Movement in a Full and Dynamic Environment using a Small Influence Map

Paulo Roberto Lafeta Ferreira—DigiPen Institute of Technology
paulo.ferreira@digipen.edu, paulo.lafeta@gmail.com

Influence Maps is a technique that gives tactical perspective for AI characters, allowing strategic decisions to be made based on the current game environment. They are layered over a geographical representation of the game map and it can have several layers. These layers are combined via a weighted sum to provide an overall idea of the suitability of each area on the map for the current decision. However, this can be very expensive if we have to calculate the influence value of too many game objects.

This article explains how to make a small influence map that can be used to create a simple intelligent movement in a full and dynamic environment, without losing too much performance. This small influence map is limited to a few cells so we can have a fast algorithm to run every frame, even with a high number of objects around the AI. With this technique, we will have an AI player moving intelligently through the environment considering advantages and disadvantages of all game objects that influence it.

Influence Maps

Influence mapping is a good AI technique for performing tactical assessment. Influence maps have been used most often in strategy games, but are also useful for many other types of games that require an aspect of tactical analysis. An influence map represents a spatial representation from an AI agent's knowledge about the world. With this, the AI agent can develop a tactical perspective of the current game state.

There is no standard algorithm for creating influence maps. The way you construct and employ influence maps depends on specific strategic and tactical needs of your particular game. This article describes an influence map technique that can be used mostly for simple tactical movements on fast action environments. To achieve this, we will have to limit our influence maps, and so it's not appropriate for strategy games, where usually it's necessary more information of the world to get inferences about the characteristics of different locations in the environment.

A Usual and Simple Influence Map

First, we'll present a usual and simple influence map, so the reader can get an idea of how an influence map works and, later, compare and understand the small influence map.

An influence map can be used in almost any type of data structure, like a square grid, hexagonal grid, or a fully 3D environment. For each cell, we add a certain type of

"influence" value we wish to consider. Thinking simple, consider only two types of objects: friendly units and enemy units. So, on each cell, we add a positive value for each friendly unit, and a negative value for each enemy unit. The specific value we add or subtract will be an estimate of the unit effectiveness.

Next, the influence of each cell is spread to nearby cells. There are several ways to propagate each cell's influence. One is to divide the influence by the distance between the object that influences and the cell where we are adding/subtracting its value. When we combine the influence of all objects, we end up with a complete influence map that gives an excellent vision of places for the AI.

An influence map in a real game can be considerably more sophisticated. Rather than simply containing a number, each of the influence map's "cells" is a repository for some amount of data about the game world, like for example combat strength, vulnerable assets, area visibility, body count, resources and passability [Tozour01]. The example implemented on this article won't need to keep more than one type of data, but it will consider some of the factors cited to calculate the influence of each object.

Updating the Influence Map

If it is necessary to calculate large areas of the influence map on a regular basis, it may be a good idea to recomputed the entire influence map on a regular basis, like for example every 1-10 seconds. This is ok in most typical strategy games, because of their pace; a faster refresh rate probably won't make a more effective AI. A second approach is demand-based refreshing, a sort of lazy evaluation technique. This is a more flexible approach, and is more efficient when you need to perform less extensive influence map analysis. We'll use this variant to compute our small influence map.

Small Influence Map

An influence map can be very costly and difficult to deal in games with a great number of dynamic objects and constant fast action. To make a good use of influence map in a simple and efficient way, we can use the small influence map technique. Small influence map is a way of calculating only the cells that matter for the AI characters, also taking in consideration which objects should influence these cells. For this article, our objective is to make intelligent movement for the Player AI, and so our small influence map will calculate only the cells that matter for this purpose. Before explaining more details, let's describe the game chosen to implement the technique. So it will be easier to understand the implementation of the technique. This game has the characteristics described where a small influence map fits well: a full and dynamic environment.

The Game Used

The game used to implement movement using a small influence map is called *Chameleon*. *Chameleon* is a multiplayer game for 2-4 players. Each player controls a creature-type character as shown in the Figure 1.

