Industrial Data
ISSN: 1560-9146
iifi@unmsm.edu.pe
Universidad Nacional Mayor de San Marcos
Perú

Santos López, Félix Melchor; Santos de la Cruz, Eulogio Guillermo
Algoritmo de PRIM para la implementación de laberintos aleatorios en videojuegos
Industrial Data, vol. 15, núm. 2, junio-diciembre, 2012, pp. 80-89
Universidad Nacional Mayor de San Marcos
Lima, Perú

Revista de la Facultad de Ingeniería Industrial
15(2): 80-89 (2012) UNMSM
ISSN: 1560-9146 (Impreso) / ISSN: 1810-9993 (Electrónico)

Algoritmo de PRIM para la implementación de laberintos aleatorios en videojuegos

Recibido: 14/09/12 Aceptado: 04/03/13

Félix Melchor Santos López1
Eulogio Guillermo Santos de la Cruz2

ABSTRACT

The Prim's algorithm, extracted from the graph theory, is easily adaptable for generating random mazes in the game development process. This study provides the theoretical framework of the algorithm, its adaptation to generate two-dimensional arrays of values, coding in Java and use of libraries provided by this programming language. It restricts the creation of the arrays to a minimum size of 11x11 to ensure that generates coherent sizes mazes and its application is for generating orthogonal called labyrinths (2D view). Finally, the measurement is made of the performance of the proposed encoding and concludes that in all tests, the average response time is less than a tenth of a second to generate maps of labyrinths.

Keywords: Prim, Mazes, Game, Graph.

PRIM ALGORITHM FOR THE IMPLEMENTATION OF RANDOM MAZES IN VIDEOGAMES

RESUMEN

El algoritmo de Prim, extraído de la teoría de grafos, es fácilmente adaptable para la generación de laberintos aleatorios en el proceso de desarrollo de videojuegos. Este estudio proporciona el marco teórico del algoritmo, su adaptación para la generación de valores en matrices bidimensionales, la codificación en Java y el uso de librerías proporcionados por este lenguaje de programación. Se restringe la creación de las matrices a un tamaño mínimo de 11x11 para garantizar que se genere laberintos de tamaños coherentes y su aplicación es para la generación de los denominados laberintos ortogonales (vista en 2D). Finalmente, se realiza la medición del rendimiento de la codificación propuesta y se concluye que en todas las pruebas realizadas, el tiempo promedio de respuesta es menor a una décima de segundo para generar los mapas de laberintos.

Palabras clave: Prim, Laberintos, Videojuegos, Grafos.

INTRODUCTION

The video game industry, today, is one of the fastest growing globally and one that employs a variety of professionals such as programmers, designers, sound engineers, test analysts, experts in physics, mathematical computing professionals, screenwriters, artists, etc. One of the most important pillars in the development of video games is to create the environment in which it takes place; this is to say the maps by which our character, machine, ship or other is going to go traveling.

For creating such maps, there are different procedures, the most popular being fixed maps once are designed in size and shape. However, more elaborate video game maps with roads and different in each instance of the game, algorithms for randomly generating these maps must be applied. This paper presents the algorithm of Prim extracted from a graph theory. "A graph is a non-empty set of objects called nodes connected by edges" [2].

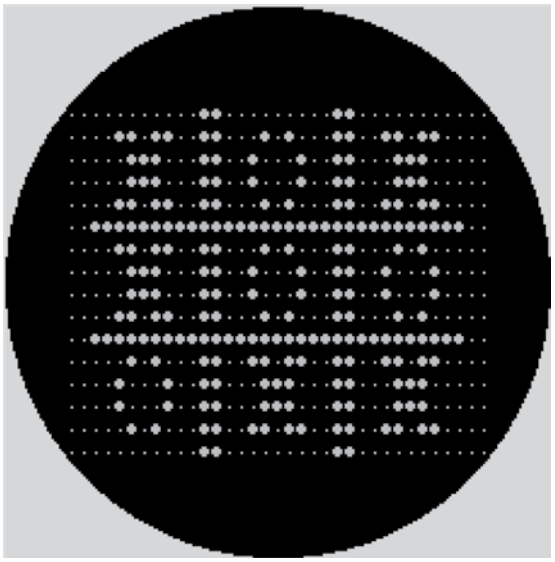
This algorithm can generate a minimum spanning tree in this article is adapted for generating random labyrinths dimensional matrices. In addition, the study covers the measurement of performance of this algorithm to generate a number of them at once. For example: 5, 10, 20, 30, 40 and 50 mazes in each iteration, obtaining results in nanoseconds which are analyzed and concluding that everyone is on average less than 0.01 seconds. Therefore, an optimum time is considered appropriate and, because it is less than one tenth of a second.

I. THE FIRST VIDEO GAME OF HISTORY

Summarizing what was designated by Ramker R. [9] In 1952 Alexander Shafto Douglas presented at the University of Cambridge the doctoral thesis in mathematics showing human-computer interaction by the popular game of noughts and crosses called OXO. It was implemented in the machine EDSAC (Electronic Delay Storage Automatic Calculator) and used the buttons on a phone to interact with users. In figures 1 and 2 can be seen the screens of the first video game in history.

