Tower Defense Test for Plunge Interactive

Octavio Martínez García (o.martinez.gar@gmail.com)

Overview

- The game goal is to defend a series of crystals agains constantly spawning enemies, that try to steal them from the player. To achieve that, the player can deploy several types of turrets that will automatically attack the enemies in their range. The player may upgrade the turrets to increase their fire damage or deploy new ones in the game scenario, as long as he has enough money to afford either the upgrade or the new turret. Money is gained by defeating enemies so as to fortify the defenses and prevent the enemies from stealing the player crystals.

Review

- The game has a scenario composed by a resizable **grid** of cells, which are planes with a Cell script attached. Those cells may be empty, occupied by turrets or by crystals. The player may deploy new turrets on empty cells only. When the player moves the mouse over the game grid the empty cells will be highlighted to show whether a new turret can be deployed there.
    - the ground object, which has a Grid class script attached, in the awake function initialises the cell array to the specified dimensions in the editor, creating a bidimensional array of the given dimensions and instantiating all the cells, as well as storing them in its cells field, in a double loop.
    - I had previously though of creating a plane dynamically and make a «cell» each of the plane's mesh divisions, but with the current solution we can have an array of objects which can have its own behaviour (cells) and also it looks simpler to achieve this way, so i opted for doing as mentioned
    - for highlighting the cells, we have used raycasting, sending a ray from the cursor position to the cells. For that we had to place them in a separate layer, to ensure that the rays only hit entities placed on that layer. That is done in the *Update* function of the GameManager class, inside the GameMaster GameObject, which handles the high level logic of the game (basically in game gui events and placement / upgrading of turrets)
    - since the turrets may have various sizes (1x1, 2x1, etc.) its necessary to find not only the cell hit by the raycast, but also its neighbor cells. This is achieved gettting the neighbors of the cell hit by the ray, calling *GetNeighborHood* in the ground (Grid) attribute of the GameManager.

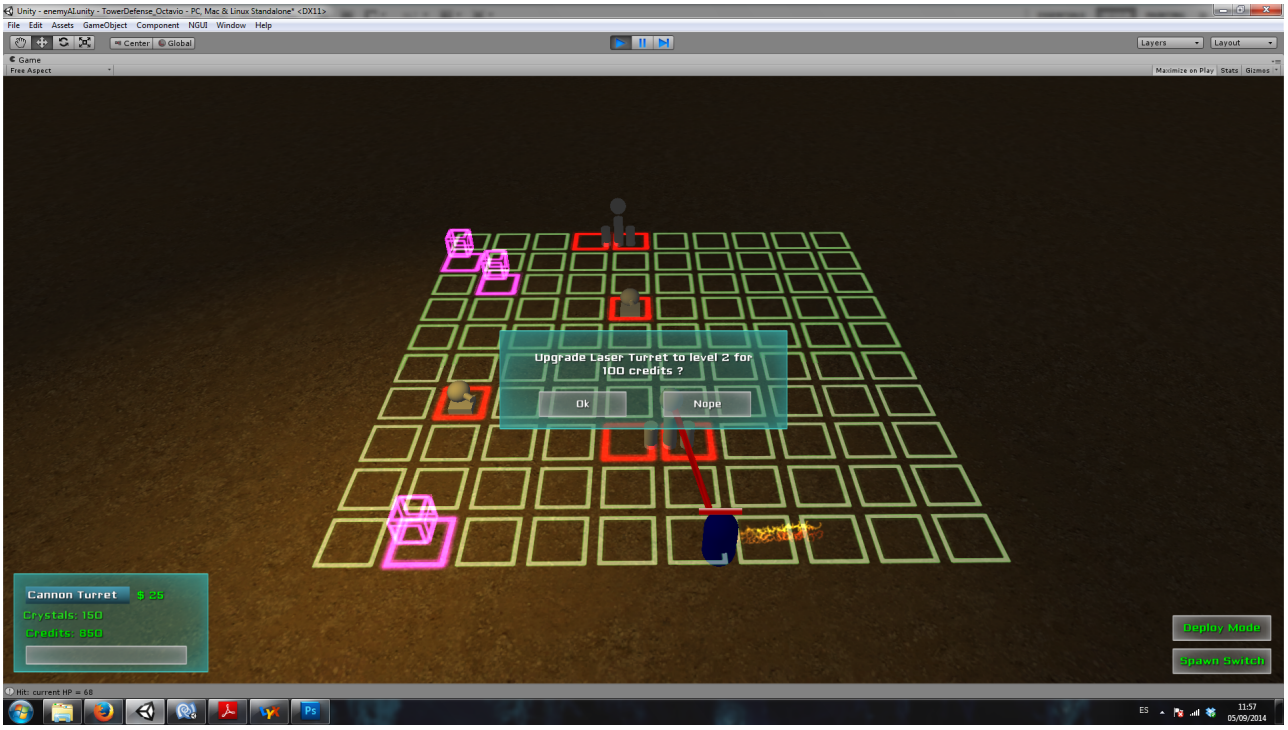
```
// get the neighboring cells of a given one based on the turret size (used when placing turrets)
public List<GameObject> GetNeighborhood (Vector2 size, GameObject targetcell)
{
        // we get the cell indexes
        Vector2 pivotCellIndex = new Vector2 (targetcell.GetComponent<Cell> ().xIndex,
                targetcell.GetComponent<Cell> ().zIndex);

        List<GameObject> neighborCells = new List<GameObject> ();
```

```
        // start at pivot cell (0,0) and get the rest based on the size
        // relative to that pivot one (e.g.:    size (2x2) ---> +0+0, +0+1, +1+0, +1+1)
        for (int i = 0; i < size.x; i++) {
                for (int j = 0; j < size.y; j++) {
                        int xCoord = (int)pivotCellIndex.x + i;
                        int yCoord = (int)pivotCellIndex.y + j;
                        if (InsideGrid (xCoord, yCoord)) {
                                neighborCells.Add (cells [xCoord, yCoord]);
                                Debug.Log ("Cell index added: (" + xCoord + "," + yCoord + ")");
                        }
                }
        }
        return neighborCells;
}
```

