CROOKED HEAD

Turn Based Strategy Framework

Documentation

Version 2.0

1. Introduction

This project is a highly customizable framework for turn based strategy games. It allows to create custom shaped maps, place objects like units or obstacles on it and play games with both human and AI players. The framework was designed to allow implementing various gameplay mechanisms easily. In this document I describe in details how to use it. In subsequent chapters I present project structure – what files it contains and which of them you're going to need, scene structure – how to set up a scene and what scripts to use, how to customize the project to fit your needs and finally recap everything in a short tutorial chapter. To get you started, I also provided a few example scenes with different kinds of units and styles. If you have any questions, you are welcome to contact me.

2. Project structure

Project structure is shown in Fig. 1. The essence of the project is code contained in Scripts folder. It is divided into namespaces to avoid collisions with other libraries and help you navigate easier. Scripts that extend the Unity Editor are stored in Editor folder. The most important script here is GridHelper - a powerful tool that will help you set up the scene.

Prefabs folder holds prefabs for human an AI players. These are the only pre-made prefabs that you will use in your scenes. The rest of them - cells and units - you will customize yourself.

The Examples folder contains examples that I prepared to get you started with the framework. Each of the examples contains its own assets, code and prefabs. The assets I used are as follows: 1 bit pack [1], Roguelike Characters [2], Roguelike/RPG Pack [3], Alien UFO Pack [4], Hexagon Tiles [5], UI Pack [6] and Kenney Fonts [7]. Those are really great assets that you may want to check out. It is worth noting that they are public domain.

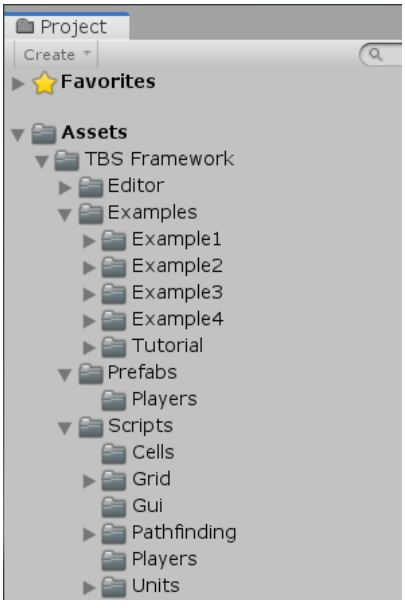

Fig. 1 - Project structure

3. Scene structure

Let's look at a scene created with simple assets available in Unity. The scene consists of a grid of hexagonal cells, a few units of three different kinds, obstacles and minimalistic user interface. Fig. 2 shows the scene.

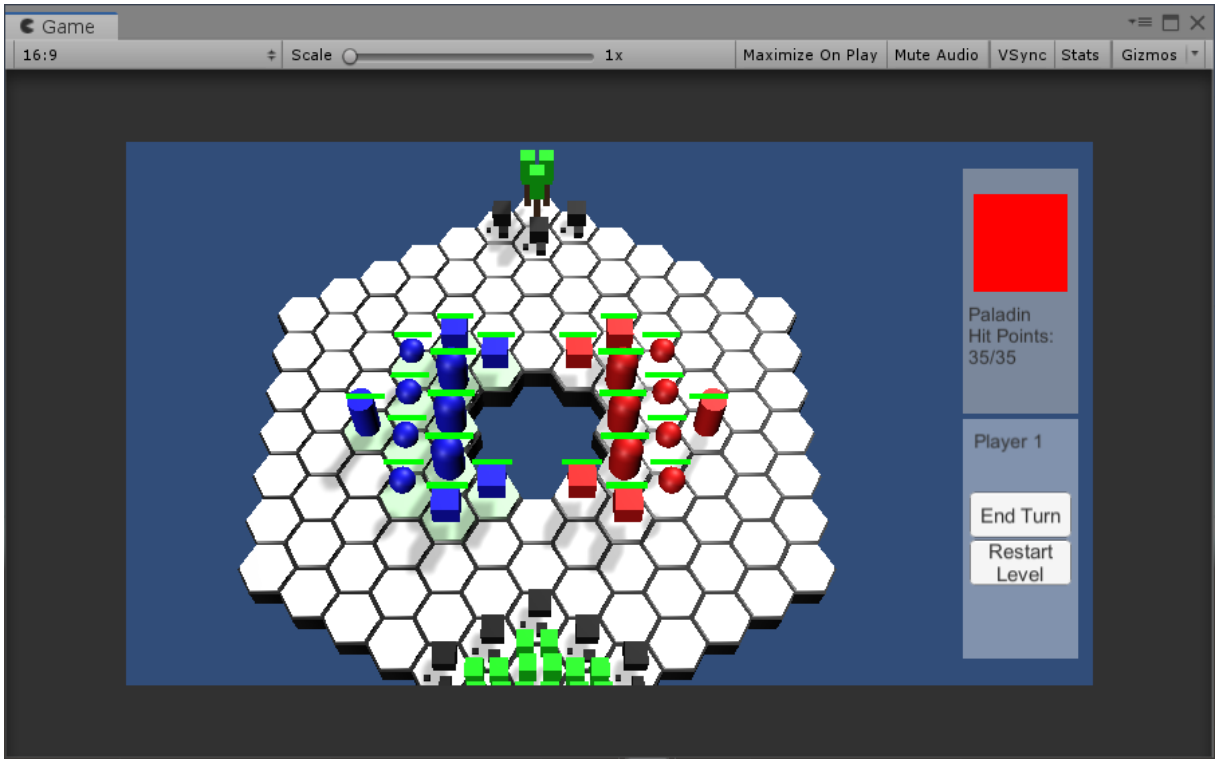

Fig. 2 - Simple scene

Doesn't look very impressive at the moment, does it? In a second we will see what can be done to customize the project. First let's take a look at the scene setup, shown in Fig. 3.

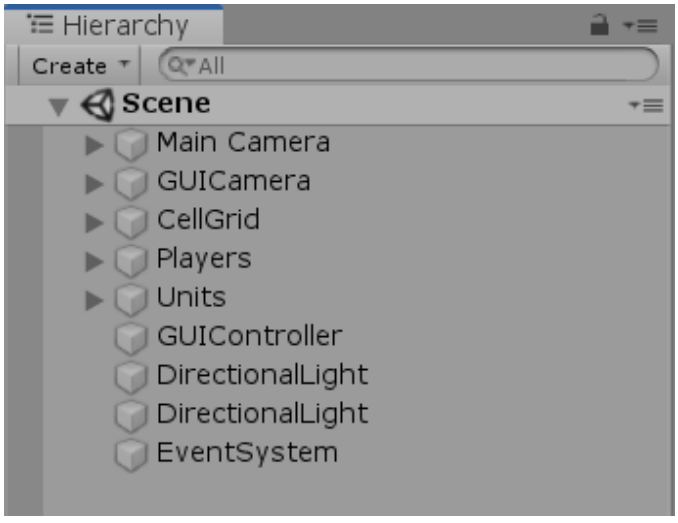

Fig. 3 - Scene hierarchy

Lights, cameras, user interface controller and event system are pretty obvious. The most important objects are CellGrid, Players and Units. Let us look into them.

3.1. CellGrid

CellGrid is the main object in the scene. It parents all the cells that the grid consists of. Fig. 4 shows CellGrid game object. As you can see, it has two scripts attached to it:

- CellGrid – Keeps track of the game, stores cells, units and players objects. It starts the game and makes turn transitions. It reacts to user interacting with units or cells, and raises events related to game progress. Basically, it's the game controller.
- CustomUnitGenerator – Script that loads units into the game.

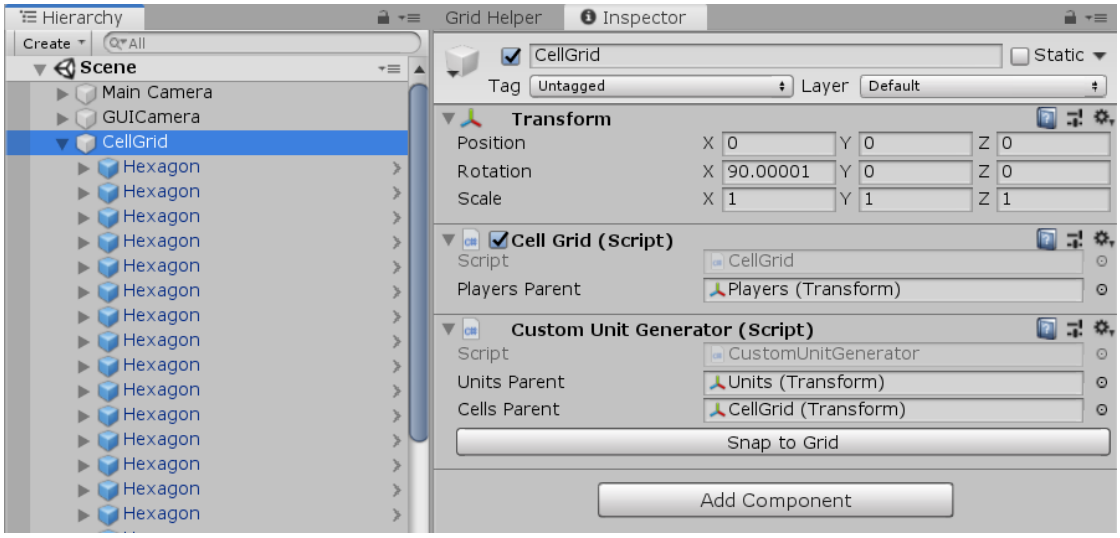

Fig. 4 - CellGrid game object

3.2. Players

Players game object holds player objects. A player is a game object with Player script attached to it. Number of players is not limited, but CellGrid script requires at least one player object to work correctly. Attribute "Player Number" must be unique to each player. It is possible to include AI players in the game by implementing `Play()` method in class derived from Player. The project contains such implementation, the AI is not very strong though.

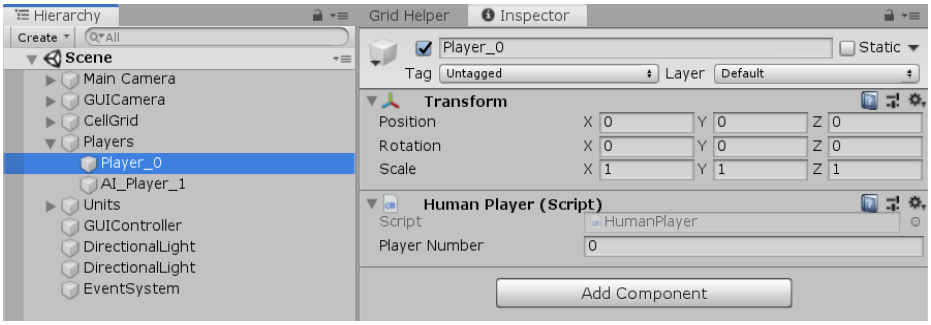

Fig. 5 - Players game object

3.3. Units

Units game object holds all units that take part in the game. Units placed outside of their parent will not work properly and will rise errors. Each unit has Player Number attribute that should correspond with Player Number attribute on Player object. Adding units that don't have any player „attached" (player with corresponding Player Number doesn't exist) is acceptable, but it will be impossible to control them.

