Coverage in Arbitrary 3D Environments

The Art Gallery Problem in Shooter Games

Eduardo Penha Castro Fantini, Luiz Chaimowicz
Computer Science Department
Federal University of Minas Gerais
Belo Horizonte, Brazil
{fantini, chaimo}@dcc.ufmg.br

***Abstract*—The Art Gallery Problem consists in determining the minimum number of observers required to cover an environment such that each point is seen by at least one observer. This is a NP-Hard problem well known in the field of computational geometry. In the literature, several restrictions are applied to 2D and 3D environments to study and solve the problem in polynomial time, for example the use of simple polygons, orthogonal and planar environments, etc. In this paper we present an approximate and polynomial solution based on metaheuristic genetic algorithms that can be applied to general 3D environments without any restriction and, therefore, applicable in shooter games and also real-world environments. The solution uses the techniques of (i) computer graphics to generate sample points in the environment, (ii) ray-mesh intersection test to generate a graph of visibility between the samples and (iii) genetic algorithms to find and optimize the minimum set of observers. The maps of the game Counter-Strike were used to analyze the placement of small groups of observers in complex environments with obstacles. The game engines Half-Life and Irrlicht were used to apply the ray-mesh intersection test in 3D environments. A series of experiments were performed and the results show that our methodology is capable of obtaining a good coverage of space with a small number of agents observing.**

Keywords—art gallery problem; computational geometry; visibility; natural computation; computer graphics; shooter games

I. INTRODUCTION

The Art Gallery Problem (AGP) was introduced in 1973 by Vitor Klee, when he asked how many stationary guards are needed to cover an art gallery room with *n* walls [1]. This visibility problem and its variations have been deeply studied ([2, 3, 4]) and support a series of applications in the real world ([5, 6, 7]), e.g., security cameras, military scout positioning, cellular antennas distribution, and optimizations.

It was shown that determining an optimal minimum set of guards to cover a polygon is NP-Hard, even for simple polygons [8]. Some approximation algorithms with logarithmic approximation ratios [9, 10, 11] and genetic algorithms [12, 13] are applicable for restricted versions of the problem, positioning guards at vertices or at points of a discrete grid. Constant-factor approximations are known for guarding 1.5D terrains and monotone polygons [14, 15, 16], and exact methods are proposed for the special cases of rectangle and triangle visibility in 2D orthogonal polygons without holes [17, 18]. In 3D environments there are studies about elevation maps restricted to planar terrains [6], and about orthogonal polyhedral environments [19], a type of constraint since it works only for orthogonal 3D polygons. In spite of these efforts, the unrestricted version of the optimization problem for 2D and 3D environments remains open.

In this scenario of unconstrained environments, we propose a methodology based on sampling techniques, ray-mesh intersection test and metaheuristic genetic algorithms to provide approximate solutions to the following problem:

Given an arbitrary three-dimensional environment, what is the minimum number of observers we need to cover the whole environment and where should we place them?

Answering this question, even approximately, is important for a large number of applications in the real world and also in the digital games field. There are several applications in games in which considering a minimum set of points able to monitor or cover a complex volume in the space is important, for example:

- Strategic positioning of players to cover a map against enemies in multiplayer shooting games.
- Effective positioning of light sources, which are limited in engines, to illuminate scenarios during the level design process.
- Positioning of cameras in the environment for transmission or recording of matches in three-dimensional games.

This work focuses on the first game application example and uses the Counter-Strike 1.6 as the experimentation platform. Counter-Strike is a mod for the game Half-Life with a partially accessible code by the Half-Life SDK and Dynamic-Link Libraries. The Counter-Strike is a widespread shooting game and has complex 3D maps that generate arbitrary polygons with holes and non planar visibility graphs. These characteristics were important for the choice of platform.

The paper is organized as follows: Section 2 will address the basic concepts of the Art Gallery Problem and related subjects while the Section 3 will explain the methodology that we propose. The Section 4 presents a series of experiments performed and their results. Finally, Section 5 brings the conclusion and directions for future work.

II. PRELIMINARIES

To support the understanding of the addressed problem, we introduce some necessary concepts and notations.

The Art Gallery Problem is a NP-Hard optimization problem. Finding an exact solution and checking its optimality are problems with exponential complexity [8].

Both in the original definition of the problem in 1973 and in this work, a guard is a stationary point placed on the environment and has a range of visibility equals to 2π, i.e. 360 degrees [1].

The minimum set of guards needed to cover an environment, polygon or samples set is called Minimum Vertex Guard (MVG) [12]. Therefore, the MVG is the exact solution for AGP. For approximate solutions, we will use the nomenclature Approximate Minimum Vertex Guard (AMVG).

In computational geometry, the visibility graph is a graph of intervisible locations made for a set of points and obstacles in n-dimensional Euclidean spaces. Each vertex of the graph represents a point, and each edge represents a visible link between two of them.

A. Notation used in our aprouch

Given an undirected visibility graph $G = \{V, E\}$, generated from the samples V of a three-dimensional environment, we must find the set AMVG, identified as S, so that $S \subseteq V$. Each edge e_{ij} indicates that v_i and v_j are visible samples to each other.

Whereas $v_i \in V$, $s_i \in S$ and $e_{ij} \in E$, we can say that S is the set with the smallest cardinality $K = |S|$ such that, for all v_i there is an edge e_{ij} between v_i and s_j or $v_i = s_j$.

Therefore, we are searching for the subset S with the smallest possible cardinality K that covers all vertices of the set of samples V.

III. PROBLEM MODELING AND METHODOLOGY

In this paper we propose a methodology for finding approximate solutions to the AGP in arbitrary environments, whether two-dimensional or three-dimensional. This process includes three distinct stages, each one with flexible use of computational techniques.

Given an arbitrary environment, we first generate a set of sample points belonging to the area or volume that we want to cover. Then we construct a visibility graph, where the samples represent the vertices and an edge represents visibility between two of them. Finally, we apply genetic algorithms to search for the AMVG on this graph.

A. Generating Samples in an Environment

The first step we perform in our model is the mapping of sample points in an environment. There are several strategies in computer graphics to generate sample points on surfaces, each with its advantages and disadvantages depending upon the application.

