CROOKED HEAD

Turn Based Strategy Framework

Documentation

Version 1.0.1

1. Introduction

This project is a highly customizable framework for turn based strategies. It allows to create custom shaped cell grids, place objects like units or obstacles on it and play a game 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 most important files are contained in Scripts/Core folder. If you don't care about the examples or want to start from scratch, you can remove all the rest. The Core folder contains 40 scripts, but to set up a basic scene you are going to need not more than 6. Scripts that extend the Unity Editor are stored in Editor folder. The code has comments on it, so I'm not going to explain it here.

The ExampleAssets folder contains assets that I used in example scenes. In some of the pictures, you can see trees and rocks from Low poly styled trees [1] and Low poly styled rocks [2], available in Asset Store. These packages are NOT included in this project though, because it is prohibited by Unity. In actual scenes, they are replaced with simplified objects made by me. Apart form that I used: Roguelike Characters [3], Roguelike/RPG Pack [4], Alien UFO Pack [5], Hexagon Tiles [6], UI Pack [7] and Kenney Fonts [8]. Those are really great assets that you may want to check out. It is worth noting that they are public domain.

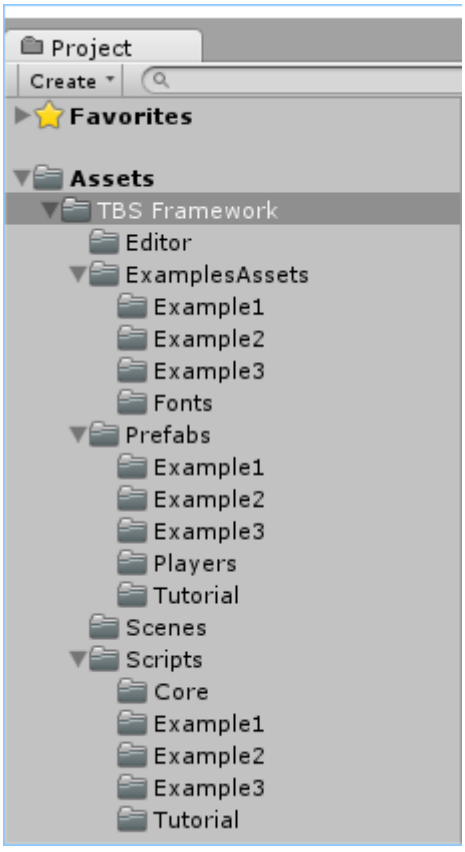

Fig 1. Project structure

The prefabs folder holds prefabs that I created for purpose of example scenes. You can use cells and units prefabs to start prototyping quickly. Players prefabs will be useful as well, though as for the AI player, you should probably create your own.

Finally, the Scenes folder contains a few playable scenes that show off some of the framework's capabilities.

3. Scene structure

Lets look at a scene created with simple assets available in Unity (and some nice looking trees and rocks). 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.

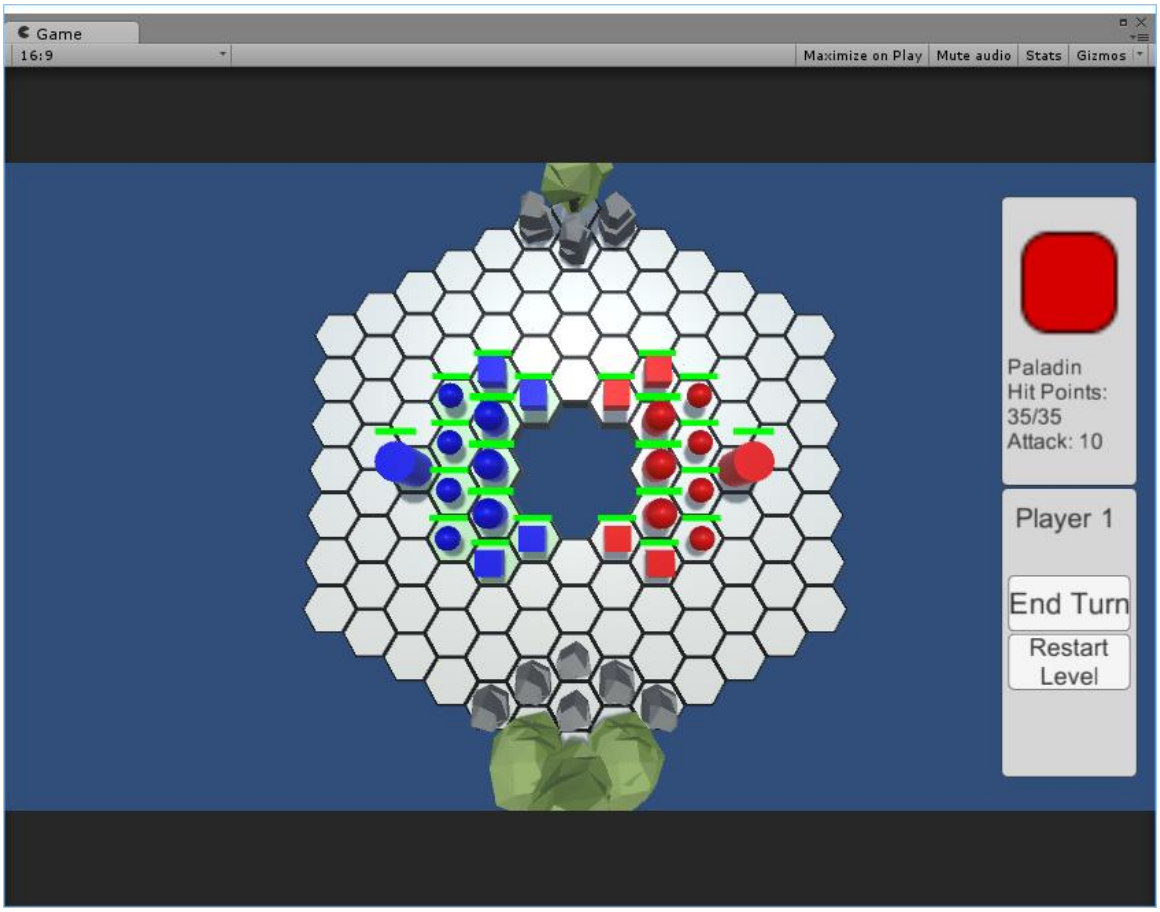

Fig. 2 – Simple scene

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 lets take a look at the scene setup, shown in Fig. 3.