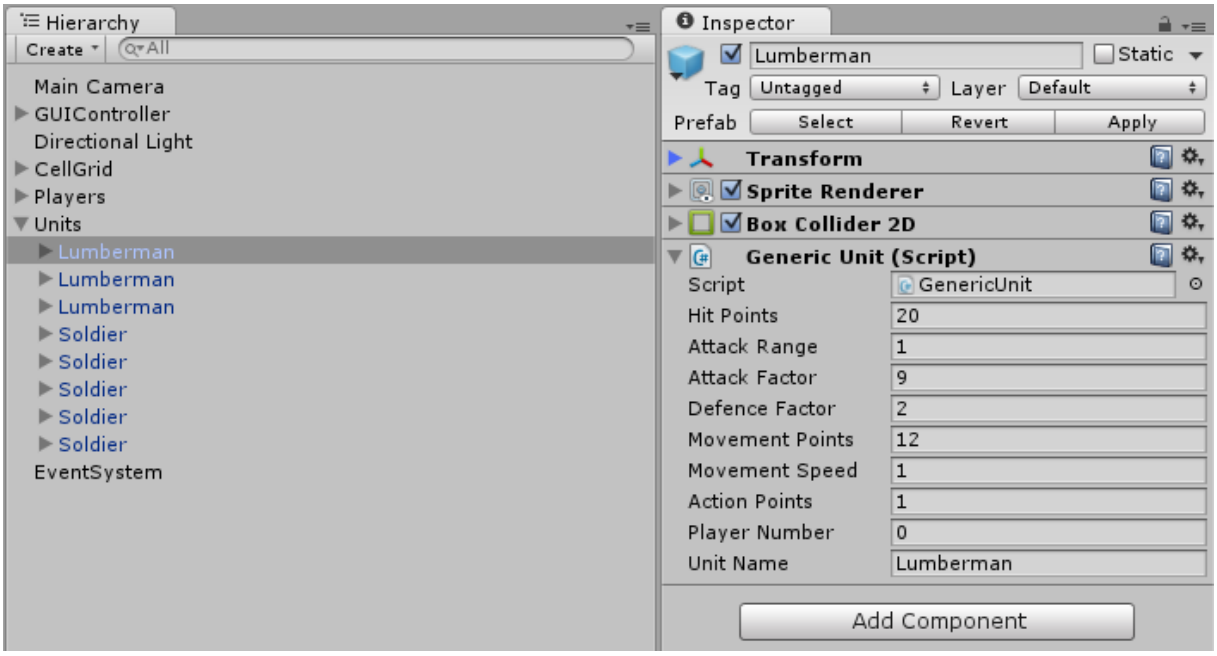

Fig. 6 - Units game object

4. Creating a scene

All this setup presented in the previous chapter may seem a bit intimidating. This is why the framework contains Grid Helper - a powerful custom editor that will prepare initial scene structure for you. In this chapter you will learn about prefabs that you need to prepare and scene generation process.

4.1. Cell prefab

Before generating a grid, you need to have a prefab of a cell. A cell is a game object with implementation of Cell script attached to it. It must also have a collider to allow mouse events to work. Sample cell prefab is presented in fig. 7 below.

There are three abstract cell classes in the project: Cell, Square and Hexagon. In your project you will be inheriting from either Square or Hexagon, depending what cell style you are going for. If you want to add another cell type, like Triangle, you would need to inherit directly from Cell class. Cell has a few abstract methods:

- `int GetDistance(Cell other)`
- `List<Cell> GetNeighbours(List<Cell> cells)`

- `void CopyFields(Cell newCell)`

Square and Hexagon classes have these three methods covered, so if you are not going for anything fancy, you don't need to worry about them. Next, there are customization methods that are also abstract and need to be coded, but at this point you may give them empty implementations. You will learn about them in the next chapter:

- `void MarkAsReachable()`
- `void MarkAsPath()`
- `void MarkAsHighlighted()`
- `void UnMark()`

Lastly, there is one more methods that you actually need to implement properly before grid generation:

- `Vector3 GetCellDimensions()`

As the name suggests, the method returns cell dimensions. It is necessary for grid generators to work - it tells them how far apart place the game objects. How do you get the dimensions? Lets say, your tile sprite is 16x16 pixels and you import with Pixels Per Unit set to 10. Therefore, the dimensions in units are 16 / 10, 16 / 10. The method would need to return Vector3(1.6f, 1.6f, 0). For 3D cells you need to check the actual 3D model size.

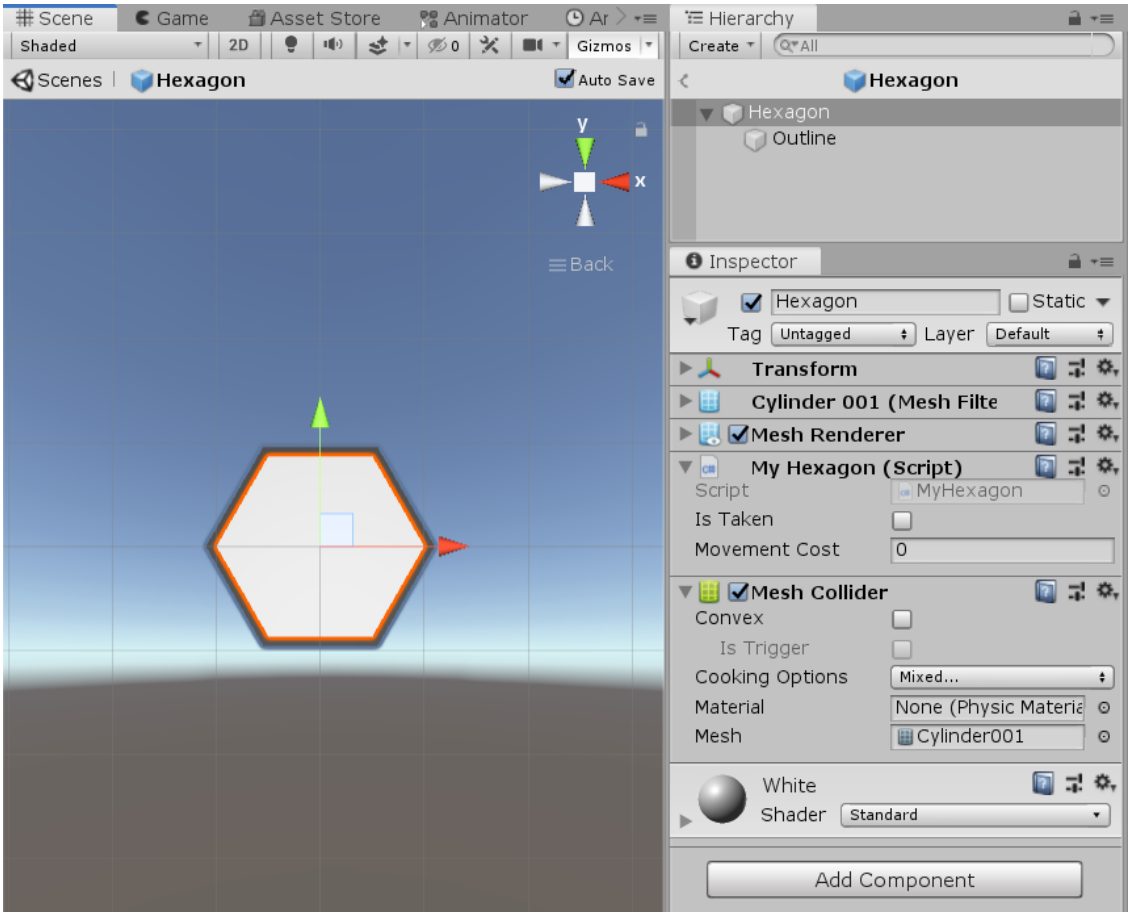

Fig. 7 - Sample cell prefab

4.2. Grid generation

If you have cell prefab prepared, you can get to grid generation. Grid generation is done with a tool named Grid Helper. You can access it by selecting Window -> Grid Helper from Unity menu. The window is shown in fig. 8.

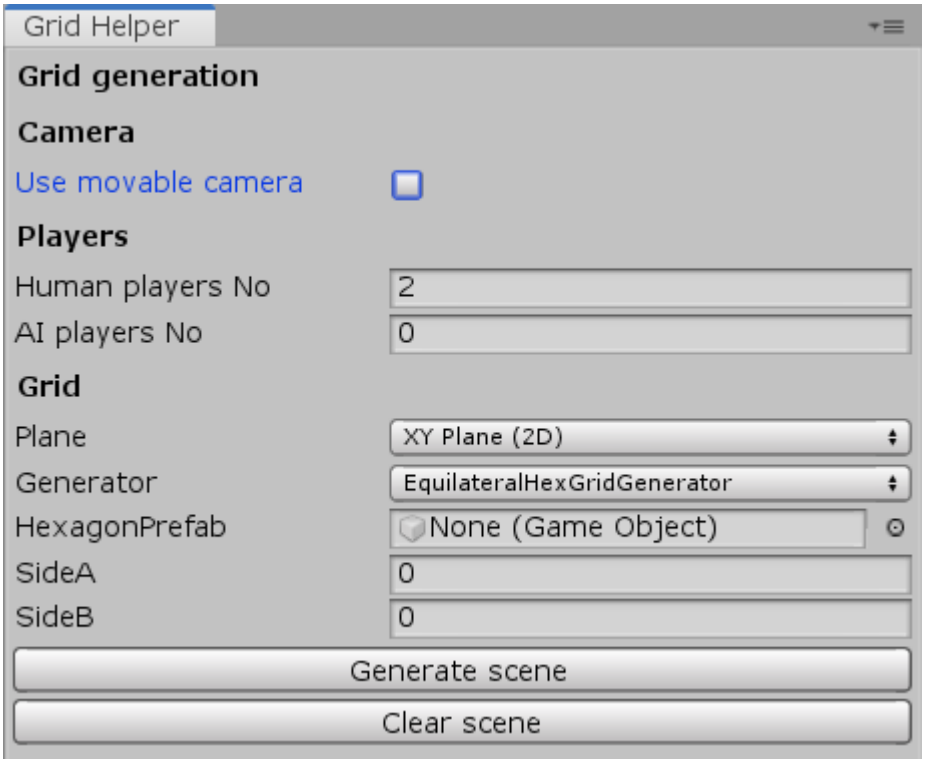

Fig. 8 - Grid helper window

The purpose of Grid Helper is to generate a basic scene structure with the parameters given by user. The meaning of the parameters is as follows:

- **Use movable camera -** whether movable camera script should be added to the scene
- **Human players No -** number of human players to generate
- **AI players No -** number of AI players to generate
- **Plane -** what plane to generate the grid in, either XY or XZ
- **Generator -** type of script used to generate the grid, basically shape of the grid. The project contains a few generators for hex grids: RectangularHexGridGenerator, HexagonalHexGridGenerator, EquilateralHexGridGenerator, TriangularHexGridGenerator and RectangularSquareGridGenerator for square grids. each generator has its own parameters - cell prefab and grid dimensions. You can create your own generators by implementing ICellGridGenerator and they will be visible in the dropdown list.

When you are done with setting the parameters, click “Generate scene“ button. The script will generate all necessary game objects described in previous chapter, like CellGrid, Players and Units. Additionally, it will make sure that there is a main camera in the scene, set it up so it shows the map, add some lighting and a simple GUI controller for making turn transitions.