Among the sampling techniques we can highlight: Uniform, Random, Stratified, Latin Hypercube, Poisson-Disk and Best Candidate Samplings. These are generally applied in two-dimensional surfaces, but can be adapted to three-dimensional environments [20]. Other possibilities include more advanced Hammersley and Halton samplings for 3D geometry [21].

It should be emphasized that the mapping of samples is limited to the volume defined by the geometric walls and obstacles in the environment. Depending upon the application, we can further reduce this volume to areas of interest. For example, generating samples only in areas where it will be necessary to monitor, which does not always encompasses the entire environment.

For Counter-Strike game, we utilize the bots (computer-controlled players) navigation waypoints as mapping samples. The waypoints are vertices of the path finding graphs, a well-known artificial intelligence technique applied in continuous three-dimensional environments for games [22]. The waypoints of Counter-Strike maps are generated from real players moving around the space during several matches. After that it is possible to make improvements manually. In Counter-Strike, each waypoint has a lot of useful information for intelligence decision of bots (see *Fig. 1*), e.g., where to camp, where to plant bombs, mobility possibilities, etc.

The region we want to monitor is the same one in which real players and bots are moving. For this reason we decided to take advantage of these waypoints for the visibility graph.

Fig. 1. Waypoint information in Counter-Strike. Inside the game the waypoints are shown as vertical bars.

Another alternative technique that we used for generating samples in the game was the uniform distribution over the volume. We applied it in regions of interest. Regions of interest which we call *green zones* are regions that a character can occupy in the environment while moving. Thus, to boost the efficiency of the process, we do not need to monitor volumes outside the *green zones*. The *green zones* are defined by the geometry of the environment and their obstacles are called *red zones*. The uniform distribution over the volume is a simple sampling technique and consists in generating equally spaced samples in the three-dimensional environment [20].

To extract sample waypoints of the game, we created a command console for Counter-Strike that accesses information

from running maps and saves them to an external file. For generating a uniform mapping, we loaded the game map in the Irrlicht[1] graphics engine, created routines to generate the uniform samples and validate some of them inside the *green regions* defined manually with the support of the software Autodesk 3D Max.

It is important to consider that our manual step to create *green zones* is optional, but can provide considerable gains in efficiency depending on the number of reduced samples.

B. Making the Visibility Graph

The second step of the methodology is the construction of a visibility graph using the technique of ray-mesh intersection test. For each pair of vertices we trace a ray starting from the first to the second, and if that ray does not collide with the geometry of the scenario we created an edge. Thus, each edge indicates that there is visibility between its two vertices.

Here we use the term ray-mesh intersection test generically, because in game development field there are a lot of names for the same technique to test the visibility between two points in three dimensional spaces (e.g. trace line test, hit test, check ray intersection, collision point from ray, etc…). All of them use a mathematical method to test whether a ray between two points intersects triangles of the given object. It is a variation of the ray casting algorithm, the most basic of many computer graphics rendering algorithms that use the geometric algorithm of ray tracing. [20].

We decided to use this technique because it is computationally cheap and one of the simplest ways to check if two points are visible between themselves in 3D spaces. The ray-mesh intersection test is widely used to ballistics in shooter games and it is present in most of the game engines.

In Counter-Strike we access the method called *TRACE_LINE* through the Half-Life engine and create a new command in the game console to generate a visibility graph between the waypoints of each map during the execution of the matches.

To generate visibility graphs out of the game, we used the method called *getSceneNodeAndCollisionPointFromRay* from Irrlicht engine. So, loading the Counter-Strike maps in Irrlicht and generating valid samples, it is possible to create the corresponding visibility graph between them.

C. Finding the Minimum Vertex Guard or an Aproximation

The third step of our methodology involves finding the minimum set of guards to cover the visibility graph. This is the Minimum Set Cover (MSC) optimization problem, also NP-Hard. We have developed two algorithms, one exact based on the backtracking paradigm and another based on metaheuristic genetic algorithms to compare some results.

As this stage is the most computationally complex of the three of our methodology, we did the analysis of time complexity of the proposed algorithms. The algorithms we used to find the MVG and AMVG provides the complexity asymptotic upper bound of our methodology.

[1] Irrlicht Engine is an open source high performance realtime 3D engine written in C++. Website: *http://irrlicht.sourceforge.net/*

1) An Exact Algorithm Based on Backtracking

The exact algorithm (see *Algorithm 1*) is sensitive to the size of the MSC and has exponential complexity of time, so it is not efficient for graphs whose MSC sets are large. The analysis of time complexity of this algorithm is given in (1), where N is the number of samples, K is the size of exact MSC and C is the size of the set cover for testing per iteration. The MSC solution is the same for MVG in a graph.

The *Algorithm 1* can run several iterations. The first one generates all the possibilities of sets with size one and tests whether each set covers all samples (*Algorithm 2*). The second iteration generates all sets with size two and so on. In this way, when we found one or more sets with size K that covers all samples, the *Algorithm 1* ends and we found out the optimal solution, the MSC or MVG.

$$\sum_{C=1}^{K} \frac{N!}{(N-C)!\ C!}\, CN \quad \text{for } 1 \le K \le N \tag{1}$$

For the worst case, when $K = N$, the time complexity is given by the following equation:

$$O\left((2^N - 1)N^2\right) \tag{2}$$

Algorithm 1: Exact Algorithm

Input: adjMatrix, |V| // Adjacency matrix and number of vertices

```
1:  for i ← 0 to i < |V| do
2:    if |MSC_solutions| != 0 then
3:      break // algorithm stop condition
4:    end if
5:    K ← i + 1 // number of elements in candidate set
6:    candidate.push(i) // candidate solution
7:    for j ← 0 to j < i do
8:      candidate[j] ← j
9:    end for
10:   if checkFullCoverage(adjMatrix, candidate) = true then
11:     MSC_solutions.push(candidate)
12:   end if
13:   flag ← false
14:   pivot ← K-1
15:   while flag = false do
16:     while positions[pivot] = (pivot + N - K ) do
17:       if pivot = 0 then
18:         flag ← true
19:         break
20:       else
21:         pivot ← pivot - 1
22:       end if
23:       if flag = false then
24:         candidate[pivot] ←  candidate[pivot] + 1
25:         for m ← pivot + 1 to m < K do
26:           candidate[m] ← candidate[m-1] + 1
27:           if checkFullCoverage(adjMatrix, candidate) = true then
28:             MSC_solutions.push(candidate)
29:             pivo ← K-1
30:           end if
31:         end for
32:       end if
33:     end while
34:   end while
35: end for
36: print MSC_solutions
```

