiplayer menu allows player to control the network bandwidth in order to achieve best multiplayer connection and gameplay. “Network Bandwidth” field changes network bandwidth on RTSNetworkManager by updating RTSNetworkManager.active.connectionConfig.InitialBandwidth value. The initial bandwidth is set to 3000 B/s which is slightly below the uNET hard limit. The player has a freedom to set this bandwidth lower or higher. The Passed Bytes is the limit for manually controllable network traffic. In NetworkSyncTransform, when synchronising positions, rotations, animation names and NavMeshAgent destinations, passed variables to [Command]s are counted towards “Passed Bytes”. Before sending to [Command] there is a check if previously sent traffic does not exceed the limit. In the case, the limit is exceeded, variables are not synchronised and are set to wait until the next synchronisation. “Passed bytes” does not account for the whole traffic and thus should be smaller than the “Network Bandwidth”.

“Enable Match Maker” button allows to enter matchmaker mode, where the new menu, shown in Fig. 87 (e) is used in order to connect with matchmaker. The upper button “Create Internet Match” creates a new match. It’s working in a similar way like when player creates LAN host. In such case it creates the match, where other players can connect to it later, after the match is created. Room name from the input box is used when creating the new match. Find Internet Match works in a similar way like when connecting as a client: the player is searching for already created room by another player. The room list is displayed and player can chose to which one to connect. Change MM Server allows to chose between 3 default servers: Local, Internet and Staging. Local one is similar to LAN multiplayer, used to connect to players within the same LAN. Internet and Staging uses Unity servers in order to set up matches that players could connect not just from the same LAN computer. “Disable Match Maker” button allows to get back to the not connected phase, i.e. shown in Fig. 87 (a).

4.4.11.6. Selection marks and health bars

Just below multiplayer button there are two ticks for enabling or disabling selection marks and health bars.



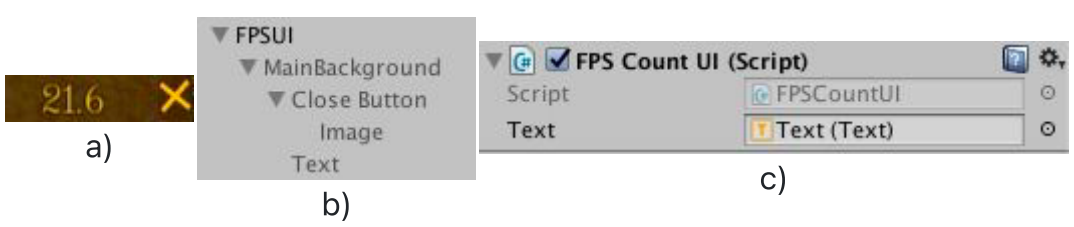
a) b) c)

Fig. 88 - Selection marks and health bars ticks gameObjects (a), SelectionMarksUI component for selection marks (b) and health bars (c).

There are two gameObjects, which represents selection marks and health bars (Fig. 88 (a)), and their setup are identical. Both has tick boxes and labels to the right from them. There are also two SelectionMarksUI components here (Fig. 88 (b-c)), with one on SelectionMarks and another on HealthBars gameObjects. **toggle** reference is used to get isOn value from corresponding Toggle components. SelectionMarks toggle has OnValueChanged action, which calls SwitchSelectionMarks() function, while HealthBars toggle calls SwitchHealthBars() function.

4.4.11.7. FPS and BattleSystem statistics buttons

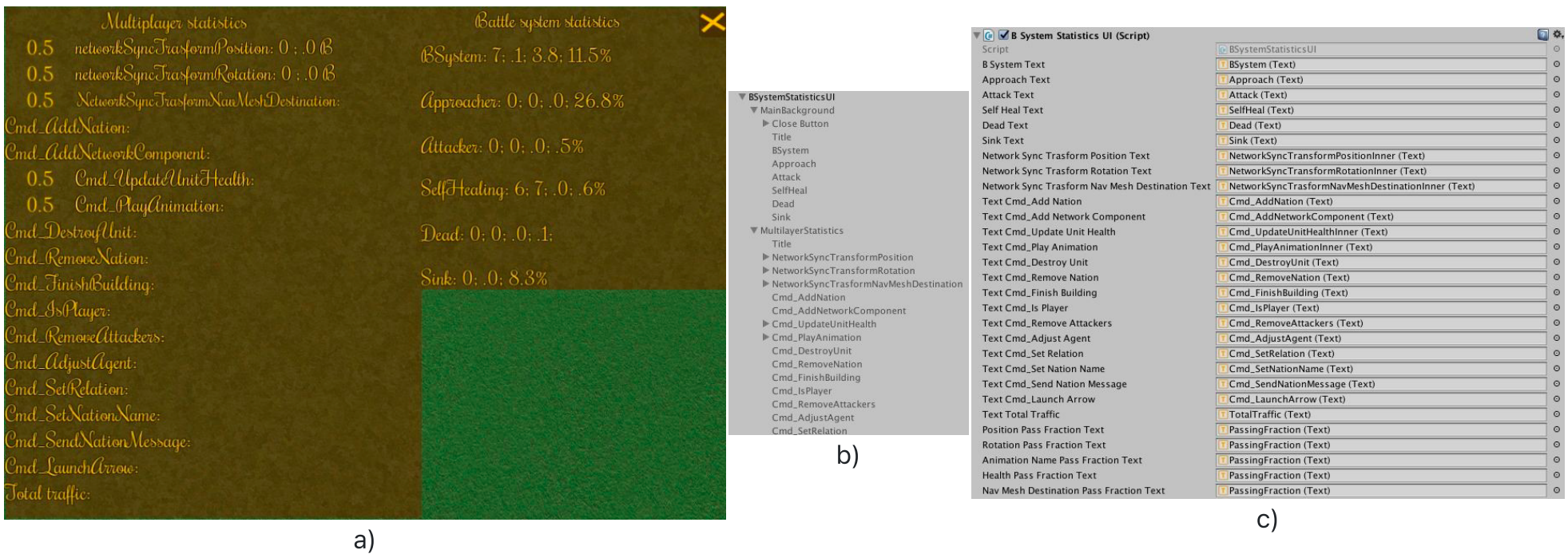
Two buttons named “FPS Counter” and “BSystem Statistics” are just below ticks of selection marks and health bars, and above “Scores” button. Buttons are used only to open FPS Counter and battle system statistics, which themselves are separate UI elements, so their function is only to enable corresponding gameObject. Both elements are used to check game performance on runtime.



a) b) c)

Fig. 89 - FPS counter UI as seen in the game (a), its gameObject setup (b) and FPSCountUI component (c).

Fig. 89 (a) shows FPS counter in the game. It appears in the top left corner of the game screen and has a number and close button. The number shows the current FPS rate of the game and close button allows to close FPS counter. FPSUI gameObject (Fig. 89 (b)) has inner gameObject, which can be enabled or disabled. This inner gameObject has a label background with Close Button gameObject and the Text , which stores counter values. This text is used as **text** reference in **FPSCountUI** component, attached to root gameObject (Fig. 89 (c)), in order to update counter values. **fpsGo** is a reference to inner FPS gameObject, which can be enabled and disabled. FPS values are received from FPSCount.active.fps.



a) b) c)

Fig. 90 - Battle system and multiplayer statistics as it appears in the game (a), its gameObject setup (b) and BSystemStatisticsUI component with all its references (c).

Battle system and multiplayer statistics appears as a larger UI element in game (Fig. 90 (a)), where numbers of units with overall performance are displayed for various battle phases. BSystemStatisticsUI gameObject setup is similar to FPSUI gameObject (Fig. 90 (b)). Two inner gameObject are used to enable or disable battle system and multiplayer panels. Battle system statistics have 7 gameObject with text labels, and 6 of them are used to display battle system performance. Multiplayer statistics have 18 such texts which displays the detailed traffic in terms of number of messages and bytes sent. Each of these gameObject has Text references in BSystemStatisticsUI component (Fig. 90 (c)). Battle system statistics values are formatted inside BattleSystem and accessed through message1 - message6 string variables.

Multiplayer statistics also have 5 input fields for a detailed control of particular synchronisations. These synchronisations are positions, rotations, NavMeshAgent destinations, unit healths and animation names. Player can increase or decrease any of these in case if other ones becomes too intense.

4.4.11.8. Scores UI

a) b) c)

Fig. 91 - Scores menu in the game (a), its gameObject setup (b), and NationScoresUI and NationScoresListUI components (c).

Scores UI are used to display scores and progresses for nations. The “Scores” button in the Options menu and the scores menu are separate UI elements in a way that button only opens scores menu.

In game scores menu appears as a big window in the middle of the screen (Fig. 91 (a)). The scores menu has two parts: the list of nations in the left and the selected nation scores in the rest of the window. The list is filled with nations and in brackets is specified master score of the nation. In the right side is shown numbers of units, buildings, damage, collected and current resources. Scores menu can be closed with close button in the right upper corner of the window.

NationScores (the child gameObject for NationScoresUI) gameObject contains 2 main child game objects (NationsScoresList and MainBackground), which represents sections of nations list and nation detailed scores (Fig. 91 (b)). NationScoresList has a regular grid, which adds button instances to represent each of the nation. MainBackground has mostly text labels, close button and resource icons.

There are two components attached on NationScoresUI: **NationScoresListUI** for managing nations list and **NationScoresUI** for managing scores for selected nation (Fig. 91 (c)). NationScoresListUI has 3 references the **nationScoresListGo** with main gameObject reference to check if it's active, **gridGo** to add new node instances as childs on it, and **listNodePrefab** with a prefab to be instantiated. This prefab can be found as NationScoresListNode set as inactive grid element. It has child gameObject to store a nation name label with its master score. **NationsScoresListNodeUI** component is attached on that prefab with a ChangeNationScoresUI() function, which is called when player presses the button to select the nation. This function sets nation in NationScoresUI as a selected nation to display its scores there.

NationScoresUI has all Text component references from corresponding labels and the UpdateScores() coroutine, which is running every 0.5 seconds in order to update all scores in the menu.

Master score is used here to represent overall progress of the nation. The table below illustrates sources and sinks of how master score is changed during various actions while in game. Developer can easily add further variety of actions in game by calling Scores.active.AddToMasterScoreDiff(scoreToAdd, nation) where scoreToAdd would be the score which is added by the action and nation - the nation to get this score change.