- *GetNeighborhood* function is shown above:
  - once we have the neighbor cells, we mark them as highlighted if they are empty by calling *TagNeighboringCells* in GameManager, which changes the tag and material of each cell in that neighborhood
  - then if we left click on an empty cell we deploy a turret there, but again, since there are several sizes, we cannon simply place it in the cell, which is why we find a middle point of the cells position by an average of their position vectors, and place the new turret on that middle point, calling *GetCellsMidPoint* on the GameManager. Previously to that deployment of the turret we have to make sure that every cell in the neighborhood is empty (*GameManager.ValidDeploy*)
  - finally if we click on a turret cell we enter the upgrade process, which brings in the level up panel and shows the cost of upgrading, giving us the chance to upgrade if we can afford it. For that since the turret is placed in the middle point of the cells (except for 1x1 ones) we have to had stored that same turret in every cell of the neighborhood, to retrieve it later when we click on any cell of those, and to modify it when we upgrade it.
  - when a turret is modified it increases its level, and with it various attributes dependant of it (damage, cost, etc.)
- There are 2 types of **turrets**:
  - Cannon turret:
    - it shoots bullets at a fixed pace at enemies within its range. It occupies 1x1 cells.
    - when an enemy (collider tagged «Enemy») enters its triggered collider it grabs its transform and releases it on trigger exit
    - then in the *Update* frame it checks if it has a target and if so it aims at the target (rotates the moving part ot «sentinel») after a lapse has passed and fires a bullet at it (again after a delay)

- it fires by creating a new instance of its projectile (a bullet) from the position of its cannon sockets, which are an array of GameObjects that are used to get the shoot initial position
    - now it only has 1 cannon, but this could be extended to several more simply by attaching more cylinders and more sockets to the array, and so it will instantiate one bullet per cannon socket
- when the bullet hits an enemy (when an enemy tagged object enters the bullet's triggered collider) it sends him a receive damage message and destroys itself