Algorithm 2: CheckFullCoverage

Input: adjMatrix, S, |V| // Adjacency matrix, solution and number of vertices

```
1:  if |S| > |V| then
2:    return false
3:  end if
4:  for j ← 0 to j < |V| do
5:    Covered[j] ← false // vector to mark coverage
6:  end for
7:  count ← 0
8:  for i ← 0 to i < |S| do
9:    if Covered[S[i]] = false then
10:     Covered[S[i]] ← true
11:     count ← count + 1
12:     for j ← 0 to j < |V| do
13:       if adjMatrix[S[i]][j] = true AND Covered[j] = false then
14:         Covered[j] ← true
15:         count ← count + 1
16:       end if
17:     end for
18:   end if
19: end for
20: if count != N then
21:   return false
22: end if
23: return true
```

2) An Aproximate Algorithm Based on Genetic Algorithms

Due to the high computational complexity of this optimization problem, we look for solutions through a metaheuristic genetic algorithm. A genetic algorithm (GA) is a search technique used to find approximate solutions for optimization problems and uses techniques inspired by evolutionary biology such as inheritance, mutation, natural selection and recombination (e.g. crossing over).

Genetic algorithms are implemented as a computer simulation in which a population of abstract representations of selected solution is used to find better solutions. The evolution starts from a set of solutions randomly created and is performed through generations. In each generation, the adaptation of each solution in the population is evaluated by a fitness function. Then, some individuals are selected via tournament for the next generation and recombined by crossing over or mutated to generate a new population. The new population is then used as input to the next iteration of the algorithm until the stopping condition is reached (see *Algorithm 3*).

Algorithm 3: Genetic Algorithm

```
1.  for r ← 1 to 30 do // number of repetitions
2.    g ← 0 // generation
3.    Initialize the random population, P(g)
4.    while target generation g not met do
5.      Fitness Evaluation of P(g)
6.      Selection by tournament on P(g)
7.      Crossing Over P(g)
8.      Mutate P(g)
9.      g ← g +1
10.     Generate P(g) from P(g −1)
11.   end while
12. end for
13. return Best Solution
```

Being a stochastic method, the experimental evaluation of the genetic algorithm should be performed with repetitions so that the results should be reported according to the average value and its standard deviation (σ).

To design a genetic algorithm it is necessary to define certain characteristics and parameters, such as:

a) Representation of individuals: An individual (i.e. a candidate solution) in our GA is represented by a binary string of the size of our sample points set. The bit value of 1 means that this is a point guard and belongs to AMVG. Otherwise this sample should be covered by another vertex (see *Fig 2*).

0	**1**	**0**	**0**	**0**	**1**	**0**	**…**	**0**	**0**	**0**	**0**	**0**	**1**

Fig. 2. Representation of individuals in genetic algorithms.

b) Method of selection: In all of our experiments, we used the tournament method with two individuals. Elitism is also applied, preserving the best individual to the next generation.

c) Fitness evaluation: In the evaluation of a candidate solution we consider the size of the cover set and the amount of samples it covers. Smaller sets with greater coverage get the best fitness. The maximum fitness value in our algorithm is 100.

d) Population size: This parameter should be a sufficient value to generate a population diversity able to avoid the premature converge of solutions.

e) Number of generations: This parameter depends on the behavior of the algorithm with respect to the convergence of the results. After the experiment, if we observe that the fitness did not converge (high standard deviation), we increase the number of generations and repeat the experiment. After running some experiments, we observed that seting the number of generations equal or greater than the number of vertices of the visibility graph, the genetic algorithm generates better results. So, we keeped it close to the number of vertices.

f) Probabilities of crossing over and mutation: These parameters are calibrated by analysing the standard deviation of the *Average of Average Fitness* (AAF) in final results of the calibration algorithm (see *Algorithm 4*). Smaller standard deviations indicates better convergence and better set of parameters for GA. In our experiments we used the random point method for the crossing over and mutation operators.

More details about GA parameters control can be seen at the reference [23].

The analysis of time complexity of our genetic algorithm is given in (3), where N is the number of samples, E is the number of edges in the visibility graph, P is the population size, G is the number of generations and R the number of repetitions.

$$O(RGP(N+E)) \quad (3)$$

Algorithm 4: Genetic Algorithm Calibration

```
for generations ← 100 to 200 do
  for population ← 100 to 200 do
    for pCross ← 0.6 to 0.9 do
      for pMutation ← 0.01 to 0.10 do
        for s ← 0 to 30 do
          GA(graph, generations, population, pCross, pMutation, seed[s])
        end for
        pMutation ← pMutation + 0.05
      end for
      pCross ← pCross + 0.15
    end for
    population ← population + 50
  end for
  generations ← generations + 100
end for
```

IV. EXPERIMENTS AND RESULTS

To explore our methodology for AGP in arbitrary three-dimensional environments, we performed a series of experiments and analyzed their results.

A. *Genetic and Exact Algorithms applied to Sparse Graphs with Known Minimum Vertex Guard*

Initially we developed a generator of sparse graphs, specifically Minimum Spanning Trees (MST), with known MVG sets to compare the results and time performance of our exact and genetic algorithms. This experiment focuses only on step three of our methodology.

Minimum Spanning Trees are the worst cases of connected visibility graphs. In a complete graph, for example, the MVG has one element that can be any vertex of the graph.

The parameters used for the genetic algorithm were: repetitions = 30, initial population = 250, mutation probability = 10% and crossover probability = 90%. For graphs with 100 vertices we performed 200 generations, for graphs with 500 vertices we run 600 generations and for graphs with 1000 vertices, 1100 generations. The results are shown in *Table 1*.