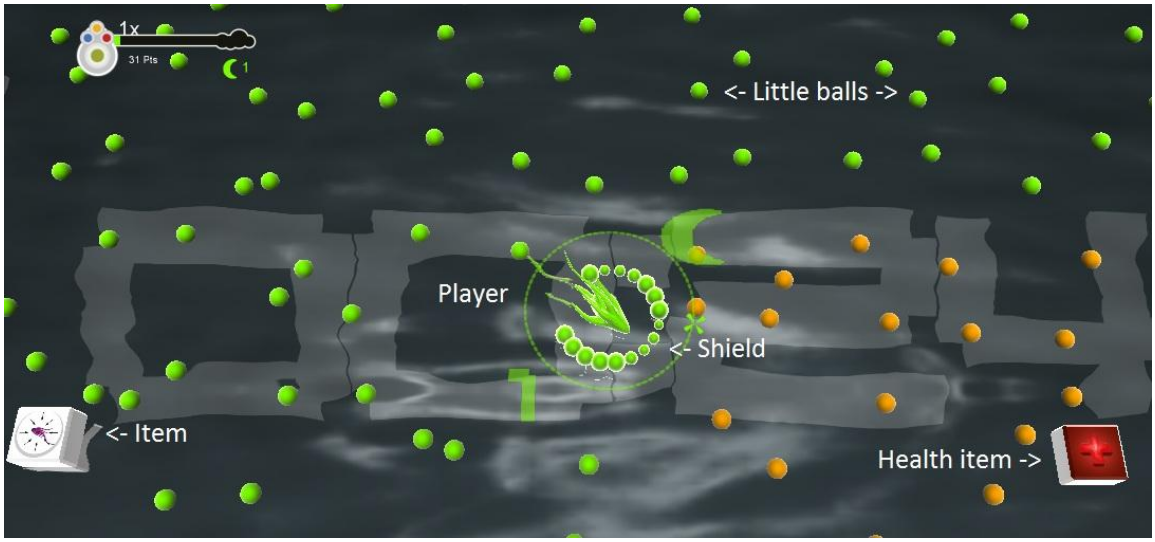

Figure 1. Screenshot of *Chameleon* with texts indicating some object types of the game.

The core aspect of this game is the colors. Almost all objects of its world can have 4 colors: green, yellow, red or blue. The player can change its color during the game and collect the balls of same color. Each ball he collects is added to his shield. Balls of different colors do damage on his shield or in the player itself, and so it should be avoided. Figure 2 below illustrates a yellow ball doing damage on a player's shield.

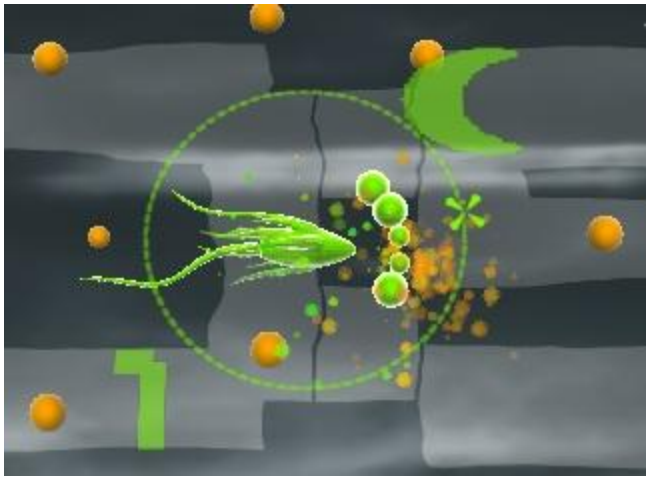

Figure 2. Yellow ball doing damage on player's green shield.

The player can shoot his balls of his shield on other players. In other words, his shield is also his ammo. The player can catch up one item to use it later. There are several different items available, which one with a different effect.

In some game modes there are some monsters with simple steering behaviors like: wandering and pursuit. They pursue and try to kill nearby players with different color, and ignore the ones with same color. Monsters are shown in Figure 3.