4.3. Tile painting

Tile Painter is a new feature of Grid Helper introduced in TBS Framework v2.0. The purpose of this tool is to paint over the grid with different kinds of cell prefabs. It is available in Grid Helper window, which you open by selecting Window -> Grid Helper in Unity menu. The interface is shown in fig. 9 below.

To use the tool, select brush radius, assign tile prefab that you want to use and enter edit mode with a button at the bottom of the interface. You can select the prefab by dragging it into the field in inspector, or simply selecting it in project explorer - the second option works only when tile edit mode is on. When in use, the tool takes control over scene view - you won't be able to select, move or interact with objects in the scene until you exit edit mode. Fig. 10 shows TIle Painter in action. A red circle indicating painting radius is drawn on the scene. It is worth noting that the tool supports native Unity undo / redo operations.

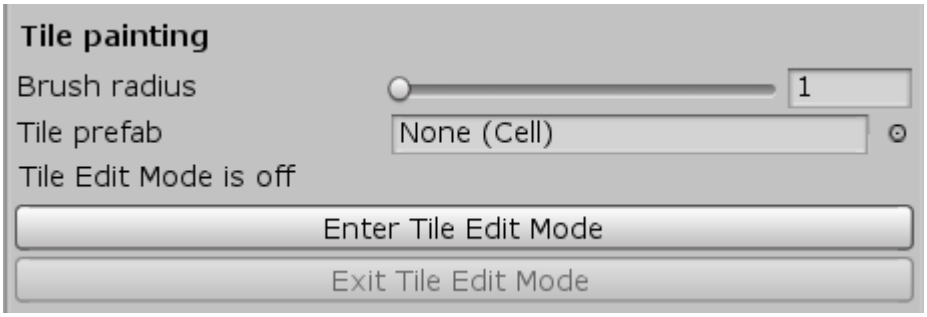

Fig. 9 - Tile Painter interface

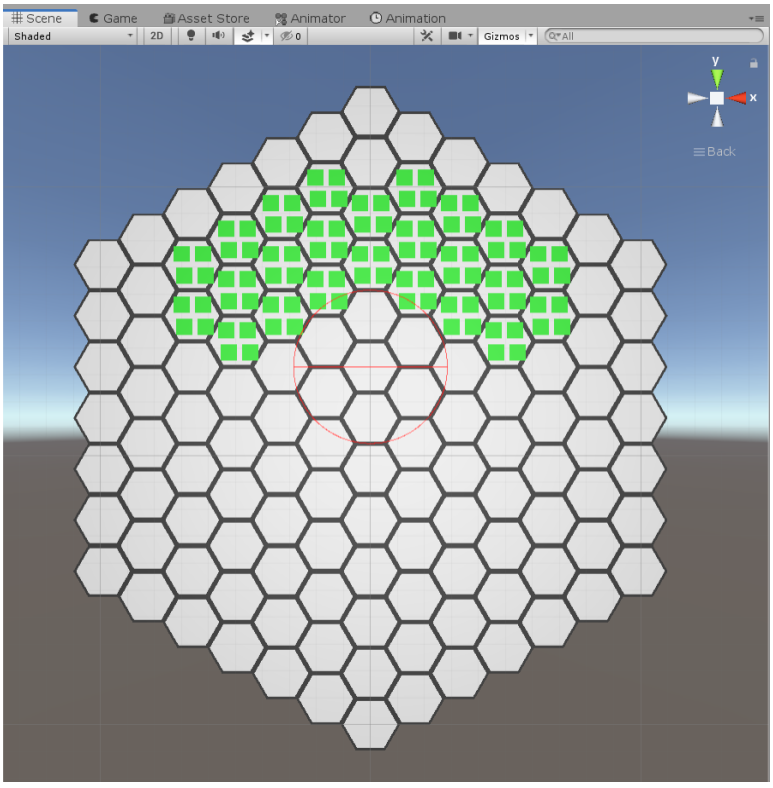

Fig. 10 - Tile Painter in use

4.4. Unit prefab

Another prefab that you need to prepare is unit prefab. A unit is a game object with implementation of Unit class and collider attached to it. Similarly to Cell, Unit has some abstract appearance customization methods that needs to be implemented in your derived classes. Again, you can code empty implementations for now. The methods are as follows:

- `void MarkAsDefending(Unit aggressor)`
- `void MarkAsAttacking(Unit target)`
- `void MarkAsDestroyed()`
- `void MarkAsFriendly()`
- `void MarkAsReachableEnemy()`
- `void MarkAsSelected()`
- `void MarkAsFinished()`
- `void UnMark()`

You will learn about these and remaining virtual methods in the next chapter.

4.5. Unit painting

Unit Painter is another new feature introduced in TBS Framework v2.0. It was designed to help with populating the grid with units. It is available in Grid Helper window accessible in Unity menu under Window -> Grid Helper. The interface is shown in fig. 11.

To use the painter, input number of player that the unit will belong to and select unit prefab. Just like in Tile Painter, you can select the prefab by dragging it into the field in inspector, or selecting it in the project explorer (only when unit edit mode is on). Finally, turn unit edit mode on with the button at the bottom of the interface and use scene view to create units. Undo / redo operations are supported. Fig. 12 shows Unit Painter in action.

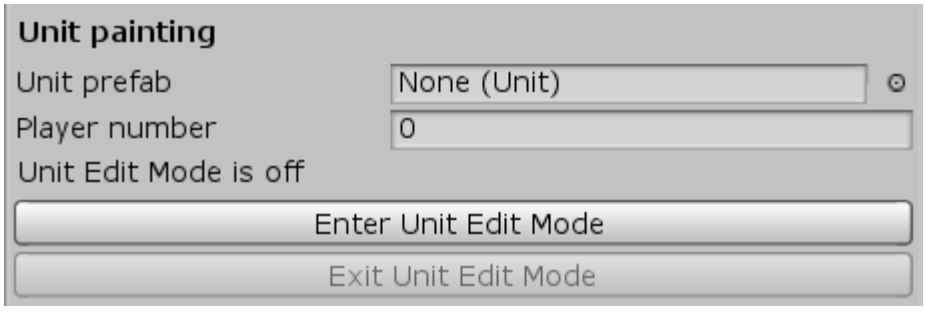

Fig. 11 - Unit Painter interface

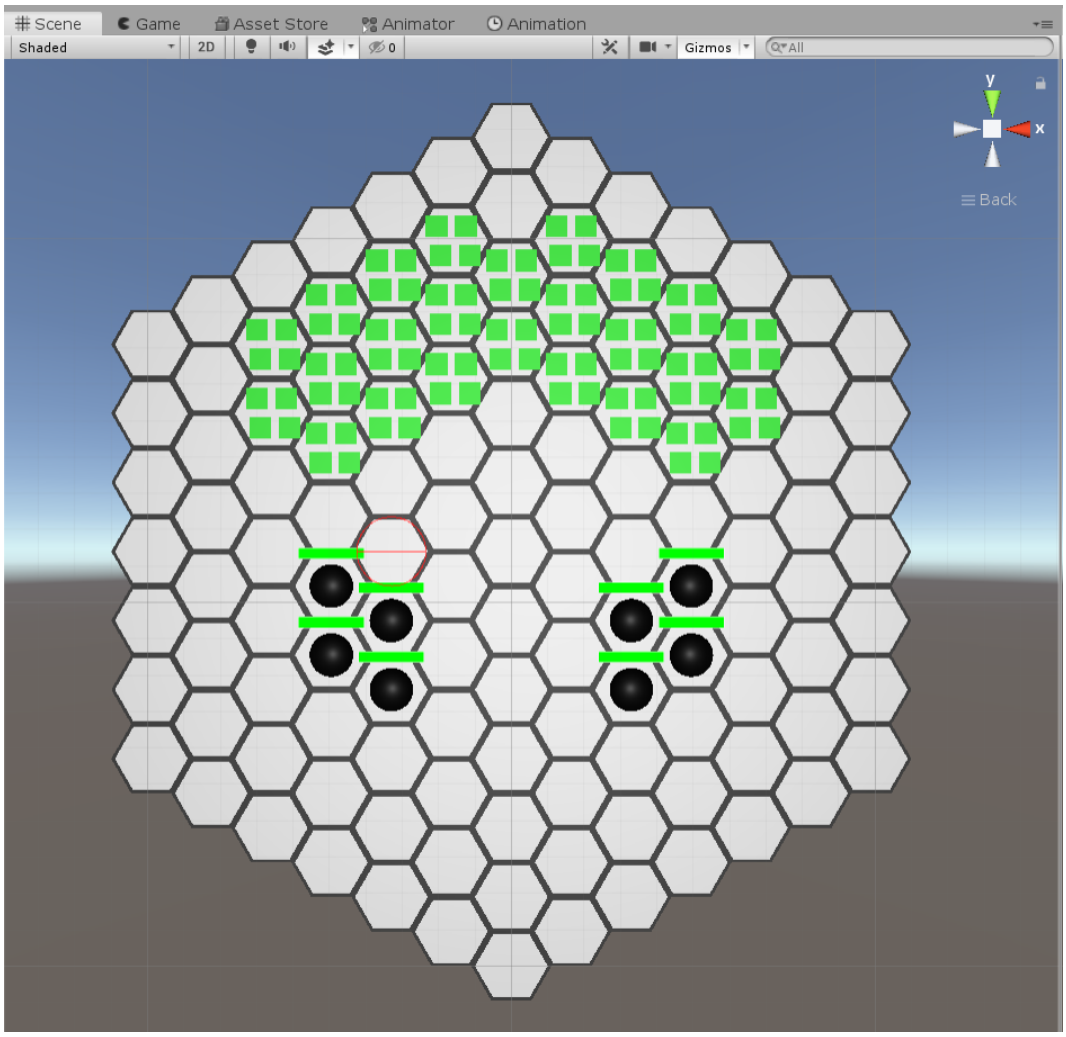

Fig. 12 - Unit Painter in use

4.6. Prefab Helper

To create you level you will probably need quite a lot of prefabs. Unity does not allow to save multiple prefabs at once. Prefab helper was created to address this issue. It is available in Grid Helper window. Currently its only function is to save multiple game objects that are selected in hierarchy into prefabs. The process is as follows:

1. Select game objects that you want to save as prefabs
2. Click “Selection to prefabs“ button on the Prefab helper interface
3. Select destination folder in dialog window

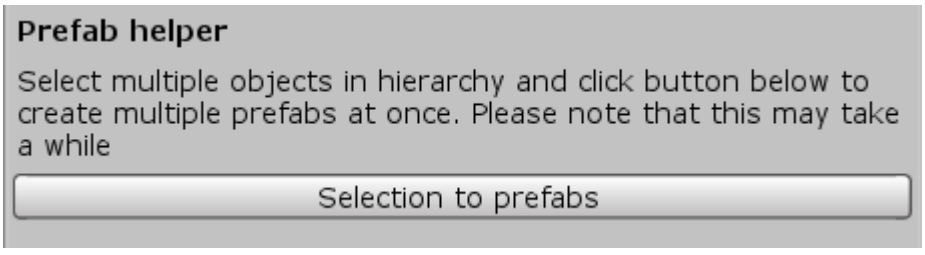

Fig. 13 - Prefab Helper interface

5. Customization

The strength of TBS Framework is the ability to customize it easily. In the project, I provided 4 examples, each with different kind of style.