TABLE I. EXACT VERSUS GENETIC ALGORITHMS RESULTS

Input Graph		**Exact Algorithm**		**Genetic Algorithm**		
Vertexes	***MVG***	***MVG***	***Time (s)***	***AMVG***	**σ[a]**	***Time (s)***
100	1	1	0.00	1	± 0.137	181.26
100	2	2	0.07	2	± 0.132	186.21
100	3	3	2.69	3	± 0.135	186.78
100	4	4	77.68	4	± 0.121	151.91
500	1	1	0.04	1	± 0.101	1.83e+003
500	2	2	10.45	2	± 0.050	1.81e+003
500	3	3	2.2e+003	3	± 0.061	1.23e+003
500	4	4	1.71e+005	4	± 0.045	0.93e+003
1000	1	1	0.23	1	± 0.101	4.48e+003
1000	2	2	141.17	2	± 0.069	4.39e+003
1000	3	3	3.90e+004	3	± 0.045	3.74e+003

a. Standard deviation from average of average fitness.

We can observe that the time consumption behavior of the exact algorithm is exponential, while the genetic algorithm practically does not changes over graphs with similar vertex quantity. We also verified that genetic algorithms found optimal solutions for this experiment with satisfactory convergence of population.

A MST graph with 200 vertices and a MVG with size 4 was used to calibrate the parameters of our genetic algorithm. The *Table 2* shows the sets of parameters that gave better convergence results. The tested parameters were: population size *P*, number of generations *G*, crossover probability *Pcross* and mutation probability *Pmut*. We are looking for sets that results in average of average fitness *AAF* near 100 with smallest standard deviation σ. Therefore, the Set 4 was the best.

TABLE II. GENETIC ALGORITHMS CALIBRATION: SOME PARAMETER SET RESULTS

Parameter set	**P**	**G**	**Pcross**	**Pmut**	**AAF**	**σ[a]**
Set 1	100	100	0.60	0.01	99.99	± 0.0008
Set 2	150	200	0.60	0.05	99.99	± 0.0008
Set 3	150	100	0.75	0.01	99.97	± 0.0033
Set 4	150	200	0.90	0.10	99.99	± 0.0000
Set 5	100	100	0.60	0.01	99.99	± 0,0008

a. Standard deviation from average of average fitness.

.

B. *Genetic Algorithms applied in Counter-Strike Waypoints*

In order to simplify the sampling in the first stage of our methodology, we take the vertices of the navigation graph of bots from the Counter-Strike maps, the waypoints. Over the years, these waypoints were improved by the community and developers, which made them an interesting basis of information for many types of research (see *Fig. 3* and *Fig. 4*).

After collecting waypoints from each map, we create the visibility graph via ray-mesh intersection test within the game itself, saving them to external files. This is the second stage of our methodology (see *Fig. 5*).

Finally, in the third stage, we apply the genetic algorithm to find the AMVG for these samples (see *Fig 6* and *Fig. 7*).

Fig. 3. Waypoints (black points) in Cs_italy map.

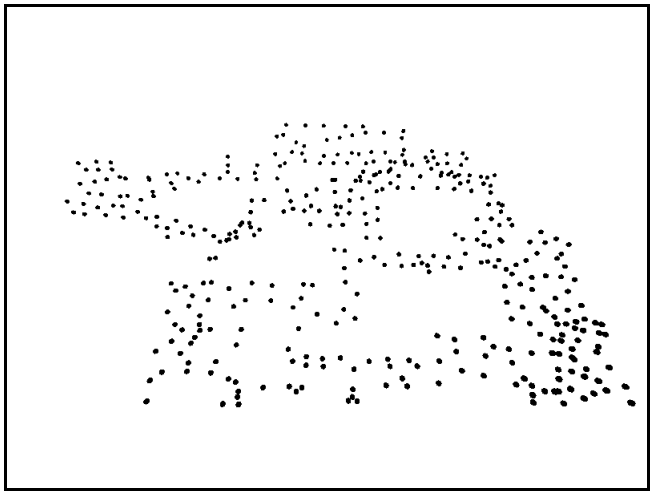

Fig. 4. Waypoints extraction from Cs_italy map.

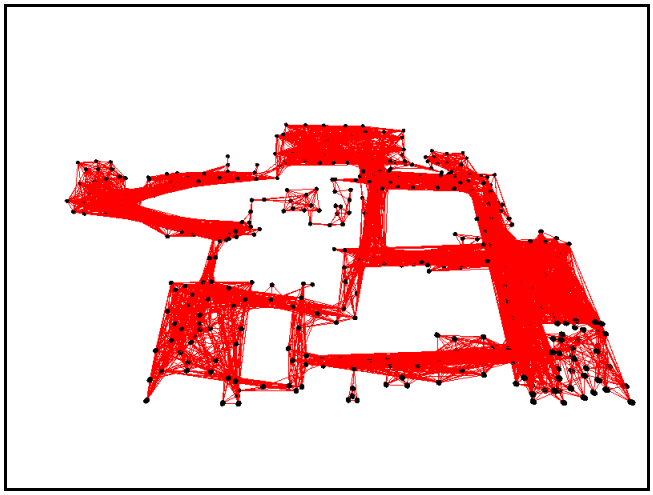

Fig. 5. Visibility graph generated by ray-mesh intersection test for Cs_italy map samples.

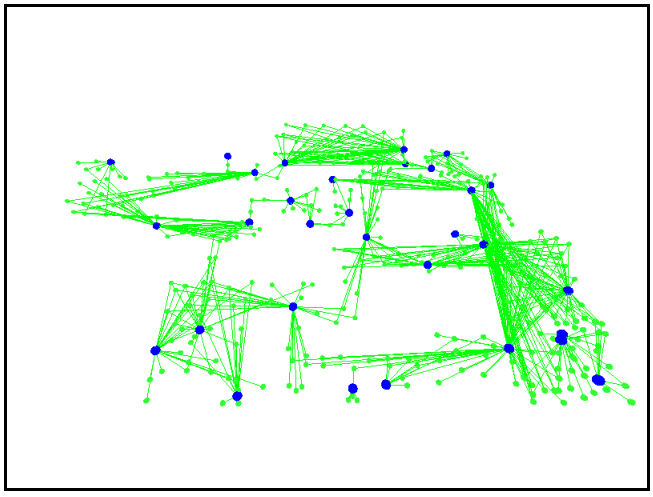

Fig. 6. Aproximate Minimum Vertex Guard for Cs_italy map samples.

