CROOKED HEAD

Turn Based Strategy Framework

Documentation

Version 2.2

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 detail 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. Contact details are at the end of the document. 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 the 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 the Editor folder. The most important script here is GridHelper - a powerful tool that will help you set up the scene.

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], Kenney Fonts [7], Backgrounds by Kronbits [8] and Aekashics Battlers [9]. Those are really great assets that you may want to check out.

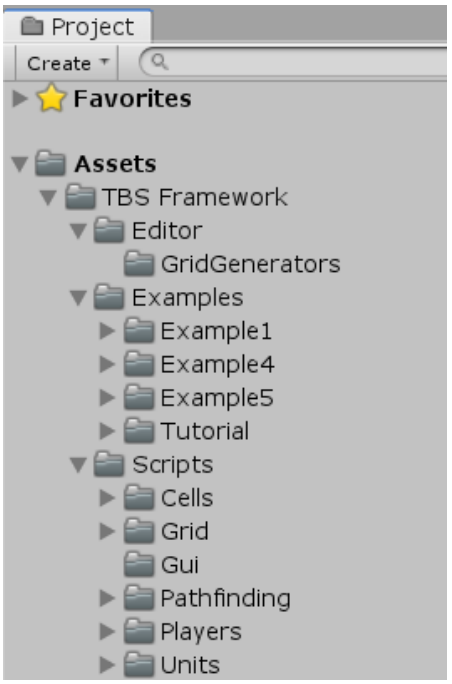

Fig. 1 - Project structure

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.

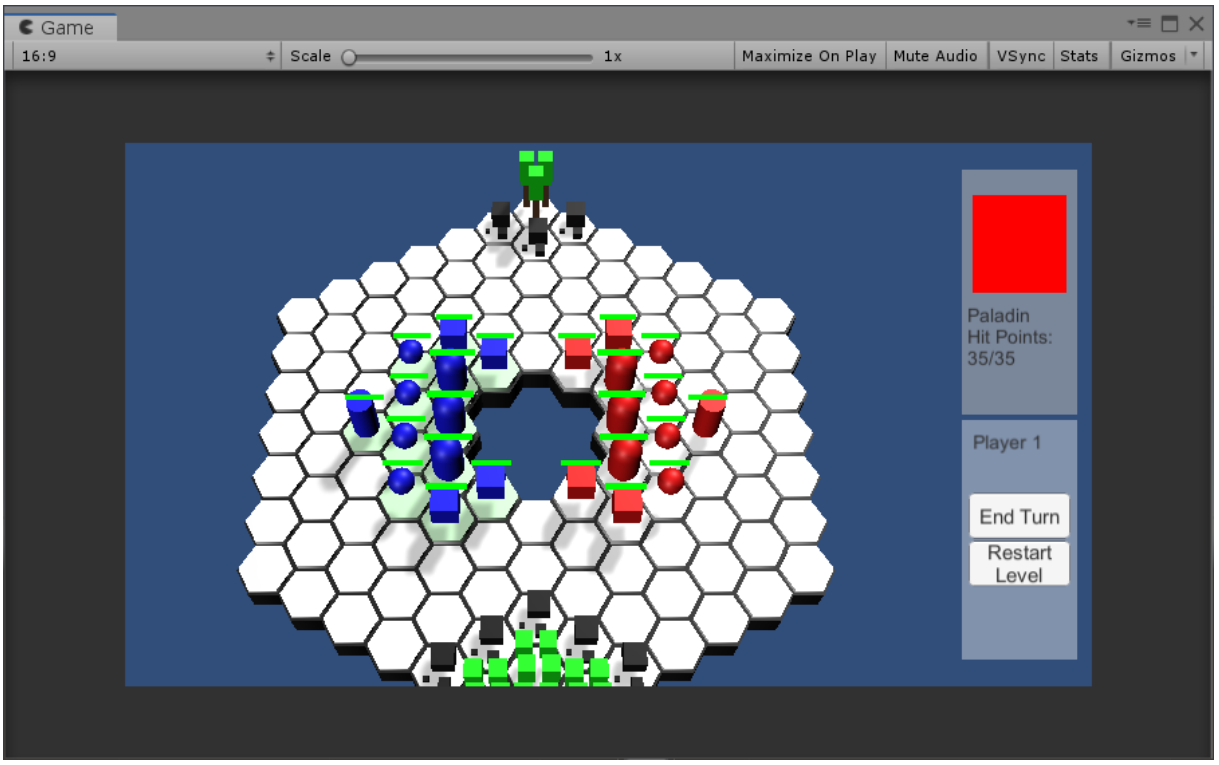

Fig. 2 - Simple scene

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.

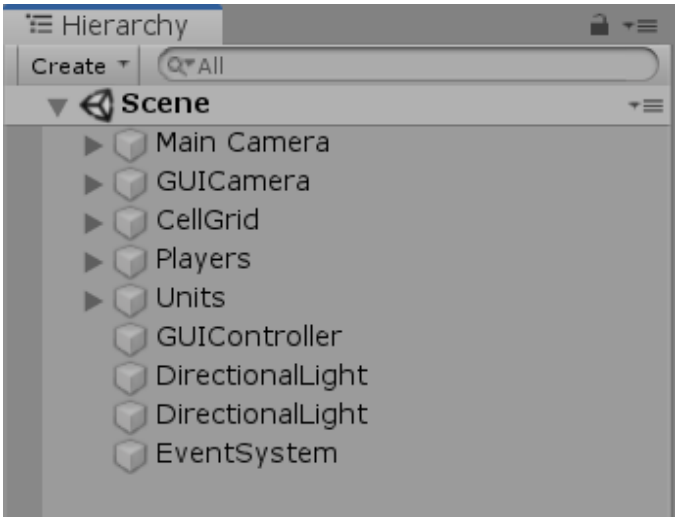

Fig. 3 - Scene hierarchy

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
- SubsequentTurnResolver - Determines which player goes next and which units can be used. Can be replaced with a different resolver, as described in "Turn transitioning" chapter
- DominationCondition - Check if the game is over and who is the winner. Can be replaced with a different resolver, as described in "Ending the game" chapter

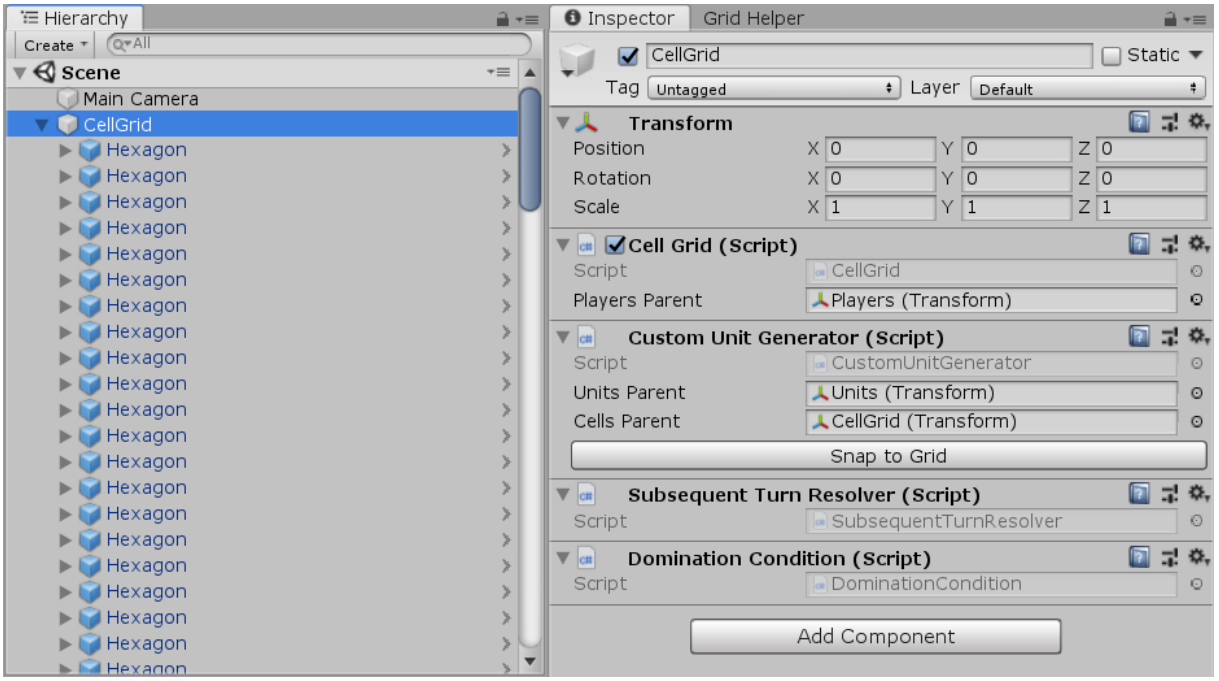

Fig. 4 - CellGrid game object

3.2. Players

Players game object holds player objects. A player is a game object with a Player script attached to it. Number of players is not limited, but the CellGrid script requires at least one player object to work correctly. Attribute "Player Number" must be unique to each player.

The project contains a strong, competitive AI player implementation. One of the next chapters describes the AI setup process in detail. There is also a NaiveAiPlayer script that was used prior to v2.1 which is marked obsolete and left for compatibility reasons.

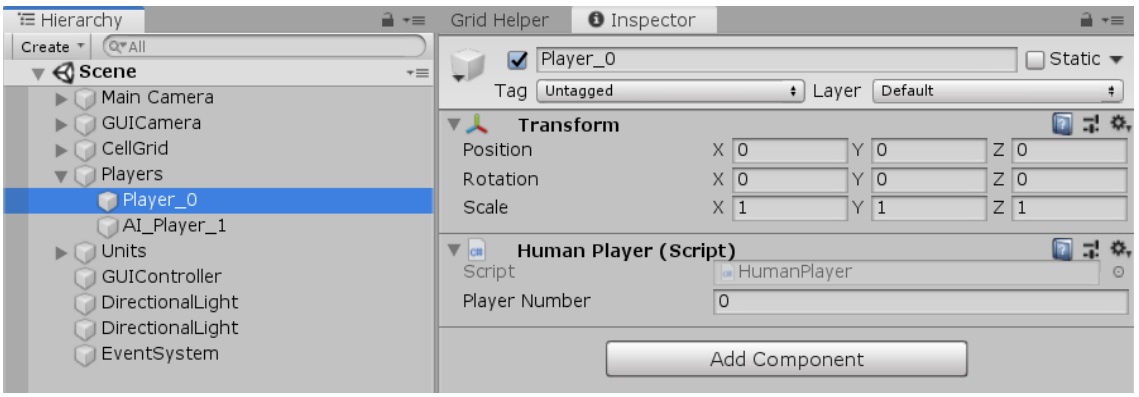

Fig. 5 - Players game object

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 raise errors. Each unit has a Player Number attribute that should correspond with the Player Number attribute on the Player object. Adding units that don't have any player „attached" (a player with corresponding Player Number doesn't exist) is acceptable, but it will be impossible to control them.