+0.5 x unit level	For damaging buildings
+0.05 x unit level	For damaging units
+0.005 x unit level	For collecting resources
-1 x unit level	For receiving damage on buildings
-0.1 x unit level	For receiving damage on unit
+1	For creating building
+0.1	For spawning unit
+0.1 x new level x new level	For leveling up units and buildings

Current resources display the current resources of the nation, while collected resources show only resources collected by working in Mining Points and chopping lumber. If they are received from population difference, they do not count here. Resource nodes are set at the start of the game by instantiating ResSlot gameObject on the grid. This allows developers, who wish to set up more resources in game, to automatically have these resources displayed between Collected and Current resources.

4.4.11.9. Error tracking

It can be useful to trace in game errors, which can't be traced via editor. It is usually helpful when error appears on the built game but does not appear while running in Editor. It's also quite helpful to trace rare errors in published game, while player is playing the game.

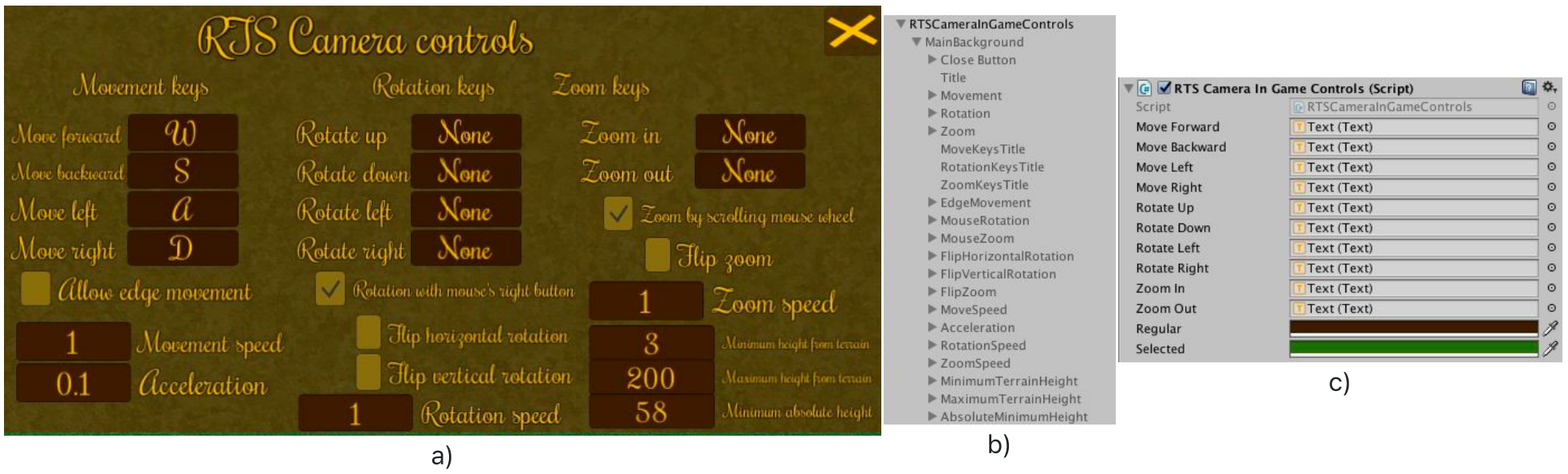
While RTS Toolkit does not provide tools by default, it is quite easy to install 3rd party tools such as "Consolation" (https://github.com/mminer/consolation) in order to be able to see in game errors. Developers can also set crash logging service such as (Unity Cloud Diagnostics) which would track and report errors and exceptions automatically when players are playing the game.

4.4.11.10. RTS Camera controls

RTS Camera controls can be changed by player in game via RTS Camera controls window (Fig. 92 a). The window can be opened by clicking RTS Camera controls button in Options menu. Properties in this menu are identical to RTSCamera settings set through Editor.

The interesting part here is key detection system. When player presses on the key, which needs to be changed, button becomes green, indicating that there is waiting for input from keyboard. Then player presses the key from the keyboard then input gets registered.

RTS Camera controls gameObject is being designed with all buttons, text fields, input fields and checkmarks as separate gameObjects (Fig. 92 b). Each button or input field has action to trigger their functions in **RTSCameraInGameControls.cs** component (Fig. 92 c), which is attached onto the root gameObject. Text fields for keys are also referenced as RTSCameraInGameControls.cs variables.



a) b) c)

Fig. 92 - RTS Camera controls menu and its gameObject setup.

4.4.11.11. RPG Camera controls

RPG Camera controls are also available for player to set (Fig. 93 (a)). The window can be opened by clicking on RPG Camera controls button in Options menu. RPG Camera controls gameObject (Fig. 93 (b)) is set in a similar way like RTS Camera controls and is driven by **RPGCameraInGameControls.cs** component (Fig. 93 (c)).

a) b) c)

Fig. 93 - RPG Camera controls menu and its gameObject setup.

4.4.11.12. Graphics settings

In game, graphics setting gives player the power to adjust the balance between speed and quality themselves. That is a huge advantage, as developer does no longer need to care which shader, material or amount of details to chose in order to make “every player happy”. From graphics menu players can choose themselves how detailed game they will play.

a) b) c)

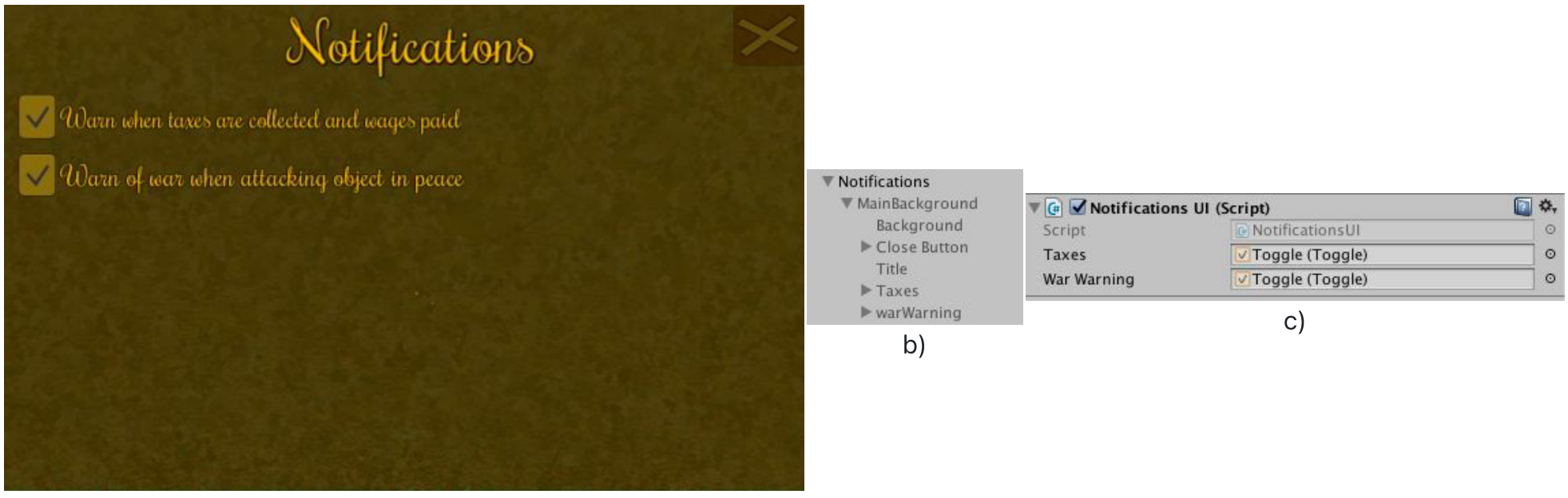
Fig. 94 - Graphics setting and its gameObject setup.

Fig. 94 (a) shows how Graphics setting menu looks in RTS Toolkit. Menu can be opened by clicking on “Graphics settings” button in Options menu. Each change is going through various in-game settings. E.g. when changing water detail, then water detail is switched in Water script. When entering new number in “Camera far clipping plane”, this is being set on Camera, Quality preset allows to chose one of the build in choice from QualitySettings levels. Terrain size and resolution dropdown allows to change the size and resolution of the terrain tiles. This will erase all current gameplay and restart the game from scratch. This can be useful to adjust terrain details at the beginning in order to get higher FPS or better graphics if default configuration is not good.

Graphics settings gameObject (Fig. 94 (b)) is set in a similar way like RTS and RPG Camera controls with corresponding references in **GraphicsSettings.cs** component (Fig. 94 (c)), which is attached on the root gameObject.

4.4.11.13. Notifications

Notifications menu allows player to enable or disable some notifications on screen, such as reports when taxes are collected and wages paid, or war warning notifications (Fig. 95). The menu can be opened through Options menu by clicking on Notifications button. Notifications gameObject with corresponding NotificationsUI.cs component is set in a similar way like RTS and RPG Camera controls.



a) b) c)

Fig. 95 - Notifications menu and its gameObject setup.

4.4.11.14. Journeys menu

Journeys menu allows player to set units on journey around the world. Menu can be opened by grouping units while Journey mode in grouping menu is enabled. It can be re-opened for already existing journey by pressing its group button. Journeys menu and it's setup is shown in Fig. 96.

a) b) c)

Fig. 96 - Journeys menu in game (a), it's gameObject setup (b) and JourneysUI component (c).

Player can set any kind of journeys by using a sequence of world coordinates, which defines waypoints of the path. If the next waypoint is within the bounds of the current map, then units start to move directly towards that point. If the next waypoint is outside the map, closest point to the next waypoint is being found on existing map and units are send to that point. If player's camera is following the group, then it is likely that as camera moves closer, new terrains will be generated. When units reach closest point, search will be repeated and the next point will be found. This way player units can travel with no bounds around the world.