Fig. 7. Approximate Minimum Vertex Guard points inside Cs_italy map.

Waypoints can provide useful information for bots. This is an artificial intelligence technique used in games where bots collects data on the environment. For comparative purposes, we used a type of existing information, i.e. the points the camp. Camp points are waypoints with information indicative for bots stay lying in wait, monitoring the environment to surprise the opponent. These points are usually occupied by snipers in the game. We compared the number of this known camp points from Counter-Strike maps with the AMVG found by the genetic algorithm. We also analyzed the percentage of samples coverage of camp points and AMVG (see *Table 3*).

The parameters used for the genetic algorithm were: repetitions = 30, initial population = 250, number of generations = 700, mutation probability = 10% and crossover probability = 90%.

In the previous experiment, the maps Cs_747, Cs_havana, De_chateau and De_torn presented sets of camp points smaller than AMVG, but with different coverage. So, we modified the fitness of the genetic algorithm to find the AMVG with coverage near of 60%, 70%, 80% and 90%. Our goal was to explore different percentages of coverage and analyze their impact on the size of AMVG. We also included other maps in this experiment, which have variations in numbers of vertices and edges. They are: Cs_italy, Cs_militia, De_cbble, De_dust and De_prodigy. The results are shown in *Table 4*.

We can observe a significant reduction in size of the AMVG for smaller coverings percentage. For maps Cs_747, Cs_havana, De_chateau and De_torn, the AMVG are smaller than the set of camp points with the same range of coverage.

The parameters used for the genetic algorithm were: repetitions = 30, initial population = 250, number of generations = 700, mutation probability = 10% and crossover probability = 90%.

TABLE III. GENETIC ALGORITHMS IN COUNTER-STRIKE MAPS

Input Map				Genetic Algorithm	
Name	***Vertexes***	***Edges***	***Camp Points / % Cover***[a]	***AMVG / % Cover***	**σ**[b]
As_oilrig	487	6313	39 / 70.2	32 / 100	± 0.006
Cs_747	388	6210	24 / 79.1	33 / 100	± 0.014
Cs_assault	470	17782	50 / 94.9	18 / 100	± 0.006
Cs_backalley	375	4592	44 / 66.7	2 / 100	± 0.112
Cs_estate	399	9350	52 / 97.7	23 / 100	± 0.009
Cs_havana	396	3883	30 / 69.9	34 / 100	± 0.010
Cs_italy	391	5116	72 / 97.4	31 / 100	± 0.008
Cs_militia	603	12194	63 / 90.7	29 / 100	± 0.011
Cs_office	386	4968	52 / 96.6	3 / 100	± 0.057
Cs_siege	557	12845	51 / 74.9	30 / 100	± 0.006
De_airstrip	521	7596	45 / 63.7	39 / 100	± 0.004
De_aztec	521	11495	61 / 81.0	31 / 100	± 0.004
De_cbble	631	14106	67 / 82.6	40 / 100	± 0.004
De_chateau	540	6627	22 / 53.5	42 / 100	± 0.005
De_dust	476	9619	43 / 81.3	26 / 100	± 0.006
De_dust2	433	7572	54 / 96.5	26 / 100	± 0.007
De_inferno	440	5206	49 / 83.2	25 / 100	± 0.009
De_nuke	618	12325	41 / 69.6	16 / 100	± 0.002
De_piranesi	572	8531	75 / 77.3	47 / 100	± 0.003
De_prodigy	337	2878	40 / 79.2	34 / 100	± 0.015
De_storm	492	10095	41 / 75.6	26 / 100	± 0.011
De_survivor	513	10121	46 / 81.7	4 / 100	± 0.048
De_torn	418	4645	33 / 58.6	36 / 100	± 0.005
De_train	521	11667	42 / 89.4	15 / 100	± 0.007
De_vertigo	383	4529	51 / 88.3	36 / 100	± 0.006

a. Number of map camp points and your respective waypoints samples coverage.

b. Standard deviation from average of average fitness.

TABLE IV. GENETIC ALGORITHMS IN COUNTER-STRIKE MAPS: AMVG RESULTS FOR DIFFERENT COVERING PERCENTAGE TARGETS

Input Map	Cover Set Size / % Cover				
	Target 60%	***Target 70%***	***Target 80%***	***Target 90%***	***Target 100%***
Cs_747	5 / 60.6	7 / 71.1	10 / 80.7	15 / 90.5	33 / 100
Cs_havana	5 / 60.1	8 / 72.0	12 / 80.8	19 / 90.1	34 / 100
Cs_italy	6 / 62.1	9 / 70.3	13 / 80.1	19 / 90.0	31 / 100
Cs_militia	6 / 59.5	7 / 70.1	7 / 79.6	12 / 89.4	29 / 100
De_cbble	6 / 60.4	9 / 70.5	14 / 80.0	22 / 90.0	40 / 100
De_chateau	10 / 60.0	13 / 70.4	17 / 80.4	24 / 90.0	42 / 100
De_dust	5 / 60.3	6 / 71.6	9 / 81.9	12 / 90.3	26 / 100
De_prodigy	7 / 60.2	10 / 71.2	15 / 80.4	20 / 90.5	34 / 100
De_torn	8 / 60.8	11 / 71.3	14 / 80.4	19 / 90.2	36 / 100

C. *Exact Algorithm applied in Counter-Strike Waypoints*

Analyzing the *Table 3*, we found some results of the genetic algorithm with reduced AMVG. It allows us to run the exact algorithm to check how close these results are of the MVG. For maps Cs_backalley, Cs_office and De_survivor we performed the exact algorithm, obtaining the results given in *Table 5*. As can be seen in these results, the genetic algorithm obtained solutions near or equal to the MVG.