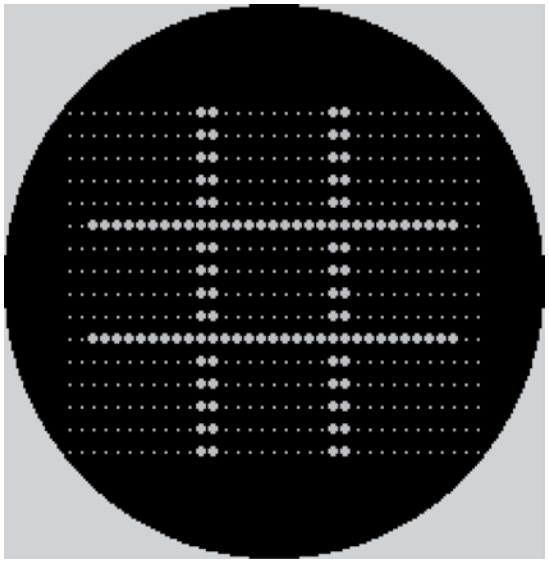
1 Informatics Engineer, PUCP. Postgraduate Diploma in Audit and Security of Information Technology, UNMSM. J2EE Analyst at SUNAT.
E-mail: fsantos@pucp.edu.pe
2 Industrial Engineer, UNMSM. Professor at Faculty of Industrial Engineering, UNMSM.
E-mail: esantosd@unmsm.edu.pe

Figure 1. OXO Start



Source: Taken from Ramker R. [9]

Figure 2. End of the OXO start



Source: Taken from Ramker R. [9]

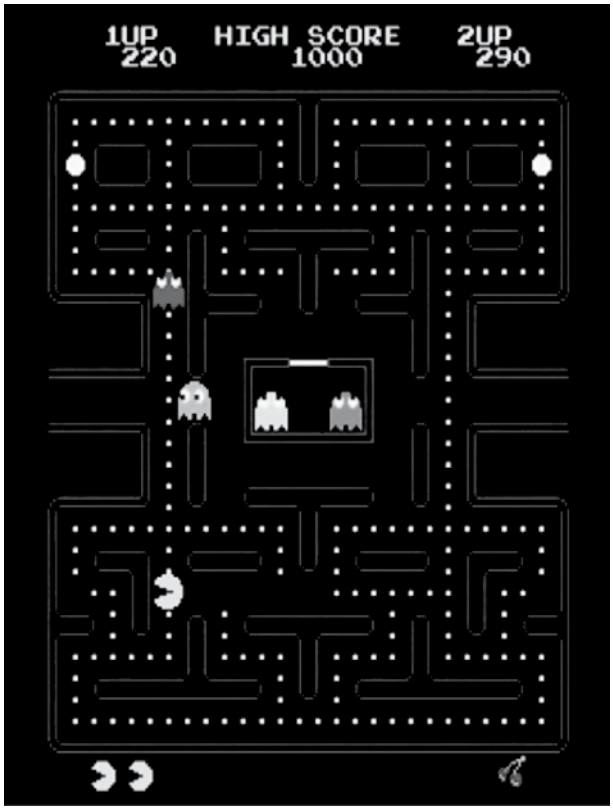
In this way began a new charge of industry information technology innovation globally, both in algorithm development needs, graphical interfaces, application of mathematics and physics, as well as in the development of increasingly powerful graphics cards, microprocessors, RAM and communication buses.

II. ORTHOGONAL MAPS AND ISOMETRIC

Within the video games world a very common pattern is the use of maps for their development. This is to say maps where the main character is moving and avoiding or facing obstacles that the game provides.

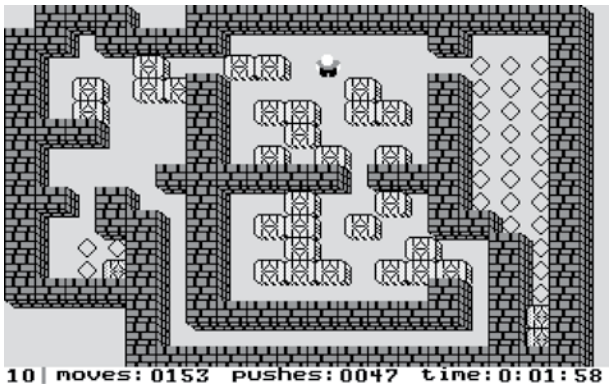
There are two types of maps: orthogonal and isometric. The maps are those classic orthogonal in 2D view and the player appreciates it in a direct view. Examples of orthogonal maps can be seen in the classic games Sokoban ® and Pac man ® illustrated in figures 3 and 4 respectively.