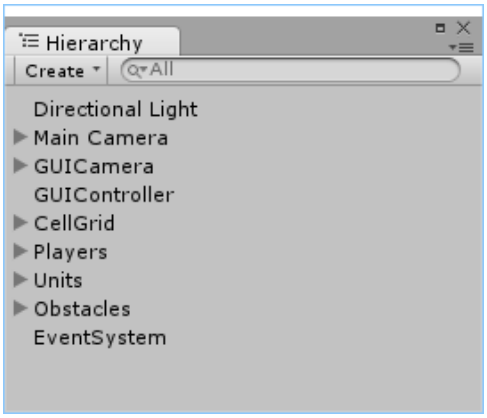

Fig. 3 – Scene hierarchy

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

CellGrid

CellGrid is the main object in the scene. It parents all the cells that the grid consists of. As you can see, it has three 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.
- CustomUnitGenerator – Implementation of IUnitGenerator. Spawning units will be explained in subsequent chapter.
- CustomObstacleGenerator – **Optional** script that places obstacles on the grid.

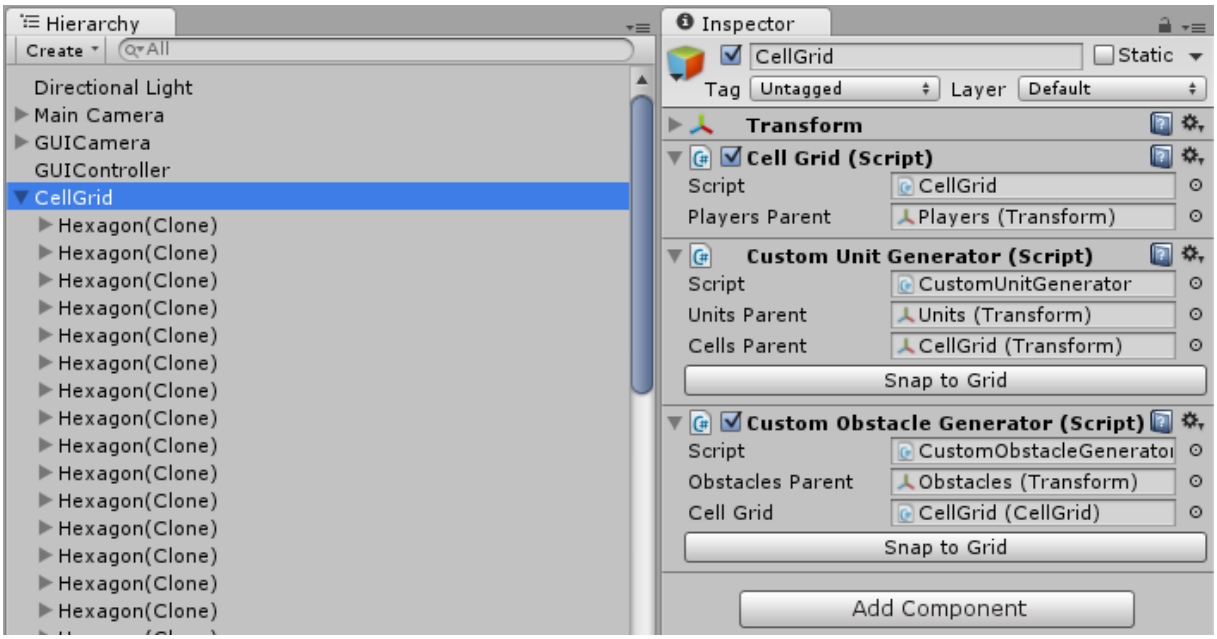

Fig. 4 – CellGrid game object

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. Adding players without any units to control is allowed, but such player will be skipped every turn and will not be able to give any input to the game. To change that behaviour, modify `EndTurn()` method in CellGrid class.

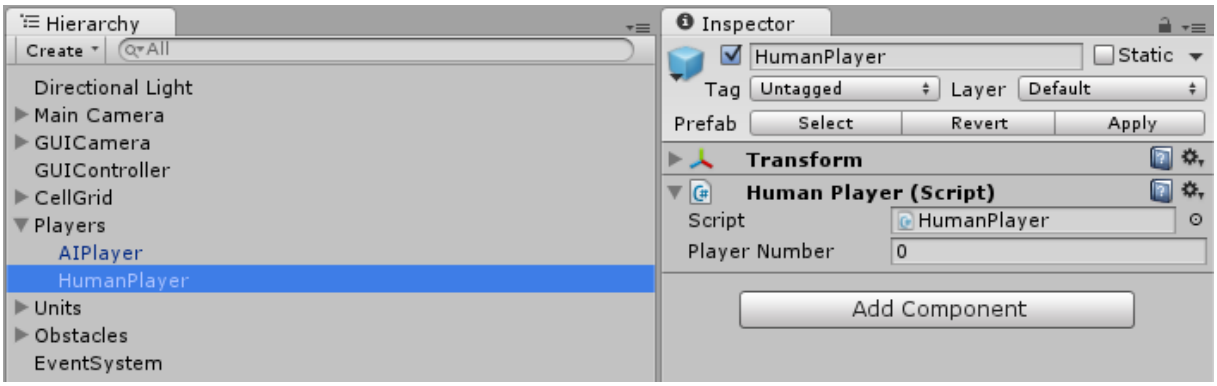

Fig. 5 – Players game object

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 the units.