TABLE V. EXACT AND GENETIC ALGORITHMS IN COUNTER-STRIKE MAPS: TIME AND MVG RESULTS

Input Map	Exact Algorithm		Genetic Algorithm		
Name	***MVG***	***Time (s)***	***AMVG***	**σ**[a]	***Time (s)***
Cs_backalley	2	2.92	2	± 0.112	1.52e+003
Cs_office	3	436.11	3	± 0.057	1.96e+003
De_survivor	3	1.41e+003	4	± 0.048	1.76e+003

a. Standard deviation from average of average fitness.

D. *Genetic Algorithms applied in Counter-Strike Samples Points*

Another type of mapping samples was tested in a Counter-Strike map. Instead of take advantage of the waypoints, we did a 3D uniform sampling in the first stage of the methodology.

We selected the map Cs_assault containing many interesting geometrical elements, such as large open areas, indoor areas, tunnels, ramps, obstacles and high areas.

In the first stage we created *green zones* and *red zones*. *Green zones* are the volumes of interest, i.e. the valid space where a character can be positioned on the map. The *red zones* are obstacles and were created to serve as subtractive volumes to *green zones*. These areas were manually created in Autodesk 3D Max, represented by rectangles, prisms and spheres (see *Fig. 8* and *Fig. 9*). The combination of these primitives allows the creation of arbitrary geometries.

When we started the process of uniform sampling, we observed that some *green zones* had excessive samples and others had samples shortage. This is due to the method of uniform sampling, because the spacing between samples sometimes does not match the volume we need to fill (e.g. narrow tunnels) and thus it was necessary to further subdivide the samples. For this experiment, we subdivided the samples until all the *green zones* were filled.

To validate each sample, we performed geometric calculations to test whether the point is inside a 3D polygon of interest and out of obstacles. See *Fig. 10* and *Fig. 11*.

In the second stage we performed the ray-mesh intersection test using the Irrlicht graphics engine, generating the corresponding visibility graph. The graphs generated by samples of waypoints and the uniform samples are significantly different, as can be seen in *Fig. 12* and *Fig. 13*.

For the last stage we applied the genetic algorithm in visibility graph to find the AMVG. The parameters used for the genetic algorithm with uniform samples were: repetitions = 30, initial population = 200, number of generations = 2000,

mutation probability = 10% and crossover probability = 90%. The comparative results are shown in *Table 6*.

TABLE VI. WAYPOINT VERSUS UNIFORM SAMPLING METHODS

Cs_assault Map			Genetic Algorithm		
Sampling Method	***Vertex***	***Edges***	***AMVG / % Cover***	σ^a	***Time (s)***
Waypoints	487	6313	18 / 100	± 0.006	0.18+e004
Uniform	8184	4110026	48 / 100	± 0.001	1.17+e008

a. Standard deviation from average of average fitness.

The uniform sampling provides a more efficient three-dimensional coverage of the sampling waypoints (see *Fig. 14* and *Fig. 15*), because it requires that the genetic algorithm covers a larger number of samples better distributed in the environment. However, the time to process a huge number of samples is higher.

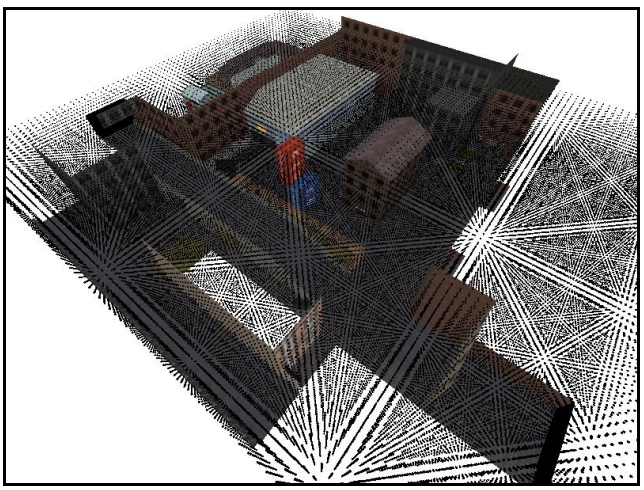

Fig. 10. Uniform three-dimensional sampling in Cs_assault map.

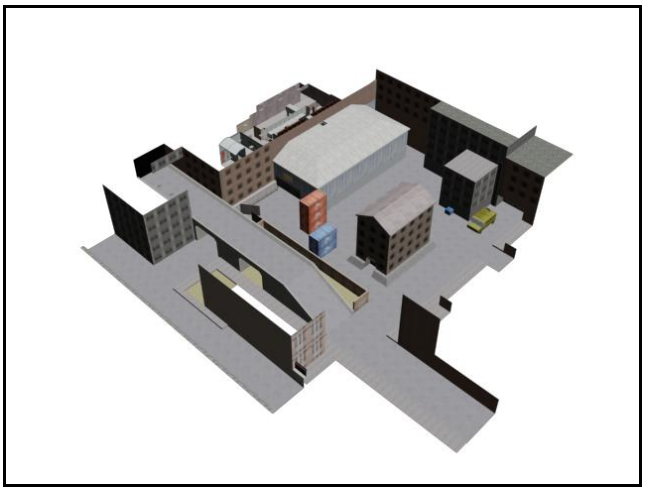

Fig. 8. Cs_assault map geometry.

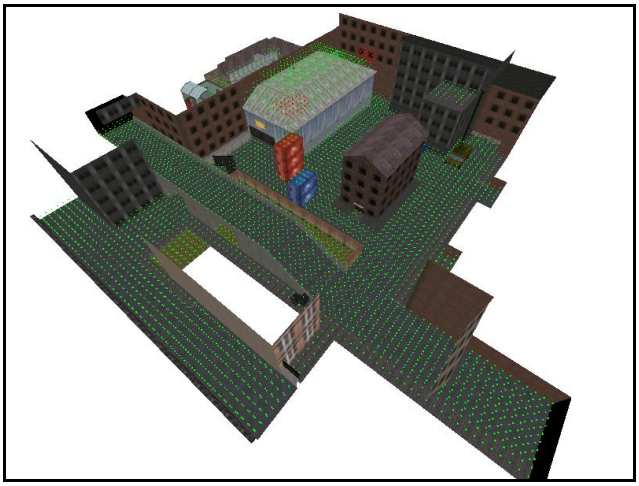

Fig. 11. Valid samples for Cs_assault map (inside green zones).