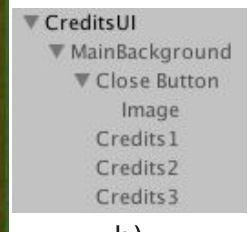
Once player opens the window, there appears only two buttons "Add position" and "Start Journey". By clicking add position player adds the first waypoint. The waypoint is in terms of N. (North) and E. (East) world coordinates. Coordinates should be float or integer numbers, they can be also negative. The next field is dropdown menu, from which player can chose to travel to one of known neighbour nations or set coordinates to home (current group centre position). By clicking again "Add position" player adds the second waypoint. Next to "Add position" button appears total distance in meters or kilometers and the total time in game time units, which will take the journey to complete. Finally when player is happy with the list of waypoints, journey can be started by pressing "Start Journey" button. Then Journey menu will get closed and units will start moving. In order to follow units, player can switch to RPG mode while units are selected. If units would reach impassable areas (big rivers, sea), player can open Journeys menu again by pressing on corresponding group button, and after modifying coordinates, to resubmit the journey.

Journey gameObject (Fig. 96 (b)) has quite standard setup - there are just buttons and input fields as active elements. Menu is controlled by **JourneysUI** component (Fig. 96 (c)). There is also a grid for waypoints list. Grid element is the first element set as inactive and being instantiated by journeys system when "Add position" button is pressed. JourneysUI holds this **positionCellPrefab** as a reference. References are also set on **closeMenu**, **distanceLabel**, and **timeLabel** in order to access them externally. **Journeys.cs** component is set on another gameObject, which controls the whole journeys system internally.

4.4.11.15. Credits button and window

Credits window is used to display game credits information to players.

a)



b)

Fig. 97 - Credits window appearance in game (a) and its gameObject setup (b).

Its appearance in game is shown in Fig. 97 (a), where are only some labels added and set text colours for them. The window has no functionalities, as its purpose is only to show credits information to the player (i.e. developer names, contacts, etc.).

CreditsUI gameObject has 4 child game objects, where 3 of them corresponding to each line labels, and the 4th is the close button, which closes the Credits window.

4.4.12. Diplomacy menu

Diplomacy menu is the UI part, which allows player to change diplomacy settings between nations. Diplomacy menu can be opened from Diplomacy button and allows to access nations list, our proposals, their proposals, our answers to their proposals and diplomacy reports.

4.4.12.1. Diplomacy button

Diplomacy button () is always visible button at the upper left corner of the screen. It is used to open nations list or close it if it's open. It has a Toggle component, which call NationListUI.FlipActivation() in order to open or close nations menu.

4.4.12.2. Nations list menu

Nations list menu appears in the middle part of the game screen with a list of nations if order that player could choose with which ones to start the dialog. Nations list appears as a transparent grid in the middle of the screen with corresponding nation faces, nation names and military icons (Fig. 98 (a)). By clicking on any of them, player can start the dialog. Only discovered nations are visible in the list.

NationList gameObject (Fig. 98 (b)) is set in a simple way that there is just used a grid with 4 column. **NationListUI** is the component, attached to the root gameObject (Fig. 98 (c)) and controls which nations to add to the list. **nationListCellPrefab** is the prefab of the whole nation cell, which can be found as NationCell prefab in Assets/RTSToolkit/Prefabs/UI directory. The prefab is instantiated when a new nation is created and the instance is added as a child gameObject to the grid, which is referenced through **nationList** variable in NationListUI. NationListUI also has **nationIcons** list, where are set all icons, which can be used for nation faces. **peaceIcon**, **warIcon**, **slavedIcon**, **masterIcon** and **allianceIcon** are the icons used to show the current military state between the player and the nation. As there can be wizzard nations in the game, **wizzardIcon** is used to show wizard icons instead of regular ones.

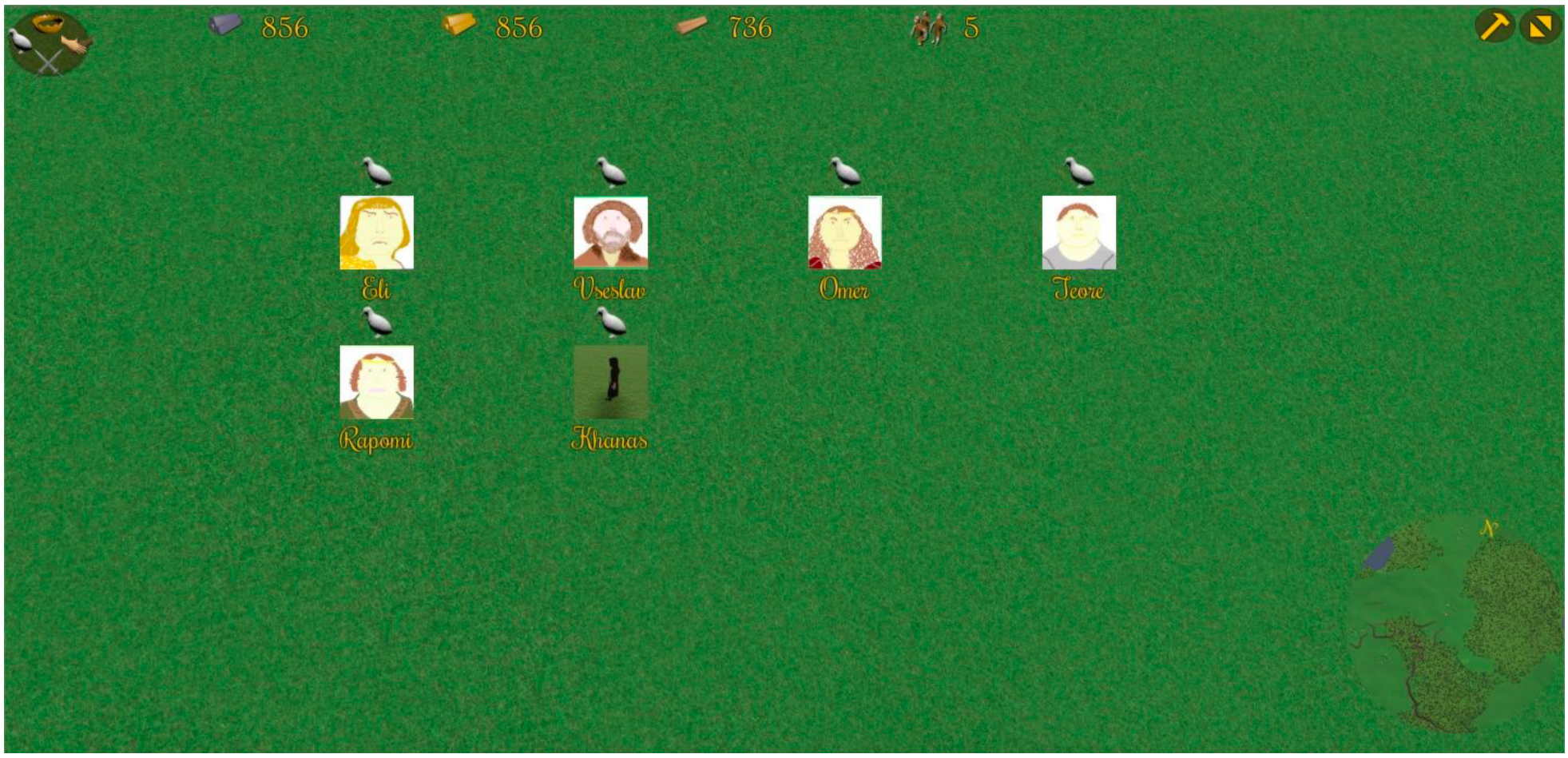
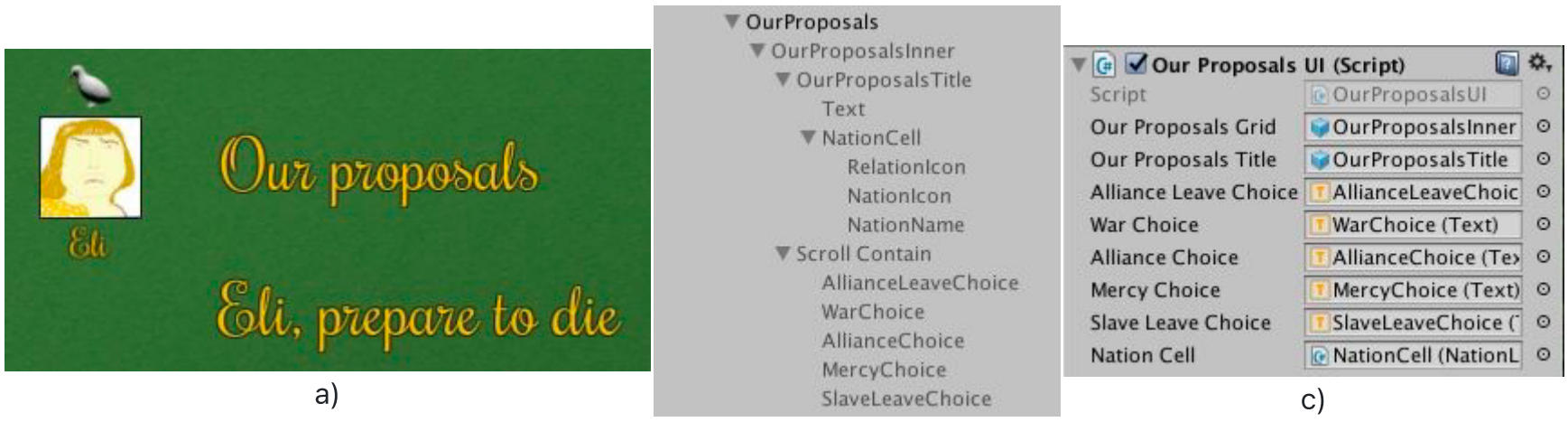


Fig. 98 - Nations list with added 7 nations into it (a), it's gameObject setup (b), NationListUI component (c), NationCell prefab (d) and NationListCellUI component (e) attached on NationCell prefab.

NationCell prefab itself has 3 UI elements (Fig. 98 (d)): relation icon at the top, nation icon in the middle and nation name text label at the bottom of the cell. Prefab has NationListCellUI attached to it with Image component references for **relationIcon** and **nationIcon**, and Text reference for **nationName**. These references are used to update corresponding images and labels.