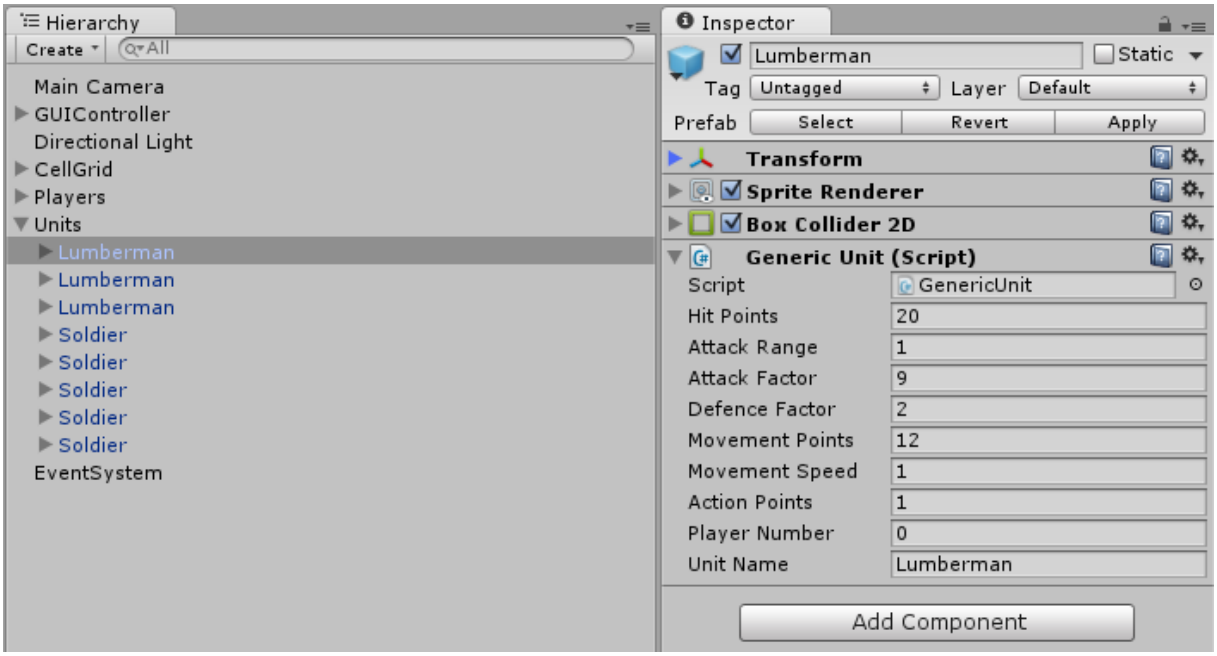

Fig. 6 – Units game object

4. Generating the Grid

Before generating a grid, you need to have a prefab of a cell. A cell is a game object with Cell script attached to it. It should also has a collider to allow mouse events to work. The project contains implementations of hexagonal and square cells, and it would be really easy to implement triangular cells as well.

If you have a cell prefab, you can proceed to generating a grid. To do so, follow these steps:

1. Create empty game object and add CellGrid script to it
2. Add script of type ICellGridGenerator to CellGrid game object. The project contains a few implementations for that:
    - HexagonalHexGridGenerator
    - RectangularHexGridGenerator
    - EquilateralHexGridGenerator
    - TriangularHexGridGenerator
    - RectangularSquareGridGenerator

    You can create your own generators by deriving ICellGridGenerator.
3. Fill in all the required parameters. All grid generators will have a CellsParent and CellPrefab parameters, other parameters will vary. CellGrid object at this step is shown in fig. 7.

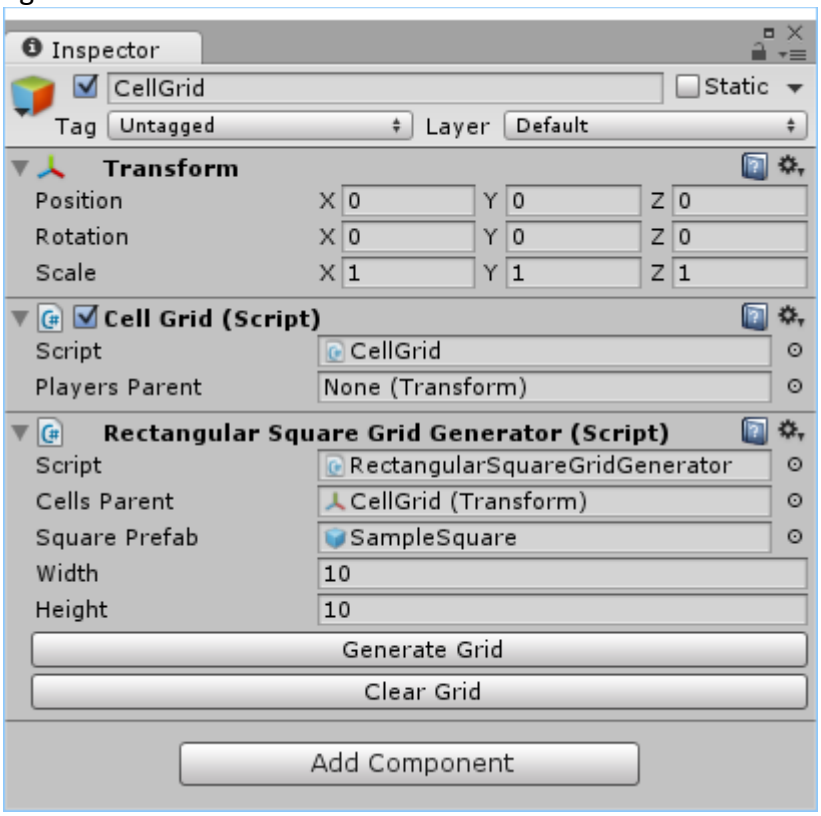

Fig.7 CellGrid game object with RectangularSquareGridGenerator attached to it.

4. All scripts derived from ICellGridGenerator (either mine or created by user) will be customized with two buttons. To create the grid, click „Generate Grid” button. The cells will be created and parented to the CellParent game object. If you are not happy with the result, click „Clear Grid” button to start from scracth.
5. When you are done with generating the grid, the ICellGridGenerator script can be safely removed.
6. At this point you can remove cells to give the grid unique shape, edit cells properties, place units, obstacles or decorators etc. Note that it is not possible to add cells to the grid manually.

5. Populating the grid with units

First, you going to need some units. Unit is a game object that has Unit script attached to it. It should also has a collider to allow mouse events to work. As mentioned before, all units that take part in the game must be children of Units game object. The project contains two implementations of IUnitGenerator class:

- CustomUnitGenerator is used, when you want to place units on the grid manually. To add a unit to the game, simply drag it to the scene and parent it to the Units parent game object. Units will snap to the nearest cell on play. If the nearest cell is already taken, the unit will be destroyed. You can snap the units manually by clicking “Snap to Grid” button. It is worth noting that snapping units to grid manually is purely visual – the cell that the unit is occupying will be set to “taken” on play.