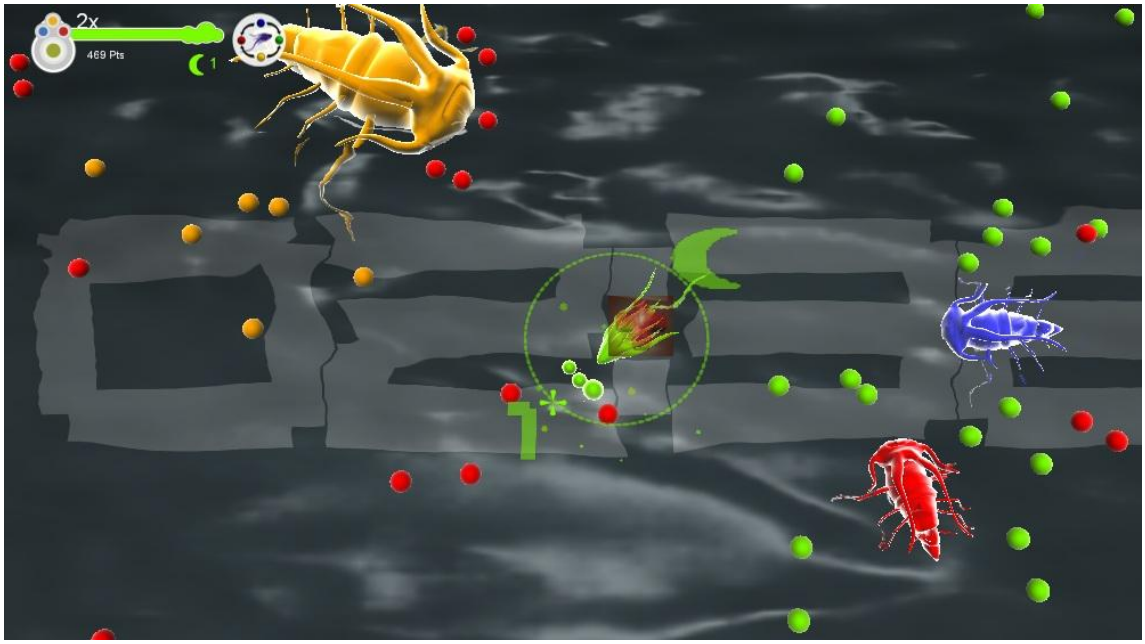

Figure 3. Simple monsters wander around or pursue players.

As mentioned before, the game can have 4 players simultaneously. To create an AI for the players considering all design aspects of the game, we decided to use influence map, but in a limited way, as the game has a lot of dynamic objects and action occurring constantly. This technique, as we will see, fits well in constructing this AI in a simple way, but with a clever and tactician aspect. Figure 4 and Figure 5 below show a game environment for 4 and 3 players respectively.