4.4.12.3. Our proposals menu

When player clicks on any of nation cells, "our proposals" menu is being opened (Fig. 99 (a)). The menu contains chosen nation icon, title and the list of reasonable proposals, which player can make for a nation. There are 5 different proposals, which player can make to another nation: alliance leave, war, alliance enter, beg mercy and slavery leave. Each proposal is only reasonable for a particular relation between player and nation. If relation is peace then player can propose war. If player is on war, then relevant proposals can be to ask to enter the alliance or to beg mercy. If player is in alliance, the proposal is available to leave the alliance. And if player is enslaved, then there is possible to leave slavery.



a) b) c)

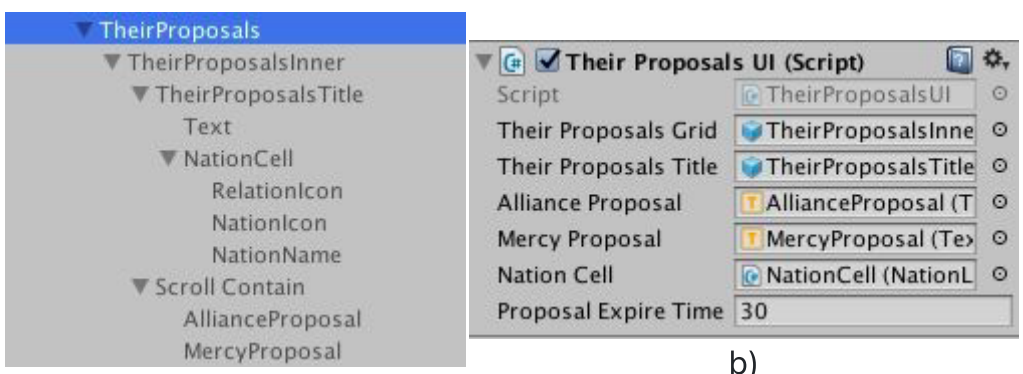
Fig. 99 - Our proposals menu as it appears in game (a), its gameObject setup (b) and OurProposalsUI component (c).

The menu consists from 3 main elements - the nation cell, title (“Our Proposals”) and the list of choices. gameObjects in this menu are set in a way to represent these 3 elements (Fig. 99 (b)): title and NationCell are not changing their relative position to the screen, so they both are set within OurPorposalsTitle gameObject branch. The list keeps changing and is set on a grid within ScrollContain gameObject branch in a way that only relevant proposal choices would show up. This is done by enabling one or another choice and keeping other ones disabled.

OurProposalsUI component is attached on the root gameObject (Fig. 99 (c)). It contains **ourProposalsGrid** and **ourProposalsTitle** in order to enable or disable our proposals menu. It also has all 5 Text references for proposal messages, as messages consist from the main body and the name. For example proposal “Let’s join alliance, Eli” has the main body, which is “Let’s join alliance,” and “Eli” is added from the nation name. The nation name is always different, depending on which nation player chose from nations list menu. OurProposalsUI also has reference to NationListCellUI in order to change the name and update the nation and relation icon.

4.4.12.4. Their proposals menu

In the identical way, like it is done with our proposals, their proposals menu is made (Fig. 100 (a-b)). Some of proposals are like statements and do not require any action from receiving nation. I.e. the proposal “We are breaking our alliance” makes changes immediately and does not require any answer or action from receiving nation. On the other hand, proposal "Let's join alliance” wants the answer from receiving nation if it accepts or not to make an alliance with sending nation. In such case there are only two proposals, which are considered as their proposals, when player can have a choice to answer: alliance proposal and mercy proposal. So this reflects the setup of the grid in their proposals and **TheirProposalsUI** component.



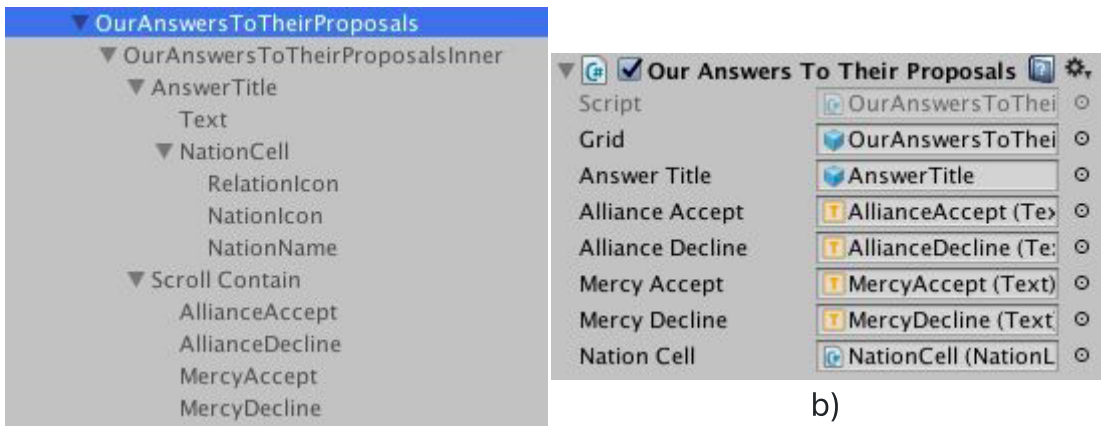
a) b)

Fig. 100 - Their proposals menu gameObject (a) and TheirProposalsUI component (b).

Their proposals can appear on players screen at any time, when nation proposes. It can also be accessed via their proposals menu, which itself is being opened if player clicks on the title “Our Proposals” while in our proposals menu. Their proposals stays for **proposalExpireTime**, after which they are removed from their proposals menu. “Their Proposals” title also has OnClick action to switch back to our proposals menu. So player clicking on these titles can keep switching between our and their proposals. If there are no their proposals, then menu is empty, with only nation icon and title label.

4.4.12.5. Our answers to their proposals

Player has a choice to answer to another nation proposal when pressed on the proposal text. As there are only two proposals, which other nation can ask to player, there are also two answers into each of them: player can accept or decline alliance or mercy proposals. If proposals are declined, nothing changes and another nation needs to make a new proposal, which player could consider later. If proposal is accepted, changes takes place immediately and relation is changed. OurAnswersToTheirProposals gameObject (Fig. 101) is set identically to the previously discussed gameObject for our and their proposals.



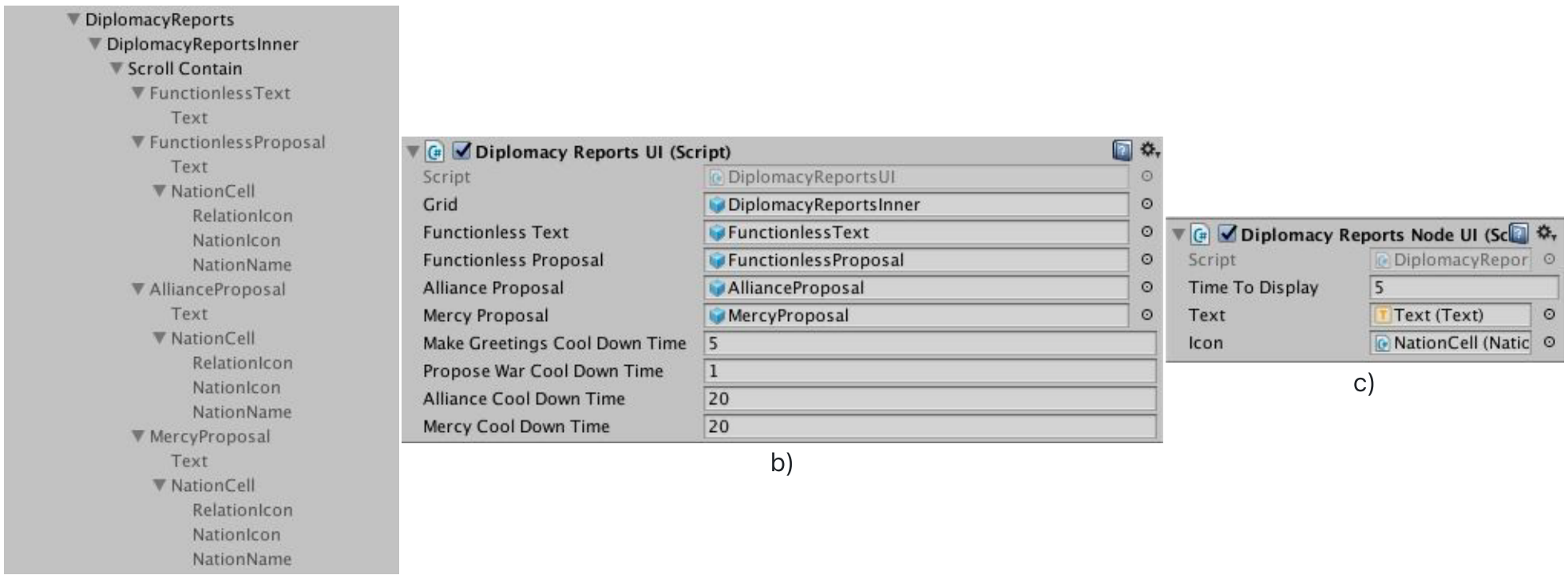
a) b)

Fig. 101 - OurAnswersToTheirProposals gameObject setup (a) and OurAnswersToTheirProposals component (b).

4.4.12.6. Diplomacy reports

Diplomacy reports menu has all reports, which can be thrown to player's screen when something happens. This includes alliance and mercy proposals, which are thrown the first time when they arrive and all other proposals, which decision depends not on the player but on another nation, making them to appear as a simple report, which does not need to answer.