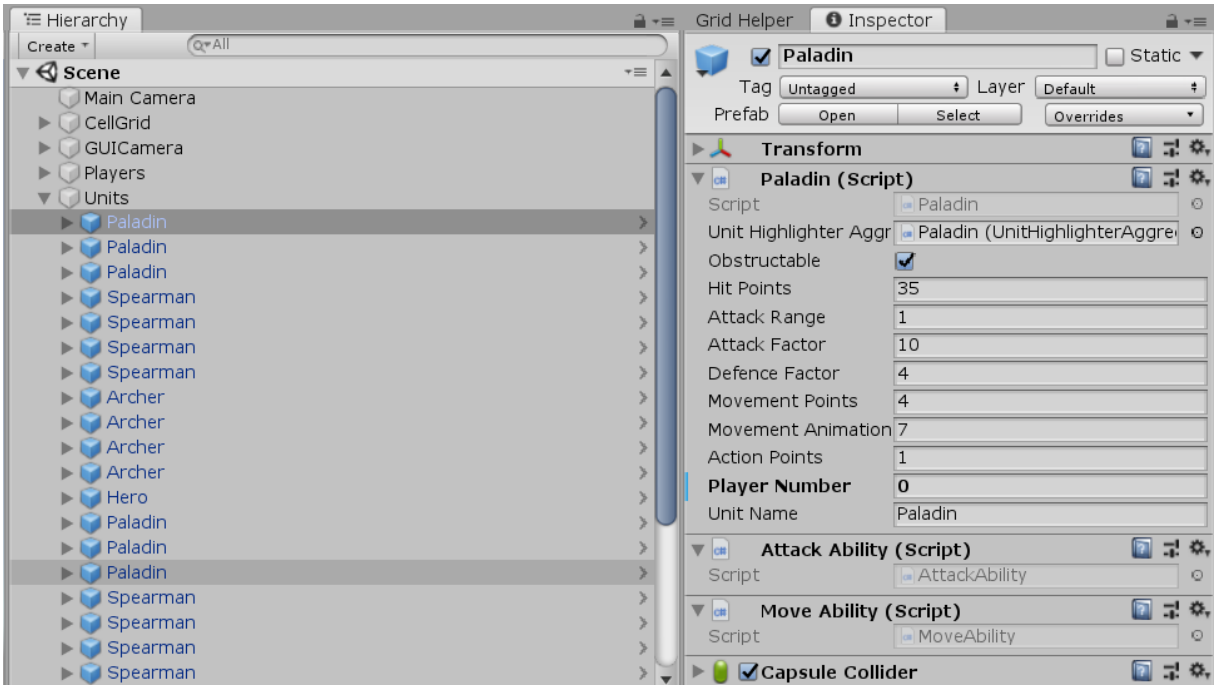

Fig. 6 - Units game object

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 the 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 virtual, 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 to 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.

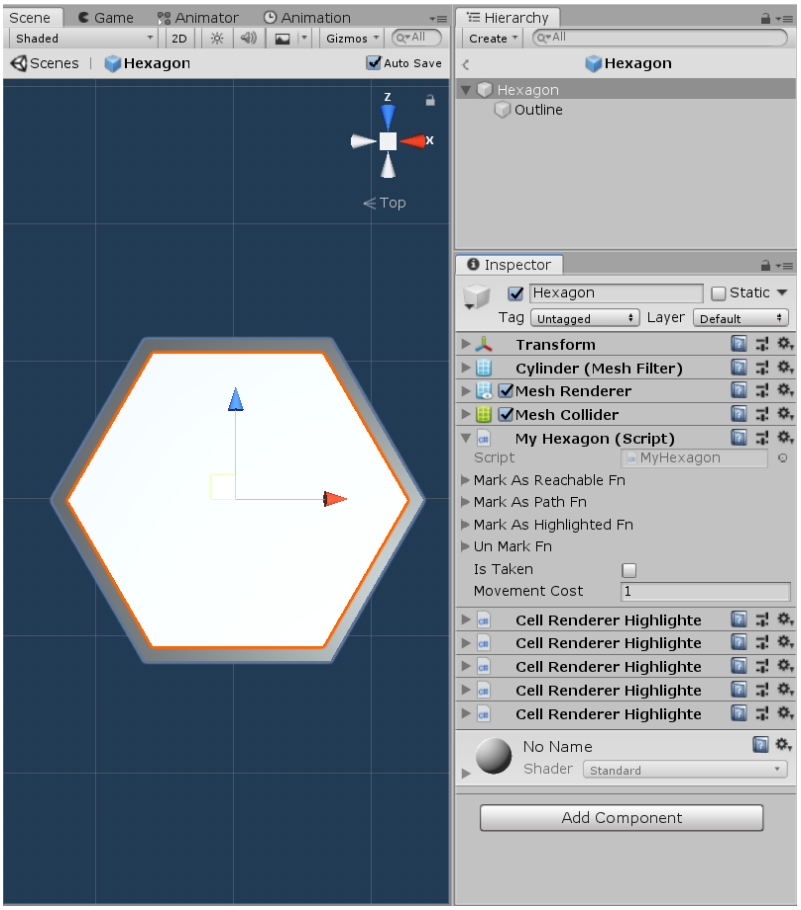

Fig. 7 - Sample cell prefab

4.2. Grid generation

If you have the 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 the 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 the 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 the shape of the grid. The project contains scripts to generate hex-tile grids in the shape of rectangle, triangle, parallelogram and hexagon, and rectangular square-tile 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 the “Generate scene“ button. The script will generate all necessary game objects described in the 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 triggering 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 the 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 the 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.

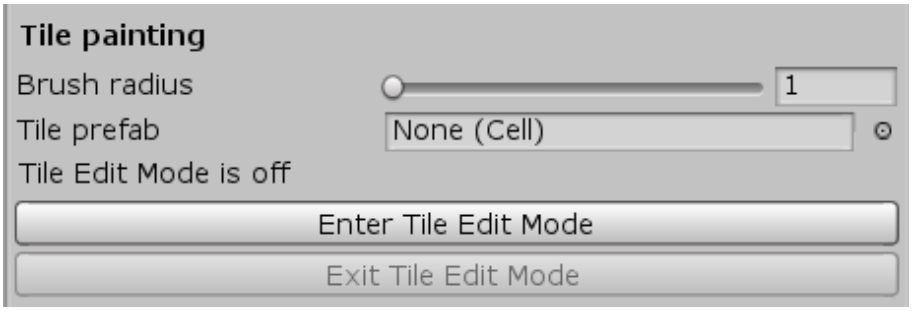

Fig. 9 - Tile Painter interface

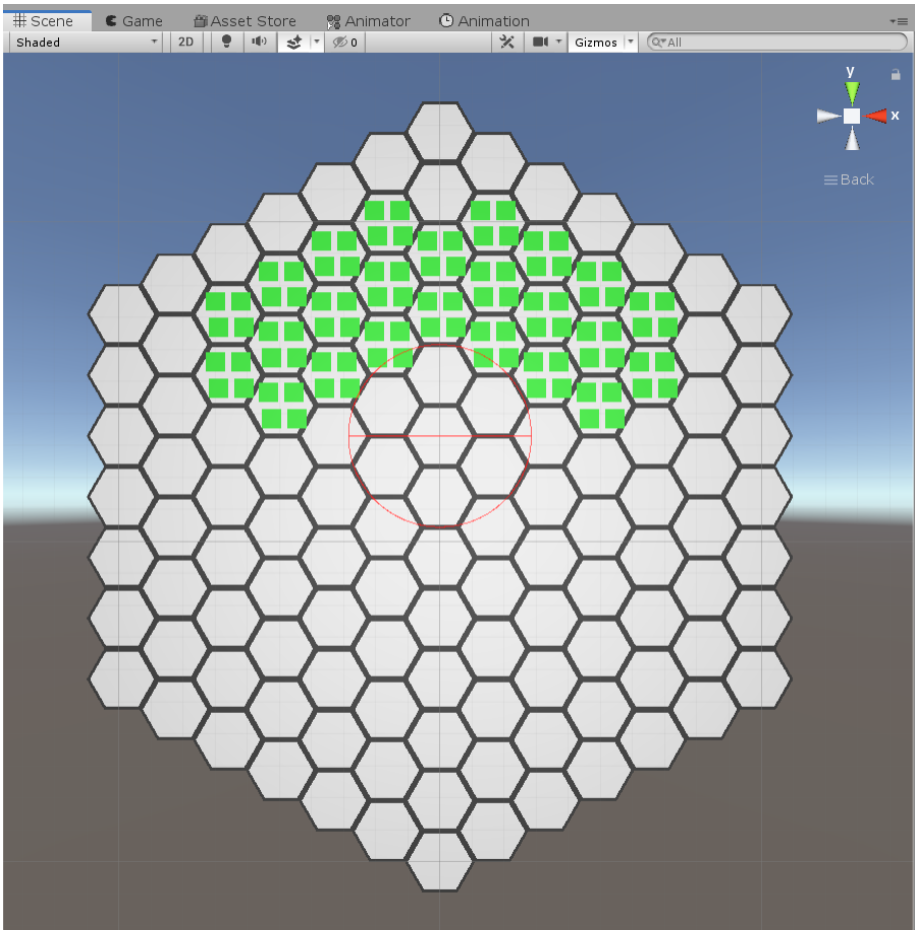

Fig. 10 - Tile Painter in use

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 virtual appearance customization methods that need 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 the Grid Helper window accessible in the Unity menu under Window -> Grid Helper. The interface is shown in Fig. 11.

To use the painter, input the number of player that the unit will belong to and select the 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.