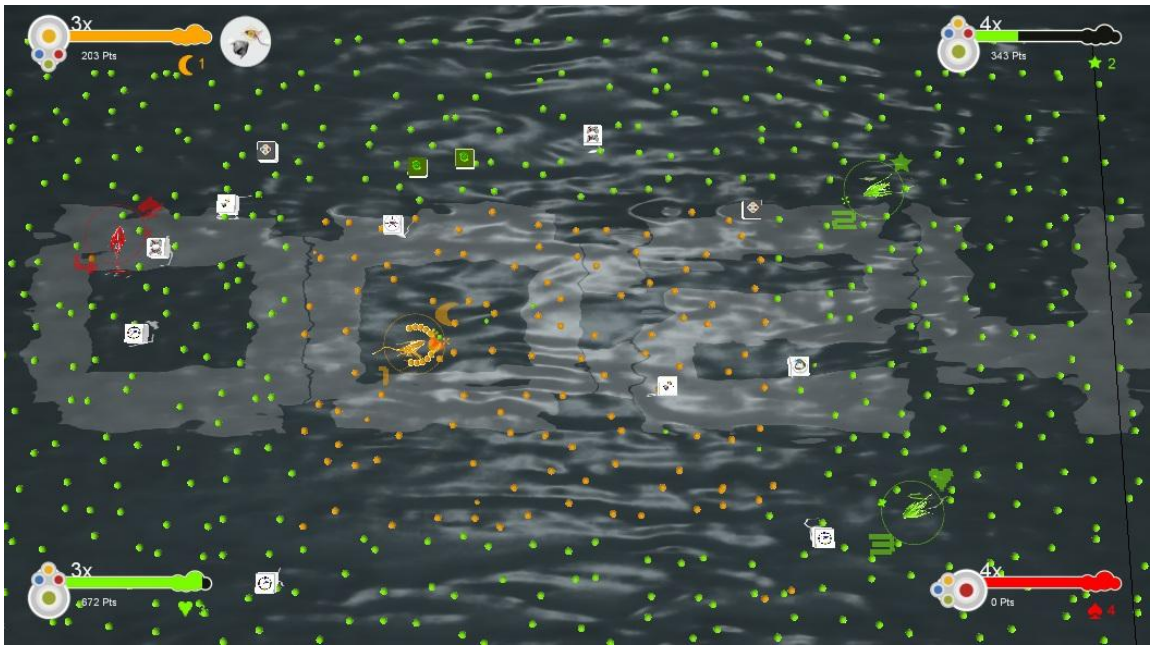

Figure 4. Game screen with 4 players.

There are different game modes like death match, capture the flag, survival, monster slaughter, etc, where the AI can be easily adapted for each one using the influence map technique. Later we will exemplify how this can be done.

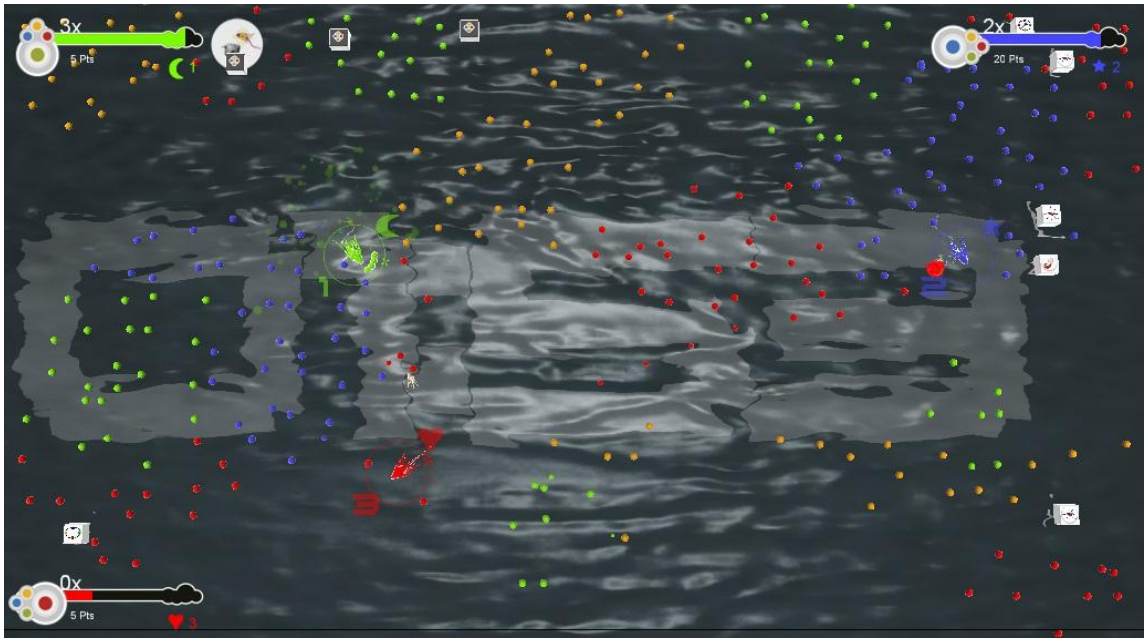

Figure 5: Death match with 3 players.

Technique implementation

In this section we will discuss with details how to create and calculate the small influence map and how to use it for Player AI movement.

Data Structure

We will use a 2D grid for the influence map, as it is simple and easy to understand and also good enough for our open environment. So, first we create a set of square cells superimposed on our game world. The size of the influence map cells is somewhat arbitrary. It's a trade-off between accuracy and efficiency. To define the cell size, we used our player model size to define our grid cell, so it could fit on it. In the end, this cells size created a good balance between accuracy and efficiency. Each cell of the grid maintains only a float number (complex games usually store more data).

Each Player AI has its own influence map (2D array of float) where it's stored calculation results of the object influences and then it's used to decide where to move the Player. Besides the influence map, we also need access to a list of all active objects in the game world that influence to do these calculations.