Figure 3. Map of Sokoban ®



Source: Taken from Eugenia M. [5]

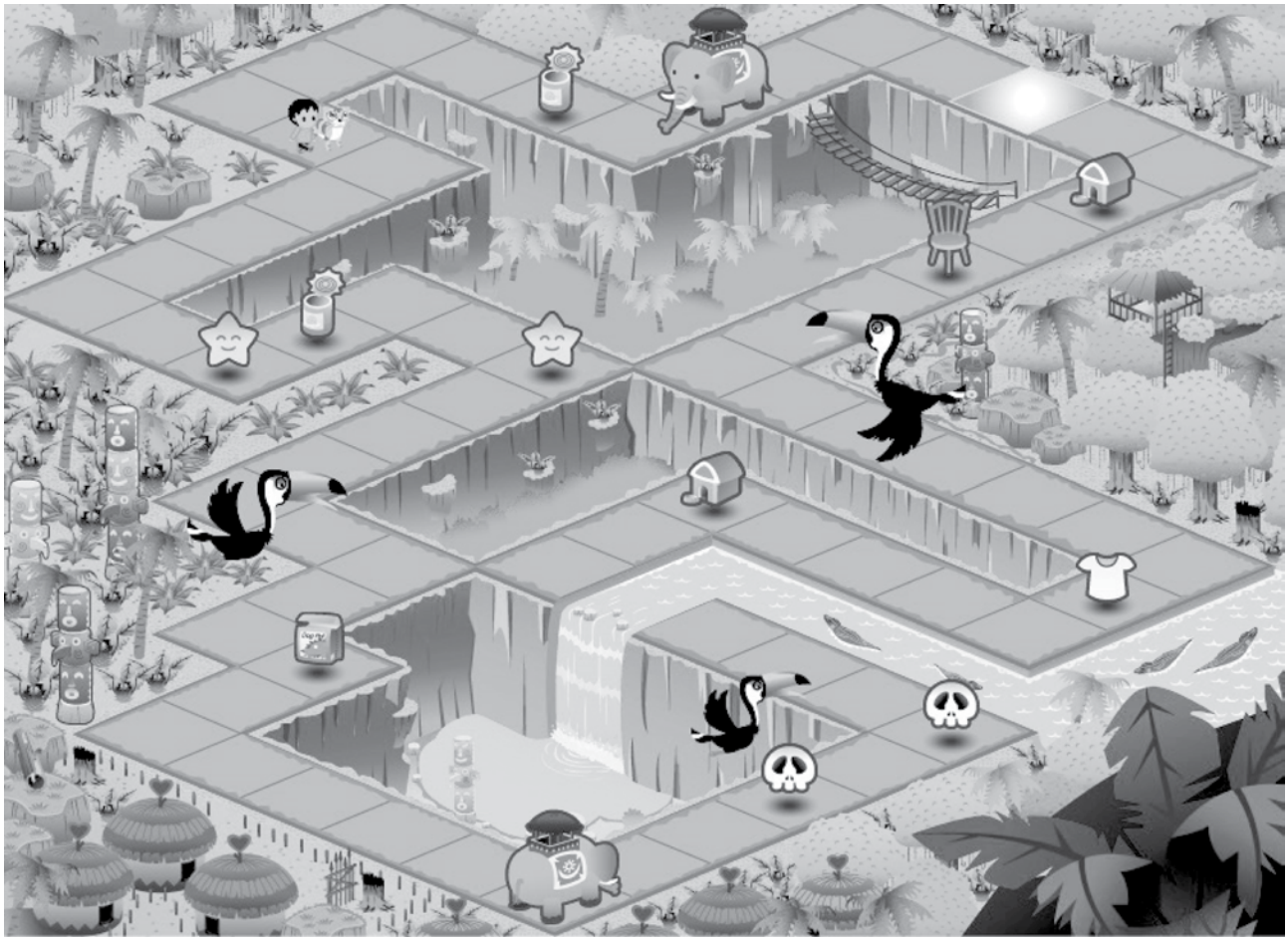
Figure 4. Map of Pac man ®



Source: Taken from Epifania A. [4]

Furthermore, the maps are those that allow to simulate an isometric 3D scene. These maps have the particularity of placing images on an inclination angle of 45 degrees. Figure 5 shows an isometric game map.

Figure 5. Isometric map



Source: Taken from Jongart D. [7]

III. IMPLEMENTATION OF THE MAPS

In game development, the maps are the central core of the development of these applications for entertainment, because it is the stage on which is based the entire project. There are a variety of ways and algorithms for generating the labyrinths. The most basic are the manually generated maps, namely algorithms are applied but not simply as templates in the case of figures 3 and 4, where the labyrinths are always the same for the game or a fixed there with labyrinths sizes and shapes are pre-established.

The implementation is based on mazes of dimensional matrices with allocation of certain securities by developers. For example, figure 6 shows a matrix of a labyrinth generated. The value 8 represents the start or entry to the labyrinth, the value 5 represents the end or output values 1 represent walls or blocks in which the character cannot be positioned and 0 values represent the possible displacements of the video game character.

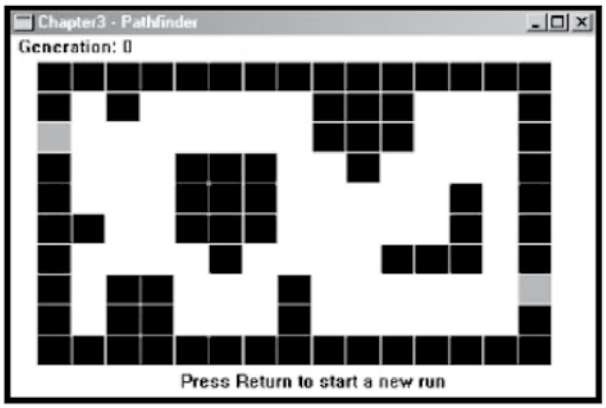
The matrix was generated with a genetic algorithm that provides the artificial intelligence field in chapters and evolutionary computation theory. Palma J [8] points out in his book that genetic algorithms are expressed by binary strings of bits, which in conjunction with a structure (chromosome) give rise to the necessary components for the use of these algorithms. It requires an initial population is called "parents" and then a "crossover" and "mutation" are generated the following generations and it is iterated for a certain number of times, resulting in a matrix with binary values that are used for implementing maps.