Alliance and mercy proposals appear for the first time through Diplomacy Reports, but they have OnClick action, which allows player to open our answers to their proposals menu and answer to it directly. These proposals also go to their proposals menu, which player can access through diplomacy menus. Reports are displayed on player's screen just for **timeToDisplay** value set in **DiplomacyReportsNodeUI** (see Fig. 102 (c)). There are also cool down periods for make greetings, propose war, alliance and mercy proposals.



a) b) c)

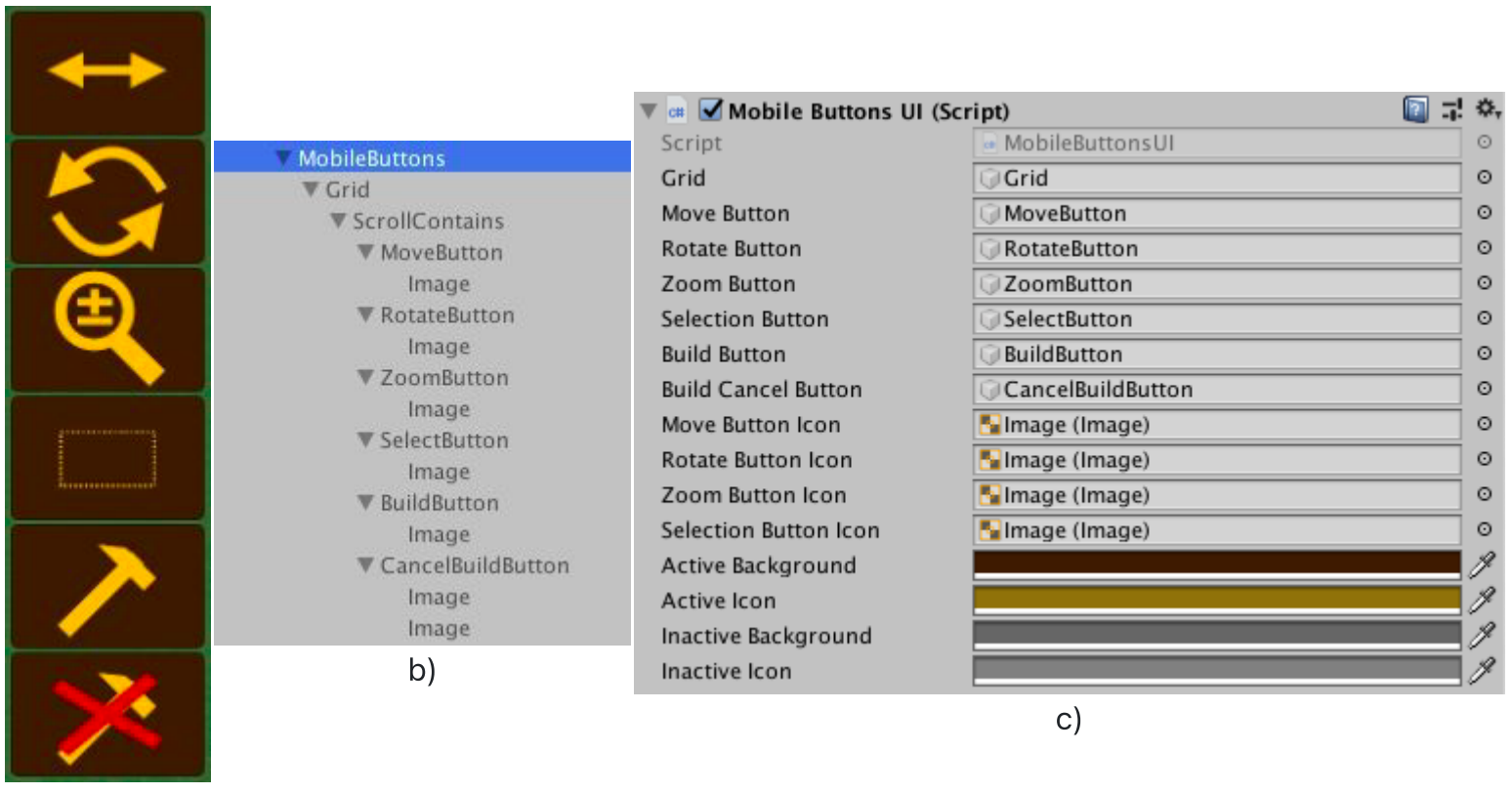
Fig. 102 - DiplomacyReports gameObject (a), DiplomacyReportsUI component (b) and DiplomacyReportsNodeUI component (c) attached on each report node.

Proposals, which does not require any answer, like "We are breaking our alliance" appears as functionless proposals. Their OnClick action is set only to close them and do not open any answers menu, as there can't be any answers to these proposals. Similarly works functionless text, which displays a simple report, for example "Taxes collected, wages paid". However, functionless texts does not have nation icon near them, as they are coming not from any nations, but as generic game reports about what's happening.

As a result, DiplomacyReports gameObject is set with these 4 children gameObject to represent each of the proposals (Fig. 102 (a)). As in the reports menu there can be visible the same proposal reports from different nations at the same time, reports are being instantiated by using these child gameObjects from DiplomacyReportsUI component references (Fig. 102 (b)).

4.4.13. Mobile buttons

Mobile buttons are used for navigation on mobile devices, which do not have keyboard and mouse (i.e. Android and iOS platforms). As there is no keyboard and mouse, on mobile devices is not possible to have full camera control, i.e. movement, rotation and zooming. However, touch screen allows to use one mode at the time. For such reason, mobile buttons allows to switch between movement, rotation and zooming modes and gives player full control of RTS camera and gameplay.



a) b) c)

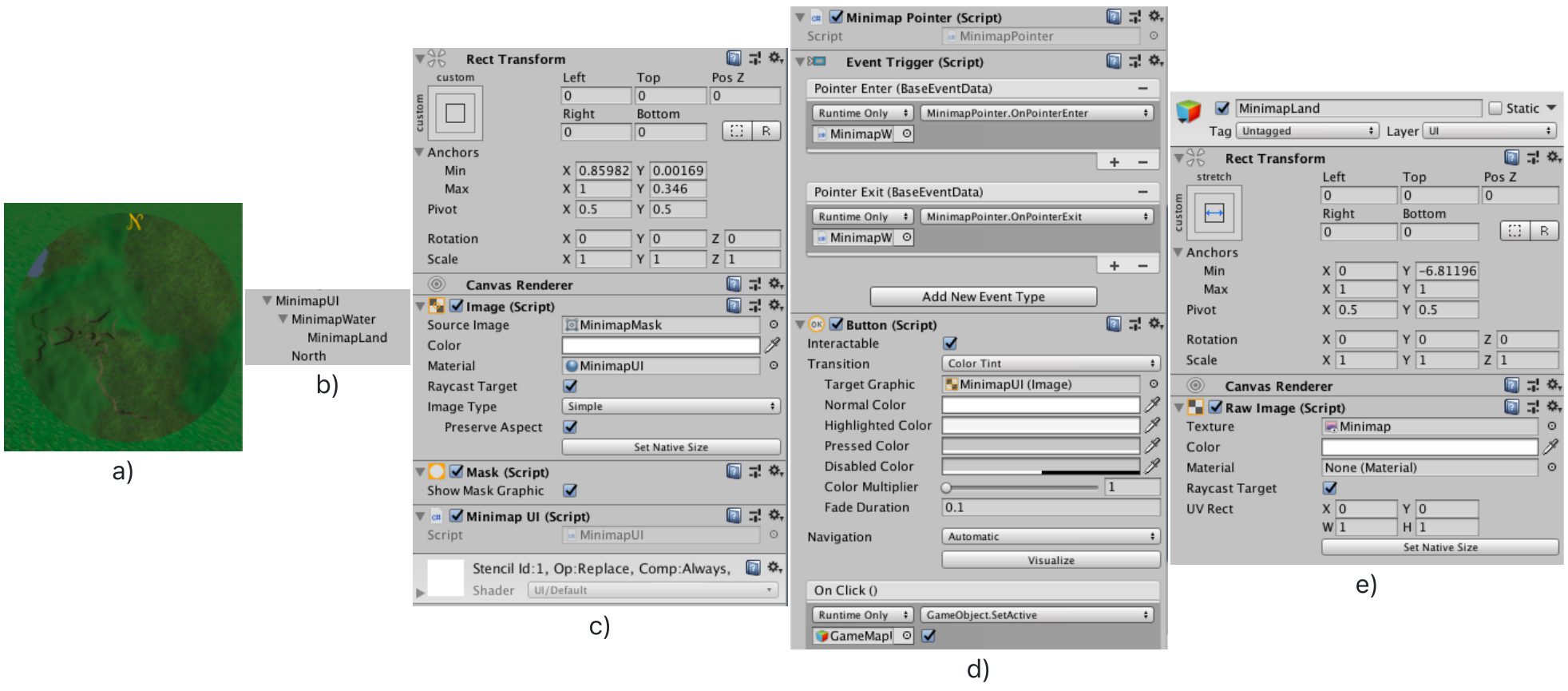
Fig. 103 - Mobile buttons as they appear in the game (a), their gameObject setup (b) and MobileButtonsUI component (c).

Mobile buttons appears at the right side of the screen only when platform is set to mobile one (Android or iOS) (Fig. 103 (a)). There are 4 buttons for camera control modes and two buttons for building modes. MobileButtons gameObject has a vertical grid, where all these 6 buttons are added (Fig. 103 (b)). **MobileButtonsUI** component is attached on the root gameObject (Fig. 103 (c)) and controls how buttons are switched in the game. If the button is enabled, its colour is set as **activeBackground** colour and icon colour is set as **activeIcon** colour. For non active buttons **inactiveBackground** and **inactiveIcon** colours are used respectively. Each button has one image for background and one image for icon (except CancelBuildButton, which has red cross cancel icon on the top of hammer icon).

MoveButton is calling ActivateMove(), RotateButton - ActivateRotate(), ZoomButton - ActivateZoom(), SelectButton - ActivateSelection() functions in MobileButtonsUI. Only one button can be activated at the time, meaning that camera is at that mode. Other buttons are in inactive state. When creating a building, ActivateBuildMode() is called with buildButton and buildCancelButton being activated. When player starts to build a building, build mode is deactivated and returned to the previous camera mode. These buttons with switching modes allows player to have a full control of RTS camera and enjoy fully featured RTS games without camera control restrictions on mobile devices.

4.4.14. Minimap

RTS Toolkit provides the minimap for the game. By default, minimap is displayed in the bottom right corner of the game's screen (Fig. 104 (a)).



a) b) c) d) e)

Fig. 104 - The minimap appearance in the game (a), its gameObject hierarchy (b) and components attached to it (c).

Minimap gameObject has relatively simple structure (Fig. 104 (b)) with only 4 gameObjects set. There is the main gameObject, which components are shown in Fig. 104 (c), and three other child gameObjects which renders the the water and land of the minimap, and the North direction icon.

The root gameObject Fig. 104 (c) has image and mask components. The image uses MinimapMask image, where is painted just a simple circle. Then Mask component allows to render the whole minimap to appear in a circle form, i.e. its edges are transparent (Fig. 104 (a)). The button component has onClick action registered, which allows the player to open the game map when player clicks on the

minimap.

MinimapLand gameObject is responsible for displaying the land on the minimap. This includes green areas, i.e. grasslands and forests. MinimapLand gameObject has only one component RawImage attached to it (Fig. 104 (e)). This component uses Minimap RenderTexture in the texture field. This way Minimap RenderTexture renders everything within the UI element. Minimap RenderTexture is also set through MinimapCamera, which is another gameObject outside MainCanvas.