Object Weights

To calculate the influence value on a cell of the limited influence map, we need to check how much each object influences it. Basically, an influence of an object in a cell is its Weight divided by the Distance between it and the cell. Let's see how to define the Weight returned by an object. Each object returns its Weight based on its and player AI's features. Below there is a description of how much Weight is returned by each object type that influences on the *Chameleon* game.

- **Item that is used later:**
    - Weight returned is a positive constant if player doesn't have any item, or zero if player already has an item (player can carry only one item, so it's not interesting to pursuit an item if he already has one).
- **Instant item** (like for example health pack. Player don't need to press a button to use it):
    - Positive constant.
- **Monsters:**
    - Negative constant.
- **Shots:**
    - Negative constant if it has different color or zero if same color (does not damage, so it doesn't care).
- **Little Balls:**
    - If same color:
        - Weight varies according to the number of balls that player already has in its shield. Great value if player has few balls (so it's good

to get more!), low value if has a high number of balls (not much interesting) or zero if shield is already full.
    - If different color:
        - If player has high number of balls in its shield, then it doesn't need to worry too much about dangerous balls and so it return a low negative value as weight. But as the number of balls is smaller, higher will be the negative number returned as weight.
- **Players**: Other players also influence on the player AI movement.
    - If both players have **same colors**, then it's not much interesting (they can't do any damage to each other) and so returns a low positive weight.
    - If they have **different colors**, then we need to calculate a weight proportional to the difference of variables that matters between the two players. In our case: Health Points and number of balls in shield. So, something similar to this formula:
        - Weight: (playerAI.HP – player.HP)*X + (playerAI.shieldCount – player.shieldCount)*Y
    - You can adjust X and Y to value more shield or HP.
    - With this formula we return a positive or negative value, depends on each player current status. If it returns a great negative value, then the player AI should avoid this player, as it's probably much stronger than him. Or if the player AI is with a much more better HP and shield, then it will return a great positive weight and so player AI will likely want to pursuit it and try to kill it.
- **Diamond** (for Get the Diamond mode):
    - Return a very high positive value. As the objective for the Capture the Diamond mode is to get the Diamond. Then, all Players AI will make this pursuit high priority.
- **Territory** (for Get the Diamond mode):
    - After getting the Diamond, the player must bring it back to his territory. So, if player has the diamond, then Territory returns a very high positive value. Else, it will return zero.

These weights are defined during the game in real-time as they are required. So, the Player AI constructs its influence map considering various aspects of the game with the weights acquired. As consequence, it will move strategically accordingly to what is happening in the game.

Calculating the Small Influence Map

There is no standard algorithm for creating influence maps, nor any single way to apply the technique. This article describes a method that seems to be the more appropriate for the great number of objects and low number of tiles that need to be calculated.

One of the main tricks of this small influence map is about which cells we need to calculate. For this, we need to analyze our goals in the game and what information

the Player AI really needs about its world. In the *Chameleon* game, the influence map is only for its movement. Not a simple movement, but a movement that implicitly involves tactical analysis through the weights of the objects. In a Strategy game, it is interesting to have a lot of cells analyzed, so the AI can see, for example, where is the "frontier" between his troop and the enemy's troops. But on some games, this isn't necessary, and the developer may only wants the Player AI to move intelligently with a subtle tactical sense of its environment.

The game action is fast and so the environment changes a lot. The Player AI only wants to know where to move and we want to save processing performance. For these reasons, we calculate the value of only the 8 tiles around the Player. The best tile around will be the chosen direction of the Player AI's movement. It is simple as that.

Each player AI has to calculate its own influence map. The pseudo-algorithm below calculates the limited influence map of a Player AI:

```
foreach cell around the Player AI // 8 cells
{
	float totalValue = 0;

	foreach object in the world that influences
	{
		// Note below that we pass the playerAI as
		// argument, so the object can evaluate which
		// weight to return
		float objValue = object.getWeight(thisPlayerAI);

		// The value is inversely proportional to the
		// distance between the object and the playerAI.
		// More far, less influence the obj will make
		totalValue += objValue / distance(object.Pos, thisPlayerAI.Pos);
	}

	cell.influenceValue = totalValue;
}
```

Now with the 8 cells around each Player AI calculated, we know which one is best to move. But when should we update the small influence map of each player? It really depends on how much performance and accuracy you want to have. But mostly, you should update the Player AI's small influence map if: the player changes its tile; or if an amount of time has passed (environment changes even if player is still); or if the player changes one of its attributes that influences objects weights (on Chameleon for example, if player changes his color).