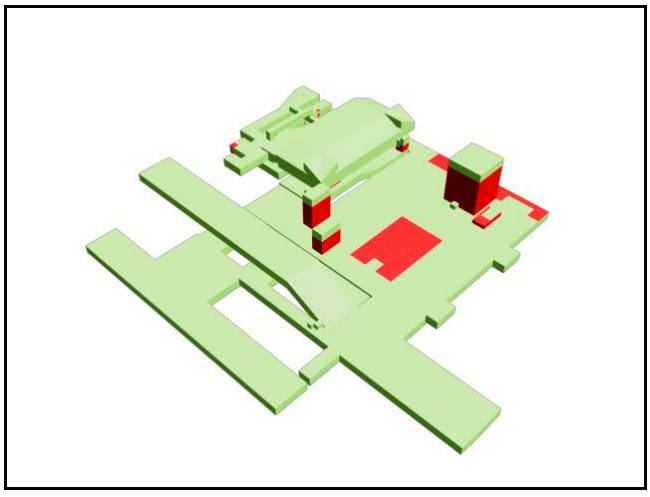

Fig. 9. Green zones and red zones from Cs_assault map.

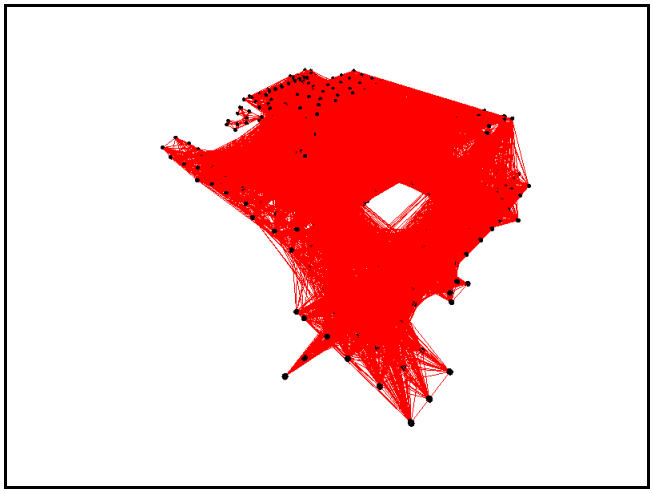

Fig. 12. Visibility graph using waypoint samples of Cs_assault map.

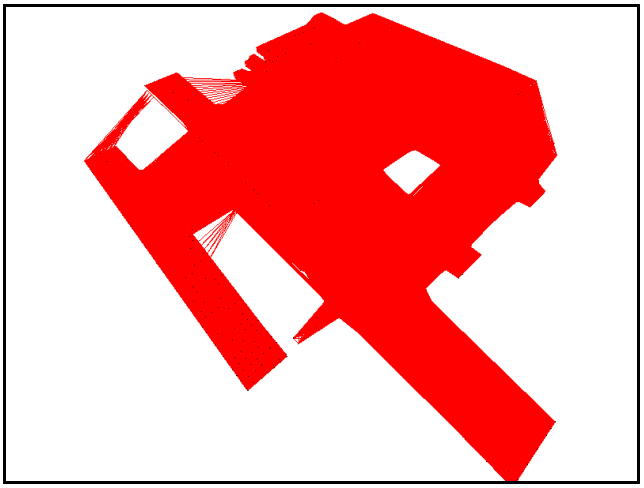

Fig. 13. Visibility graph using uniform sampling for Cs_assault map.

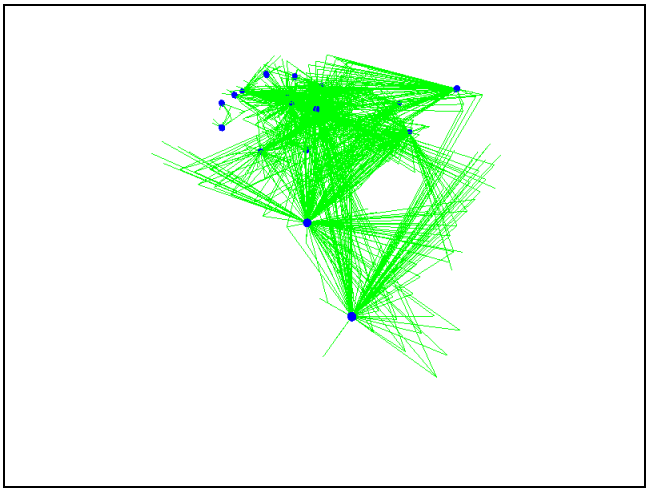

Fig. 14. Approximate Vertex Guard for waypoint samples of Cs_assault map.

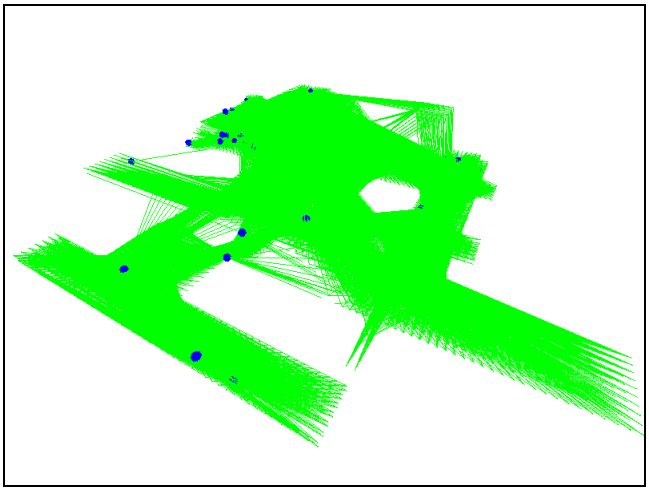

Fig. 15. Approximate Vertex Guard for uniform sampling of Cs_assault map.