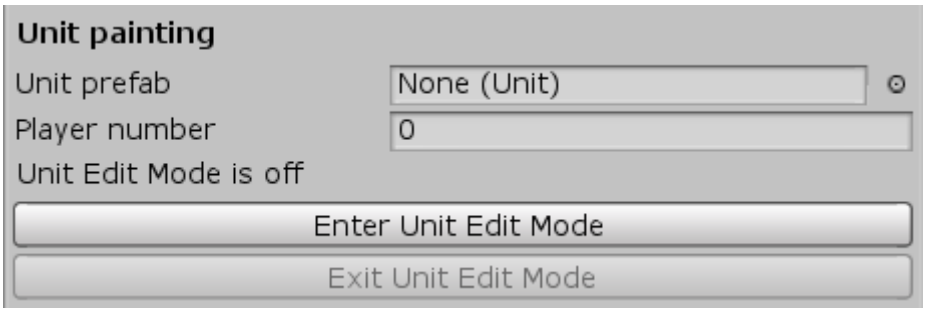

Fig. 11 - Unit Painter interface

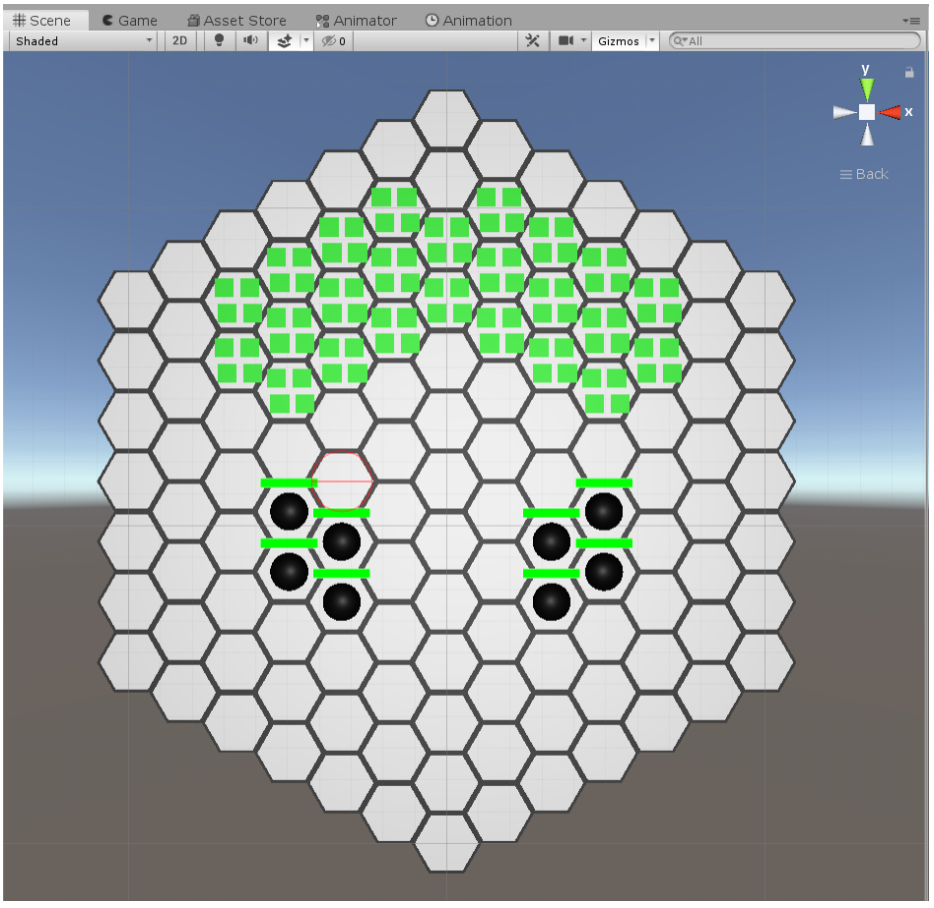

Fig. 12 - Unit Painter in use

4.6. Prefab Helper

To create your 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 the Grid Helper window. Currently its only function is to save multiple game objects that are selected in the hierarchy into prefabs. The process is as follows:

1. Select game objects that you want to save as prefabs
2. Click the “Selection to prefabs“ or “Selection to prefabs (variants)” button on the Prefab helper interface. The latter option saves prefabs as variants.
3. Select destination folder in dialog window

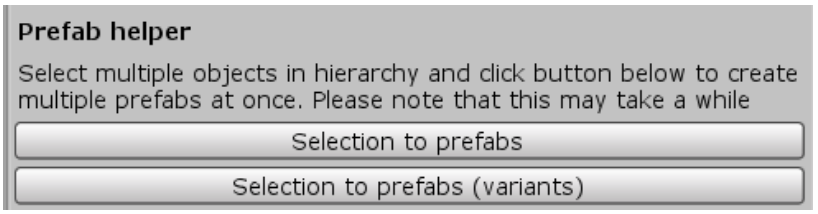

Fig. 13 - Prefab Helper interface

5. Customization

The strength of TBS Framework is the ability to customize it easily. In the project, I provided 3 examples, each with different kinds 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 kinds of cells – triangular for example.

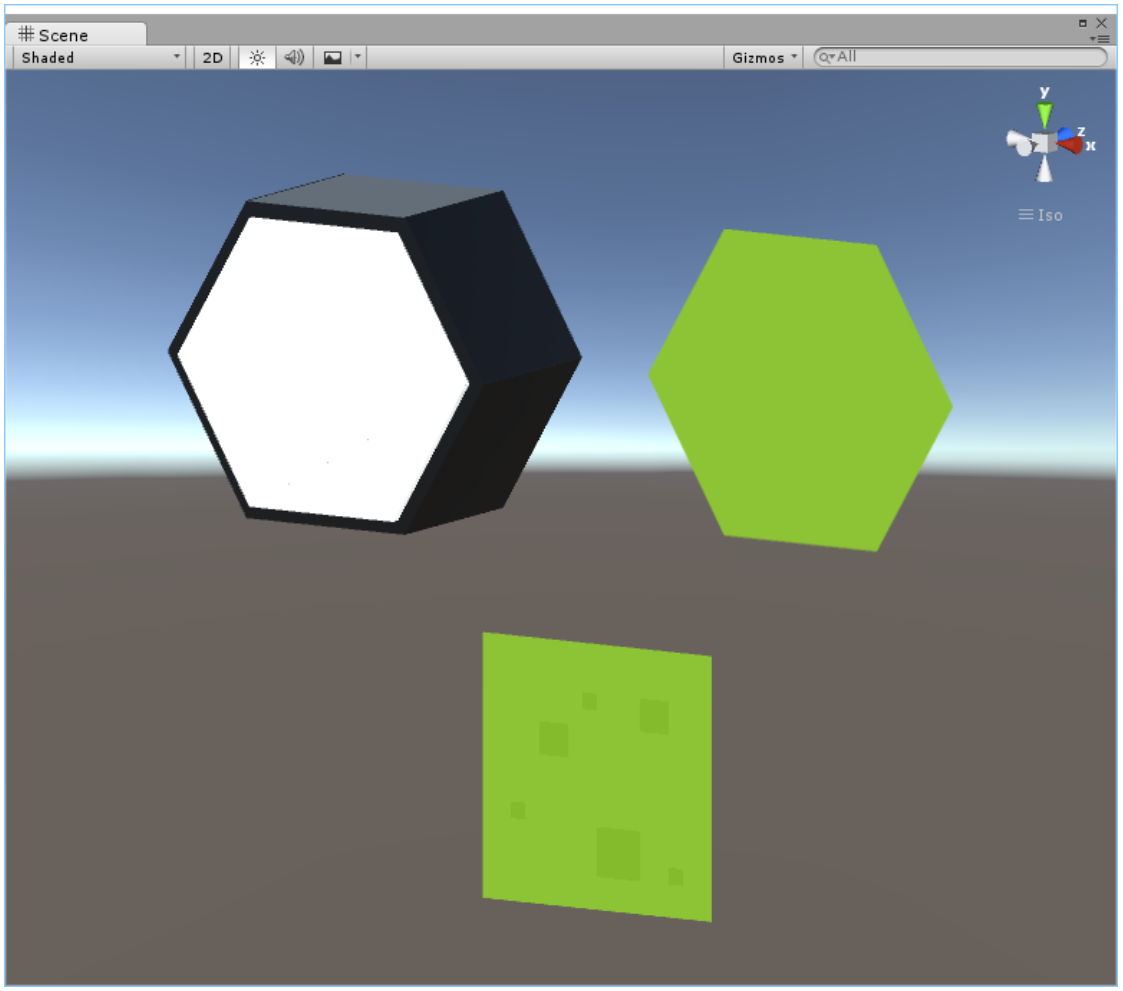

Fig. 14 - Different kinds of cells

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

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

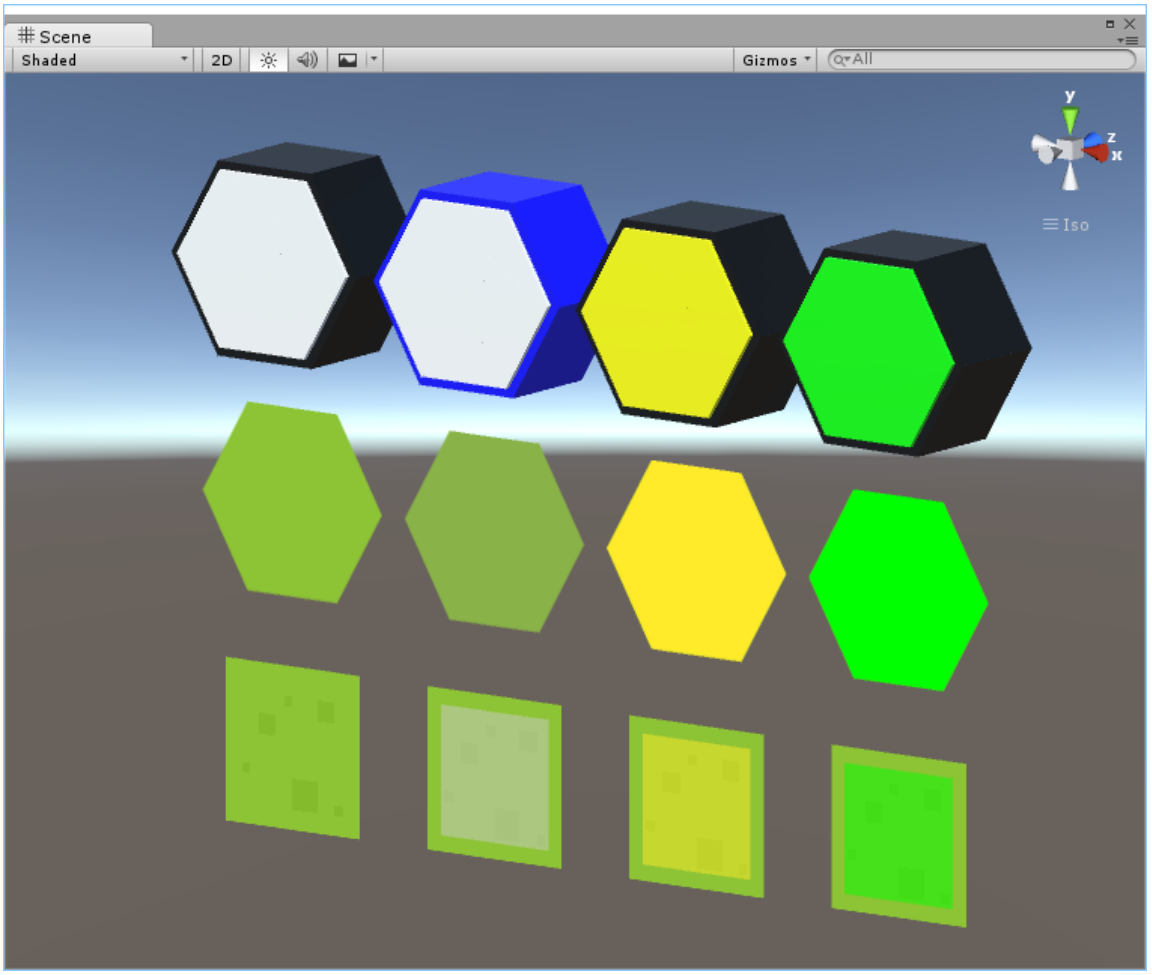

Fig. 15 - Cells appearance in different states

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

- `int GetDistance(Cell other)`

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

- `List<Cell> 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:

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

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

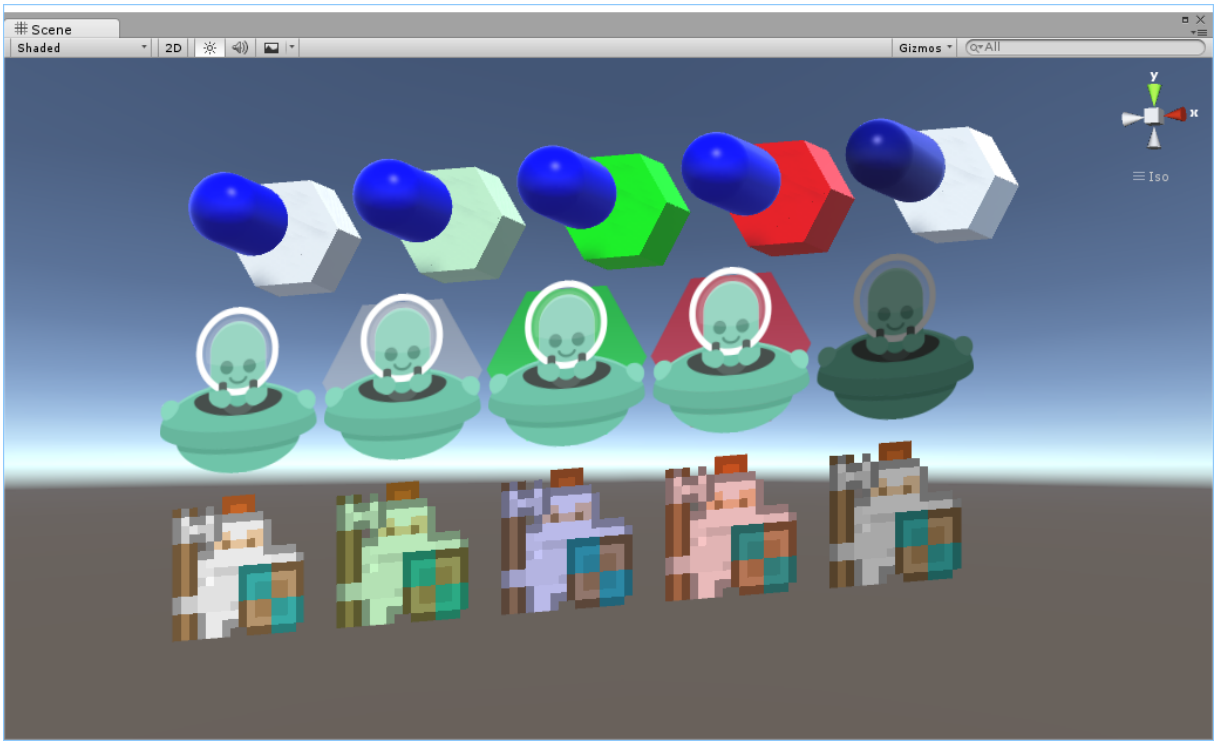

Fig. 16 - Units appearance in different states

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

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

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

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

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

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

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

- `void OnDestroyed()`

Called when the unit is destroyed

- `bool IsUnitAttackable(Unit other, Cell sourceCell)`

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

- `AttackAction DealDamage(Unit unitToAttack)`

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

- `void AttackActionPerformed()`

Method is called after the attack was performed.

- `int Defend(Unit aggressor, int damage)`

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

- `void DefenceActionPerformed()`

Method called after defence action was performed.

- `bool IsCellMovableTo(Cell cell)`
- `bool 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 a given cell.

- `void 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. Unit special abilities

Unit ability system is a powerful tool that facilitates adding customized behaviour to units. It's intuitive, reusable and encapsulated in a way that is easily accessible both for human and AI players. Basic unit abilities like moving and attacking are no exception, they are also implemented in this manner. It is such an important topic, that it deserves to be discussed in detail in it's own chapter.

5.3.1. How to use it

To make an ability usable by a unit, simply add the ability script to the unit prefab. Moving and attacking abilities are added to a gameobject by default when Unit script is attached to it.

Implementing new abilities is done by extending Ability class. In the implementation you define how the ability is visualized to the player and what happens when it is activated. The public interface is as follows:

- `void Display(CellGrid cellGrid)`
- `void CleanUp(CellGrid cellGrid)`

These methods are invoked only for human players. The purpose of `Display` is to visualize the ability, like highlighting cells that are available to move, or enemies that are in range. When the ability is deselected, the `CleanUp` method is run to return the grid to default appearance.

- `void OnUnitClicked(Unit unit, CellGrid cellGrid)`
- `void OnUnitHighlighted(Unit unit, CellGrid cellGrid)`
- `void OnUnitDehighlighted(Unit unit, CellGrid cellGrid)`
- `void OnCellClicked(Cell cell, CellGrid cellGrid)`
- `void OnCellSelected(Cell cell, CellGrid cellGrid)`
- `void OnCellDeselected(Cell cell, CellGrid cellGrid)`` `

Another batch of methods invoked only for human players. Can be used to display contextual information about the ability, like highlighting cells affected by area of effect attack. `OnUnitClicked` and `OnCellClicked` are usually used to execute the ability.

- `void OnAbilitySelected(CellGrid cellGrid)`
- `void OnAbilityDeselected(CellGrid cellGrid)`
- `void OnTurnStart(CellGrid cellGrid)`
- `void OnTurnEnd(CellGrid cellGrid)`
- `void OnUnitDestroyed(CellGrid cellGrid)`

These methods are run regardless of the type of player that is using the ability. OnTurnStart and OnTurnEnd methods could be used to implement passive abilities.

- `bool CanPerform(CellGrid cellGrid)`

Method used to define whether the ability can be used - for example it could check if the unit has any movement points left to move

- `IEnumerator Act(CellGrid cellGrid)`

Finally, `Act` method is where you define what the ability actually does. It is run as a coroutine, because most abilities take some time to execute - for example an animation is played.

It is important to remember to run the method as a coroutine, since calling it like a regular function will have no effect. Because of that, the base Ability class has a utility method called `Execute`. It should be used to apply the ability instead of using `Act` directly. The method will run the coroutine for you and also accepts two functions as parameters that will be run before the coroutine starts, and after it finishes. There are two more convenience methods with pre-action and post-action already setup - `HumanExecute` which blocks input while the ability runs and `AIExecute` with empty pre-action and post-action.

5.3.2. How does it work

As mentioned before, ability scripts should be attached to unit gameobject. Ability selection is handled by the `CellGridStateAbilitySelected` object, which takes a list of abilities as a parameter. By default, all abilities attached directly to the unit are selected. Different `CellGridState` scripts are described in the next chapter.

The system can be leveraged to select abilities individually. Have a placeholder ability that will hold a list of selectable abilities and represent each one as a button. Clicking on the button should set the GridState to `CellGridStateAbilitySelected` with specific ability passed as the parameter.

5.3.3. Conclusion

The ability system brings a whole new level of depth into the Turn Based Strategy Framework. Features that previously were awkward to add, now are easy and intuitive to implement. I would like to thank the TBSF community for constantly asking me about different gameplay mechanics, your questions always inspire me to think about ways to improve the Framework.

5.4. 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.

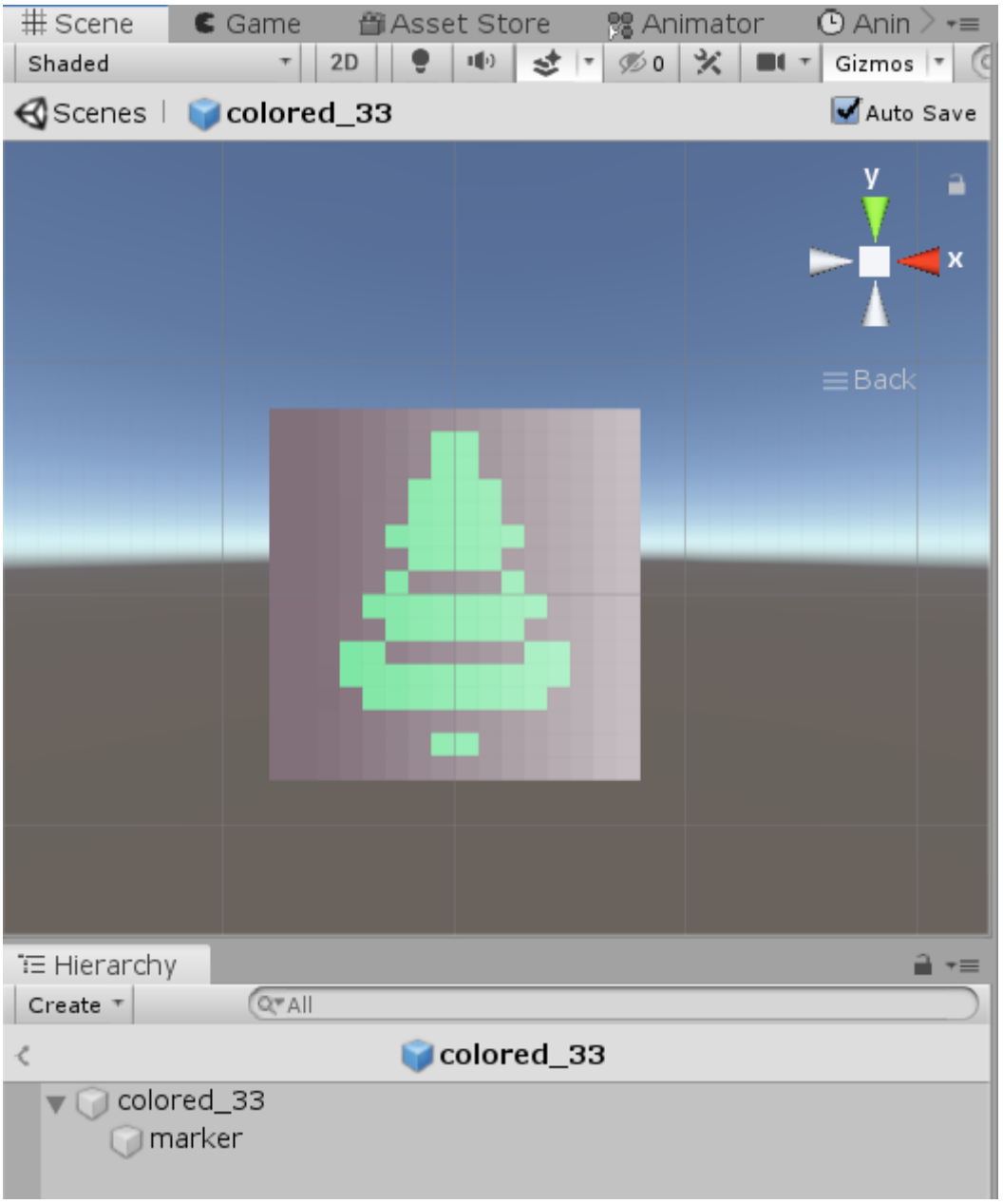

Fig. 17 - Sample cell prefab

The prefab consists of a parent object that holds components required by Cell (Cell script and 2D collider) and a child object called "marker". Marker is a 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 colours in each of the states. In the UnMark method marker is set to transparent. Fig. 18 shows how this works.