5.1. Cell customization

First let's look at cells that I created, shown in Fig. 14. As you can see, they can be 3D objects, sprites, hexagons or squares. It is also possible to implement different kind of cells – triangular for example.

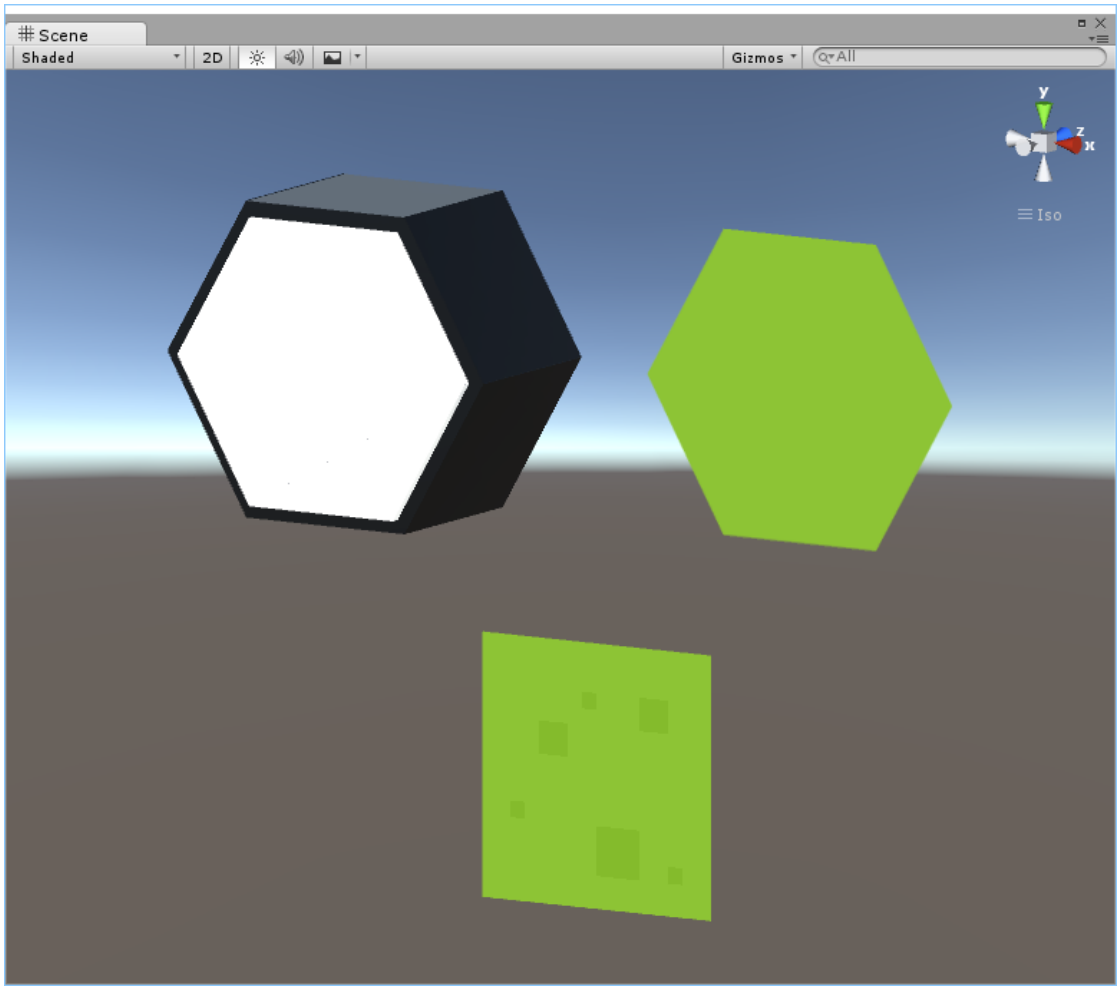

Fig. 14 - Different kinds of cells

Cells can be programmed to change appearance depending on state that they're in. To do so, just override appropriate methods in class derived from Cell. Available methods are:

- `MarkAsReachable()`
- `MarkAsPath()`
- `MarkAsHighlighted()`
- `UnMark()`

Let’s look at cells that are in different states, shown in Fig. 15. From left the cells’ appearance is: normal, highlighted (when mouse is over the cell), marked as reachable (by currently selected unit), marked as path (of currently selected unit). I used a lot of grey, yellow and green here because I think they look nice, but of course you are not restricted to it. The „markers” don't have to be colours – they can be images, particle effects or whatever you can think of.

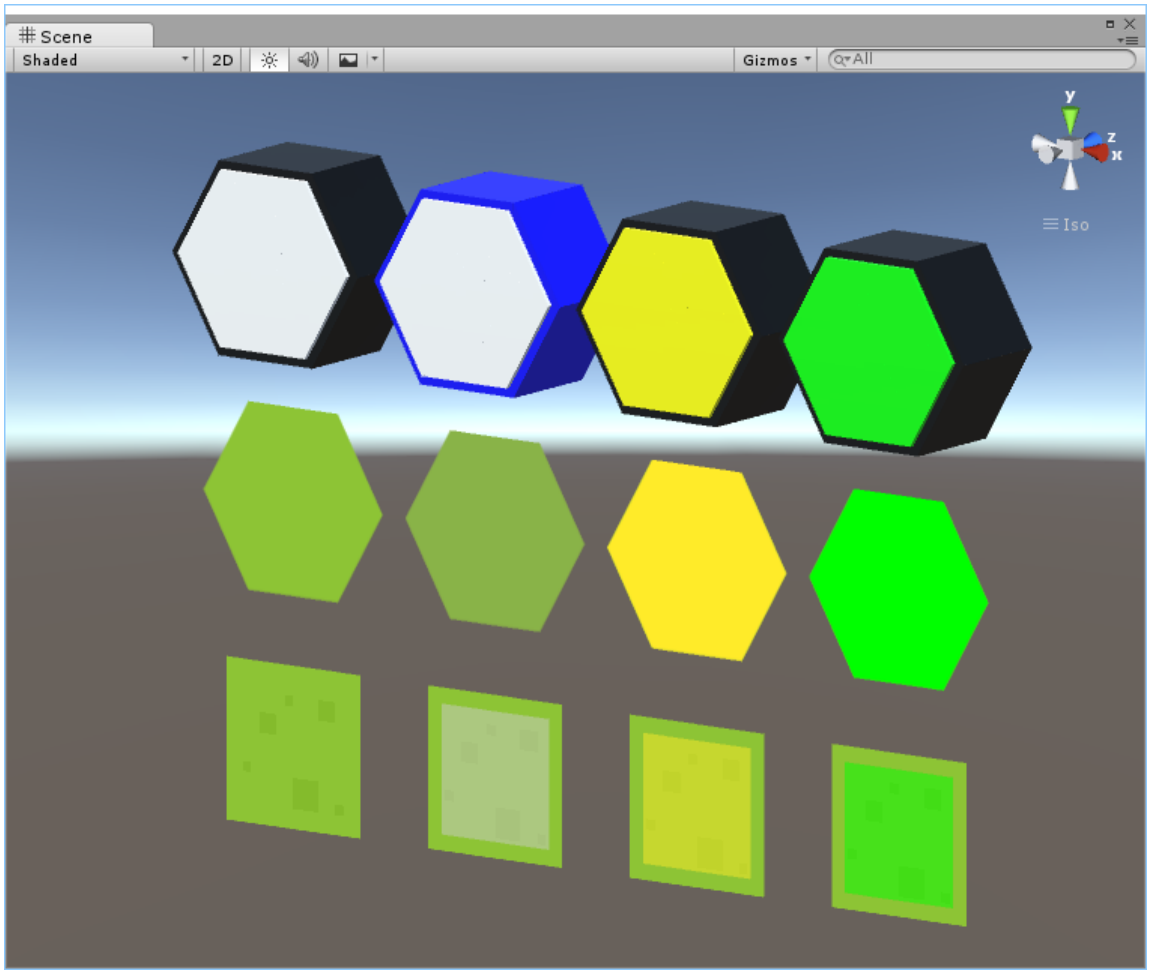

Fig. 15 - Cells appearance in different states

There are two abstract methods that can change cell grid behaviour as a whole:

- `GetDistance(Cell other)`

Returns distance to cell given as parameter. Manhattan norm is used by default. Distance is used in pathfinding and calculating attack range.

- `GetNeighbours(List<Cell> cells)`

Method returns a list of adjacent cells. Neighbouring cells are used in pathfinding functions.

5.2. Unit customization

Similarly to cells, units' appearance can also be customized by overriding appropriate methods:

- `MarkAsFriendly()`
- `MarkAsSelected()`
- `MarkAsReachableEnemy()`
- `MarkAsFinished()`
- `MarkAsDefending()`
- `MarkAsAttacking()`
- `MarkAsDestroyed()`
- `UnMark()`

Units in different states are shown in Fig. 16. From left units appearance is: normal, marked as friendly unit, marked as selected unit, market as enemy unit that is in range of attack, marked as finished (can't move and attack in this turn anymore).

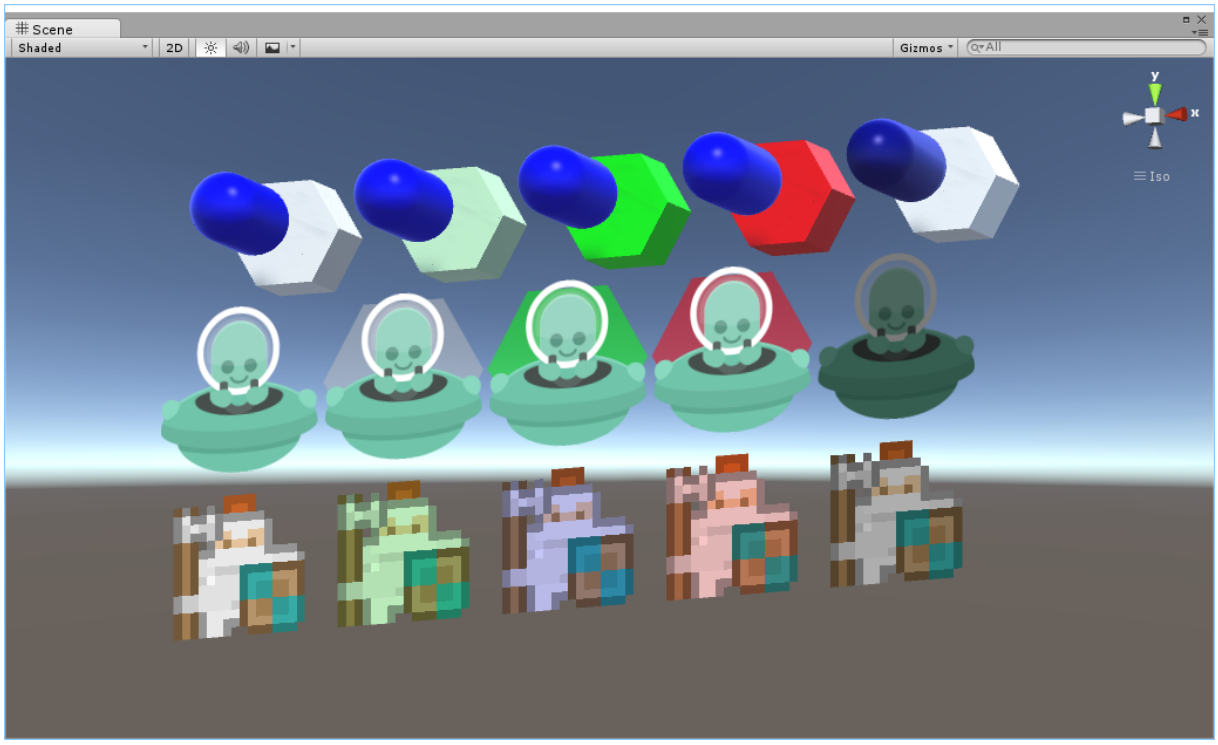

Fig. 16 - Units appearance in different states

Apart from appearance, unit's behaviour can also be customized in various ways. Methods available to override are:

- `OnMouseDown()`
- `OnMouseEnter()`
- `OnMouseExit()`

These three are called by Unity when user clicks on the unit, highlight it with mouse and when mouse pointer exits unit's collider. The methods invoke UnitClicked, UnitHighlighted and UnitDehighlighted events.