- Laser turret: it shoots a continuous laser beam at enemies as long as they stay at range. It takes up 2x1 cells.
    - when an enemy enters its triggered collider it also grabs its transform and releases it on trigger exit also
        - this could be refactored into the base class, but maybe other turrets would have a different behaviour here, so we keep it in the children
    - in the *Update* function, as long as an enemy is in range the turret shoots a continuous laser beam at it by creating an instance of its projectile (a laser beam)
    - this laser beam is basically a lineRenderer (seems the most straightforward solution to render a beam) with a LaserBeam class attached. This has only 2 vertexes, at the turret and at the target, that simulate a straight laser beam, and 2 particle effects, one for the emission at the turret position, and another for the laser hit (a flame) at the target
    - the *Update* function of the beam is responsible for setting its vertexes and creating the hit effect at target, which, when happens, sends a message to the enemy to receive damage, and since its called once per frame (Update) while he is in range, it results in damage over time kind of damage (so we keep the base damage of the turret low - 0.5f)
- Both types inherits from a base turret class, so as to have the basic turret logic and attributes in one base class, and also the chance to extend it with more types in the future
- Also there are situations where we have to retrive a general class and call some methods of it (e.g. *LevelUP*) letting each object produce its own behaviour, so here the inheritance is a natural solution



- The game has a **GUI** with several options shown in the screenshot above: (made with the NGUI plugin)
  - bottom left panel:
    - a list of turrets than can be deployed with its respective costs
      - when an option of the list is selected it calls the function *OnSelectionChange* in the GameMaster object (which has a GameManager class, which updates the currently selected turret type and the GUI cost label
    - 2 labels showing the current amount of crystals (resource to protect) and credits (money)
      - they are updated in the GameManager via the *UpdateScore* function which updates the various UILabel text in the GUI, and is called whenever any of those quantities changes (in start, at kill enemy, at deploy turret, etc.)
    - a progress bar used to time up the turret **upgrade**. You cannon upgrade a second turret until the 1st one is done. (i.e. until the bar fills up)
      - this is achieved again in the GameManager through a coroutine that fills up the bar over time when we confirm the upgrade in the upgrade panel
      - since it has to fill over a specified period and its used outside *Update*() , the use of coroutines is a natural way of achieving this

- bottom right panel:
  - deploy button: switches between deploy mode on / off. This enables / disables the grid and therefore the placement of new turrets
  - spawn switch: toggles the spawning of enemies between on and off
    - its mostly used for developing and debugging purposes, since the use of it in-game would obviously break the game balance
- mid panel:
  - its the upgrade panel that shows up when you click on a turret giving you the chance to upgrade it, provided that there is enough money
    - depending on the button you click the process of leveling up a turret initiates and the panel hides again
- grid:
  - as it is shown the grid is composed of smaller planes which highlight when the mouse is over them, or show themselves marked in different colors when occupied by turret or resource.
  - internally the cells are stored in a 2D array stored in the ground object, which has a Grid class attached. As it is shown in the image, the grid is rotated from what will be its normal position, and shows the rows horizontally and the columns vertically.
    - I thought of changing that, but since you can move the camera around, from a player point of view it makes little sense to define any of the sides as rows or columns. Maybe with a fixed camera it would make more sense to arrange it as a normal matrix with rows vertically and columns horizontally
- I chose to use NGUI over the regular Unity GUI system cause it seems like a better approach, since it's more flexible and straightforward

- The player can move the **camera** with (W,A,S,D or direction keys) and rotate it around with the mouse and r-click pressed. The camera can also be reset by clicking the mid mouse button
  - you can also adjust the speed and rotation speed of the camera from the editor.
- The **enemies** will spawn periodically from random locations at the right of the scene (2 last rows) and will move searching for a resource cell, according to a Breadth First Search pathfinding algorithm.
  - the Grid has the responsibility of creating the spawn portals and spawning the actual enemies (since it knows the state of the cell array). When it spawns an enemy (Instantiates a clone of the current enemy) it passes the starting

cell to the enemy. This is done to enable the enemy finding its path via the BFS algorithm.

