Tower Defense with Augmented Reality

Bogdan-Mihai Păduraru
Faculty of Computer Science,
"Alexandru Ion Cuza" University of Iasi
General Berthelot, No. 16
bogdan.paduraru@info.uaic.ro

Adrian Iftene
Faculty of Computer Science,
"Alexandru Ion Cuza" University of Iasi
General Berthelot, No. 16
adiftene@info.uaic.ro

ABSTRACT
The current project presents the creation of a game where the interaction is based on standard elements like keyboard and mouse and also with cardboard drawings used to represent the game map. The main aims of this project were both to offer a better user experience for the game players and to help students or interested persons to understand better how the A* algorithm works. The idea of this project came when Augmented Reality applications started to appear on the mobile market, offering a different experience than current standards.

Author Keywords
Augmented reality; Unity; A* algorithm.

ACM Classification Keywords
H.5.2. Information interfaces and presentation (e.g., HCI): User Interfaces. H.3.2. Information Storage and Retrieval: Information Storage.

General Terms
Human Factors; Design.

INTRODUCTION
We live in an era where stress affects us more and more every day. Relaxation by playing video games has become a normal activity for most of us. Most people now own a smartphone or tablet which they carry it everywhere [3]. These devices contain applications that help them stay in touch with friends or family (Facebook, Skype, chat apps), visualize multimedia content (movies, music, online TV) and also video games that must be up to date with the consumers need of relaxation and spending small breaks time.

This project will present an application that was built with the help of Augmented Reality with its purpose being to offer a relaxation alternative for those who use it. Right now the number of similar applications is pretty small, the most popular one being AR Defender2.

In what follows, the paper is structured as follows: a state-of-the-art section where similar application that use Augmented Reality are presented, a section for system architecture where it shown the basic elements of the system we build, and a final part where it shows conclusions and suggestions for future work.

STATE-OF-THE-ART
The world of Augmented Reality started to be more and more present in applications for tablets or smartphones. We will now present the most important applications that are similar to our application.

AR Defender 2
This application was one of the first to come on the mobile market, being developed and published by Bulkypix at the beginning of 2013 [1]. By mixing augmented reality components with 3D objects that represent different structures and characters inside the game this application immediately drew the attention of users. One of the features that made this application so successful was the networking that made it possible to play together with friends at the same time.

PulzAR
Initially published as a single game by the existing developer Exient Ltd, this application was later converted into a package of games, all of them using Augmented Reality technology [5]. One key feature this application had when it launched was the platform where the games could be played, PS Vita. Another interesting thing to notice was the launch date, 12 June 2012, only four months after the initial launch of the PS Vita console.

Toyota 86 AR
Another application that uses Augmented Reality technology, Toyota 86 AR was developed by Toyota Company in order to showcase the driving experience of their newest car model [6]. Made for Android and IOS, this application contains a single scenario where as soon as one of the games drawing maps made by Toyota are recognized, the user can start controlling the Toyota 86 car model in a virtual environment which serves as a real simulator for those who want to inspect and test the form, components, acceleration ratio, drifts etc.

SYSTEM ARCHITECTURE
In this section the main objective is to give a better understanding of the events that occurs in the current application, to familiarize with some of the used concepts and to offer some details about their implementation.

Game matrix

The first thing that must be prepared once the level starts is to build the game map both visually on the screen and internally in memory. Because in each scene there are only two objects that must be recognized by the Vuforia SDK [8], we can hold direct references to them. Once we know the object on which the map will be build, we can access its components and get details about the map size with the GetSize method that return a bi-dimensional vector where we find the width and height. Another important thing that we must be aware of is the dimension of a 1x1 piece of the matrix in relation with the world space. For this we have created an enum structure for an easier configuration, leaving the user to make the decision before the game starts.

Once this is done, we must decide where to place the start and finish point (represented in the image with "S" and "F") on the XY axis in our map from a visually point of view. Let's consider the rectangular shaped below as one of our game drawings where we will build our map. One more thing to remember is that each object in Unity is represented by a position vector that describes the coordinates of its center [7].