- RandomUnitGenerator is used to spawn given number of units in random positions. Please note that even though the script has Number of Players filed, the players game objects still need to be added to the scene manually.

Using CustomUnitGenerator is preffered way of spawning units. The other script was created to showcase the possibility of implementing other methods of spawning units and probably will not be very useful. It is important not to use both scripts at the same time – such setup will not work. If the desired behaviour is to spawn some units manually and some randomly, a new script will be required.

6. Customization

The strength of this project is the ability to easily customize it. I provided 3 examples, each with different kind of style. First lets look at cells that I created, shown in Fig. 8. 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.

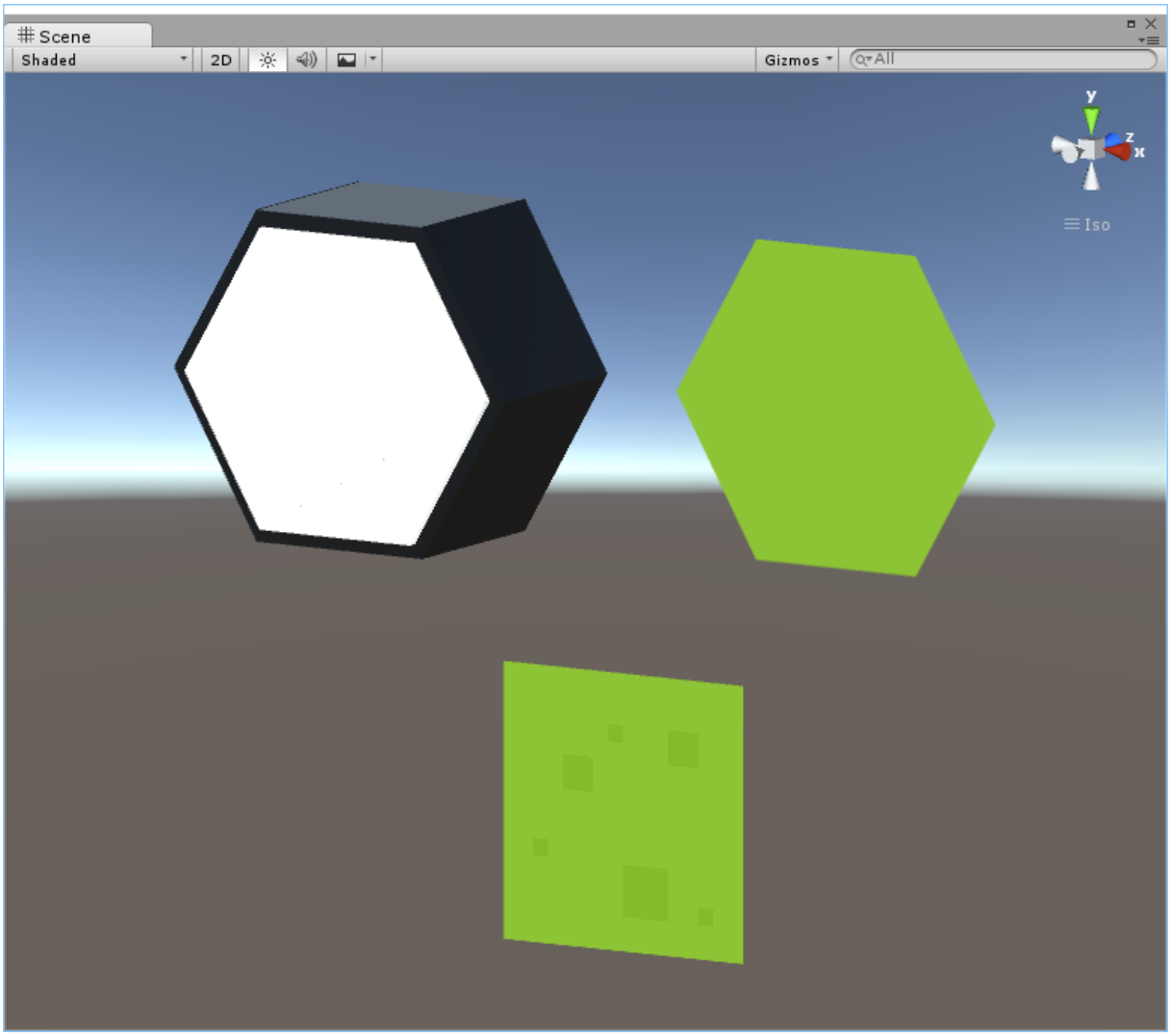

Fig. 8 – Different kinds of cells

Cells can be programmed to change appearance depending on state that they're in (I use term „state" here in colloquial sense, as it is not related to a state design pattern). To do so, just override appropriate methods in class derived from Cell. Available methods are:

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

Lets look at cells that are in different „states", shown in Fig. 9. From left the cells' appearance is: normal, highlighted, 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.