Figure 6. Matrix for generating the labyrinth

```
{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1,
8, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1,
1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1,
1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1,
1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1,
1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1,
1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 5,
1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
```

Source: Taken from Buckland M. [1]

Figure 7. Labyrinth generated



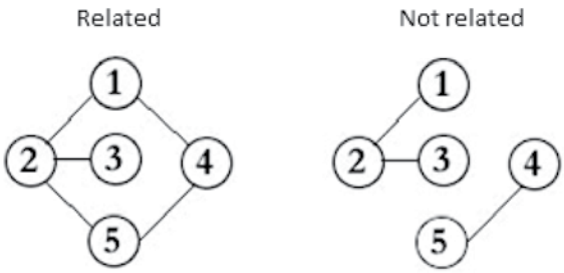
Source: Taken from Buckland M. [1]

Figure 7 shows the implementation of the labyrinth graph based on the values of the matrix in figure 6.

IV. MATHEMATICAL DEFINITION PRIM'S ALGORITHM

Prim's algorithm is a special case of the minimum spanning tree generic providing graph theory. It defines a graph "as a set of objects called nodes or vertices and the vertices of pairs of bonding lines for calls edges, which can be oriented or not" [6]. Also, it must be a related type; this is to say all pairs of vertices must be connected by an edge. Next, in figure 8, shows an illustration of related and not related graphs.

Figure 8. Types of graphs



Source: Taken from Chartrand G. [2]

Initially, the algorithm begins with the following equation:

$$A = \{(v, v.t) : v \in V - \{r\} - Q\}$$

Where:

- A : Represents the graph equation
- v : Represent each vertex
- vt : The minimum weight vertex connects to the other vertex.
- R : Represents the path of minimal spanning tree to grow.
- V : Represents all nodes in the graph.
- Q : Represents the lowest priority queue.

When the algorithm terminates the queue of the lowest priority Q is null and it´s expressed by the following equation:

$$A = \{(v, v.t) : v \in V - \{r\}\}$$

The central idea of the equation is to make iterations, where the node is always looking for that edge results in the lower value. These iterations are performed until the lowest priority Q queues zero.

This is followed by an example to explain the application of Prim algorithm on a graph.

In figure 9 (a) can be seen the nodes from the letter "a" to "i" and their values in each of the edges bound. From this graph, it is necessary to select a node at random, in this case the node "a". After the connection is evaluated next vertices and the values assigned to the edges, selecting the lowest value. In this case it selects the edge of value 4 corresponding to node "b", as shown in graph (b).

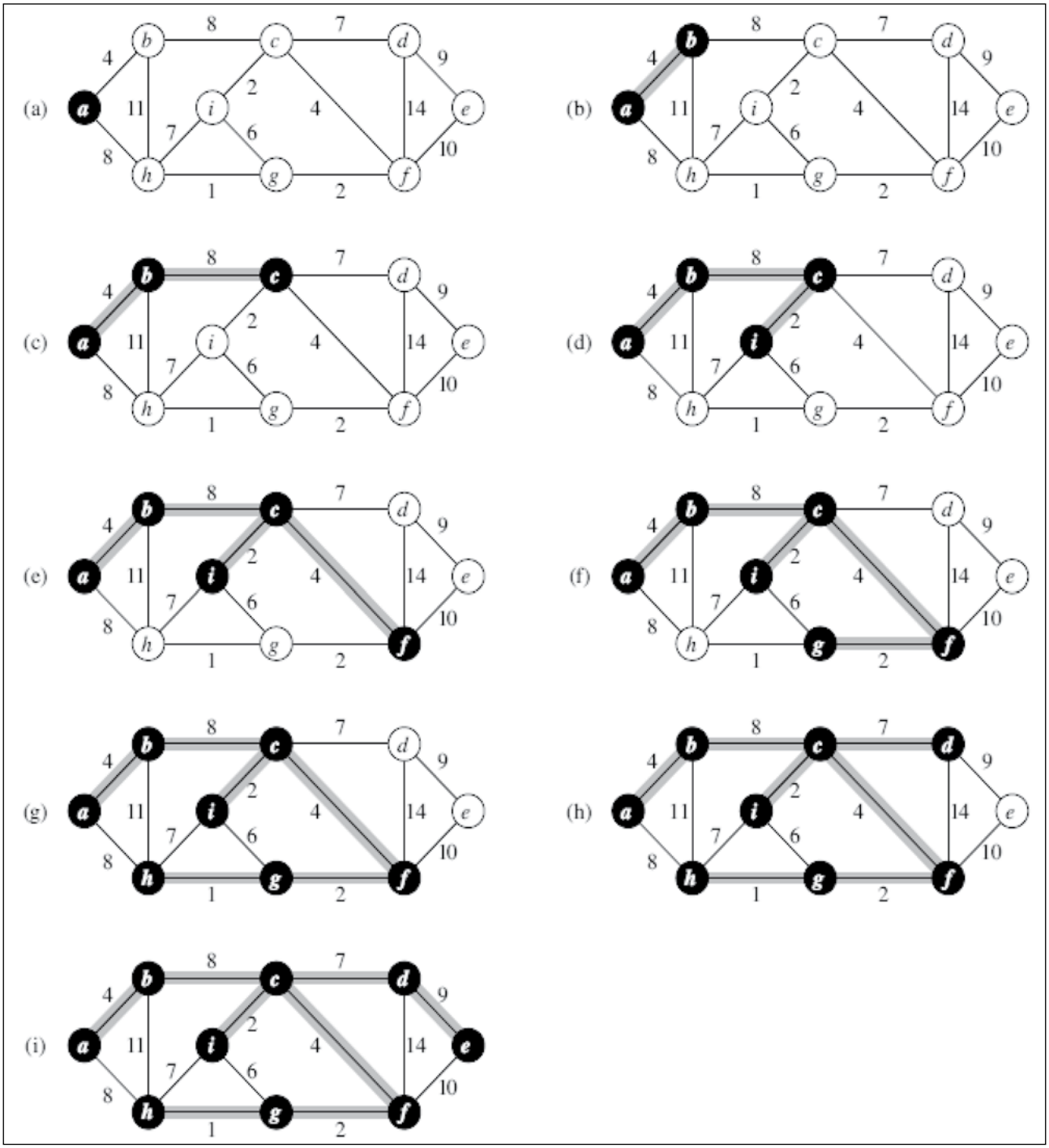
Next, the new nodes to be evaluated are "a" and "b" and proceeds to re-select the next node whose edge is the next lower value. It will be appreciated that the following edges: "bc" with a value of 8, "ah"