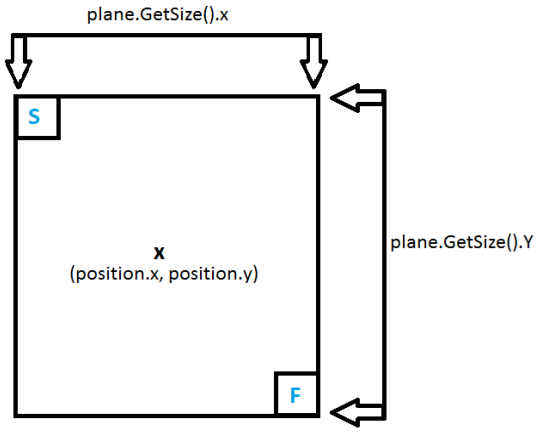

Figure 1. General representation of a game map

The only thing that remains is to create the matrix that will correspond to the game map. Since the A* script contains the functionality of finding an optimal path based on a given map, it will be responsible with the matrix creation and all its details.

Detecting the map drawing

For the map drawing detection in a level we have used the *OnTrackingFound* method that is found inside the *CustomEventHandler* class which was made available by the Vuforia SDK. This script is attached to an object that will soon by detected and offers some details about the object, like its name by using an instance of the class *TrackableBehaviour*, called *mTrackableBehaviour*.

Sending the message when an augmentable object was found is done like this:

```
MapCreator.instance.MapFound(mTrackableBeha
viour.TrackableName);
```

Once this message was sent, we continue with adding obstacles for our current level and then, we search for the optimal path for our enemies that will soon be instantiated. We can see in Figure 2 how this map looks in our application.

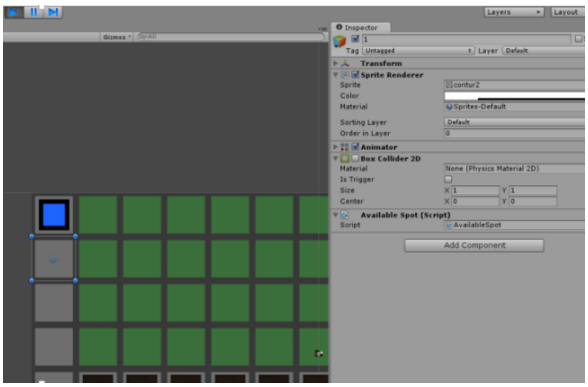

Figure 2. Map space object.

Adding obstacles for current level

When sending the message that our augmentable object was found, we establish our current level index, after which we start instantiating obstacles on the map.

Using two "for" instructions from 0 to the maximum number of lines respectively columns, we will move through each space of our game map. One important thing while doing the obstacles initialization is to register them on our map:

```
Astar.instance.RegisterObstacle(i, j);
```

- where *i* represents the index of our current line and *j* the index of our current column. In order to instantiate an object we used the Instantiate method from the *MonoBehaviour* class:

```
Vector2 pos = GetWoorldCoordsFromMatrix
Coords(new Vector2(i,j));

GameObject c = Instantiate(obstacle, new
Vector3(pos.x, pos.y, 0), Quaternion.
identity) as GameObject;
```

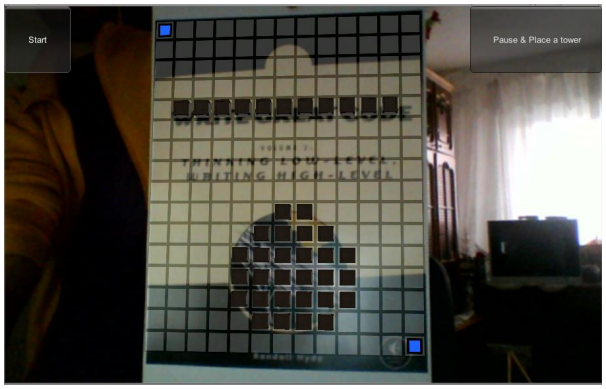

Figure 3. Showing the complete map of a level.

The map from each level is built according to the black portions from the image (see Figure 3). In the matrix representation of the level we say that a space is available when its value is 0, and an occupied space has its value equal to 1. Thus when the pathfinder algorithm will run this representation will help it search for the optimal path and avoid all obstacles.