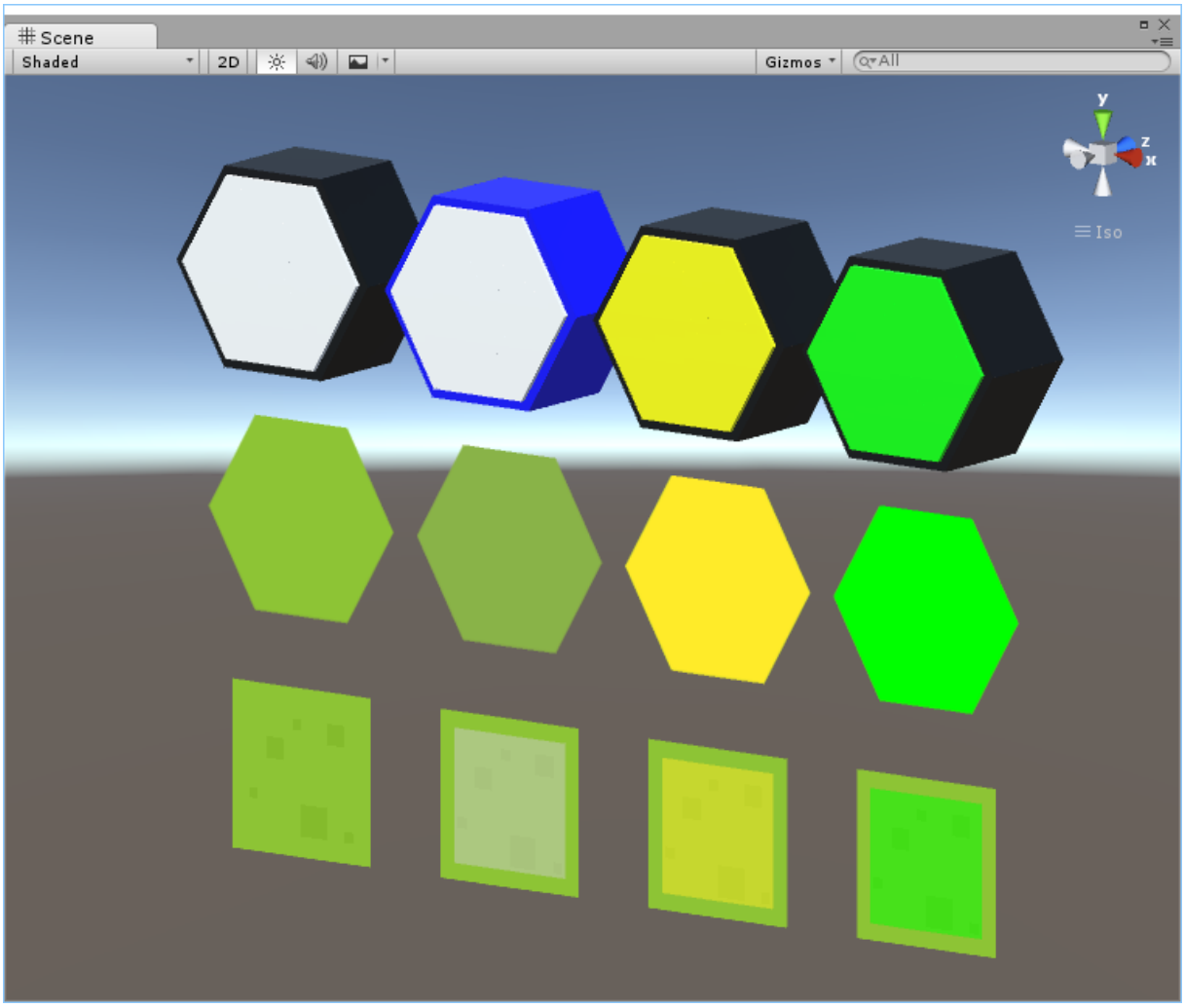

Fig. 9 – Cells appearance in different states

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

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

Units in different states are shown in Fig. 10. 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).

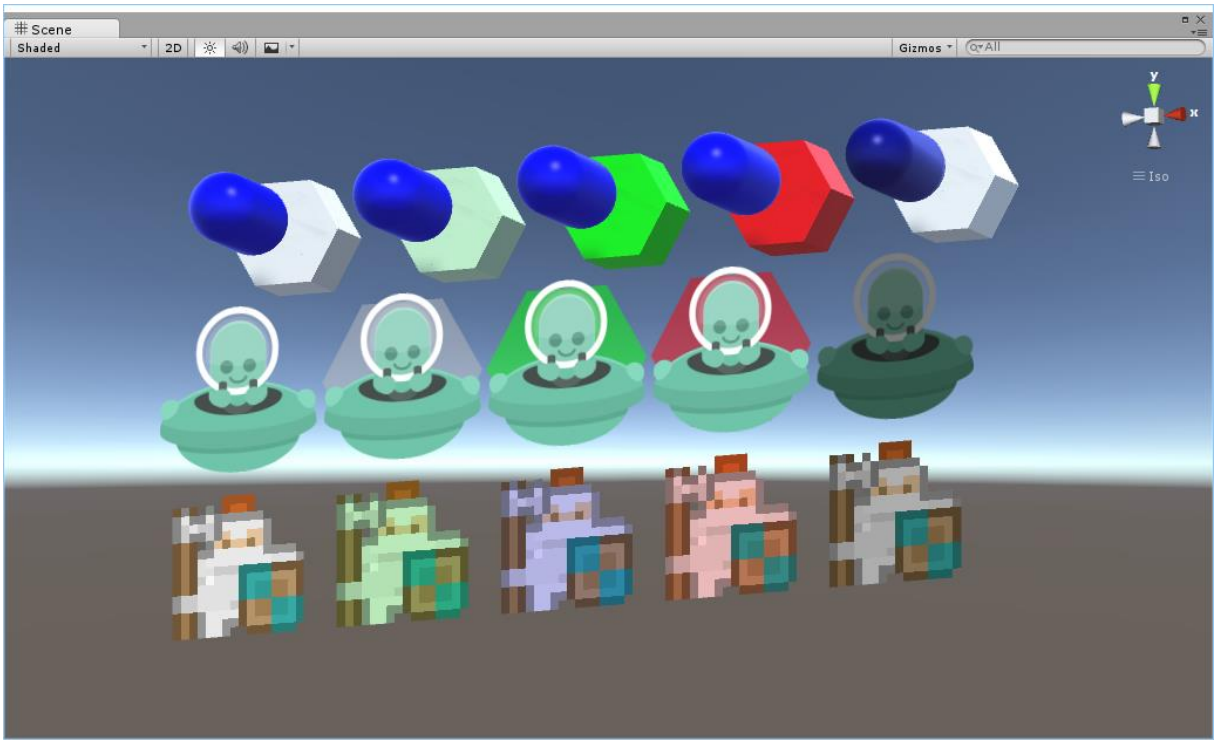

Fig. 10 – Units appearance in different states

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

- `OnMouseDown()`
- `OnMouseEnter()`
- `OnMouseExit()`
- `OnUnitSelected()`
- `OnUnitDeselected()`
- `OnTurnStart()`
- `OnTurnEnd()`
- `OnDestroyed()`
- `OnUnitSelected()`
- `OnUnitDeselected()`
- `IsUnitAttackable(Unit other, Cell sourceCell)`
- `DealDamage(Unit other)`
- `Defend(Unit other)`
- `Move(Cell destination, List<Cell> path)`
- `IsCellMovableTo(Cell cell)`
- `IsCellTraversable(Cell cell)`

Their purpose is described in the code. Below I present a few examples of unit customization. In example shown in Fig. 11, flying saucer is allowed to move over water and obstacles, while units that are on the ground are not.