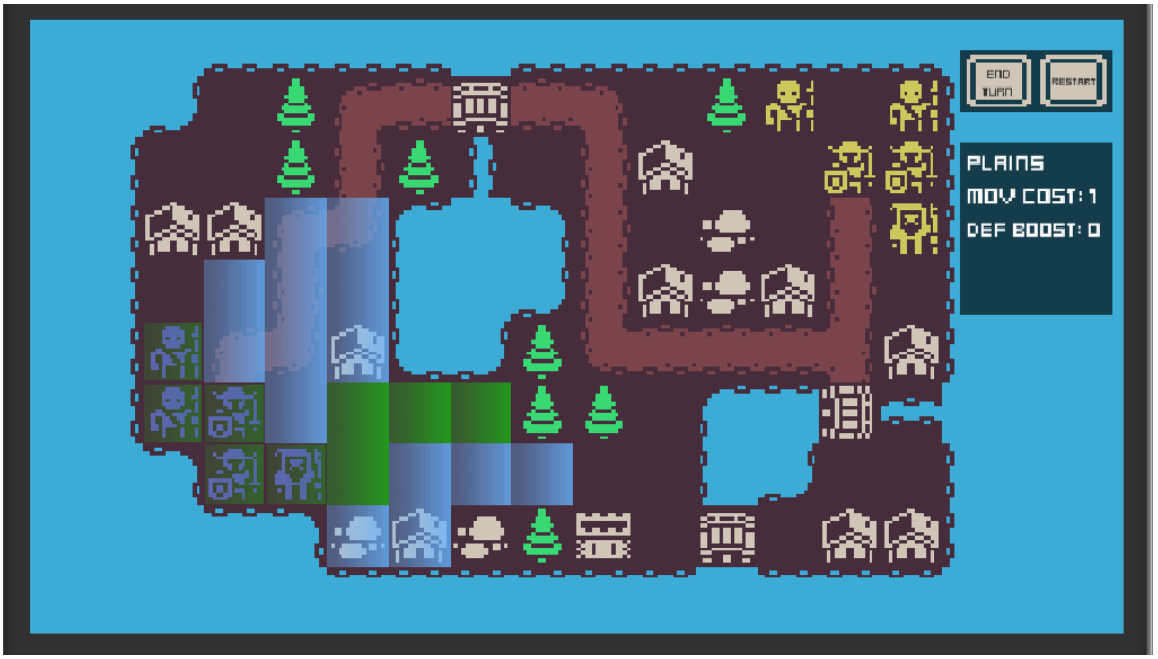

Fig. 18 - Sample cell customization

Next, let's try unit behaviour customization. In example shown in fig. 19, the 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 a class derived from Unit, that will represent an 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 a class derived from Alien, that will represent a flying alien unit. This time we have to override a few more methods, as there are 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. In `IsCellTraversable` we define that the unit can move through any cell that is not taken by another flying unit. Next, in `Move` method we set IsSkyTaken field to false on the cell that the unit is leaving and true on the cell that the unit is moving to (`Move` method in base class does the same thing to IsTaken field). Finally, when the unit is destroyed, IsSkyTaken field is set to false. There is no need to override `IsCellMovableTo` method because flying units are not allowed to finish the move over water.