Finding the optimal path using A* algorithm

A* is the most popular choice when it comes to pathfinding since its very flexible and can be used in a big range of contexts, which is the primary reason we used it.

The base idea of A* [4] comes from Dijkstra algorithm [2] because it can find the shortest path, but it also contains elements from Greedy algorithm since it is based on a minimum constant function for guidance during the search, called also a heuristic function. The success of this algorithm consists in combining these two information's from Dijkstra algorithm [2] (which favors nodes that are closer to the *start* node) and Greedy algorithm (that favors the nodes that are closer to the *end* node). In the standard terminology when we discuss this algorithm, $g(n)$ represents the function that computes the exact cost from *start* to any node, and $h(n)$ represents the heuristic function that estimates the cost from a current node to the *end* node. The logic behind this algorithm is to balance these two function when moving towards the final node. Thereby, at each iteration this algorithm will chose the solution that has the smallest value which is associated to the function $f(n) = g(n) + h(n)$.

In a map represented as a matrix there are numerous heuristic functions for movement, however we will mention only two of them, the first one being used in this project.

First the Manhattan distance that allows movement in four direction in a matrix represented map: *north*, *south*, *east*, and *west*. This choice is the most popular because it is easy to compute with the following function:

```
function heuristic(node, goal){
    dx = abs(node.x - goal.x);
    dy = abs(node.y - goal.y);
    return D * (dx + dy);
}
```

An important thing to notice is the value D which is a constant and must be assigned at the beginning. For a shortest path computation and keeping the heuristic function "admissible" it is recommended to assign a small value for this. Besides, there are different proposals related to this constant that take into consideration more aspects of a map like different terrain that imply different traversal costs or the number of the obstacles in a scene, but mostly this constant variable is assigned with 1, representing the simplest case scenario which was also used in this project.

When we need a more detailed traversal of the map we can use eight directions which implies a heuristically function that takes into account diagonal computation. This function has the same effect as the one mentioned earlier when it receives the four classical directions, but it can also support new directions: north-east, north-west, south-east, and south-west. The function that computes the distance between two given nodes with this type of movement is:

```
function heuristic(node){
    dx = abs(node.x - goal.x);
    dy = abs(node.y - goal.y);
    return D * (dx + dy) + (D2 - 2 * D) *
min(dx, dy);
}
```

The constant variable D was discussed above, but we can see a new variable, D_2. The value assigned to this is strictly connected to the value of D with the given formula: $D_2 = sqrt(2) * D$.

Once these details have been discussed we can move on to present the A* algorithm. There are two data sets, OPEN and CLOSED. In the first set we add nodes that are possible candidates when computing the path. Initially this set contains only the start node. The CLOSED set contains nodes that have already been examined. Initially this set is empty. If we had to give a visual representation of these sets we could say that the first represents the frontier and the second shows the inside area of the visited zones. Also each node will have a reference to its parent or better said to the node through we reached it in

our search, being able to determine the path by going recursively through each parent of a node once we found the final node.

The algorithm consists of a while loop that will pop out at each step the best node from the OPEN set (the node with the lowest value of its f function). If that node is the final one, the algorithm will stop. Otherwise, the node will be deleted from the OPEN set and added to CLOSED. Then we examine all its neighbors. In any of these neighbors is in the CLOSED set, they will be skipped. Also, if one of them is in the OPEN set we skip them, avoiding a double check. If none of these conditions are met, the node will be added to the OPEN set and we will set its parent reference with the current node and compute its cost equal to $f(n) = g(n) + h(n, n')$, where n' represents the extracted node.

```
OPEN = priority queue containing START
CLOSED = empty set
while lowest rank in OPEN is not the GOAL:
  current = remove lowest rank item from
OPEN
  add current to CLOSED
  for neighbors of current:
    cost = g(current) +
movementcost(current, neighbor)
    if neighbor in OPEN and cost less than
g(neighbor):
      remove neighbor from OPEN, because
new path is better
    if neighbor in CLOSED and cost less
than g(neighbor): **
      remove neighbor from CLOSED
    if neighbor not in OPEN and neighbor
not in CLOSED:
      set g(neighbor) to cost
      add neighbor to OPEN
      set priority queue rank to
g(neighbor) + h(neighbor)
      set neighbor's parent to current
reconstruct reverse path from goal to start
by following parent pointers
```

In this project the OPEN and CLOSED set have been represented as simple generic lists that contains element of type *AstarNode*, which is a data structure create by us with the following representation:

```
public int x;
public int y;
public AstarNode parrent;
public double g;
public double rank;
```

where "*x*" and "*y*" represent the coordinates from the game matrix, "rank" represents the value of the function $f(n) = g(n) + h(n)$, the rest of them being easily to figure out from their name.