- `OnUnitSelected()`
- `OnUnitDeselected()`

Methods called when unit is selected and deselected. Being selected means that current player clicked on the unit and is about to use it.

- `OnTurnStart()`
- `OnTurnEnd()`

Called on all units belonging to a player when he starts and ends his turn.

- `OnDestroyed()`

Called when unit "dies".

- `IsUnitAttackable(Unit other, Cell sourceCell)`

Returns a boolean value indicating if unit can attack another unit, given as parameter, from a cell given as parameter.

- `DealDamage(Unit unitToAttack)`

This is where you calculate how much damage an attack should cause to another unit.

- `AttackActionPerformed()`

Method is called after attack was performed.

- `Defend(Unit aggressor, int damage)`

This is where you calculate how much damage was actually caused by an attack. Why is it separated from DealDamage method? You may want to apply some additional factors - like terrain defence bonus - that will affect the damage.

- `DefenceActionPerformed()`

Method called after defence action was performed.

- `IsCellMovableTo(Cell cell)`
- `IsCellTraversable(Cell cell)`

These methods return a boolean indicating if a unit can end its move on a given cell and if it can move through given cell.

- `Move(Cell destination, List<Cell> path)`

Method moves a unit from one cell to another, using given path

Remember that these methods are virtual and have some default behaviour implemented. In your derived code you should call base implementation or integrate base code in your class.

5.3. Customization examples

Below I present a few examples of project customization. First, let's see how I approached cell visual customization. Sample cell prefab is shown in fig. 17.

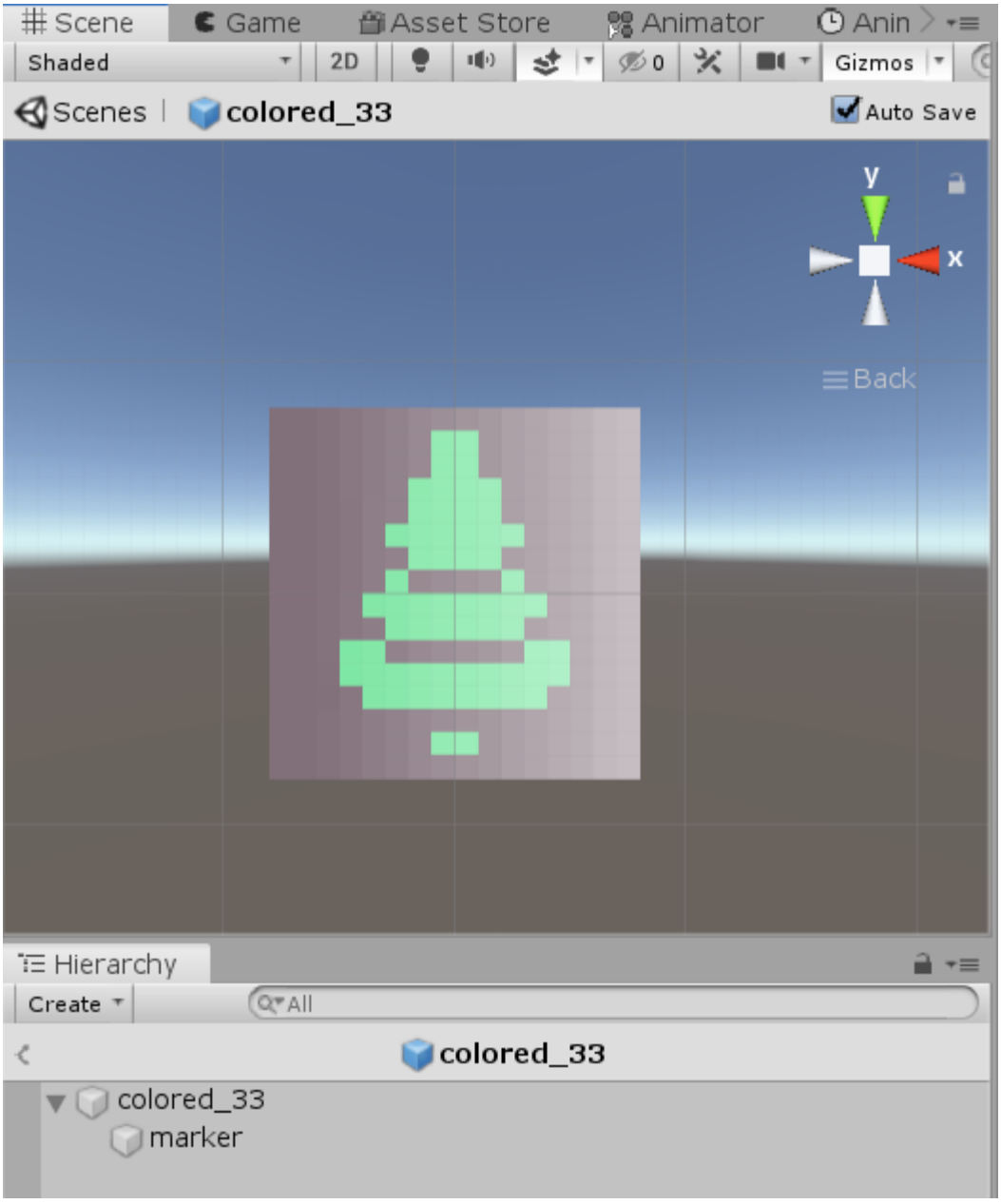

Fig. 17 - Sample cell prefab

The prefab consists of parent object that holds components required by Cell (Cell script and 2D collider) and child object called "marker". Marker is pale overlay that you can see in the picture. This is how it looks like in the code:

```
public override void MarkAsReachable()
{
    SetColor(new Color(0.4f, 0.7f, 1f, 0.8f));
}
public override void MarkAsPath()
{
    SetColor(new Color(0, 1, 0, 0.5f));
}
public override void MarkAsHighlighted()
{
    SetColor(new Color(0.8f, 0.8f, 0.8f, 0.5f));
}
public override void UnMark()
{
    SetColor(new Color(1, 1, 1, 0));
}
private void SetColor(Color color)
{
    var highlighter = transform.Find("marker");
    var spriteRenderer = highlighter.GetComponent<SpriteRenderer>();
    if (spriteRenderer != null)
    {
        spriteRenderer.color = color;
    }
}
```

Basically what this does is set marker to different color in each of the states. In UnMark method marker is set to transparent. Fig. 18 shows how this works.

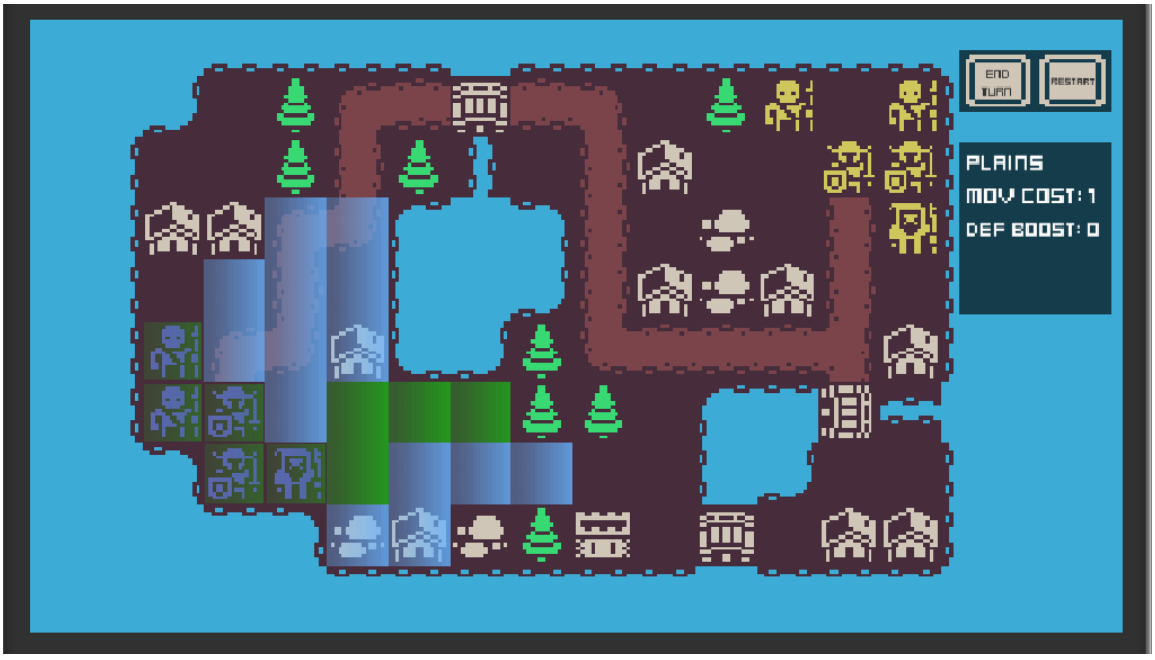

Fig. 18 - Sample cell customization

Next, let's try unit behaviour customization. In example shown in fig. 19, flying saucer is allowed to move over water and obstacles, while units that are on the ground are not.