Fig. 11 – 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`:

```
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.
}
```

I called this class `Alien`.

- 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:

```
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();
}
I called this class FlyingAlien.
```

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 to 3 contains implementation of such system. To get that effect, simply create three subclasses of Unit, and override their `Defend` methods:

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

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

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

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. 12.

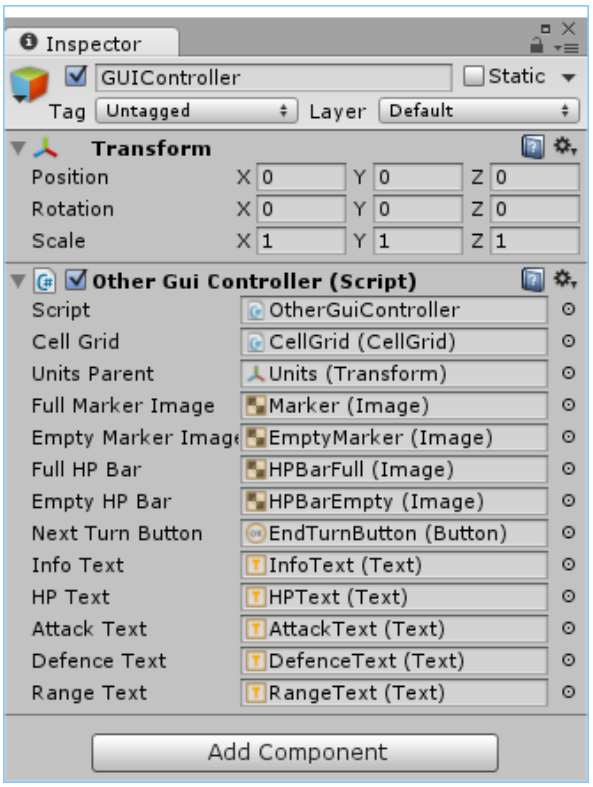

Fig. 12 – GUIController structure

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. Two examples of UI can be seen on Fig. 2 and Fig. 11. Another approach is shown in Fig. 13.

Fig. 13 – Different kind of UI

7. 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.

1. First thing you want to do is create a new scene in Unity editor.
2. Create a cube by clicking Game Object -> 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 it's colour to grey when highlighted, yellow to indicate that it is reachable and green to mark it as path. The code looks like this:

```
using UnityEngine;

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, and drag it to prefab folder to create a prefab.
6. Create three empty objects by clicking Game Object -> Create empty in Unity editor. Name the objects CellGrid, Players Parent and Units Parent.
7. Add the following scripts to the CellGrid game object: CellGrid.cs, RectangularSquareGridGenerator.cs and CustomUnitGenerator.cs.
8. Fill in all the parameters. The scene hierarchy should look like in Fig. 14.

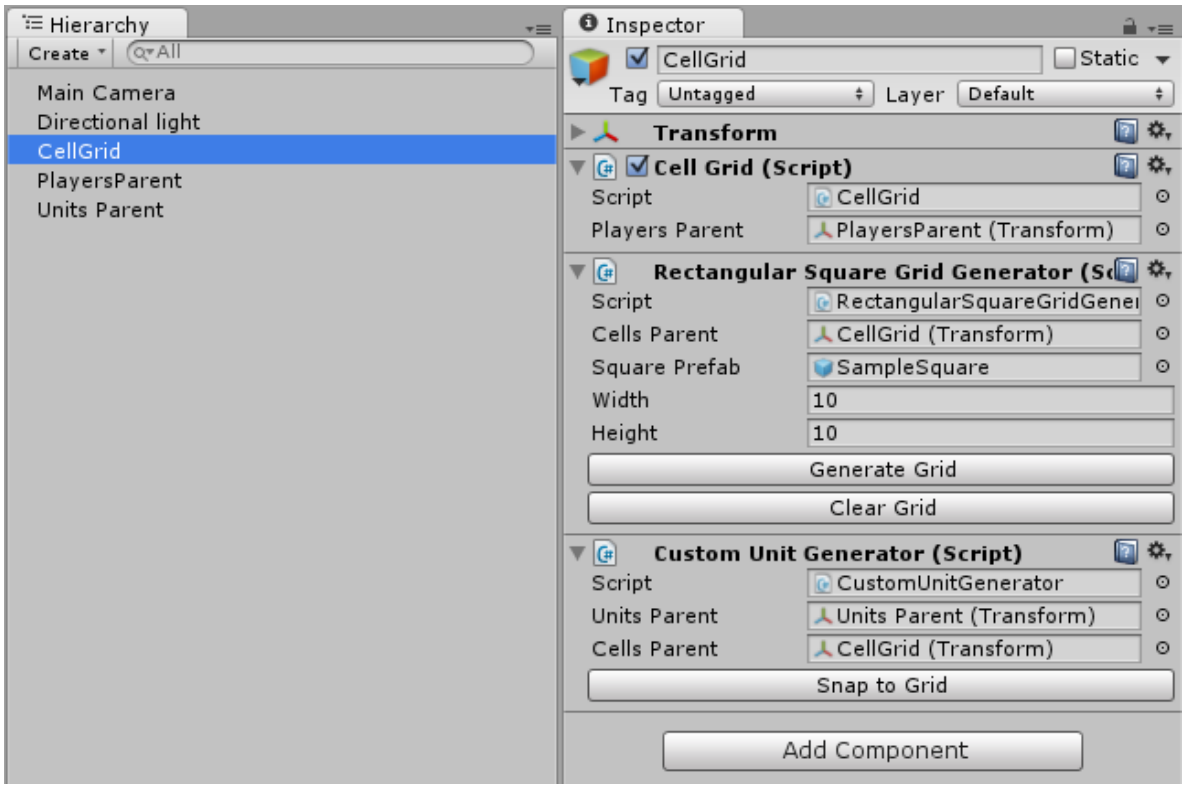

Fig. 14 – Scene hierarchy at step 8th

9. Once the parameters are filled in, simply click “Generate Grid” button to create the grid. Scene view will look like it is shown in fig. 15.

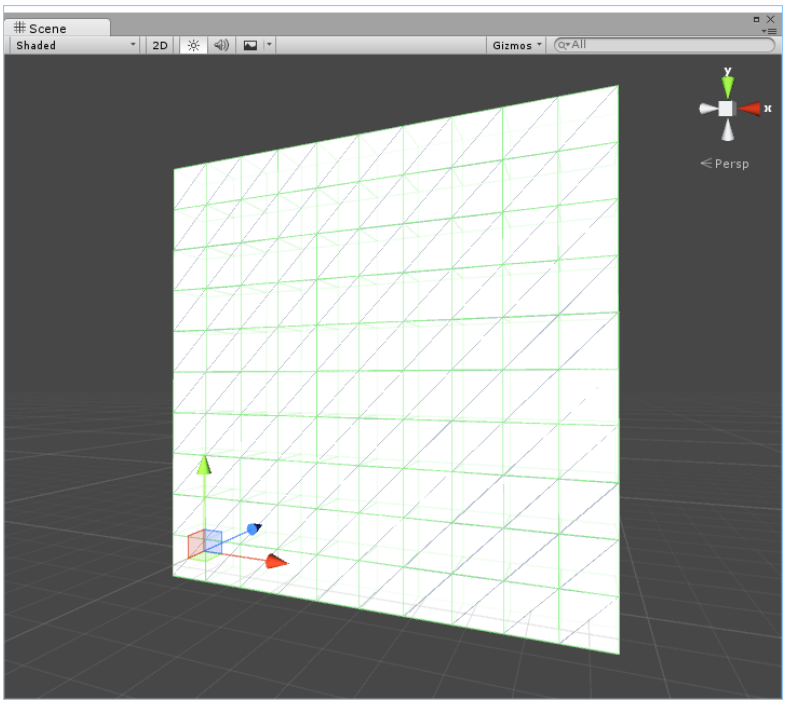

Fig. 15 – Scene view at step 9th

10. At this point, RectangularSquareGridGenerator script can be safely removed from the CellGrid game object.

11. 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 will ommit implementation of some functions.

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

}
```