Another example could be creating a unit countering system, similar to rock – paper – scissor game. Example scene 1 contains implementation of such a system. To get that effect, simply create three subclasses of Unit, and override their `Defend()` methods. In the method we check for the type of the other unit and multiply the damage accordingly.

```
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 the 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.

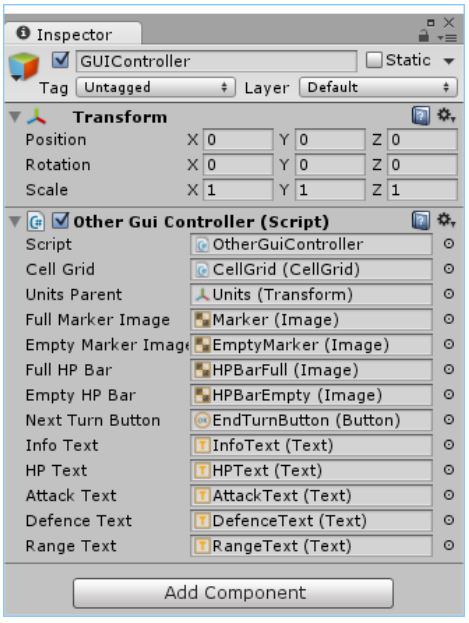

Fig. 20 - GUIController structure

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 the UI should react to them. For a 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. The AI Player

The project contains an AI implementation that can be customized and extended easily. While rule-based, it can offer a real challenge to the human player. Subsequent chapters describe how to use and customize the system.

6.1. Overview

To use the AI, have a player gameobject with an AIPlayer script attached to it. While generating the grid, Grid Helper will set up the players for you, so there's no need to do it manually. The script requires a UnitSelection component, basic implementation is added by default.

The AI is rule-based and uses AIAction scripts that are attached to individual units. The AIAction represents a single action that the unit can take. There are usually multiple actions attached to the unit, the AIPlayer processes each one in order, evaluates if the action should be executed and acts accordingly.

6.2. AI Actions overview

Each action corresponds to some unit abilities, usually one, but potentially more. On the other hand, a single ability can be used in multiple AI Actions. The project contains two basic AI Actions related to basic unit abilities - `MoveToPositionAIAction` and `AttackAIAction` and a few more specific actions used in example scenes. The AIAction public interface is as follows:

- `bool ShouldExecute(Player player, Unit unit, CellGrid cellGrid)`

Determines whether the action should be executed by the AI

- `Precalculate(Player player, Unit unit, CellGrid cellGrid)`

Processes the game state to determine what the action should be, for example which unit to attack. The idea is that this information should be available before the `Execute` method runs

- `IEnumerator Execute(Player player, Unit unit, CellGrid cellGrid)`

Executes the action using underlying unit ability. Just like `Ability.Act` method it is also run as coroutine, for the same reasons

- `ShowDebugInfo(Player player, Unit unit, CellGrid cellGrid)`

Displays debug information. Debugging AI Actions will be covered in next subchapter

- `CleanUp(Player player, Unit unit, CellGrid cellGrid)`

Clears any fields that `Precalculate` might set up and return the grid to default appearance

6.3. Basic AI Actions

The project contains two basic AI Actions related to basic unit abilities - MoveToPositionAIAction and AttackAIAction that correspond to MoveAbility and AttackAbility respectively. A basic brain with these two AIActions is added to every gameobject with Unit script attached to it.

Both abilities select their targets - cells to move to, or units to attack - using evaluators. There are a few different evaluators provided with the project and it's fairly easy to add more.

6.3.1. MoveToPositionAIAction

MoveToPositon action uses MoveAbility to move the unit. Most of the heavy lifting is done in the `ShouldExecute` method - it checks if there is any cell on the grid that has a higher score than the current cell. Score is calculated as a sum of scores from cell evaluators. It is worth noting that a cell can be selected that is beyond movement range. In such a case, the unit will either move as far as possible towards the cell, or select a cell with top score along the path. This behaviour is controlled by the ShouldMoveAllTheWay flag that can be set in Unity inspector. A cell evaluator has the following interface:

- `float Weight`

Weight assigned to this evaluator, by default the weight is 1. Can be set in Unity inspector

- `void Precalculate(Unit evaluatingUnit, Player currentPlayer, CellGrid cellGrid)`

Caches any data that could be reused by consecutive Evaluate calls

- `float Evaluate(Cell cellToEvaluate, Unit evaluatingUnit, Player currentPlayer, CellGrid cellGrid)`

Calculates score for given cell

Use a combination of evaluators included in the project, or implement one yourself. To use an evaluator, simply attach it alongside MoveToPositionAIAction. Evaluators included in the project are as follows:

- `DamageCellEvaluator`

Scores cell based on damage that can be inflicted from given cell

- `DamageRecievedCellEvaluator`

Scores cell based on damage that can be received on given cell

- `DistanceCellEvaluator`

Scores cell based on the amount of turns it takes to get there

- `AlliesNearbyEvaluator`

Scores cell based on number of adjacent allies

- `PositionProximityCellEvaluator`

Scores cell based on proximity to other cell

- `UnitProximityCellEvaluator`

Scores cell based on proximity to other unit

- `RandomCellEvaluator`

Assign random score to cell, for a bit of variety in gameplay

6.3.2. AttackAIAction

AttackAIAction action uses AttackAbility to attack enemy units. ShouldExecute checks if there are any units in attack range, Precalculate selects top target based on score from evaluators, and Execute initiates the attack. UnitEvaluator has interface similar to CellEvaluator:

- `float Weight`
- `float Evaluate(Unit unitToEvaluate, Unit evaluatingUnit, Player currentPlayer, CellGrid cellGrid)`

To use an evaluator, attach it next to AttackAIAction. There are two unit evaluators included in the project:

- `DamageUnitEvaluator`

Scores unit based on damage that can be inflicted

- `HPUnitEvaluator`

Scores unit based on hit points it has left

6.4. Debugging AI Actions

While watching the AI play, you may wonder why it made a specific decision. This is where the AI debugging mode comes into play. It is toggled on / off by the DebugMode flag on AIPlayer script.

The debug logs come from the AIAction.ShowDebugInfo method. Basic AIActions have it implemented out-of-the-box, you'll need to implement it yourself in your custom AIActions. The logs can be as simple as prints to the Unity console, there is also a built-in cell and unit colouring mechanism that facilitates visualising data.

When debug mode is on, you'll get prompted to press a button to select the next AI unit and trigger subsequent actions. Before the action is executed, debug info will be displayed. Fig. 22 shows distribution of cell scores for selected unit. Green indicates high scores, red lower. Top scoring cell is highlighted in blue, and the cell that the unit will actually move is highlighted in violet. Clicking on any cell will print it's score and it's individual components to the console.

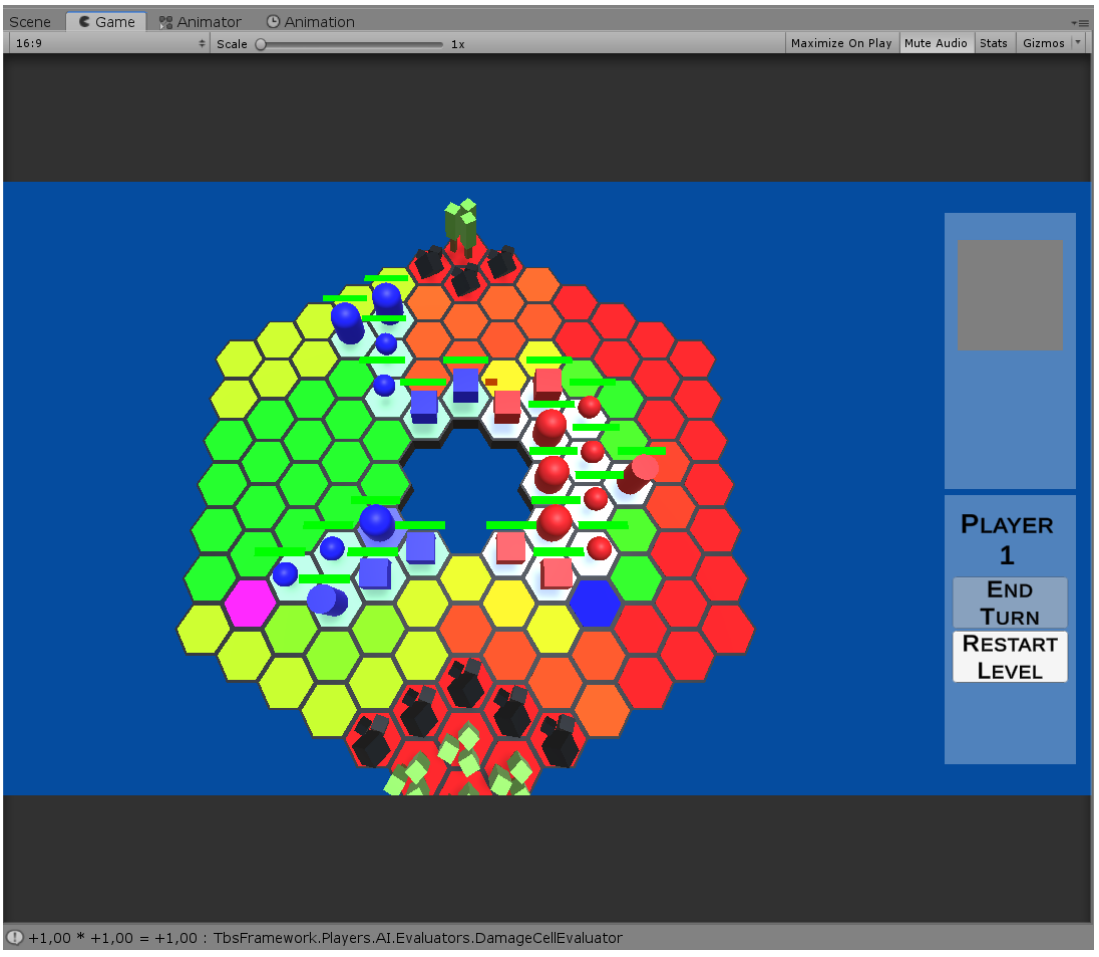

Fig. 22 AI Debug mode - cell score distribution

6.5. Unit Selection order

Another aspect to consider is the order in which the AI selects units. The component responsible for this is UnitSelection and has the following interface:

- `IEnumerable<Unit> SelectNext(List<Unit> units, CellGrid cellGrid)`

The project contains two basic implementation of unit selection:

- MovementFreedomUnitSelection

Selects units based on their ability to move. It is added to AI Player by default.

- SubsequentUnitSelection

Selects units one by one

6.6. Conclusion

The new AI system gives a solid foundation to create competent computer controlled players that are fun to play against. The next step would be to integrate the Framework with the Unity ML Agents package. Using reinforcement learning for the AI will be another breakthrough for the Framework. I'm looking forward to implementing it in one of the future releases.

7. Game state management

In this chapter we take a look at mechanisms responsible for user interaction with the game, turn transitioning and handling game over conditions.

7.1. User interaction

Selecting units, moving and attacking is delegated by CellGrid to subclasses of CellGridState class (through CellGrid.CellGridState property). If you are familiar with the 'state' design pattern, you will recognize this mechanism easily.

CellGridState is an abstract class that contains callback methods invoked when a cell is selected, deselected, clicked on, and a method that is called when any unit is clicked on. Apart from that, there are `OnStateEnter` and `OnStateExit` methods used for initialization and cleanup. These methods are called by the CellGrid object in the scene. All of the methods are virtual which means that they can be overridden in derived classes.

The project contains three implementations of grid states. In the Framework version 2.1, handling unit movement and attacking was decoupled from CellGridStateUnitSelected, and the state itself was replaced with CellGridStateAbilitySelected. The new state is used to execute abilities, as described in one of

the previous chapters. CellGridStateWaitingForInput only implements OnUnitClicked method and changes grid state to CellGridStateAbilitySelectedif a friendly unit is clicked. Lastly, CellGridStateBlockInput does not implement any methods, basically disabling user interaction with the game. It is used when it's AI turn and when the game is finished.

7.2. Turn transitioning

The turn ends when the CellGrid.EndTurn method is called. In the examples provided with the project, AI calls the method directly through its reference to the CellGrid object and human players use a GUI button to end the turn.

Version 2.1 of the Framework introduced a mechanism to customize turn transitioning - the turn resolver. The mechanism allows to select the next player and units that can be used in a given turn. To use a resolver, simply add the script to CellGrid gameobject. You can create your own resolver by implementing TurnResolver class, or use resolvers that are already implemented:

- `SubsequentTurnResolver`

Players take turns according to their player number. This is a default resolver added to CellGrid by the Grid Helper

- `SpeedTurnResolver`

Players take turns according to the speed of their units. Only one unit is allowed to move each turn

The following things happen when a turn is changing: `OnTurnEnd` method is called on all units that were playable last turn, next player is selected and `OnTurnStart` method is called on all units that will be active next turn. Finally, the `Play` method is called on the current player object. If the player is human, CellGridStateWaitingForInput is assigned as grid state and the game can be interacted with, otherwise user input is blocked and AI does its thing.

7.3. Ending the game

Starting with the Framework version 2.1 it is possible to customize game ending conditions. To do so, either implement `GameEndCondition` class, or use resolvers that are already implemented:

- `DominationCondition`

Basic condition that was used so far, destroy all enemy units to win. This condition is added to CellGrid by default by the Grid Helper

- `PositionCondition`

Reach a specific cell to win

- `ObjectiveDestroyCondition`

Defeat a specific unit to win

- `TurnLimitCondition`

Last a specific amount of turns to win

To use a condition, simply add the script to CellGrid gameobject. There can be multiple different resolvers active at once, and each condition can be applied to a different player. For example, one player needs to escort a Hero unit to safety (PositionCondition) in X turns (TurnLimitCondition), before enemy reinforcements arrive. Another player's objective is to either defeat the Hero (ObjectiveDestroyCondition), or stall long enough to stop him from escaping.

It is worth noting that checks on conditions are run only in the following events: unit moved, unit died, turn ended. If you want to run it at any other moment, you need to call `CellGrid.CheckGameFinished` manually.

Once the game is over, the `CellGrid.GameOver` event is invoked. The arguments that come with the event include a list of winning players and a list of losing players, which you can use in GUI to display the result.

8. Tutorial

In this section we will go through the process of creating the simplest possible scene from scratch. The scene will consist of a grid of cube cells, cube units, and cube obstacles. You can find the finished scene in the 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 the 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. 24

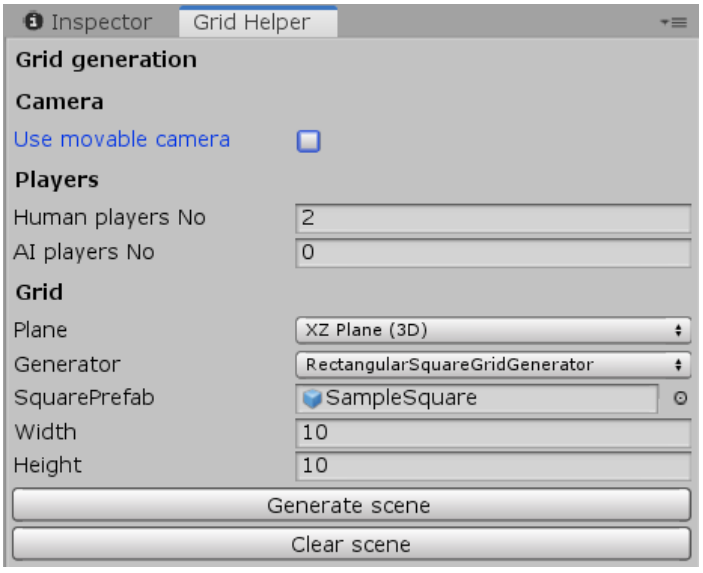

Fig. 24 - Grid helper with parameters filled in

8. Once the parameters are filled in, click on the “Generate scene” button in the Grid helper window. Scene hierarchy at this point is shown in fig. 25, and scene view is shown in fig. 26

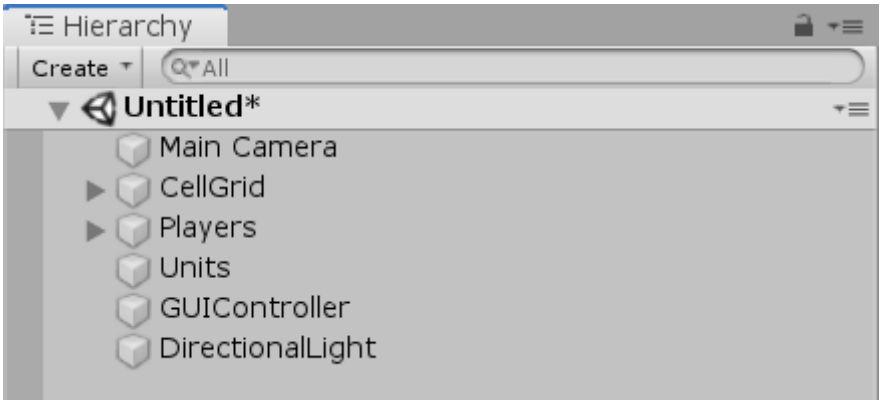

Fig. 25 - Scene hierarchy at step 8

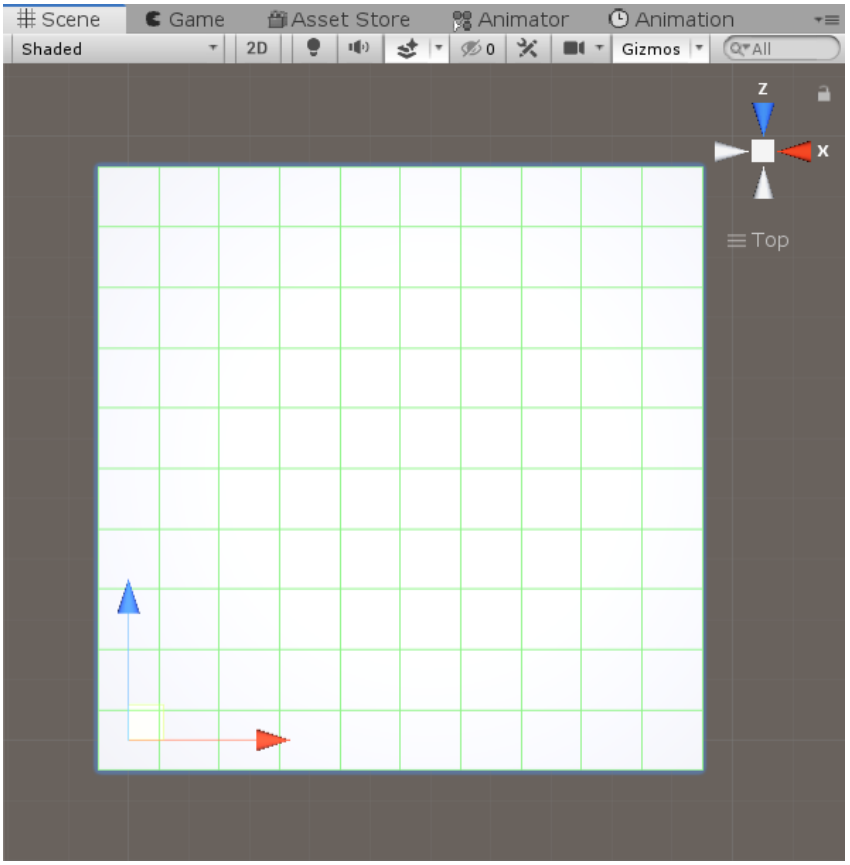

Fig. 26 - Scene view at step 8th

9. Now is the time to add units to the scene. Create a new script, name it SampleUnit and make it inherit from Unit. We'll make the unit turn light green if it is friendly, green when selected, red when it is attackable and gray when it is finished for the turn. Please note that we are accessing the Renderer component in the child object. We'll get to that in a second.