with a value of 8 and "bh" with a value of 11. Thus, it appears that the less valuable are the edges "bc" and "ah" with value 8, then select one of them randomly, for example "bc" as is shown in the graph (c).

The above steps are repeated until all nodes are connected. It connects the edges "ci", "cf", "fg", "gh", "cd" and "of" as seen in the graphs (d), (e), (f), (g), (h) and (i) respectively of figure 9.

Finally, the graph (i) shows the final result of the implementation of Prim algorithm, making all nodes interconnected through by at least one edge.

Figure 9. Prim graph algorithm



Source: Taken from Cormen T. [3]

V. ADAPTATION OF THE PRIM ALGORITHM FOR GENERATING MAZES

As noted in previous sections, two-dimensional arrays are required to generate mazes. It is therefore necessary to specify the possible values as developers that have each one of the elements of the matrix.

- B à Block or wall where the character cannot be positioned.
- à Blank box where the character can be positioned and / or move.
- O à Box candidate to be added to the low priority Q queue.
- F à Box border considered the current generated box in blank.

The following restrictions apply to the maze:

- The edges of the labyrinth will be the type of B cells, i.e. block or wall where the character does not move.
- The labyrinth has a width of a box for the displacement of the character.
- The minimum size 11 x 11.
- Labyrinth dimensions are always odd numbers. For example, 27x33.
- The algorithm allows the generation of orthogonal labyrinths.

.

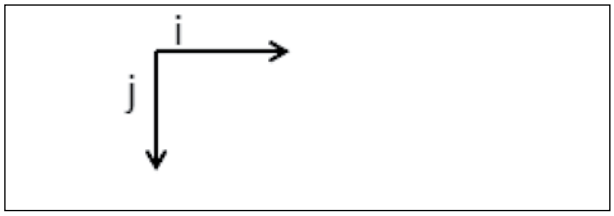
Being the labyrinth one dimensional array indicates the orientation of the ordinate abscissa i j as is shown in figure 10.

Figure 10. Orientation

```
B B B B B B B B B B B B B
B O B F B   B F B O B O B
B B B B B B B B B B B B B
B O B O B F B O B O B O B
B B B B B B B B B B B B B
B O B O B O B O B O B O B
```

Source: Own elaboration

Figure 11. Iteration start



Source: Own elaboration

Here are the steps to generate random mazes with Prim's algorithm:

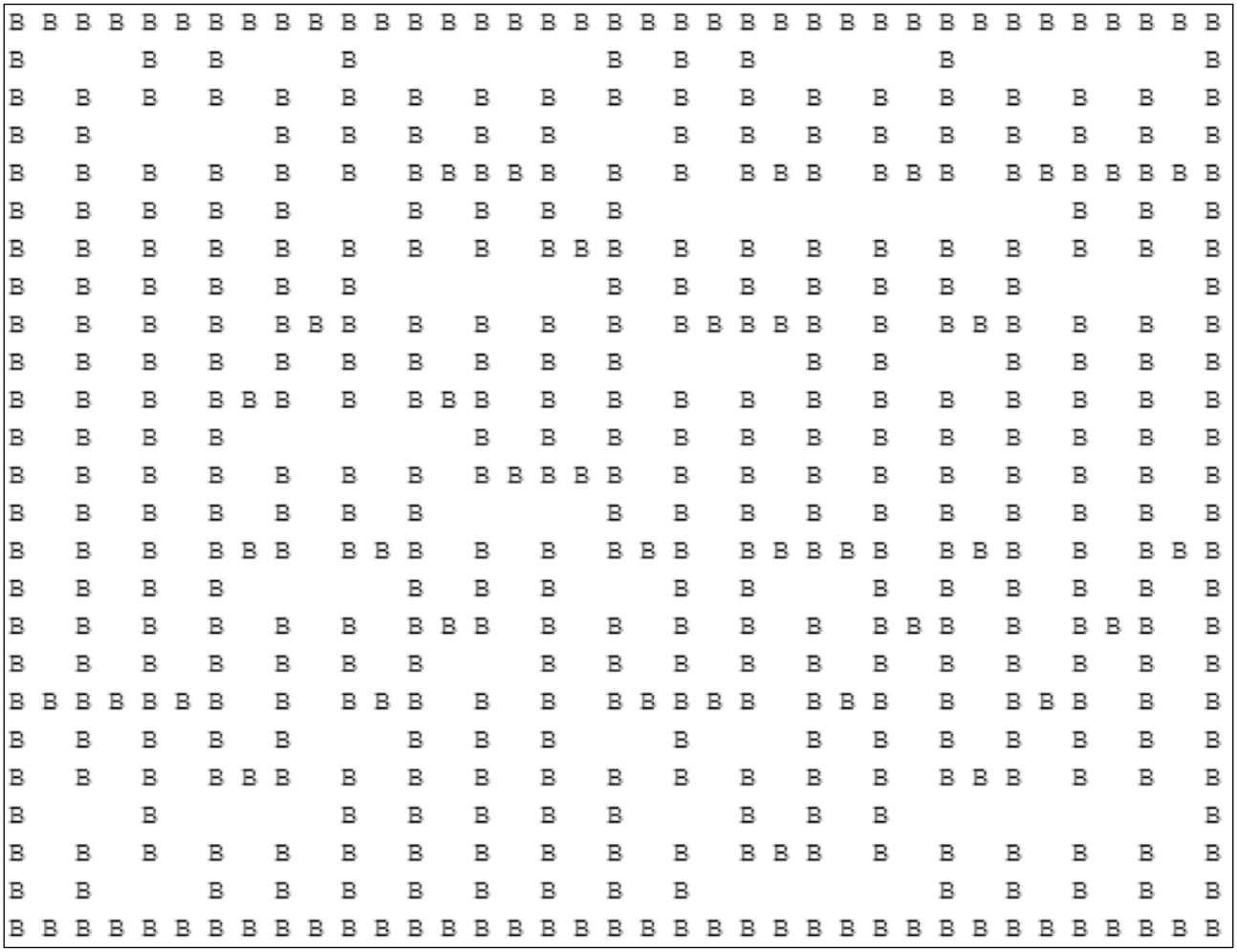
1. Random numbers whose values M and N are greater than or equal to 11 are obtained.
2. It creates a two-dimensional array of M x N with all its elements with the value of B (or block wall).
3. With the matrix created, it proceeds to load the O values, this is to say boxes candidates for the path of the labyrinth. Applies the constraint that the edges of the map cannot contain the value of O, so the positions [i, 0], [0, j], [i, N] and [M, j] do not take the value of O. Furthermore, it is restricted to only the positions i and j values for the pair take the value O.
4. With the values O loaded, this is to say boxes candidates to be added to the low priority Q queue, we proceed to select a random target and place. In figure 11 shows that the position [6, 2] is selected as a target.
5. Next, with the box blank will be selected to add to the neighbors next to the lowest priority queue, this is to say to mark them as a border in the matrix. In figure 11 can be seen the positions [4.2], [8.2] and [6.4] marked with a F.
6. Borders are added to the low priority queue. Q = {[4, 2]; [8.2]; [6, 4]}.
7. In the low priority Q queue we proceed to select a random value. For example [4, 2] and mark target position.
8. Then we see that the values of the positions [4.2] and [6.2] are marked with white and proceeds to join them, this is to say to place the target value in the [5,2].
9. F borders are added to the [4,2] and add those that are new to the lowest priority Q queue.
10. It is removed the position [4.2] of the low priority Q queue.
11. Steps 7), 8), 9) and 10) are repeated until the lowest priority queue is empty.

By applying the steps of the algorithm proceed to obtain a map of random size and shape as is illustrated in figure 12.

Source: Own elaboration

IMPLEMENTATION IN JAVA

Figure 12. Labyrinth generated



VI. IMPLEMENTATION IN JAVA

As discussed in the previous section, we proceed to implement the algorithm in the Java programming language in its Standard Edition.

In figure 13 depicts the Source code of this algorithm. Then we proceed to explain some important lines of code.

- The name of the public class is Maze as it is displayed on line 9.
- These are created to use as constant values in the matrix elements (O, , F, B) as shown on lines 11, 12, 13 and 14.
- In lines 18 and 19 are created by the function Math.random() M and N dimensions of the labyrinth.
- In line 25 the function iniciarLaberinto initializes the array of size MxN generated, establishing the walls (B) and the boxes candidates (O) to the lowest priority Q queue. Moreover, this function is responsible for validating that the maze is at least 11x11.
- In lines 27 through 33 shows the implementation of the first box blank to select at random.
- In line 40 the function evaluarVecinos marks borders (F) in the matrix maze [ ] [ ], further adding them to the lowest priority queue instantiated on line 36.
- In lines 43 through 58 we proceed to enter a “While” loop until the low priority list Q contains elements.
- Within the loop, proceed to obtain a random element of lowest priority list Q as is shown in line 48. After 50 to 53 lines is necessary to assess whether there are close neighbors and existence, that element is interconnected with randomly chosen.
- Finally, in line 55 evaluarVecinos method is again invoked to update the priority queue with borders F minimum found for the item chosen randomly. Then proceed to remove that item from the list Q as it is displayed on line 57.