Fig. 19 - Flying saucer moving over water

The steps to achieve such effect are as follows:

- Create class derived from Cell that has two new attributes:

```
public GroundType GroundType;
public bool IsSkyTaken;
//Indicates if a flying unit is occupying the cell.
```

Where `GroundType` is an enum that looks like this:

```
public enum GroundType
{
    Land,
    Water
};
```

I called this class `MyOtherHexagon`

- Create class derived from Unit, that will represent alien unit. It should override methods `IsCellMovableTo` and `IsCellTraversable`. I called this class `Alien`:

```
public override bool IsCellMovableTo(Cell cell)
{
    return base.IsCellMovableTo(cell) &&
    (cell as MyOtherHexagon).GroundType != GroundType.Water;
    //Prohibits moving to cells that are marked as water.
}
public override bool IsCellTraversable(Cell cell)
{
    return base.IsCellTraversable(cell) &&
    (cell as MyOtherHexagon).GroundType != GroundType.Water;
    //Prohibits moving through cells that are marked as water.
}
```

- Create class derived from Alien, that will represent a flying alien unit. This time we have to override a few more methods, as there is more things to take care of. I called this class `FlyingAlien`:

```
public void Initialize()
{
    base.Initialize();
    (Cell as MyOtherHexagon).IsSkyTaken = true;
}

public override bool IsCellTraversable(Cell cell)
{
    return !(cell as MyOtherHexagon).IsSkyTaken;
    //Allows unit to move through any cell that is not occupied by a flying unit.
}

public override void Move(Cell destinationCell, List<Cell> path)
{
    (Cell as MyOtherHexagon).IsSkyTaken = false;
    (destinationCell as MyOtherHexagon).IsSkyTaken = true;
    base.Move(destinationCell, path);
}

protected override void OnDestroyed()
{
    (Cell as MyOtherHexagon).IsSkyTaken = false;
    base.OnDestroyed();
}
```

As you can see, this is pretty straightforward. Another example could be creating unit countering system, similar to rock – paper – scissor game. Example scenes 1 contains implementation of such system. To get that effect, simply create three subclasses of Unit, and override their `Defend()` methods:

```
public class Spearman : MyUnit
{
    protected override int Defend(Unit other, int damage)
    {
        var realDamage = damage;
        if (other is Archer)
            realDamage *= 2;//Archer deals double damage to spearman.

        return realDamage - DefenceFactor;
    }
}
public class Archer : MyUnit
{
    protected override int Defend(Unit other, int damage)
    {
        var realDamage = damage;
        if (other is Paladin)
            realDamage *= 2;//Paladin deals double damage to archer.

        return realDamage - DefenceFactor;
    }
}
public class Paladin : MyUnit
{
    protected override int Defend(Unit other, int damage)
    {
        var realDamage = damage;
        if (other is Spearman)
            realDamage *= 2;//Spearman deals double damage to paladin.

        return realDamage - DefenceFactor;
    }
}
```

Last thing that I would like to cover here is user interface. The idea was to base it entirely on events. What you should do, is give your `GUIController` structure similar to this shown in Fig. 20.

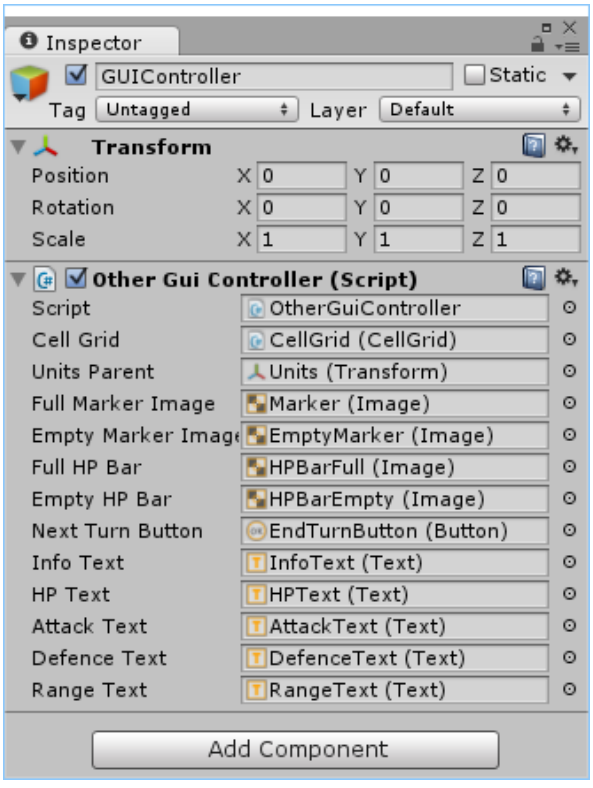

Fig. 20 - GUIController structure

The most relevant attributes here are Cell Grid and Units Parent. They allow you to subscribe to CellGrid's and units' events, and then define how UI should react to them. For complete list of available events, please refer to the code. Three examples of UI can be seen in fig. 2 fig. 18 and fig. 19. Another approach is shown in fig. 21.

Fig. 21 - Different kind of UI

6. Tutorial

In this section we will go through the process of creating the simplest possible scene from scratch. The scene will consist of grid of cube cells, cube units, and cube obstacles. You can find finished scene in Examples / Tutorial folder.

1. First thing you want to do is create a new scene in Unity editor.
2. Create a cube by clicking GameObject -> 3D Object -> Cube in Unity editor. This will be our cell prefab. Note that the cube has a Box Collider attached to it by default. Otherwise you would have to attach a collider yourself.
3. Now it's time to do some coding. Create new script by clicking Create -> C# Script in Project panel. Give the script a name, for example SampleSquare.
4. SampleSquare should inherit from Square class and override some methods responsible for cell's appearance. We will make it change its colour to grey when highlighted, yellow to indicate that it is reachable and green to mark it as path. The code looks like this:

```
class SampleSquare : Square
{
    public override Vector3 GetCellDimensions()
    {
        return GetComponent<Renderer>().bounds.size;
    }

    public override void MarkAsHighlighted()
    {
        GetComponent<Renderer>().material.color = new Color(0.75f, 0.75f, 0.75f);
    }

    public override void MarkAsPath()
    {
        GetComponent<Renderer>().material.color = Color.green;
    }

    public override void MarkAsReachable()
    {
        GetComponent<Renderer>().material.color = Color.yellow;
    }

    public override void UnMark()
    {
        GetComponent<Renderer>().material.color = Color.white;
    }
}
```

5. Attach the script to the cube, set movement cost parameter to 1, and drag it to project explorer to create a prefab.
6. Open Grid helper by selecting Window -> Grid helper
7. Fill in the parameters in the Grid helper window. Correct parameter values are shown in fig. 22

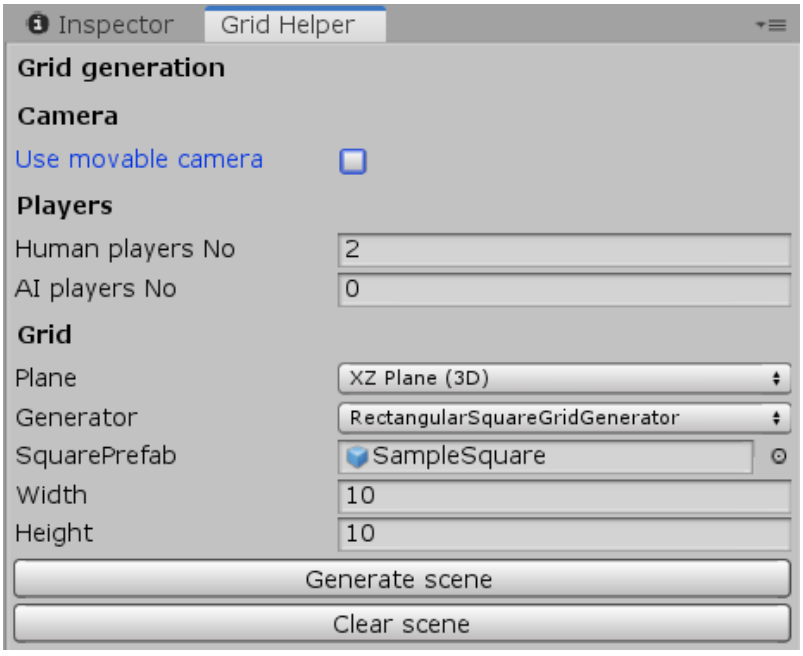

Fig. 22 - Grid helper with parameters filled in

8. Once the parameters are filled in, click Generate scene button in Grid helper window. Scene hierarchy at this point is shown in fig. 23, and scene view is shown in fig. 24

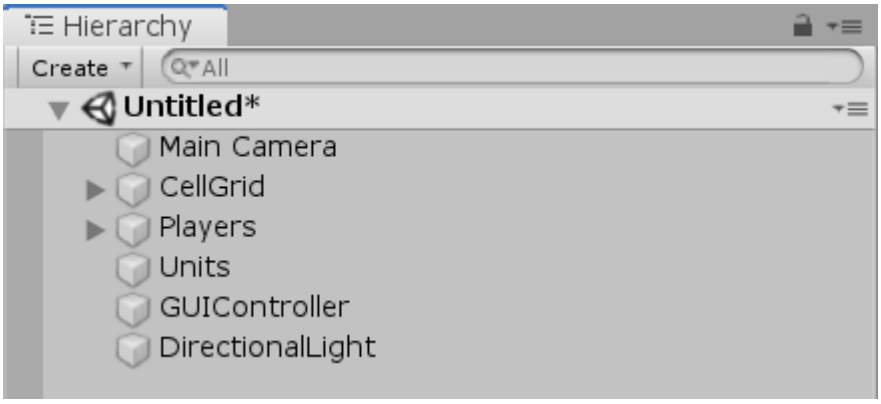

Fig. 23 - Scene hierarchy at step 8

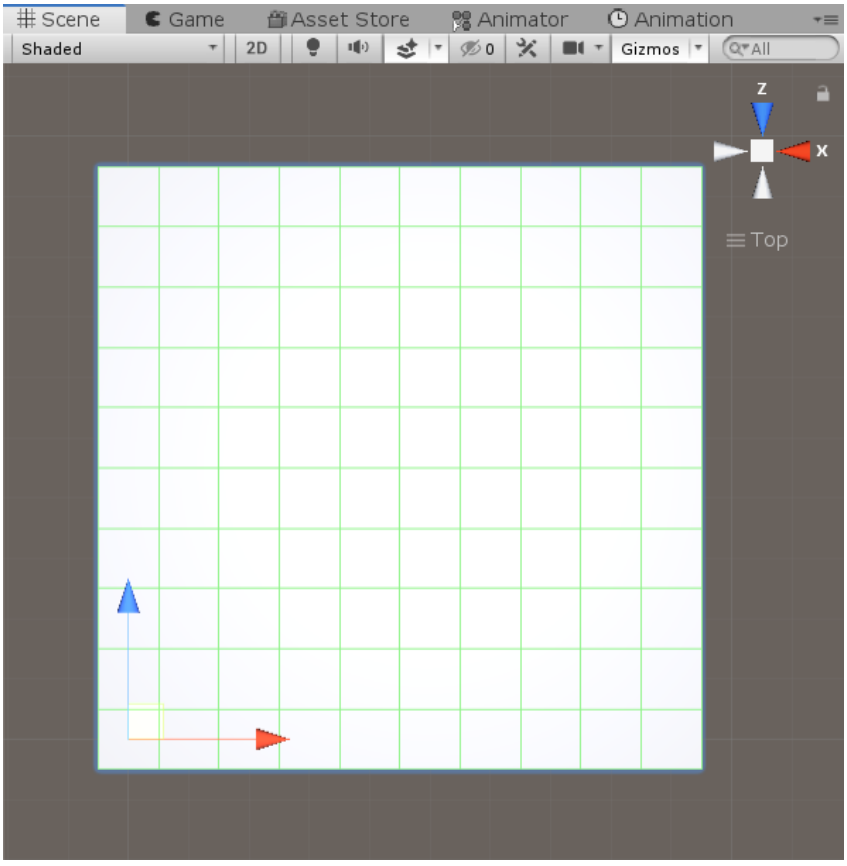

Fig. 24 - Scene view at step 8th

9. Now is the time to add units to the scene. Create new script and name it SampleUnit. The class should inherit from Unit. For the purpose of this tutorial I am omitting some functions here, but please note that all abstract methods need to be implemented. Refer to Examples / Tutorial / Scripts / SampleUnit in case of any doubt.

```
using TbsFramework.Units;
using UnityEngine;

public class SampleUnit : Unit
{
    public Color LeadingColor;
    public override void Initialize()
    {
        base.Initialize();
        transform.localPosition -= new Vector3(0, 0, 1);
        GetComponent<Renderer>().material.color = LeadingColor;
    }
    public override void MarkAsFriendly()
    {
        GetComponent<Renderer>().material.color = LeadingColor + new Color(0.8f, 1, 0.8f);
    }

    public override void MarkAsReachableEnemy()
    {
        GetComponent<Renderer>().material.color = LeadingColor + Color.red;
    }

    public override void MarkAsSelected()
    {
        GetComponent<Renderer>().material.color = LeadingColor + Color.green;
    }

    public override void UnMark()
    {
        GetComponent<Renderer>().material.color = LeadingColor;
    }
    public override void MarkAsAttacking(Unit other)
    {
    }

    public override void MarkAsDefending(Unit other)
    {
    }

    public override void MarkAsDestroyed()
    {
    }

    public override void MarkAsFinished()
    {
    }
}
```

10. Create two new materials and set them to two different colors.
11. Create two new cubes that will represent units. Attach materials and SampleUnit script to the cubes.
12. Fill in parameters in SampleUnit script. Values that I selected are shown in fig. 17, but feel free to experiment. Make sure to set Leading Color parameter of both units to different values - accordingly to colors of materials that you created.