However, MinimapLand only renders ground areas and not water. The water, which includes rivers, lakes and seas is rendered through MinimapWater gameObject, which is parent gameObject to MinimapLand. MinimapWater gameObject has Image, MinimapPointer and EventTrigger components attached. The Image has set no source image but instead uses solid blue colour to render water. MinimapPointer works together with EventTrigger to trace when player's mouse is over the minimap but is not related with water rendering. Instead it detects when the mouse is on minimap in order to hide or show other UI elements which player is using at the moment.

There is also "N" letter visible at the edge of the minimap, which stands for "North". This way the UI is displaying both - the minimap and compass. The letter in game is coming from North gameObject, which is parent to MinimapUI gameObject. Only Text component is attached to North gameObject, which displays the letter "N".

4.4.14.1. Minimap camera

Minimap camera is responsible for displaying everything what is visible in the minimap. The minimap is set in a simple way, where is shown everything visible to the player in a regular gameplay window, but instead it is shown from the vertical projection. In order to do that, secondary orthographic camera was used with its RenderTexture set instead of regular rendering to the screen. For such purpose there is only a single gameObject set with Camera and MinimapFollow components (Fig. 105).

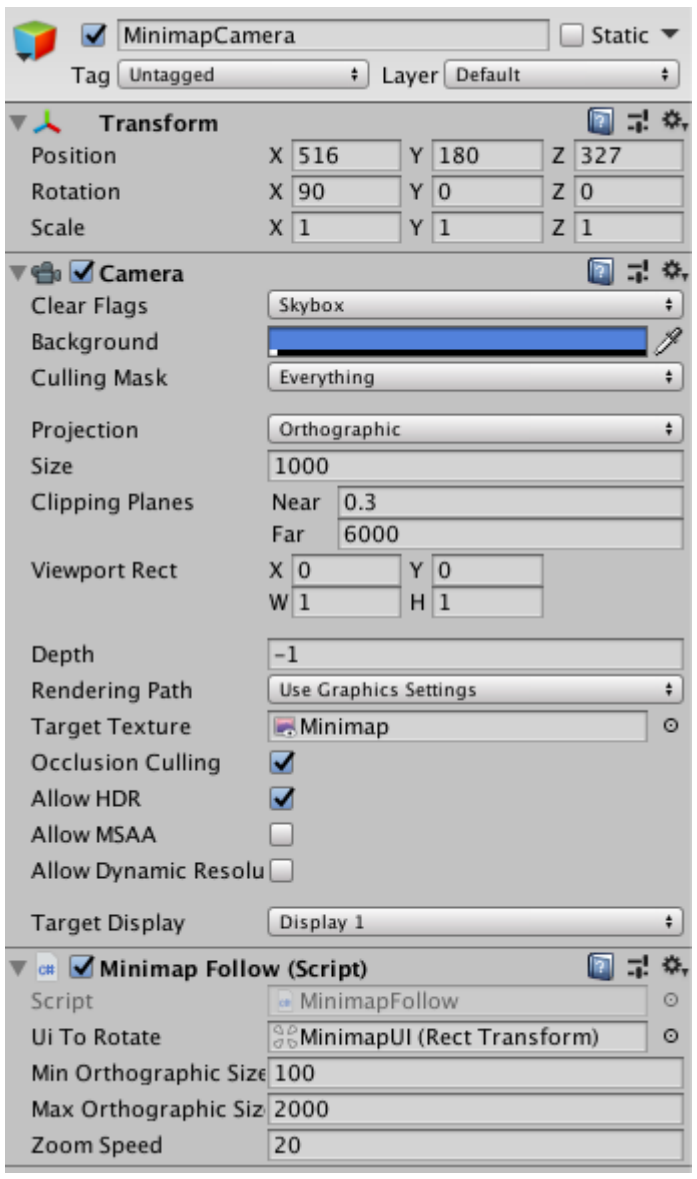


Fig. 105 - Minimap camera components.

As Camera is set with RenderTexture, which is displayed through the MinimapLand, the minimap is rendered automatically with no other scripts needed. The MinimapFollow is only used to follow the camera. The script traces the main game gamera and puts minimap camera position at the same position. When player moves RTS or RPG camera, the minimap camera moves together. The rotation of minimap camera is also mirrored from main game camera. As the MinimapUI gameObject is rotated, there are also rotated its child gameObjects, which includes both the MinimapLand and North gameObjects.

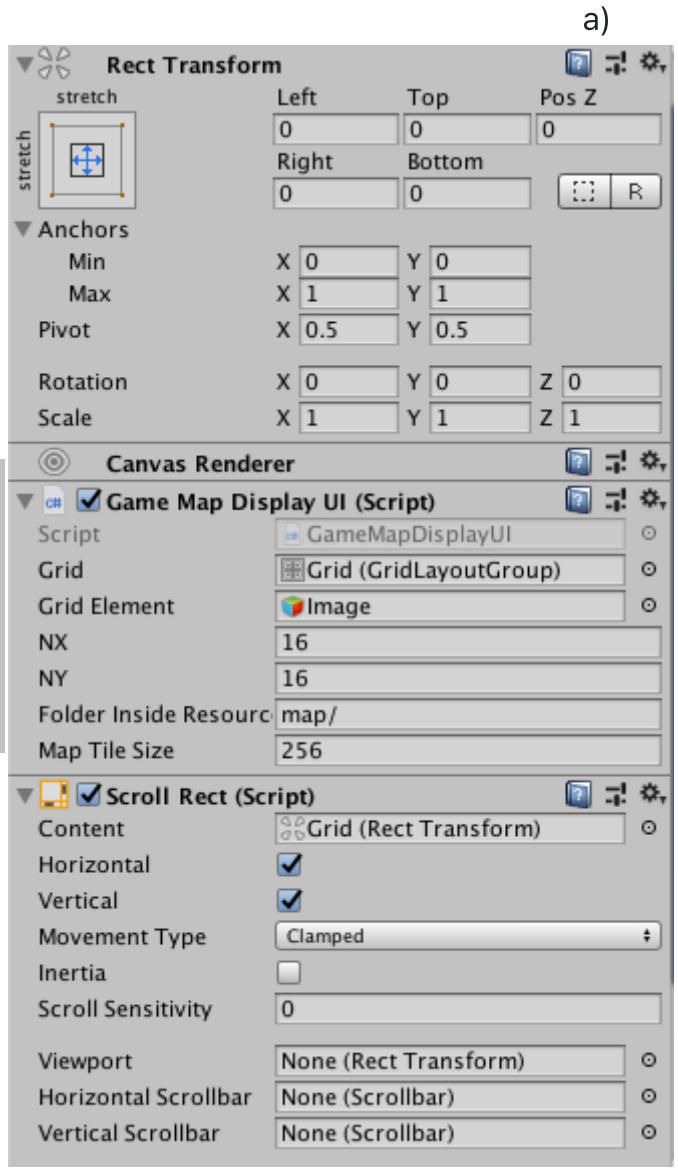
Player can also zoom in or out when the pointer is placed on the minimap and mouse wheel scrolled. In order to detect when the pointer is over the minimap, MinimapPointer.active.isPointerOnMinimap is used in order to detect such cases. Zoom of minimap is performed between **minOrthographicSize** and **maxOrthographicSize**.

4.4.15. Game map

Minimap allows player to view only a small area of the world. The large scale view is done through the game map. In RTS Toolkit the player can open the game map by clicking on the minimap. The map takes appears in the full screen (Fig. 106 (a)). Player can also slide the game map by holding left mouse button and moving the mouse around. This allows player to view areas behind visible boundaries.

a)

b)



c)

Fig. 106 - Game map UI appearance in game (a), its gameObject setup (b) and main gameObject components (c).

The GameMapUI gameObject (Fig. 106 (b)) has several child and sub-child gameObjects. They are responsible for various functionalities of the game map such as map sliding, tiling, closing and showing compass.

GameMapDisplayUI script is attached to the main gameObject (Fig. 106 (c)) and is responsible for main controls of how the game map is displayed in game. **grid** variable is the gameObject onto which game map tiles are instantiated. The prefab for each tile is set through **gridElement**. When creating the game map, at the start of the game, **gridElement** is instantiated **nX*nY** times, and all instances placed as child gameObjects for **grid** gameObject. Each instance is placed on the grid in the way that each tile is joined edges of other four neighbouring tiles. This way the game map is made from multiple map tiles, joined together as a single continuous map, without gaps between tiles.

By default there are set nX=6 and nY=8. This means that there are 6*8=48 meap tiles in total. Each map tile is displayed as a texture, which was prebaked before the game is launched. All 48 textures are saved in Assets/RTSToolkit/GameMap/Resources/map directory. Naming of textures is set in the following way “map_i_j” where i is between 0 and 5, and j is between 0 and 7 (by default). The location is inside Resources folder and is loaded via the Resources.Load() function. So Unity sees the folder, where are game map textures as “map/”, which is also set as **folderInsideResources** variable in GameMapDisplayUI component. The last variable in GameMapDisplayUI is mapTileSize, which tells how big each tile should be displayed in pixels.

Once textures are loaded, they are set on corresponding grid element instances. Here each grid element instance has RawImage component, where each tile texture is set.

The sliding of the game map is done through ScrollRect component set on the GameMapUI gameObject. ScrollRect has set the whole grid gameObject to be sliding to any direction. MovementType is set to be clamped in order to keep the game map inside the screen. As a result, when player slides the game map to the edge of the last tile, there is not possible to slide any further.

There are also set several other helping elements in the game map UI. The Background is set as tiled background in case if the game map would be needed to move outside of the screen boundaries. This allows to see background wood texture in empty space. CloseButton is set in the same way as for other UI elements where are set two images and the button action in order to disable the main gameObject. There is also set GameMapTitleText gameObject, which displays the title of the game map. The title for player is then visible at the top of the screen when the game map is open. Finally there is Compass gameObject. This is just a static gameObject with image component, which displays the compass texture.

4.4.15.1. Game map creation

Game map is based on textures, which are used to display it. So baking these textures is the main part. Baking texture methods are fully done here in RTS Toolkit from scratch, and thus available in the package. Baking of textures works in a similar way as displaying the minimap on the runtime. The difference is that textures are saved into files as large terrains are scanned to display such large scales, which can't be practically exposed on the runtime.