All the implementations were done in C/C++ (for Microsoft Visual Studio 2010). The above described methods were tested on a PC featuring an Intel(R) Core(TM) i7 CPU 860 at 2.80 GHz and 4 GB of RAM.

V. Conclusions and Future Work

We can conclude that the proposed methodology for the Art Gallery Problem in arbitrary three-dimensional environments obtains satisfactory solutions of Approximate Minimum Vertex Guard close to optimal results in some experiments.

We demonstrate that our methodology is able to cover all samples of a visibility graph and converge to small coverage sets. So, no matter if we are working with simple or arbitrary polygons from the environment, the most important is to make good mapping of samples based on our interest of monitoring.

In Counter-Strike game we found approximate solutions that cover all samples of the environment, and also sets of guard points smaller and most efficient than the set of camp points known for the maps.

We can affirm that sampling is a stage that requires careful in our approach, because it interferes directly in the quality of the results. Starting with good sampling sets it is possible to obtain satisfactory results using our methodology. So, this study given significant progress for Art Gallery Problems applied in unrestricted environments, enabling new research possibilities.

As future work, we intend to explore other techniques of genetic algorithms to increase the diversity of individuals trying to find better solutions. We also want to investigate others sampling methods, taking advantage of the concept of *green zones*, already implemented in this work, to optimize the process.

Another experiment we want to perform is the cameras placement, where we restrict the positions of observers and apply a reduced angle of visibility to them. It would be interesting to use for recording or live streaming of shooting games.

Acknowledgment

The authors would like to acknowledge the partial support of CAPES, CNPq and Fapemig in the development of this work.

References

[1] R. Honsberger, *Mathematical Gems II*, Mathematical Association of America (1976), 104-110.

[2] J. O'Rourke. Art gallery theorems and algorithms. Oxford University Press, New York, 1987.

[3] J. Urrutia. Art gallery and illumination problems. In J.-R. Sack and J. Urrutia, editors, Handbook of Computational Geometry. North-Holland, 2000.

[4] T. S. Michael. How to guard an art gallery and other discrete mathematicaladventures. Johns Hopkins Press, 2009.

[5] Görkem Safak. The Art-Gallery Problem: A Survey and an Extension. Master's thesis, School of Computer Science and Engineering Royal Institute of Technology, Sweden, 2009.

[6] Marengoni, Maurício, Draper, Bruce A., Hanson, Allen R. and Sitaraman, R.. "A system to place observers on a polyhedral terrain in polynomial time.." Image Vision Comput. 18 , no. 10 (2000): 773-780.

[7] A. Nüchter, H. Surmann, and J. Hertzberg, "Planning robot motion for 3d digitalization of indoor environments," in Proc. of the 11th International Conference on Advanced Robotics (ICAR), 2003.

[8] D. T. Lee and A. K. Lin. Computational complexity of art gallery problems. IEEE Trans. Inform. Theory, 32(2):276-282, 1986.

[9] A. Efrat and S. Har-Peled. Guarding galleries and terrains. Information Processing Letters, 2006.

[10] S. K. Ghosh. Approximation algorithms for art gallery problems. Proc. of the Canadian Information Processing Society Congress, pages 429-434, 1987.

[11] H. González-Banos and and J-C. Latombe. A randomized art-gallery algorithm for sensor placement. In Proc. 17th Annu. ACM Sympos. Comput. Geom., pages 232-240, 2001.

[12] A.L. Bajuelos, S. Canales, G. Hernández, A.M. Martins: Optimizing the Minimum Vertex Guard Set on Simple Polygons via a Genetic Algorithm, in WSEAS Transactions in Information Science and Applications 5 (11), 1584-1596 (2008).

[13] A.L. Bajuelos, S. Canales, G. Hernández, and A.M. Martins. Minimum vertex guard problem for orthogonal polygons: a genetic approach. In Proc. 10th WSEAS International Conference on Mathematical Methods, Computational Techniques and Intelligent Systems (MAMECTIS'08), pages 78-84, 2008.

[14] B. Ben-Moshe, M. J. Katz, and J. S. B. Mitchell. A constant-factor approximation algorithm for optimal terrain guarding. In Proc. 16th ACM-SIAM Symposium on Discrete Algorithms, pages 515-524, 2005.

[15] J. King. A 4-approximation algorithm for guarding 1.5-dimensional terrains. In Proc. 7th Latin American Sympos. on Theoretical Informatics, vol. 3887 of Lecture Notes Comput. Sci., pages 629-640, Springer-Verlag, 2006.

[16] B. J. Nilsson. Approximate guarding of monotone and rectilinear polygons. In Proc. 32nd Internat. Colloq. Automata Lang. Prog., vol. 3580 of Lecture Notes Comput. Sci., pages 1362-1373, Springer-Verlag, 2005.

[17] C. Worman and M. Keil. Polygon decomposition and the orthogonal art gallery problem. Internat. J. Comput. Geom. Appl., 2006.

[18] Couto, M. C., de Souza, C. C., and de Rezende, P. J. 2008. Experimental evaluation of an exact algorithm for the orthogonal art gallery problem. In Proceedings of the 7th International Conference on Experimental Algorithms (WEA'08). Springer-Verlag, Berlin, 101-113.

[19] G. Viglietta. Guarding and searching polyhedra. Ph.D. Thesis, University of Pisa, 2012.

[20] Kevin Suffern. Ray Tracing from the Ground Up. A K Peters, Ltd., 1 edition, 2007. ISBN 978-1-56881-272-4.

[21] T. T. Wong, W. S. Luk, and P. A. Heng, "Sampling with Hammersley and Halton points," Graphics tools: The jgt editors' choice, 2005.

[22] I. Millington. Artificial Intelligence for Games. Morgan Kaufmann, 2006.

[23] Eiben, A.E., Hinterding, R., Michalewicz, Z.: Parameter Control in Evolutionary Algorithms. IEEE Trans. Evol. Comput. 3(2), 124–141 (1999).