In order to help the players of the game, all details of this algorithm can be presented to the students in class before using this application. In the future we intend to add an option to the application, where the players can see these details of the A* algorithm.

Starting the game and details about the game mechanics

Once the first six activities from the beginning of this chapter have been successful, in the top left corner of the screen a button named "*Start*" will appear, its role being to start the game (See Figure 3).

Since the last activities refer to the start and end of the game, something that is knows through the *EnemyController* class, this one will be responsible the send messages to the *UIManager* class to update the screen interface according to the game state. Once the game has been started the *EnemyController* class will handle the instantiation and management of the enemies. When an enemy has been created he will receive a confirmation message that will initialize all of its data, and after that it will be registered in a list of active objects of the class that created it. After all this setup has been done for an enemy, the class will launch the enemy activity. Each enemy contains two main components, *UnitMovement* and *EnemyStats*.

The first component will send a request to the A* script for the optimal path from the starting point to end point, and it will received a list of Vector2 objects that contains the coordinates that must be followed step by step. Once this list is obtained the object will start moving from its initial position to the next point from the list. For the object movement we have used a function which is found in the *MonoBehaviour* class, *Vector3.MoveTowards*() which has the following signature:

```
public static Vector3 MoveTowards(Vector3
current, Vector3 target, float
maxDistanceDelta);
```

The first parameter represents the initial position from which we start, the second represents the final position that we want to reach, and the third parameter represents a measuring unit that will point the object towards to destination, or for a better understanding it can be viewed as a speed that dictates how fast the object reaches its destination. In order to move an object with this function

we must use it inside the *Update* function (function that is being called at the beginning of each game frame).

The second component, *EnemyStats*, contains details about the object life time. When an enemy is attacked it will lose points from its current health and once this value reaches below zero he will autodestruct, but not before sending a message to the script that is responsible for the enemy management.

Once an enemy started its activity the only interaction he will have will be with the tower components and their projectiles represented through the *TowerLogic* and *BulletLogic* classes. These contain functionality for physical interaction between object with the help of the *OnTriggerEnter* and *OnTriggerExit* functions.

Another activity, that takes place after the game is started, is the tower placement in one of the free spaces of the map (see Figure 4). For this we have used another function from *MonoBehaviour* class, *OnClikc*(). This function gets called each time the user physically presses the click button of the mouse and the cursor is over the object that contains a script that has this function and a collision component, which in our case is a *BoxCollider*. For this we have built a new class called *AvailableSpot* that is attached to each object that represents a piece of our game map. Once the obstacles have been added and the enemy path is known, the collision component will be disabled for the objects that meet these conditions so that the player can no longer interact with them since they are occupied spaces. As we can see from the picture below that the green spaces are available for building towers and by selecting one of the occupied spaces we find the collision component (*BoxCollider*) inactive.

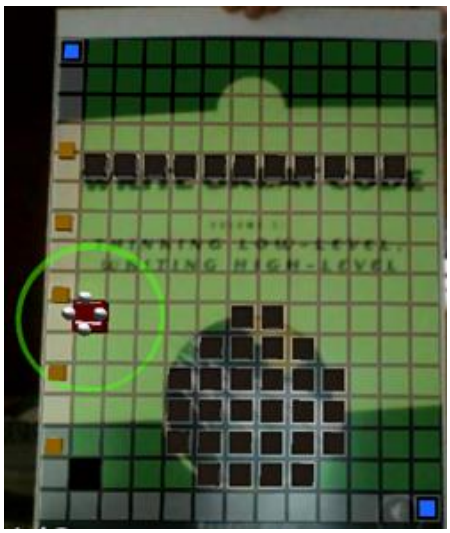

Figure 4. Displaying available spots and a tower

At this step, we can use different possible configurations for the added towers, which can help the students to understand better how the algorithm A* works. There are cases when after adding a tower it is possible not to find a path from the *start* point to the *end* point, there are cases when the added tower doesn't influence the initial solution of the A* algorithm, or there are cases when A* algorithm build another solution. All these cases must be analyzed and discussed with the students.