The game map baking camera components are set on a single object and shown in Fig. 107.

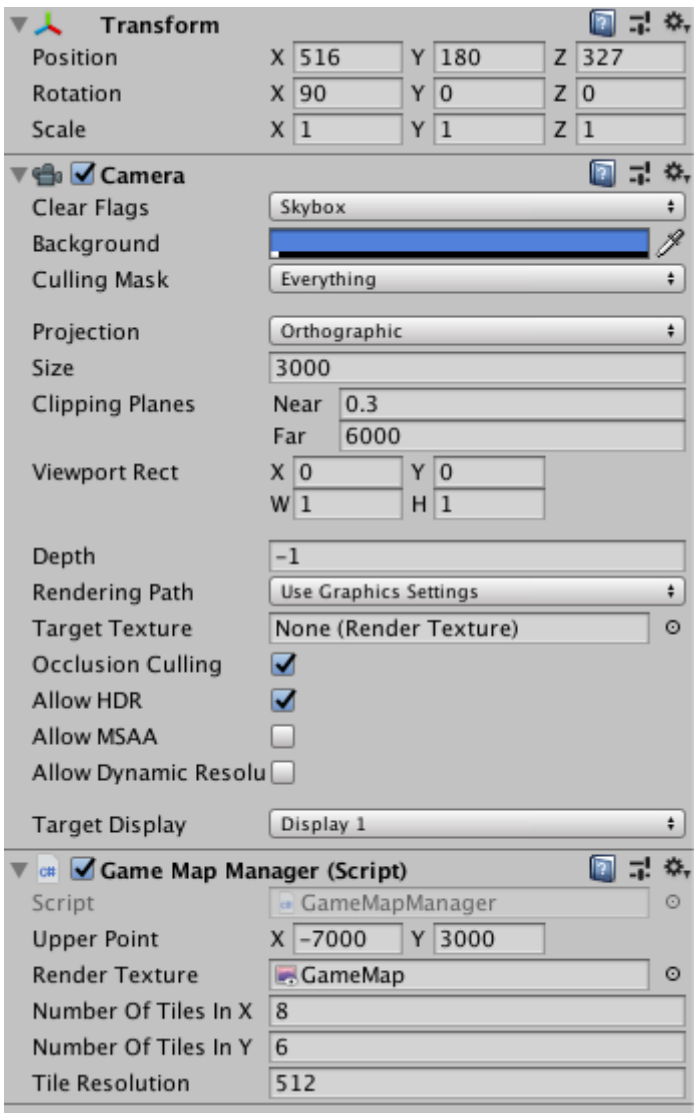


Fig. 107 - Game map texture baking camera components.

There are two components set: Camera and GameMapManager. The Camera is set as orthographic and exposed vertically downwards.

GameMapManager is responsible for controlling how camera gameObject moves, takes snapshots and saves textures into files. Camera moves within the grid, which size is **numberOfTilesX*numberOfTilesY**. Each cell has the size equal to the orthographic camera **size**. By default it is set to 3000. **upperPoint** is the location in the game world where taking snapshots begins. **tileResolution** allows to control the resolution of tile textures.

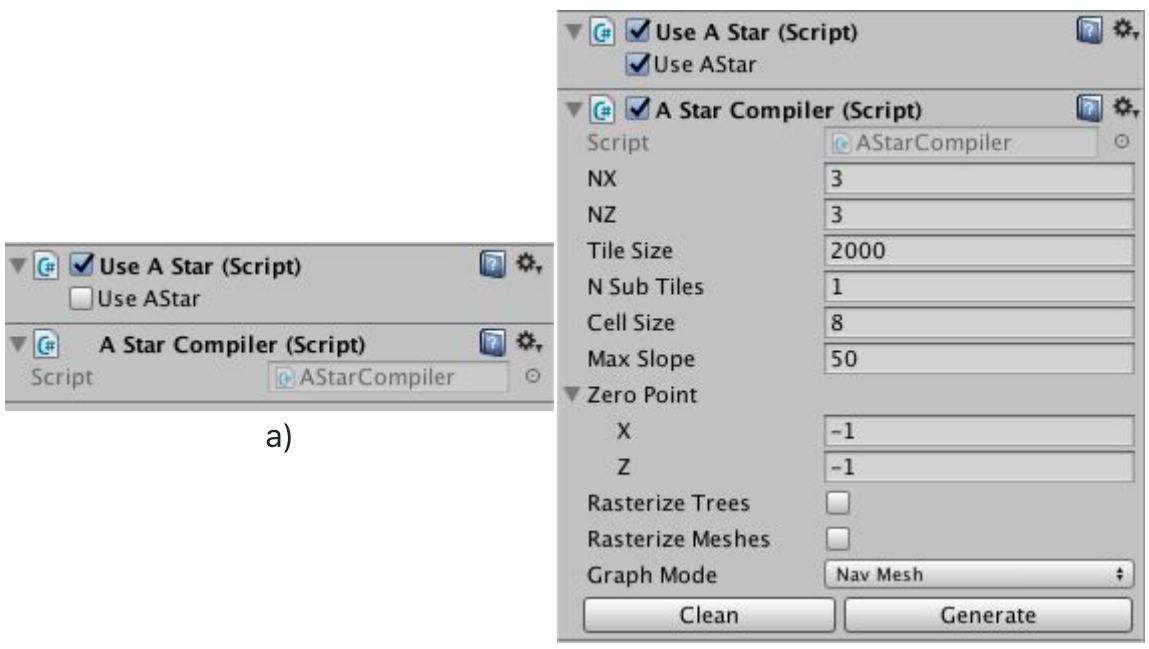
In order to start baking minimap textures, user needs to enable GameMapCamera which is disabled by default. It can be also helpful to unset buildNavigation in active GenerateTerrain component as it would be faster to just bake snapshots and not building navigation. In such case all nations in NationSpawner should be set as not to spawn and Animals gameObject set as inactive. By starting play in editor, user should be able to see camera jumping between terrain tiles every several seconds.

5. A* Pathfinding Project extension

RTS Toolkit supports usage of A* Pathfinding Project (APP) (http://arongranberg.com/astar/). APP is more advanced solution than Unity's build in navigation system and allows much more control. It also adds new possibilities, such as baking NavMesh on new procedurally generated terrain tiles, etc. RTS Toolkit does not include APP itself, as it is a separate asset, so users, who wish to use APP, should buy it separately.

5.1. Setting up A* Pathfinding Project

Once imported, make sure that AstarPathfindingProject folder is inside Assets directory. In the main scene can be found AStarCompiler gameObject with **UseAStar** and **AStarCompiler** components attached on it (Fig. 108). APP can be enabled or disabled. By default it is disabled and **useAStar** boolean tick on UseAStar component can be clicked to enable it. Make sure to wait a few seconds as there is recompilation running while switching between enable and disable states. Developers can find that while compilation is running, there is a small wheel icon ( ) rotating in the low right side of unity's screen. After recompilation is done, wheel icon disappears. During the recompilation "ASTAR" is being registered to Scripting Define Symbols in Player Settings. It is also changing component on rts unit prefabs in order use APP navigation instead of Unity's, so it's a good idea to make a backup before switching on and off APP (even if it is very unlikely, there could go something wrong and there might be a risk to lose properly working prefabs if some components would get incorrectly added or removed).



a) b)

Fig. 108 - UseAstar and AStarCompiler components when APP is disabled (a) and enabled (b).

After recompilation completes, AStarCompiler parameters becomes visible (Fig. 108 (b)) and APP extension is ready to use.

5.2. AStar compiler

AStar compiler is used to automatically set APP on the RTS Toolkit with baking NavMesh on the terrain tiles. It also regenerates NavMesh when terrains are updated, i.e. when camera is moving. Fig. 108 (b) shows parameters how AStarCompiler is set by default. **nX** and **nZ** gives the number of tiles. **tileSize** sets the size of the tile in distance units. **nSubTiles** is number of subtiles to be splited for each tile (i.e. with the Fig. 108 (b) setup there would be 3x2=6 tiles in x and z directions, and 6x6=36 tiles in total on the terrain). **cellSize** sets resolution for NavMesh. It's value is in distance units, i.e. in the given example the subtile size would be 2000/2 = 1000. The number of cells in the subtile is 1000/8 = 125, and the total resolution of the the entire NavMesh is 125x6 = 750 (or just 750 x 750 cells). In other words, cell is the smallest element on the NavMesh, which can be resolved and cellSize gives its size in world distance units.

zeroPoint gives tile index at of the tile, which has minimum values in x and z. For example when there are 3x3 terrain tiles, the central on will have index 0,0 but the minimum one would be -1,-1. This can be used if initially terrain starts at different tile indices.

rasterizeTrees and **rasterizeMeshes** sets Recast generator to rasterize trees on the terrain or meshes, so that these areas would not be walkable. **graphMode** allows to chose between NavMesh and Grid. If NavMesh is chosen then Recast graph generator will create NavMesh on the terrain, while if it's Grid chosen, regular grid graph would be used instead.