- the Grid also spawn enemies with random level, which have different color, hit points and cash rewards based on level.
- when the enemy is created, and after some initialisation (in *Awake*()), is finds the cell grid and it gets its path via BFS (in *Start*())
- the enemy has a next waypoint field which is used to move from one point to the next one. Those waypoints are retrieved from the path, which is stored in a queue of cells (since its the most efficient structure for storing and retrieving cells in a FIFO way).
- each enemy has a life bar that follows him in his movement and is updated when he takes damage
- in the *Update* function the enemy will move smoothly from the current position to the next waypoint and look for a new waypoint when he reaches the next one. If we want to make the movement of the enemy update dynamically when we place new turrets we have to recalculate the path before getting the new waypoint, by calling to the BFSGraph function.
- the **BFS algorithm** used is shown below:

```
// BFS pathfinding algorithm.
// iterates over the cells of the grid starting at the root (spawn node)
// adding the parameters and children of each node and thus creating the tree
// Returns the path from start node to end one as a queue
private Queue<Cell> BFSGraph (Cell startNode)
{
        Cell final = null;

        Queue<Cell> graph = new Queue<Cell> ();
        // mark the first as root
        startNode.parent = null;
        startNode.isStart = true;
        startNode.isEnd = startNode.tag == "Cell_Resource";
        startNode.visited = true;

        // enqueue the root node
        graph.Enqueue (startNode);

        // get its children
        Cell[] startChildren = GetNodeChildren (startNode,
                        startNode.xIndex, startNode.zIndex).ToArray ();

        // Enqueue them all in a loop
        foreach (Cell startChild in startChildren) {
                        graph.Enqueue (startChild);
        }
        // while the queue is not empty
        while (graph.Count > 0) {
                // Get the first node of the queue
                Cell c = graph.Dequeue ();
                // Check if its a resource one (target)
                if (c.isEnd) {
                                // Make him final node if target
```

```
                                // and exit loop if so
                                final = c;
                                break;
                        }
                        // get the children of c and enqueue them
                        Cell[] children = GetNodeChildren (c, c.xIndex,
                                                c.zIndex).ToArray ();
                        foreach (Cell child in children) {
                                graph.Enqueue (child);
                        }
                }
                // If final is null we could not find a path
                if (final == null) {
                        Debug.Log ("Final node is NULL");
                        return null;
                }
                // Get the parent node of the final node
                Cell parent = final.parent;
                List<Cell> cellList = new List<Cell> ();

                // While the parent is not the start node
                // This will go back up parent of each node
                // and then define the parent as current node
                while (parent != null) {
                        cellList.Add (final);
                        final = parent;
                        parent = final.parent;
                }
                // We add the last one
                cellList.Add (final);
                // The list as is goes from target to start
                // so we reverse it to get the right order
                cellList.Reverse ();
                // we clear the initial queue
                graph.Clear ();
                // and we enqueue every list node
                Debug.Log ("Path is as follows");
                foreach (Cell cell in cellList) {
                        graph.Enqueue (cell);
                }
                // clear visited nodes
                ClearGrid ();
                return graph;
        }
```

- *GetNodeChildren* is explained below as well:

```
// get the children of the given cell (node), based in 8-connectivity
private Queue<Cell> GetNodeChildren (Cell node, int xIndex, int zIndex)
{
        // we pass through all neighbors in an 8-connectivity basis and add them to the node childre
        // as long as they are not visited or outside the grid or occupied by turrets
        for (int i = -1; i < 2; i++) {
                for (int j = -1; j < 2; j++) {
                        if (InsideGrid (xIndex + i, zIndex + j)) {

                                GameObject plane = enemyGrid [xIndex + i, zIndex + j];
                                Cell child = plane.GetComponent<Cell> ();

                                if (!child.visited && child.transform.tag != "Cell_Occupied") {
```

```
                                        child.parent = node;
                                        child.isStart = false;
                                        child.isEnd = child.transform.tag == "Cell_Resource";
                                        child.visited = true;
                                        node.children.Enqueue (child);
                                    }
                                }
                            }
                        }
                        return node.children;
}
```

- NOTE: dynamic pathfinding (uncomment sentence in SetNextWP()) is theoretically possible with this implementation, but I haven't been able to test it properly since it again causes memory overflow problems at least in my PC.
- The **crystals** will spawn from random locations at the left side of the initial scene (2 first rows). When a crystal is reached by an enemy its amount of resource will diminish unitl its completely depleted

Comments

- For convenience, ease of debugging and clarity, almost all the code is commented
- I believe that the easiest part has been the camera movement, as it can be done just by getting the inputs from keyboard / mouse, and translating / rotating the camera accordingly
- On the other hand the implementation of the enemy AI has been pretty tough, partly because it's a wide topic and thus a lot of documentation, but most of it is generic, and i had to adapt it to this particular scenario. Also i have had some performance issues with the BFS algorithm, not sure if they are caused by my PC (just 4GiB RAM), of by an inefficient implementation. Also the implementation of dynamic pathfinding hasn't been tested precisely due to that bad performance issues.