12. Create two new materials and set them to two different colours.
13. Create two new cubes that will represent units. Attach materials and SampleUnit script to the cubes.
14. Create a few units and distibute them on the grid. Fill in all the parameters on units objects. Divide units over two players by assigning Player Number parameters. Remember that all units must be children of Units Parent object. You can click “Snap to Grid” button on CustomUnitGenerator script to check if the units are in desired positions. This is not necessary, the units will snap to the nearest cell when the game starts anyway. My setup at this point is shown in Fig. 16.

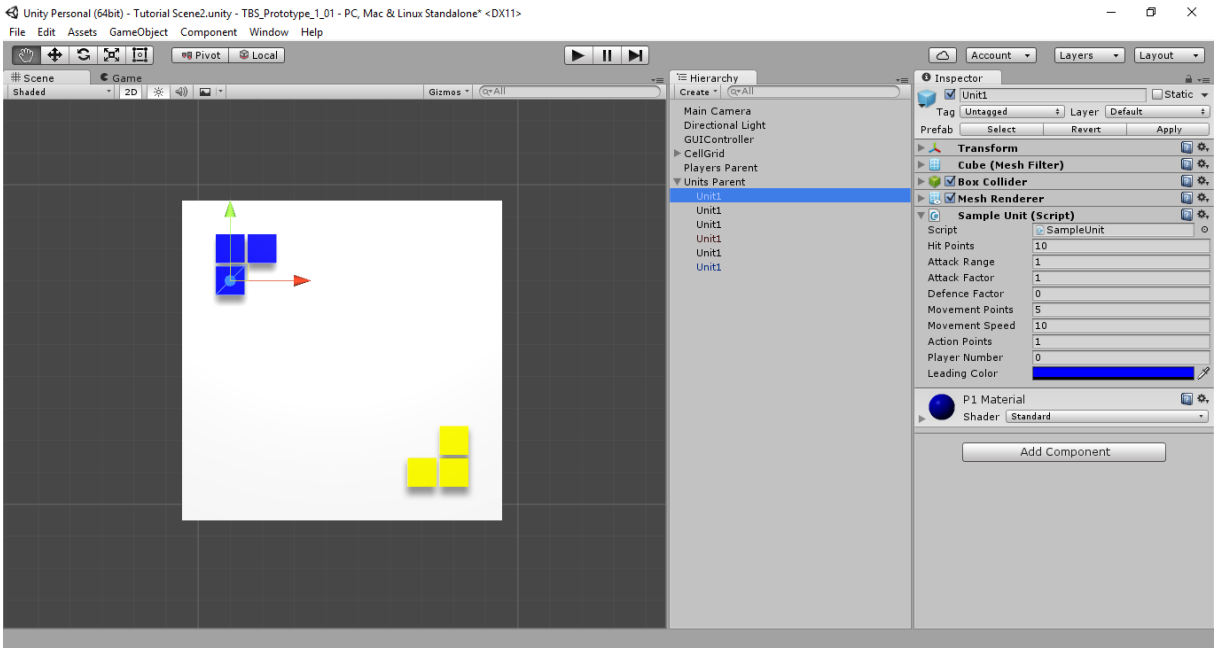

Fig. 16 – Scene setup at step 14th.

15. Add players prefabs to the Players Parent object. You can find them in Prefabs/Players folder. Both HumanPlayer or AIPlayer will be fine. Fill in Player Number parameter on the players objects. The numbers should correspond to the numbers that were assigned to the units.
16. Lets add some obstacles to the scene. To do that, create new cube, attach black material to it, duplicate and distribute it on the scene. You don't need to add any script to obstacles, unless you want to add some custom behaviour to it.
17. What you need to do, is set IsTaken parameter on cells that the obstacles are occupying. We will use a script for that. Create empty game object, and assign it as parent of all obstacles that you created. Next, add CustomObstaclesGenerator script to the CellGrid object and fill in the parameters. To check if obstacles end up in desired positions, click "Snap to Grid" button on CustomObstacleGenerator script. The IsTaken parameter will be set to true on play.
18. Lastly, we need to take care of user interface. To keep it simple we will deal only with making turn transitions. Create empty object, and attach the folowing script:

```
using UnityEngine;
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.
        }}}
```

That concludes the tutorial. Fig. 17 shows the scene setup after the last step. Fig. 18 shows the result. The created scene is playable, though perhaps not particularly interesting to play. It's up to you to change it.