Figure 13. Code Java

```
9   public class Maze {
10
11      public static final char OUT = 'O';
12      public static final char FRONTIER = 'F';
13      public static final char IN = ' ';
14      public static final char BLOCK = 'B';
15
16      public static void main(String[] args) {
17
18          int M = (int) (Math.random()*20 + 10); if(M%2==0) {M = M+1;}
19          int N = (int) (Math.random()*20 + 10); if(N%2==0) {N = N+1;}
20
21          char maze[][]=new char[2*M][2*N];
22
23          List listaZ = new ArrayList(); Map posZ = new HashMap();
24
25          iniciarLaberinto(listaZ,maze, posZ, M, N);
26
27          int posRandomInicial = (int) (Math.random()*listaZ.size());
28          posZ =  (Map)listaZ.get(posRandomInicial);
29          int i = Integer.parseInt(posZ.get("i").toString());
30          int j = Integer.parseInt(posZ.get("j").toString());
31
32          //Se marca el primer IN al azar
33          maze[i][j]=IN;
34
35          //Se crea la lista mínima
36          List Q = new ArrayList();
37          Map posQ = new HashMap();
38
39          //Se procede a asignar los vecinos
40          evaluarVecinos(maze, Q, posQ, i, j, N, M);
41
42          int randomQ;
43          while (Q!=null && !Q.isEmpty()){
44              randomQ = (int) (Math.random()*Q.size());
45              posQ = (Map)Q.get(randomQ);
46              i = Integer.parseInt(posQ.get("i").toString());
47              j = Integer.parseInt(posQ.get("j").toString());
48              maze[i][j]=IN;
49
50              if((i-2>=0) && maze[i-2][j]==IN && maze[i-1][j]==BLOCK){maze[i-1][j]=IN;}
51              else if((i+2<=2*N) && maze[i+2][j]==IN && maze[i+1][j]==BLOCK){maze[i+1][j]=IN;}
52              else if((j-2>=0) && maze[i][j-2]==IN && maze[i][j-1]==BLOCK){maze[i][j-1]=IN;}
53              else if((j+2<=2*M) && maze[i][j+2]==IN && maze[i][j+1]==BLOCK){maze[i][j+1]=IN;}
54
55              evaluarVecinos(maze, Q, posQ, i, j, N, M);
56
57              Q.remove(randomQ);
58          }
59      }
```

Source: Own elaboration

VII. ALGORITHM PERFORMANCE

To take a measurement of the performance of this algorithm is necessary to place a line in the Source Code. In line 17 and line 58 of figure 12 we add the following Java functions respectively:

Long tiempo_ini = System.nanoTime();

Long tiempo_fin = System.nanoTime();

The function nanoTime() provided by Java allows a numerical value of the current date expressed in nanoseconds. Therefore, to calculate the processing time simply apply the following subtraction:

Long time = tiempo_fin – tiempo_ini;

Additionally, minimal changes are made to the algorithm to generate a given amount of labyrinths each time it runs. Simply enclose the entire code for the

main () of figure 12 in a loop a specified number of iterations. This change is necessary to make runs by changing the amount of generated mazes in these. Table 1 shows the results of 10 runs for the generation of 5, 10, 20 30, 40 and labyrinth 50.

As a result, the table 2 shows the average time to generate 5, 10, 20, 30, 40 and 50 based labyrinths data in table 1, expressed in seconds rounded to four decimal places.

Finally, a figure 14 show in a graph depicts the increase in nanoseconds before a higher number of mazes generated. In other words, we see a reasonable increase in processing time with the increase in the number of mazes in each iteration.

Table 1. Time in nanoseconds of the algorithm

# of mazes Run	5	10	20	30	40	50
1	25100354	37383187	66987288	92819763	89280761	96249970
2	39245483	44996200	76434120	81207293	96203893	95426656
3	24432231	52134360	93389011	94756613	89914005	92501380
4	30888190	53493324	90121673	82727208	96911053	91012823
5	27527099	53009192	77247834	89248122	92009248	86443799
6	33279737	50209668	86170855	86339805	84164886	98480565
7	25106754	57479661	82021326	81133056	105141952	91453119
8	27646452	43775788	83331973	85521610	89377075	95432096
9	28601918	60932588	70575248	84874287	87789964	100370700
10	32442023	50192710	82701929	94108649	95315943	96362924