Figure 6 shows an image illustrating the eight tiles around the Player AI calculated. Then, Player AI goes in the direction of the greatest value.

Figure 6. Two items nearby between green balls seems more interesting for the AI than the other item on the top-left

Full environment: too many objects to calculate

Some games have a very high number of objects in the screen that influences the Player AI. In *Chameleon*, there are a lot of little balls in the screen. They can be seen as a kind of terrain, but they aren't static, so we cannot pre-compute their influence value. Calculate all of them constantly can be a big hit in the performance of the game. Also, the ones that are far may not be that important for the player AI's decision. Figure 7 shows this situation on *Chameleon*, where there are a lot of little balls in the environment.

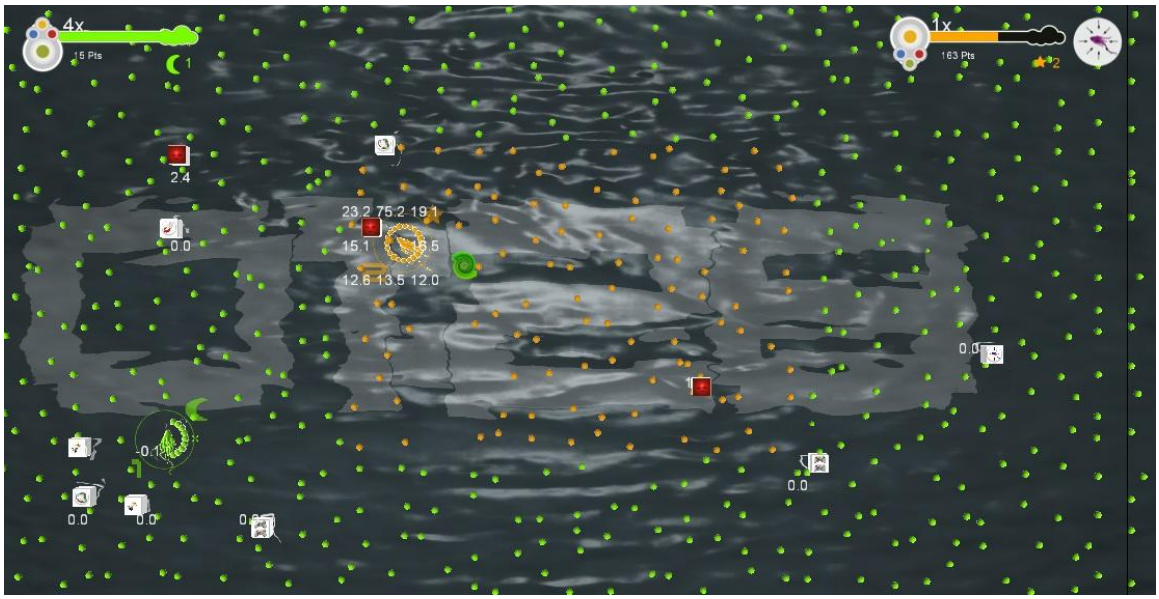

Figure 7. Too many objects to calculate the influence can hit the speed performance of the AI.

The solution made to this was simply to considerate only the little balls nearby the Player AI. To do this, it's necessary to constantly update the positions of the little balls in a separate grid. With this grid, we then apply a sort of "Breadth-first search" algorithm with limited depth (in our implementation we look 4 levels of depth). With a greater depth, the AI gains a better vision of its environment, but the game loses speed performance.

In the end, to calculate the value of each tile around the player, the algorithm gets the influence value (weight / distance) of every object of the game and little balls nearby the Player AI.

Different Game Modes

Different game modes can be easily adjusted on the Player AI. For example, in a "Get the Diamond" mode (like "Capture the Flag") players must get a Diamond and bring it back to its territory. To make the player AI prioritizes the Diamond on its movement, we just need to give the Diamond a very high positive weight. With this, player will pursue its main objective, the Diamond, but also considering other aspects of the environment, with less interest. For example, a player AI is going into the direction of the diamond, but on his way there's a good item, so the player will probably go get it and then go back into its pursue. After getting the diamond, we just need to adjust the territory weight to a great value, so the player AI will try to bring its diamond there. Figure 8 shows a "Get the Diamond" screen from Chameleon.