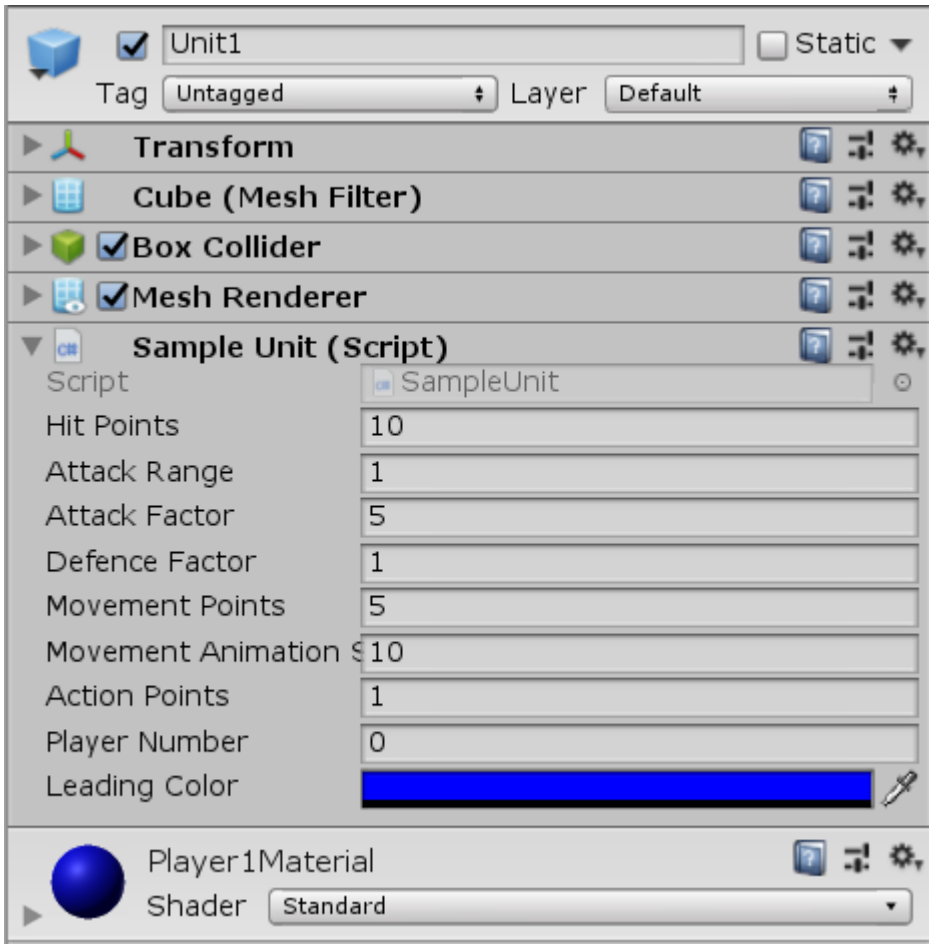

Fig. 25 - SampleUnit parameters

13. We will use Unit Painter to add some units to the game. Open Grid Helper by selecting Window -> Grid Helper. Drag and drop your unit ino “Unit Prefab” field in Unit Painter. Set “Player Number” to 0 and click “Enter Unit Edit Mode“ button. Unit Painter interface is shown in fig. 26.

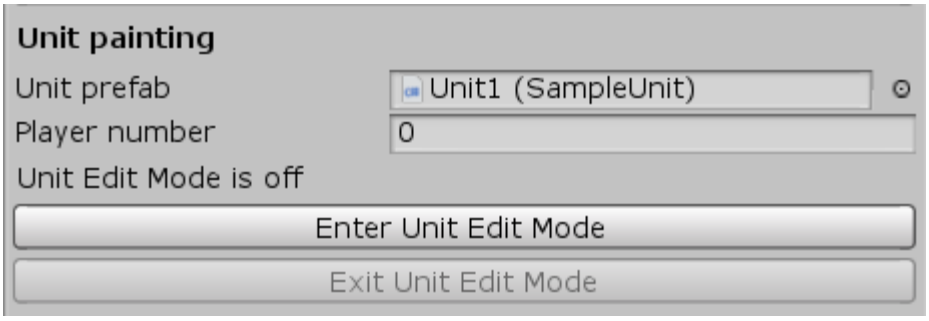

Fig. 26 - Unit Painter interface

14. While you are in Unit Edit Mode you are able to place units on the grid by clicking on cells in Scene View. Create a few units now. Change unit prefab and player number in Unit Painter interface to create units for both teams. Fig. 27 shows my scene setup at this point.

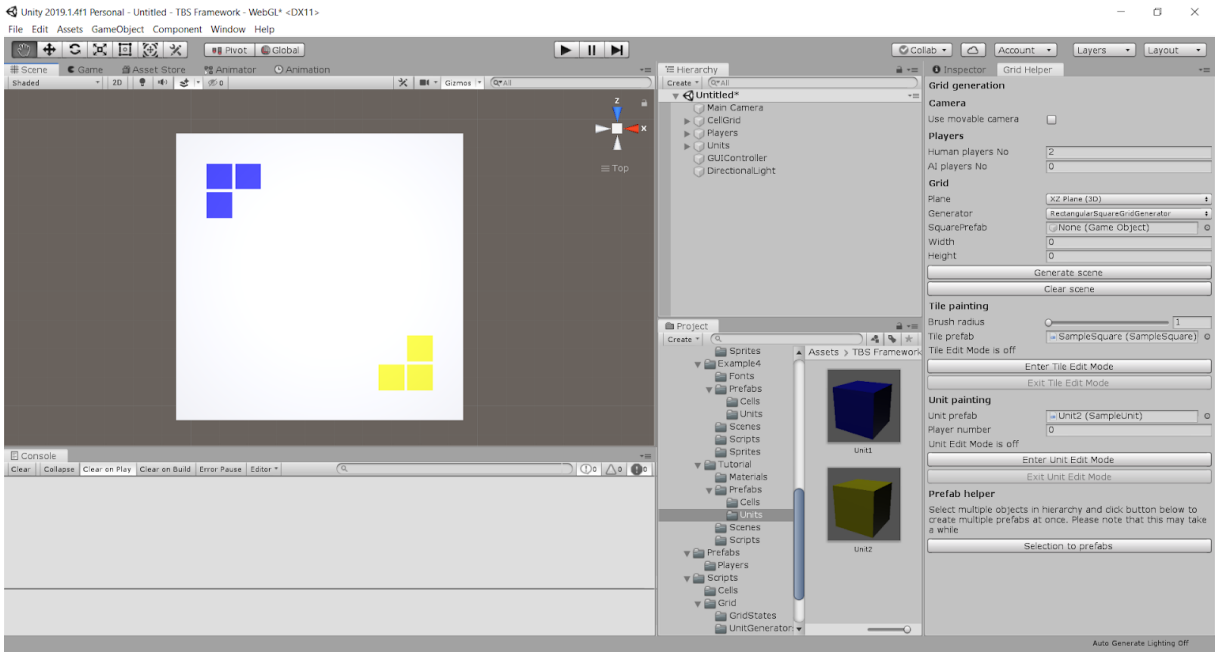

Fig. 27 - Scene setup at step 14th.

15. Let's add some obstacles to the scene. We will create another cell prefab for that purpose. Create new cube and attach black material to it. Duplicate existing cell prefab and attach the cube. The prefab is shown in fig. 28. Remember to set IsTaken field on the new prefab to true.

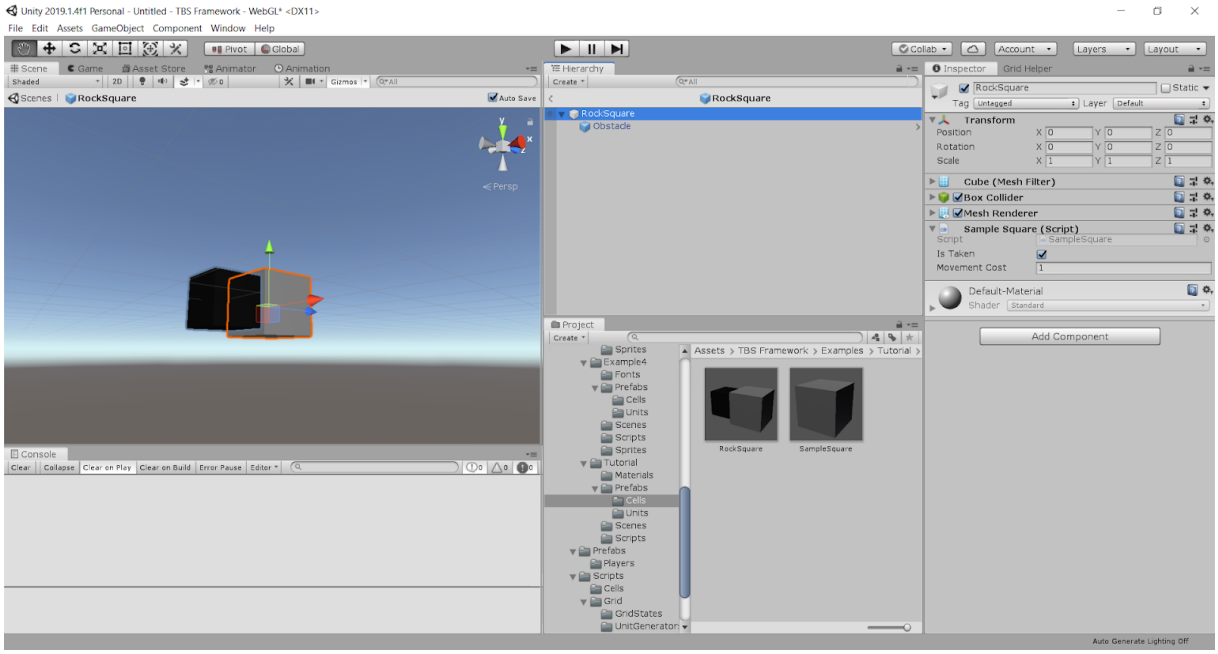

Fig. 28 - Another cell prefab

16. We will use Tile Painter to add the cell that we prepared to the grid. Tile Painter is accessible from Grid Helper window, just like Unit Painter. Drag and drop your cell prefab int “Tile Prefab“ field and click on “Enter Tile Edit Mode“ button. Use Scene View to place some new cells. Fig. 29 shows Tile Painter interface.