```
public class SampleUnit : Unit
    {
        public Color LeadingColor;
        public override void Initialize()
        {
            base.Initialize();
            GetComponentInChildren<Renderer>().material.color = LeadingColor;
        }

        public override void MarkAsFriendly()
        {
            GetComponentInChildren<Renderer>().material.color = new Color(0.8f, 1, 0.8f);
        }

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

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

        public override void MarkAsFinished()
        {
            GetComponentInChildren<Renderer>().material.color = Color.gray;
        }

        public override void UnMark()
        {
            GetComponentInChildren<Renderer>().material.color = LeadingColor;
        }
    }
```

10. Create two new materials and set them to two different colors.

11. Create two empty game objects and attach SampleUnit to them. Add cubes to the game objects and attach materials to them. Save the units as prefabs.

12. The reason we're adding cubes as children of units is that the unit gets snapped to the center of the cell that it is located on. We want to offset the cube position, so it appears on top of the cell, instead of inside of it. To do this, set the cube position to 0.9 units on the Y axis.

13. Finally, we need a collider on our parent game object. Add a box collider and offset it by 0.9 units as well, by setting the Y value of its center field.

14. Fill in parameters in SampleUnit script. Values that I selected are shown in fig. 27, but feel free to experiment. Make sure to set the Leading Color parameter of both units to different values - accordingly to colors of materials that you created.

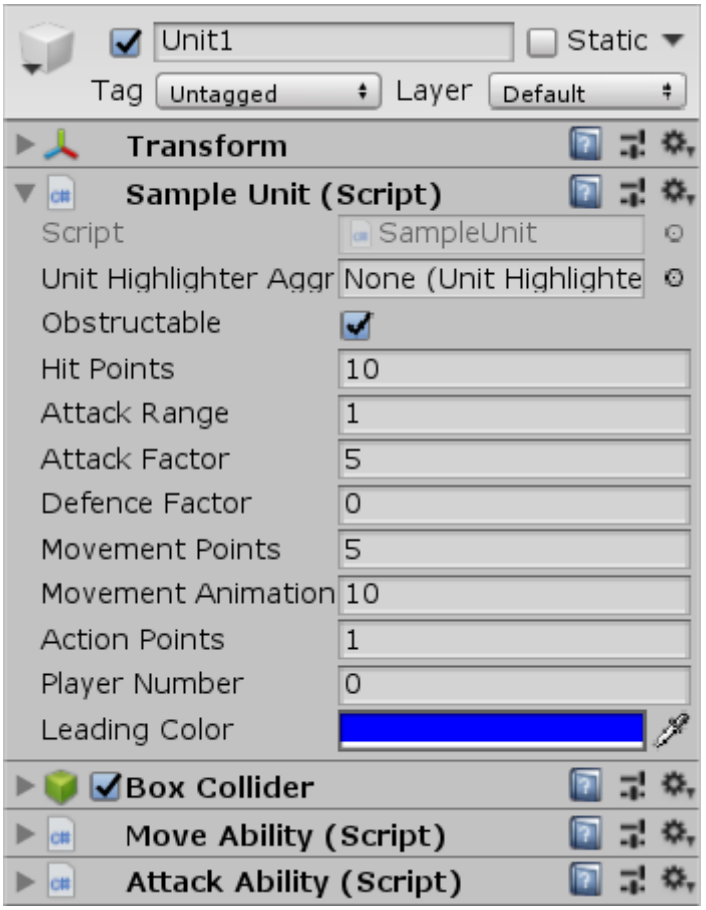

Fig. 27 - SampleUnit parameters

15. We will use Unit Painter to add 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 the “Enter Unit Edit Mode“ button. Unit Painter interface is shown in fig. 28.

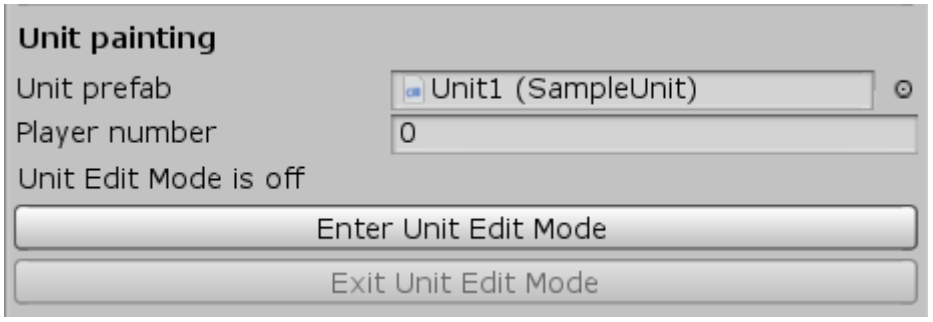

Fig. 28 - Unit Painter interface

16. 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. 29 shows my scene setup at this point.

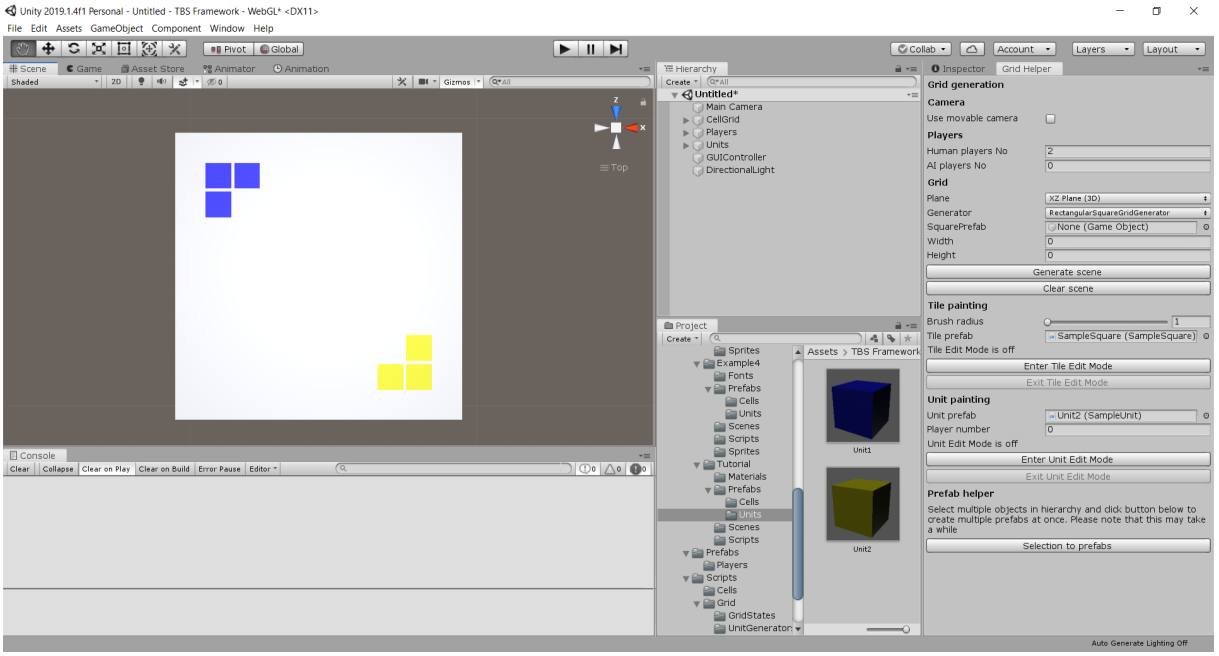

Fig. 29 - Scene setup at step 16th.

17. Let's add some obstacles to the scene. We will create another cell prefab for that purpose. Create a new cube and attach black material to it. Duplicate existing cell prefab and attach the cube. The prefab is shown in fig. 30. Remember to set IsTaken field on the new prefab to true.

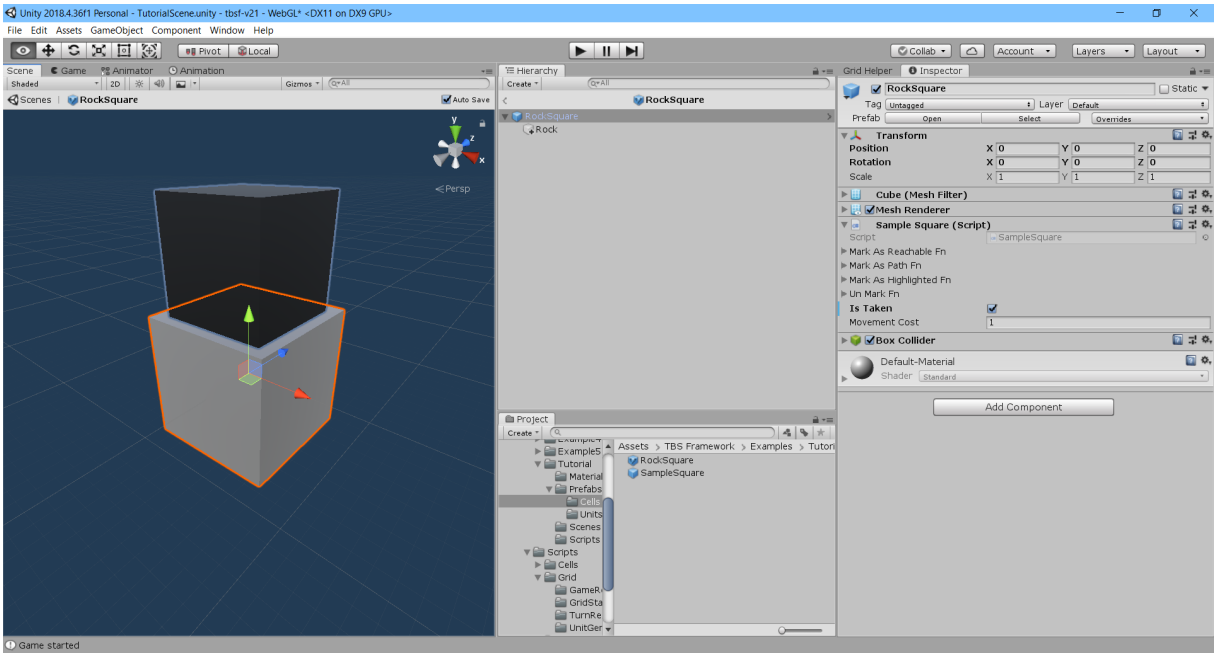

Fig. 30 - Another cell prefab

18. We will use Tile Painter to add the cell that we prepared to the grid. Tile Painter is accessible from the Grid Helper window, just like Unit Painter. Drag and drop your cell prefab into the “Tile Prefab“ field and click on the “Enter Tile Edit Mode“ button. Use Scene View to place some new cells. Fig. 31 shows Tile Painter interface.

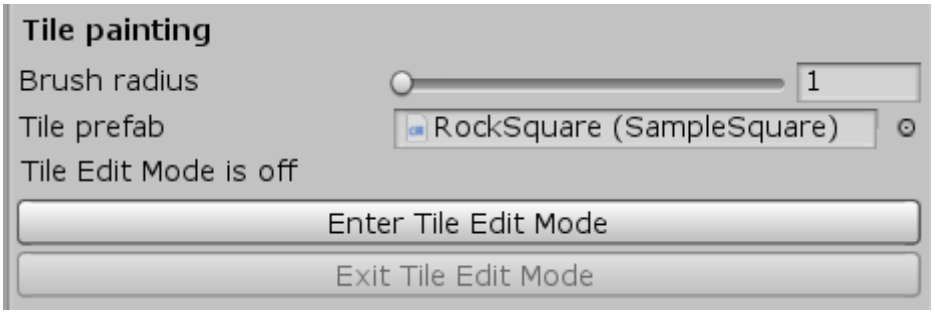

Fig. 31 - Tile Painter interface

19. 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 the "m" key on the keyboard. The code goes like this:

```
public class GUIController: MonoBehaviour
{
  public CellGrid CellGrid;
  void Update()
  {
    if (Input.GetKeyDown(KeyCode.M) && !(CellGrid.CellGridState is CellGridStateAITurn))
    {
      CellGrid.EndTurn(); //User ends his turn by pressing "m" on keyboard.
    }
  }
}
```

Fig. 32 - Simple GUI controller code

That concludes the tutorial. Fig. 33 shows the scene setup after the last step. Fig. 34 shows the level running. The created scene is playable, though perhaps not particularly interesting to play. It's up to you to change it.

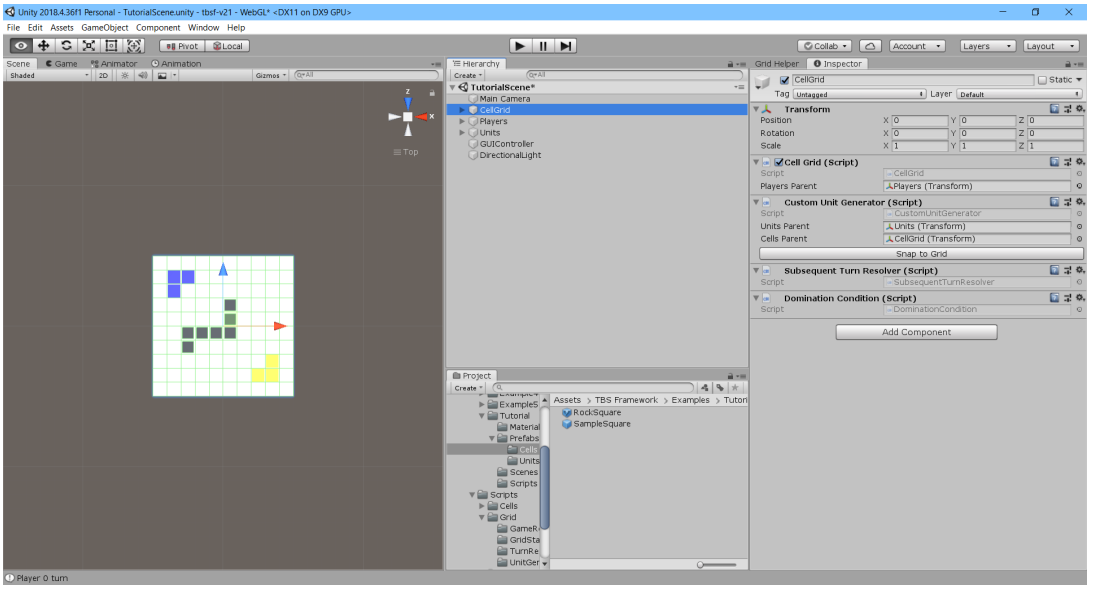

Fig. 33 - Final setup of the scene

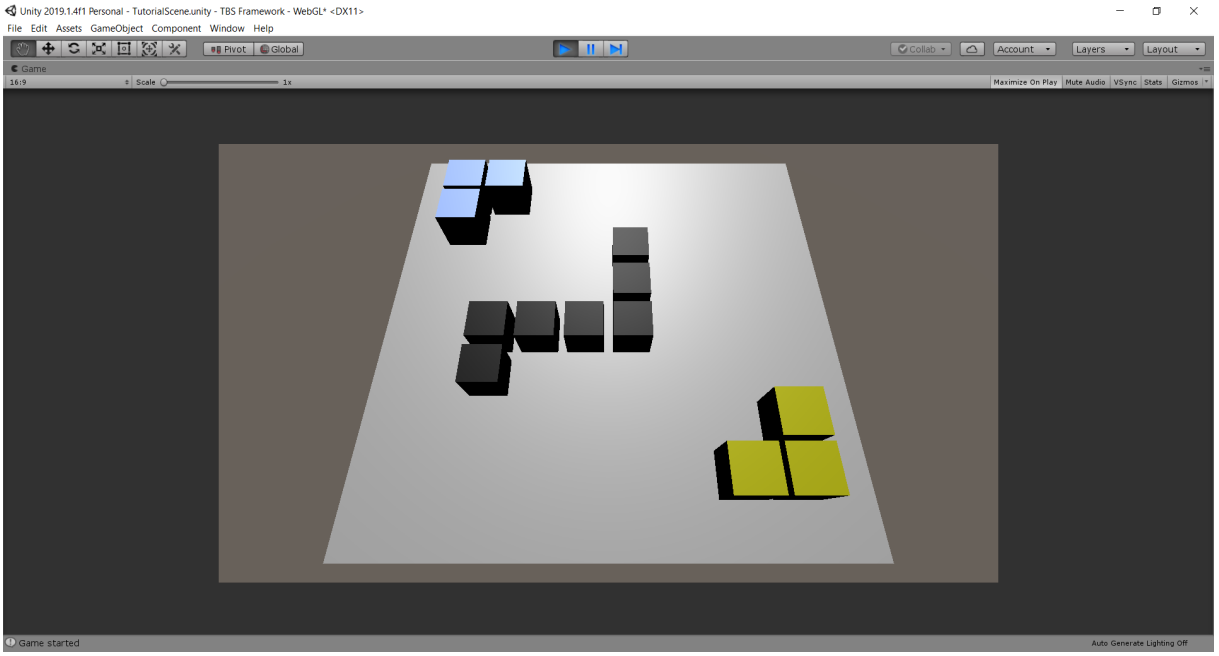

Fig. 34 - Finished scene

9. 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)."

10. 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/ and a Discord server at https://discord.com/invite/uBJNPJHFjB. 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.

11. 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 code, suggestions or ideas for new features. Any feedback will be appreciated. I hope you find my work useful.

12. 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
[8] Kronbits, Backgrounds, https://kronbits.itch.io/backgrounds
[9] Aekashics, Librarium Bundle, https://aekashics.itch.io/

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

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 the 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 the XY plane, while in v1.2.1 it was the 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 the XY plane now. Rotating your old prefabs on the x axis should do the trick.

APPENDIX B - Upgrade from v2.0x to v2.1

1. **First of all, make backup of your project before upgrading**
2. Scene setup
    a. Add `SubsequentTurnResolver` component to CellGrid gameobject
    b. Add `DominationCondition` component to CellGrid gameobject
    c. Expose `Is2D` field in CellGrid.cs script by removing `HideInInspector` attribute
    d. In Unity Editor, set `Is2D` field on CellGrid gameobject to true, regardless if your scene is 2D or not
3. Add `MoveAbility` and `AttackAbility` to your unit prefabs