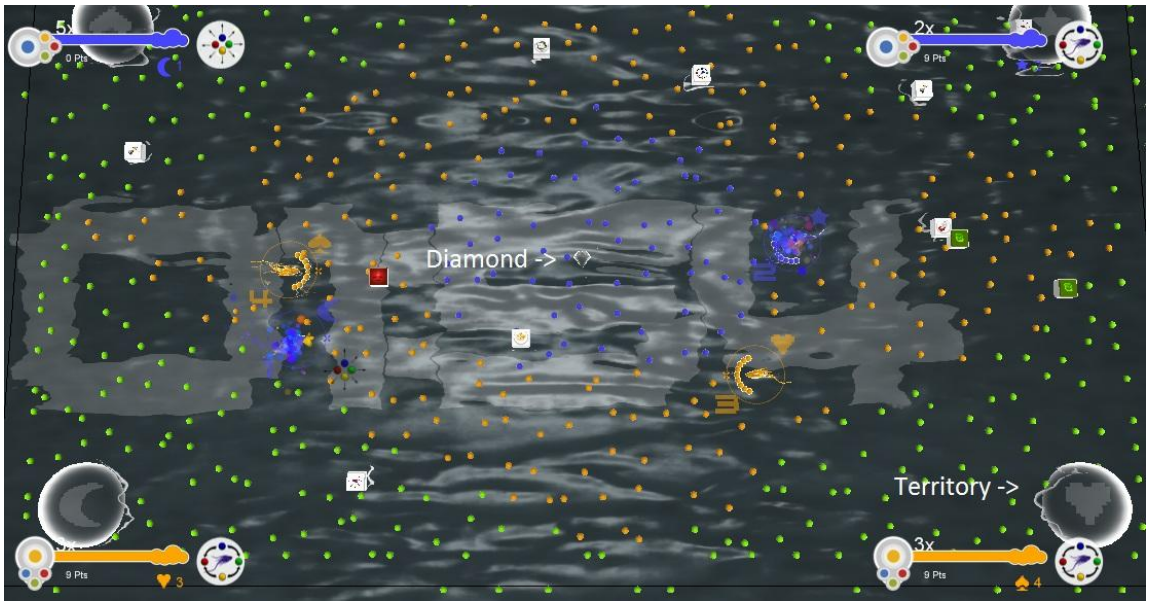

Figure 8. “Get the Diamond” mode on *Chameleon*.

Possible Problems

The algorithm may find some situations that make the Player AI stuck, like in a “flip-flopping” situation. For example, the Player AI finds out that the best tile is the top, and so it goes up. As it gets on the up tile, its influence map shows to go down and so it keeps on a vicious cycle. On our Chameleon implementation, this rarely happened in Game Modes where the weights were adjusted. Also, was noticed that this was more likely to happen with 4 players at the same time than with 2-3 players. A possible solution to this is to store the last cell positions that the Player went and if it founds out to be repetitive, then make a direction decision and stick with it until enough time has passed.

Another matter that we need to be aware is when all the tiles around the player AI get the same value. On this situation, we chose to make the player AI to not move. So it gets still, as if it’s thinking or waiting something to happen. Usually this occurs when environment is “static” without anything “interesting” for the Player AI.

Conclusion

The small influence map showed a good performance, both in processing speed as in results about Player AI movement. It only calculates the cells of the influence map that really matters for the Player AI in a short period of time and gives it a movement with subtle tactical sense. It’s not a complex algorithm, but still, gives the human player an impression that the Player AI knows what it is doing. Its movement won’t always be the best one, as the weights are an estimative made by the programmer, but mostly will do what seems to be best in that moment. Just like a human player does, estimates of what he thinks its best, which can goes right or wrong. This leads to a more immersive

experience for the human player, as he feels the player AI as a ‘live’ player making reasonable decisions in its movement during the game.

References

[Tozour01] Tozour, Paul, et al, “Influence Mapping” Game Programming Gems 2, Vol 2, Charles River Media: pp. 287-297