Description of a level and necessary steps to finish

In this section we will present one of the built levels and all the steps that are necessary in order to finish it.

The top-left and bottom-right corners represent the *start* and *end* point of the level. By pressing the *Start* button the enemies will be instantiated, and their main purpose is to reach the end point. Our goal is to prevent them from doing this. By activating the "Pause & Place a tower" the game will be paused and the user can select one of the available spaces, represented by green, where he can place towers that will destroy the enemies. Once a spot has been selected, a confirmation button will appear bellow, after which the game can be resumed or return to the state where it waits for the user to repeat the action we just mentioned.

Once all enemies have been eliminated, a message will be displayed on the screen and we can return to the main menu scene by pressing the "*Escape*" key (See Figure 5).

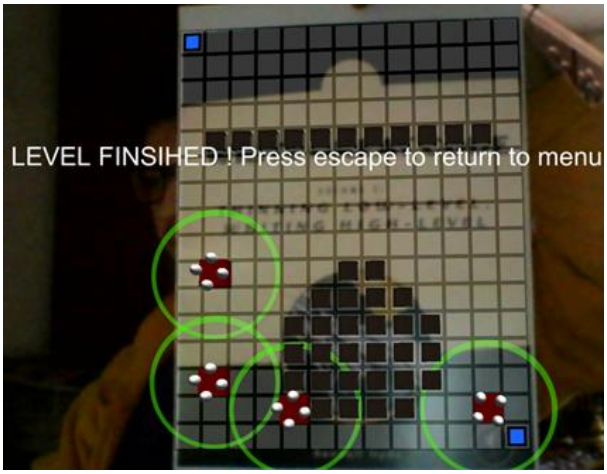

Figure 5. Finishing a level

CONCLUSIONS

As mentioned in introduction, the aim of this project was to create an educative game including elements of Augmented Reality. Matter of our using Unity, the Vuforia SDK's and developed algorithms we managed to

develop an app that offer the possibility of transforming a part of it in real world game space. Thus users can increase interest since the board on real interaction represents a degree of novelty for attracting Player. Also, the user can add obstacles on the map and create different possible configurations, and in this way he will understand better how the A* algorithm works.

This application can be extended to other platforms, but also we can add new features such as connecting multiple participants to the same game session, creating new game mechanics such as upgrading towers, creating multiple terrain types, each with a cost different to be covered, adding new models for representing objects in scene and even the ability to add their own drawings with easy to use editing interface obstacles.

When we first started this project, we did not have in plan a different interaction for the players, but as we continued working we soon realized that this type of interaction would give the game a better feel and experience.

ACKNOWLEDGMENT

This work is partially supported by POC-A1-A1.2.3-G-2015 program, as part of the PrivateSky project (P_40_371/13/01.09.2016).

REFERENCES

1. AR Defence 2: https://itunes.apple.com/en/ app/ar-defender-2/id559729773?mt=8 (Last time accessed on 29 May 2017).
2. Dial, R. B. Algorithm 360: Shortest-path forest with topological ordering. *Communications of the ACM*. 12 (11), (1969), 632–633.
3. Digital Marketing Megatrends 2017: http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/ (Last time accessed on 30 June 2017).
4. Hart, P. E., Nilsson, N. J., Raphael, B. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics SSC4*, 4 (2), (1968), 100–107.
5. PulzAR: https://goo.gl/3FWM8n. (Last time accessed on 29 May 2017).
6. Toyota 86 AR: http://www.toyota86ar.com/. (Last time accessed on 29 May 2017).
7. Unity: https://unity3d.com/ (Last time accessed on 29 May 2017).
8. Vuforia: https://developer.vuforia.com/ (Last time accessed on 29 May 2017).