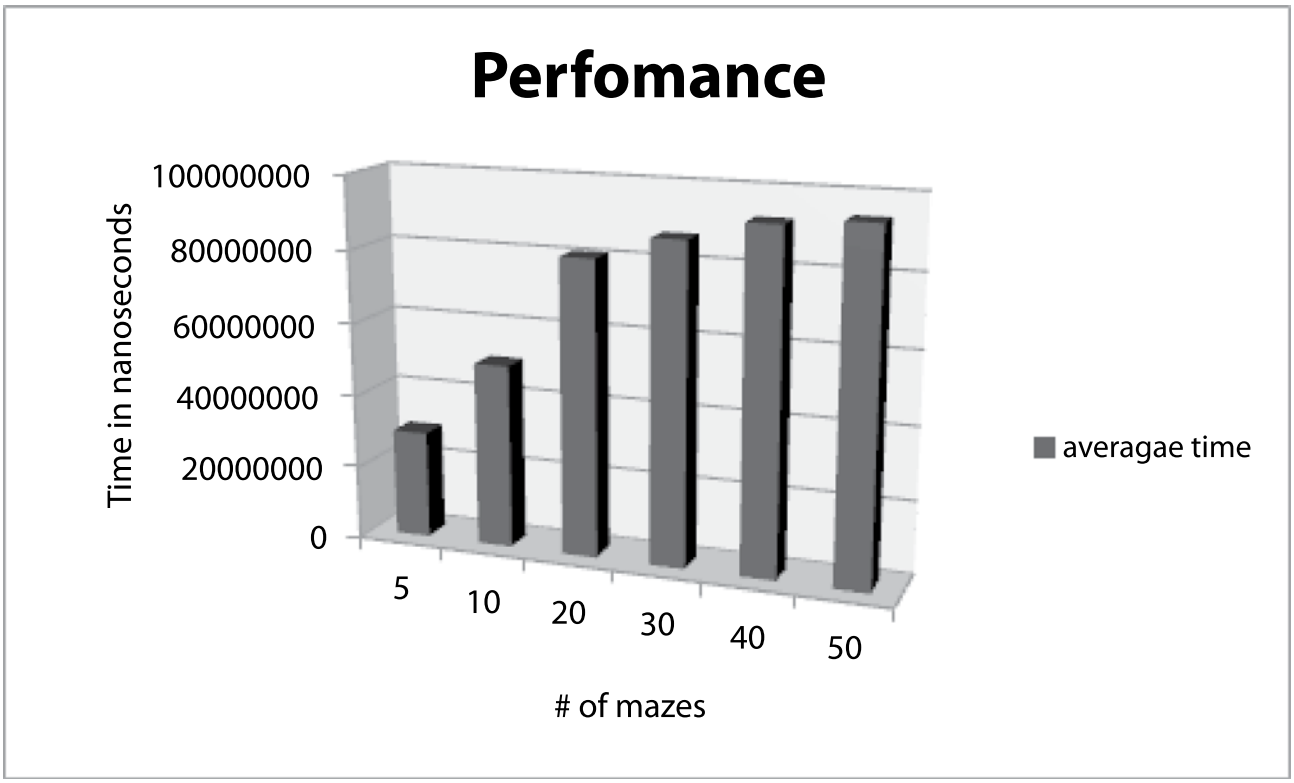
Source: Own elaboration

Table 2. Average times in seconds

	5	10	20	30	40	50
Average time	0,0294	0,0504	0,0809	0,0873	0,0926	0,0944

Source: Own elaboration

Figure 14. Graph Prim algorithm performance



Source: Own elaboration

VIII. CONCLUSIONS

1. It shows that you can adapt the Prim algorithm, which provides graph theory, to generate random mazes games by two-dimensional arrays.
2. The performance of this algorithm is quite efficient, because tests showed generally not take more than a tenth of a second to generate considerable as 50.
3. The Java programming language provides libraries and functionality quite useful for coding the Prim algorithm used in this case were Math. random() to generate random numbers and System.nanoTime() for performance measurement.

BIBLIOGRAPHY

[1] Buckland M. LaMothe A. (2002). A1 Techniques for Game Programming. Premier Press. USA.

[2] Chartrand G. Lesniak L. (2000). Graphs & Digraphs. CRC Press. USA.

[3] Cormen T. Leiserson C. Rivest R. (2009). Introduction to Algorithms. Third Edition. The MIT Press, Cambridge, Massachusetts. USA.

[4] Epifania A. (2009). “Soko-ban”: Cuando del SGA podía ser tolerable.http://www.epimundo.com/2009/11/soko-ban-cuando-el-cga-podia-ser-olerable.html. (visited 13-01-2013).

[5] Eugenia M. (2008). Los juegos más influyentes (1ª parte).http://www.kamegame.com/2008/03/17/los-juegos-mas-influyentes-1%C2%AA-parte/. (visitado el 12-01-2013).

[6] Gonzáles R. (2011). Algoritmo de Prim. http://esteban-gzz.blogspot.com/2011/07/algoritmo-de-prim.html. (visited 12-01-2013).

[7] Jongart D. (2008). Isometric Map_Jungle. http://jongart.deviantart.com/art/Isometric-Map-Jungle-75169895 (visited 26-02-2013).

[8] Palma J. Marín R. (2008). Inteligencia Artificial. Técnicas, métodos y aplicaciones. Mc Graw Hill. España.

[9] Ramker R. (2011). Historia de los Videojuegos: El origen y los inicios. http://www.otakufreaks.com/historia-de-los-videojuegos-el-origen-y-los-inicios. (visited el 12-01-2013).