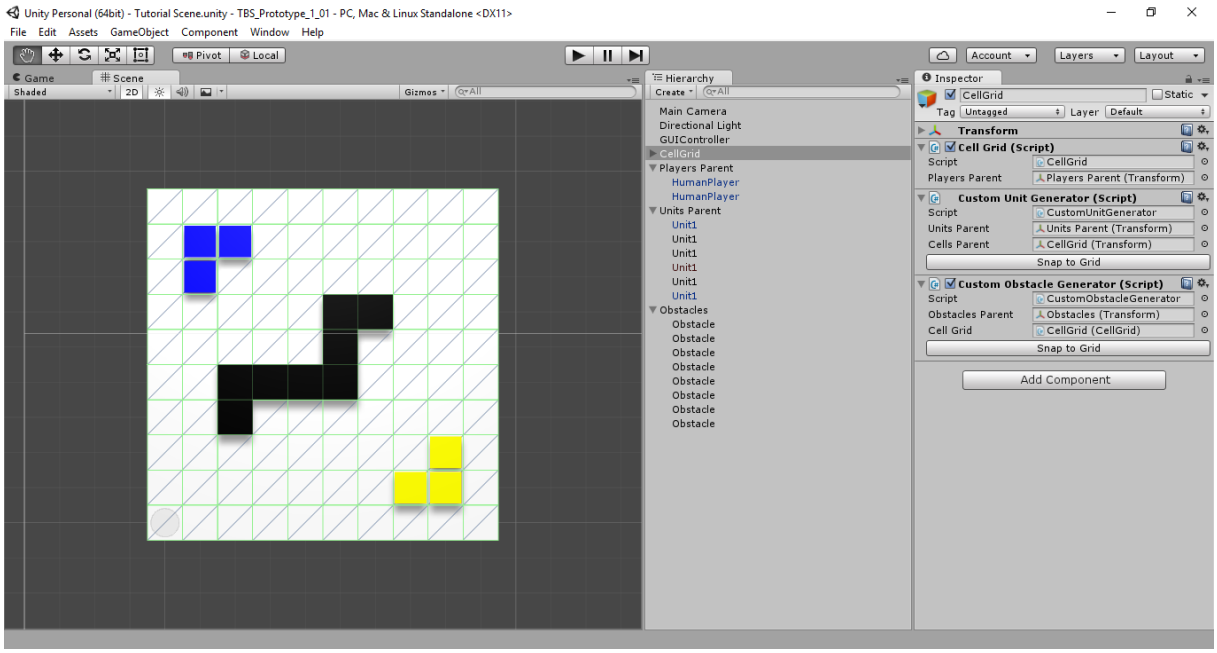

Fig. 17 – Final setup of the scene

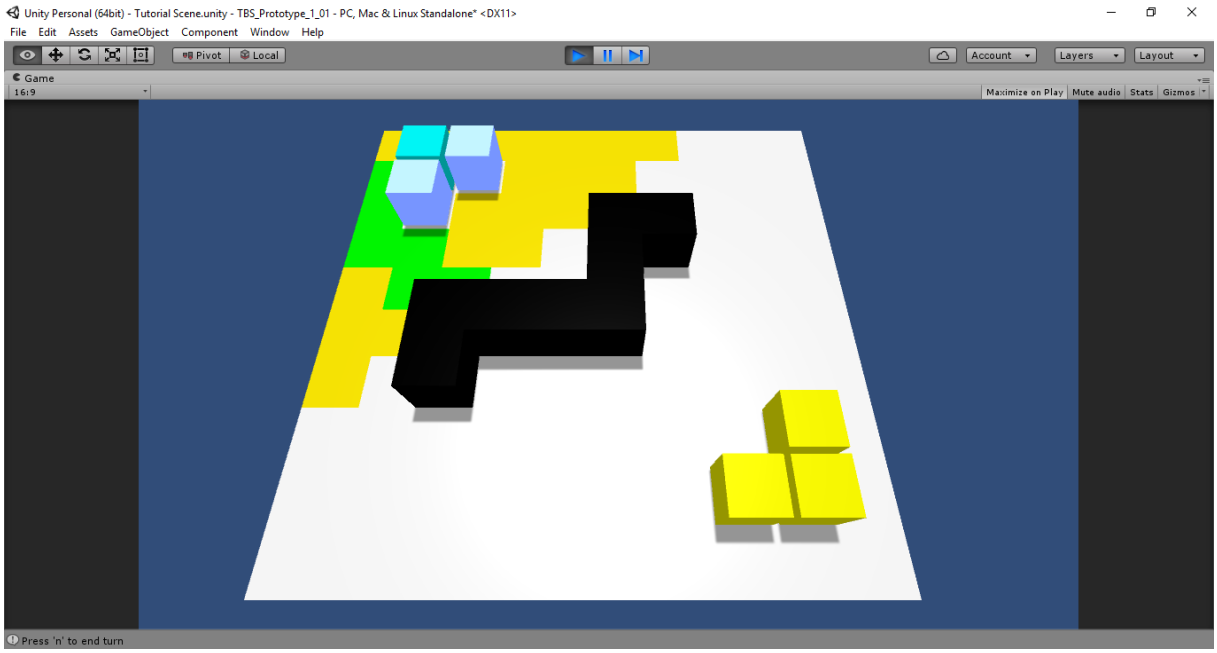

Fig. 18 – Finished scene

8. 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.

9. References

[1] Daniel Robnik, Low poly styled trees,
https://www.assetstore.unity3d.com/en/#!/content/43103
[2] Daniel Robnik, Low poly styled rocks,
https://www.assetstore.unity3d.com/en/#!/content/43486
[3] Kenney, Roguelike Characters, http://www.kenney.nl/assets/roguelike-characters
[4] Kenney, Roguelike/RPG Pack, http://www.kenney.nl/assets/roguelike-rpg-pack
[5] Kenney, Alien UFO Pack, http://www.kenney.nl/assets/alien-ufo-pack
[6] Kenney, Hexagon Tiles, http://www.kenney.nl/assets/hexagon-tiles
[7] Kenney, UI Pack, http://www.kenney.nl/assets/ui-pack
[8] Kenney, Kenney Fonts, http://kenney.nl/assets/kenney-fonts