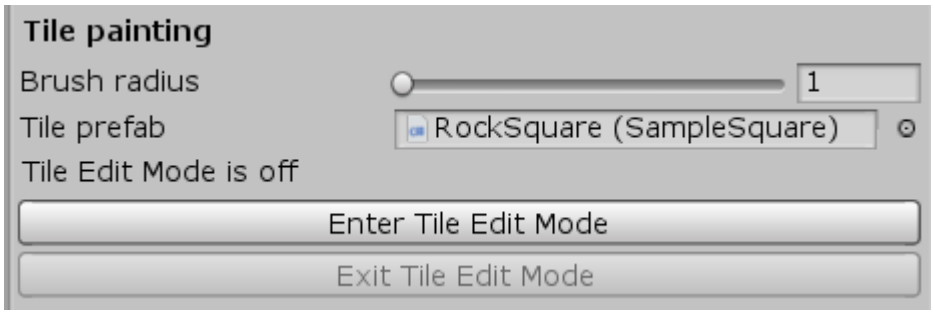

Fig. 29 - Tile Painter interface

17. Grid helper script attaches a very simple GUI controller script to the scene, so you don't need to worry about that. The script is concerned only with making turn transitions, which is done by pressing N key on the keyboard. The code goes like this:

```
public class GUIController : MonoBehaviour
{
   public CellGrid CellGrid;
    void Update ()
    {
        if(Input.GetKeyDown(KeyCode.N))
        {
            CellGrid.EndTurn();
            //User ends his turn by pressing "n" on keyboard.
        }
    }
}
```

Fig. 28 - Simple GUI controller code

That concludes the tutorial. Fig. 29 shows the scene setup after the last step. Fig. 30 shows the level running. The created scene is playable, though perhaps not particularly interesting to play. It's up to you to change it.

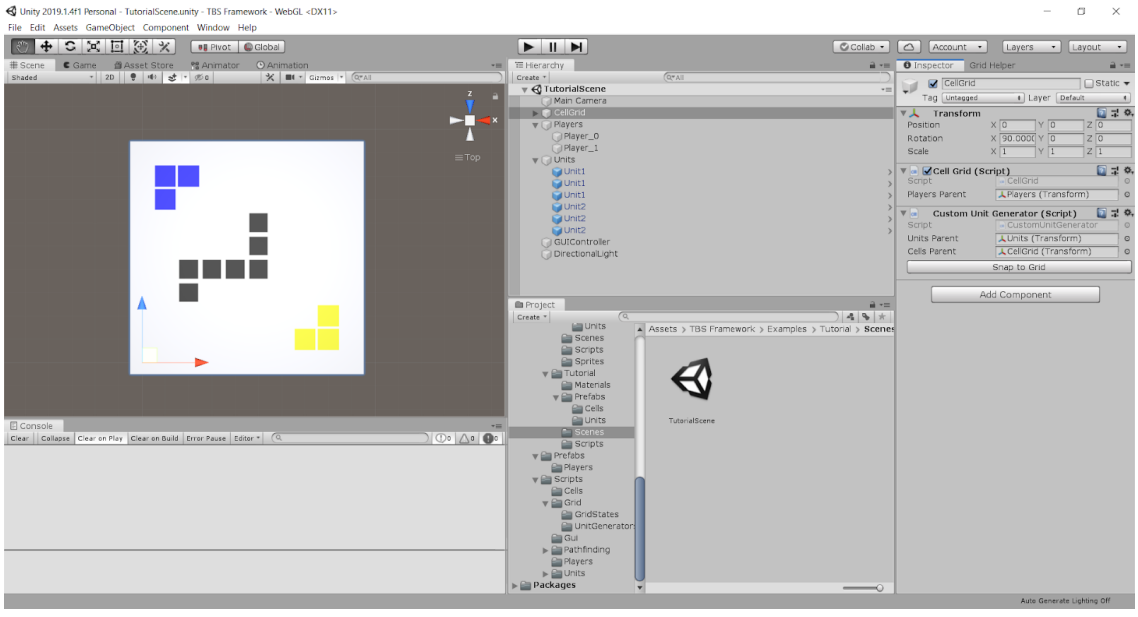

Fig. 29 - Final setup of the scene

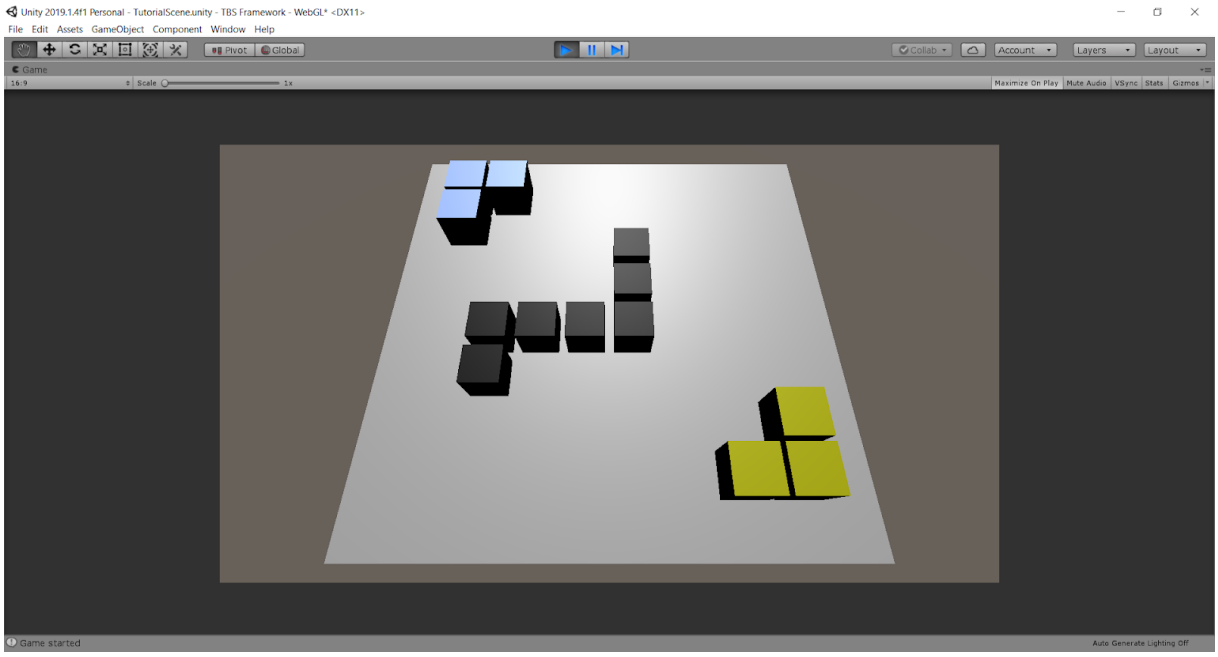

Fig. 30 - Finished scene

7. License

Turn Based Strategy Framework is covered by the same license as all the other assets on Unity Asset Store. Please refer to https://unity3d.com/legal/as_terms for full text, I will quote only relevant fragment here:

"2.2.1 Non-Restricted Assets. The following concerns only Assets that are not Restricted Assets:

Licensor grants to the END-USER a non-exclusive, worldwide, and perpetual license to the Asset to integrate Assets only as incorporated and embedded components of electronic games and interactive media and distribute such electronic game and interactive media. Except for game services software development kits ("Services SDKs"), END-USERS may modify Assets. END-USER may otherwise not reproduce, distribute, sublicense, rent, lease or lend the Assets. It is emphasized that the END-USERS shall not be entitled to distribute or transfer in any way (including, without, limitation by way of sublicense) the Assets in any other way than as integrated components of electronic games and interactive media. Without limitation of the foregoing it is emphasized that END-USER shall not be entitled to share the costs related to purchasing an Asset and then let any third party that has contributed to such purchase use such Asset (forum pooling)."

8. Support

Please feel free to contact me with any questions at crookedhead@outlook.com. Apart from that, there is a Unity Forum thread where I provide support at https://forum.unity.com/threads/turn-based-strategy-framework.704129/. Usually I reply within 48 hours. You can be sure that as long as the Turn Based Strategy framework is on the Unity Asset Store, the support will be there.

9. Conclusion

In this document I gave a description that should be sufficient for you to start creating your own games with this framework. If this is not enough, please study comments on the code and sample scenes that I provided. I will be happy to hear your opinions about the API or my coding, suggestions or ideas for new features. Any feedback will be appreciated. I hope you find my work useful.

10. References

[1] Kenney, 1 bit pack, https://kenney.nl/assets/bit-pack
[2] Kenney, Roguelike Characters, http://www.kenney.nl/assets/roguelike-characters
[3] Kenney, Roguelike/RPG Pack, http://www.kenney.nl/assets/roguelike-rpg-pack
[4] Kenney, Alien UFO Pack, http://www.kenney.nl/assets/alien-ufo-pack
[5] Kenney, Hexagon Tiles, http://www.kenney.nl/assets/hexagon-tiles
[6] Kenney, UI Pack, http://www.kenney.nl/assets/ui-pack
[7] Kenney, Kenney Fonts, http://kenney.nl/assets/kenney-fonts

APPENDIX A - Upgrade from v1.1.2 to v2.0

Version 2.0 of the project introduces changes that break compatibility with its previous versions. I didn't test upgrading very extensively, so I would advise to leave your old projects as is and use v2.0 for your new projects. Nevertheless, please find upgrade instructions below.

1. **First of all, make backup of your project before upgrading**

2. Upgrade your project to Unity 2018 or above

3. Add namespaces to your code

In v1.1.2 all scripts were thrown in Core folder without any namespaces. It was confusing to navigate and could cause collisions with code from other sources. These issues were fixed in v2.0. Your IDE will help you with assigning appropriate namespaces.

4. Fix Defend and DealDamage methods

In v2.0 DealDamage only returns damage that should be caused and Defend only returns damage that was actually caused to a unit. If you need to add some code after damage is applied, there are two new methods: AttackActionPerformed and DefenceActionPerformed.

5. Override MovementAnimation in classes derived from Unit

In v2.0 movement is done in XY plane, while in v1.2.1 it was XZ plane. Simply copy the code from MovementAnimation in Unit to your derived class and change

```
Vector3 destination_pos = new Vector3(cell.transform.localPosition.x, cell.transform.localPosition.y,
transform.localPosition.z);
```

to

```
Vector3 destination_pos = new Vector3(cell.transform.localPosition.x, transform.localPosition.y,
cell.transform.localPosition.z);
```

6. In your scenes, CellGrid gameobject will have missing scripts. Just add CellGrid.cs and CustomUnitGenerator.cs and fill in their parameters.

7. Player prefabs will also have missing scripts. Assign HumanPlayer.cs and NaiveAiPlayer.cs to appropriate prefabs.

8. Lastly, MovementPoints and MovementAnimationSpeed fields will be zeroed out in unit prefabs, because type and name of the fields changed. You need to reassign them.

The steps above will make your old scenes playable. To generate new scenes it is important to know that cell prefabs should be in XY plane now. Rotating your old prefabs on x axis should do the trick.