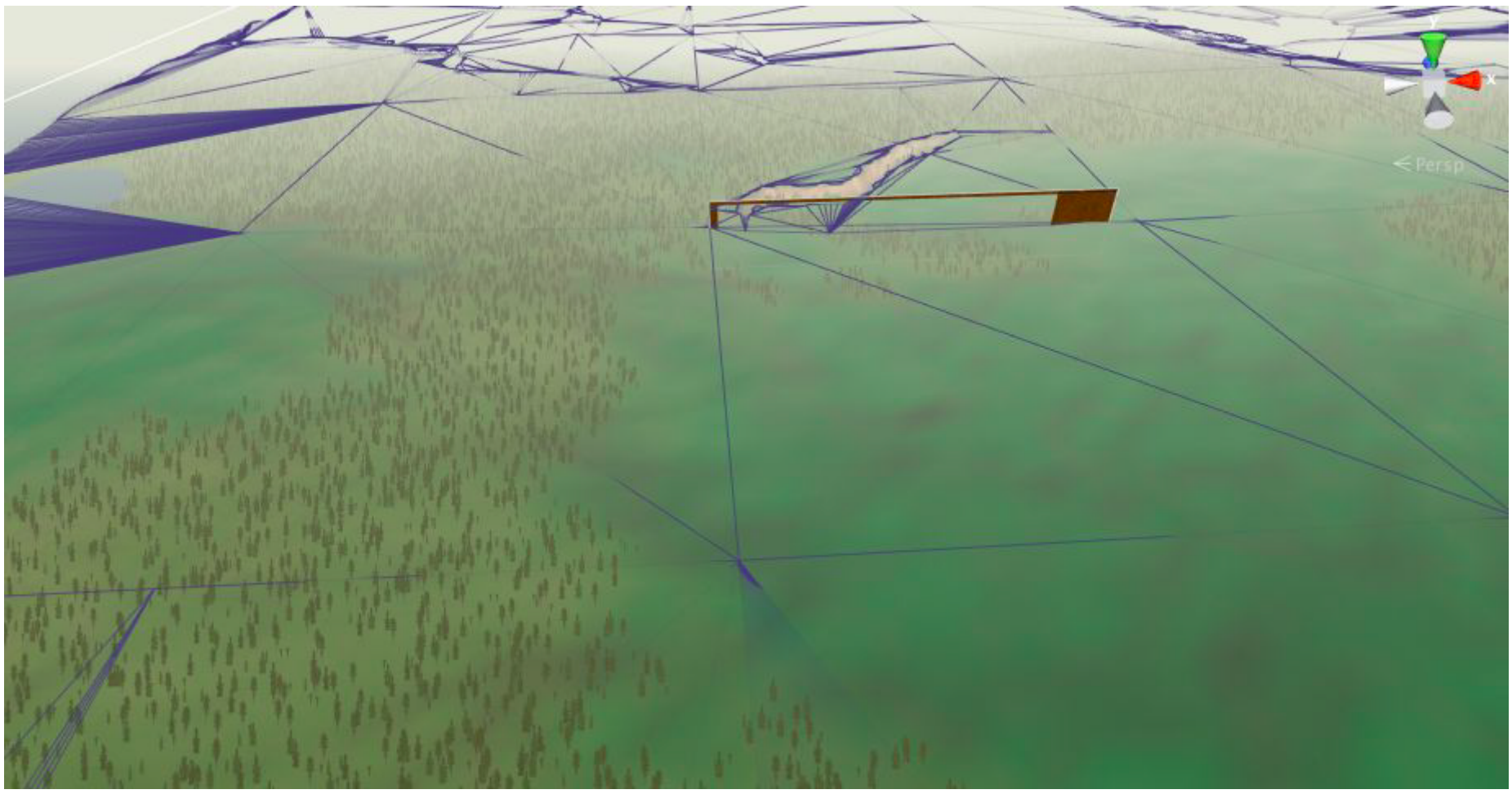


Fig. 109 - The view of APP generated NavMesh on RTS Toolkit terrains from Editor view window.

When parameters are set, developer can press “Generate” to generate the NavMesh. It takes about 5-20 seconds to generate it, after which on AStarCompiler gameObject are added two child gameObjects: A* and RVO Simulator. Once generated, terrains will be covered by NavMesh graphs (Fig. 109) to see where NavMesh is added. After NavMesh is generated, all these landscapes becomes walkable and any units can walk there.

5.3. RTS units

When switching to APP, unit and building prefabs, which are registered in RTSMaster.rtsUnitTypePrefabs are being changed. The changes are that on movable units **NavMeshAgent** component is removed, but instead **AgentPars**, **Seeker**, **FunnelModifier** and **SimpleSmoothModifier** components are added. **radius**, **height**, **maxSpeed** and **stopDistance** values on AgentPars are copied from NavMeshAgent before its removal, so that unit behaviour would not change when switching between Unity and APP navigation. On buildings **NavMeshObstacle** is replaced by **NavmeshCut** components in the same way. When switching back to Unity’s navigation, everything is done in the inverted way. This allows to switch the entire navigation system with automatically setting up all prefab units just in a single useAStar tick switch.

5.4. Movement and local avoidance

Movement and local avoidance allows to move units along their paths. In the extension it is done in half manual way by changing agent transform position and rotation. The movement and avoidance is done in the same step within ManualRVOs component, which is attached on RVO Simulator gameObject. The system is set in a way that movement would be set by calling UnitPars.MoveUnit() function independently which navigation system is used. So MoveUnit() calls NavMeshAgent.SetDestination() if Unity’s navigation is used and AgentPars.manualAgent.SearchPath() if APP navigation is used. For this reason, there is almost no difference in a visual view of how units move in the game when using one or another navigation system. The difference comes that there is easier to handle larger number of units with APP at the same FPS counts than Unity’s navigation. APP also has much more customisable parameters, which can be adjusted.

5.5. Installing, uninstalling and managing APP

If there is a need to uninstall or reinstall APP, it needs to be done in the following ways:

Installing:

1. Put APP files in the same project as RTS Toolkit.
2. Turn on useAStar tick.
3. Press Generate button in AStarCompiler.

Uninstalling:

1. Press Clean button in AStarCompiler.
2. Turn off useAStar tick.
3. Delete APP files from the project.

Enabling or disabling APP does not require to put or delete any files, but APP has to be installed previously.

Enabling:

1. Turn on useAStar tick.
2. Press Generate button in AStarCompiler.

Disabling:

1. Press Clean button in AStarCompiler.
2. Turn off useAStar tick.

Making these operations in another order can be a recipe for disaster: there may appear errors if files are deleted first before disabling useAStar first. If such thing happens, make sure to clean ASTAR in “Scripting Define Symbols” which can be found in PlayerSettings. It is also good to always have a backup before doing any of these operations and carefully check if everything works as expected after them: if something goes wrong, there is always easiest way to restore everything from a well working copy.

6. Quick scripts

Quick scripts are used to quickly do a particular thing. It can be camera teleport, spawn specific number of units when player presses specific key, etc. These scripts are helpful for debugging and making quick demos to demonstrate one or another feature. Quick scripts can be found in Assets/RTSToolkit/Scripts/QuickScripts directory.

6.1. Spawn quick armies

SpawnQuickArmies is a script which allows player to quickly put particular units around nation centre (Fig. 110). Parameters are the following ones:

n	The number of units to spawn.
nation	The nation to which units will be added.
radius	The enclosed circle radius within which units should be added.
unitType	rtsUnitId representing value, i.e. 13 will spawn arsonists.
key	The key in the keyboard to press in order to spawn units.
randomiseRotation	Units will be spawned with random rotations.

So the following component in Fig. 110 would spawn 50 arsonist units within 50 distance units around player nation centre when player presses “q” key in the keyboard. Multiple components can be added on the same gameObject - that gives mixed armies. By pressing key second, third, etc. times, more units will be added. This allows to quickly simulate a battle of two fronts by putting two armies close to each other (makes AI nations to propose a war), when the fight begins.

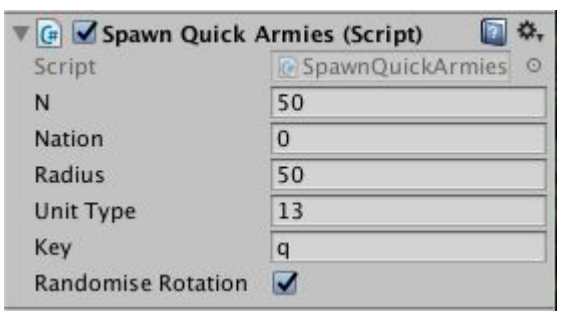


Fig. 110 - QuickSpawnArmies component.

6.2. Quick kill armies

Similar like spawning, **QuickKillArmies** allows player to quickly remove specific type units from the game. For example component in Fig. 111 will allow player to kill all player archers when “e” key is pressed. **fullDeath** just sets health to negative values and allows units to die and sink after that. Quick spawn and kill armies allows also to check if spawning and unsetting pipelines are working correctly, i.e. does not throw errors when new units are added and removed from the game.

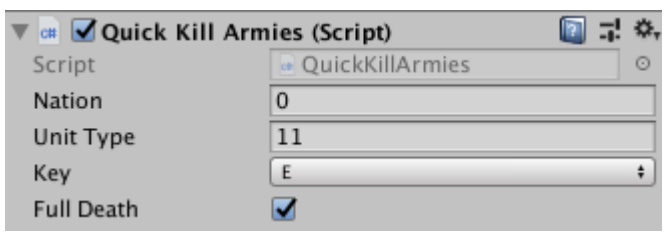


Fig. 111 - QuickKillArmies component.

6.3. Camera teleport

CameraTeleport component allows player to quickly teleport camera to **teleportPosition** (and back on second click) when player press **teleportKey** (Fig. 112). CameraTeleport can be useful to check how loading and unloading terrain tiles work, by using teleportPosition at a big distance from the current camera position.

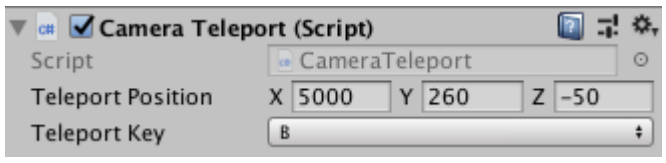


Fig. 112 - CameraTeleport component.

7. Credits

- Unity Standard Assets are used for some particle systems, water, night time greying post processing effect, etc: https://docs.unity3d.com/560/Documentation/Manual/HOWTO-InstallStandardAssets.html
- Unity NavMeshComponents (MIT License) is included for runtime NavMesh generation: https://docs.unity3d.com/560/Documentation/Manual/HOWTO-InstallStandardAssets.html
- KDTree class used for accelerating search of nearest neighbours: http://forum.unity3d.com/threads/29923-Point-nearest-neighbour-search-class
- Unit rectangle selection script was based on Camera Marquee tutorial and was adopted to RTS Toolkit needs: http://paulbutera.wordpress.com/2013/04/04/unityrtstutorialpart1marqueeselectionofunits/
- Classes adopted from procedural terrain generation tutorial: http://code-phi.com/infinite-terrain-generation-in-unity-3d/
- Humanoid model mesh has been created with MakeHuman and modelled with Blender (added weapons, armour, UV maps and animations).
- Some CC0 textures for model UV maps has been used from http://opengameart.org/ (UV maps are baked in Blender and original textures are not included in RTS Toolkit).
- Extension created for A* Pathfinding Project: http://arongranberg.com/astar/