A Generic Stochastic Method to Arrange Virtual Creatures in a 3D-Scenario

Iván Piza, Félix Ramos, Fabiel Zúñiga
Cinvestav del IPN, Guadalajara, Jalisco, México
{hpiza, framos, fzuniga}@gdl.cinvestav.mx

Abstract

We propose a generic stochastic method to handle the space planning problem [Eastman73] inherent in declarative modeling [Donikian93] of virtual scenes. Genetic Algorithms [Holland75] is an adaptive and heuristic search algorithm based on the evolutionary ideas of natural selection; it is capable to find good solutions within a reasonable time. In this work, solution representation and fitness calculation are chief issues. The proposed method has been implemented as part of a framework useful to design virtual scenes [Piza04]. The aim of this work is to let users arrange virtual creatures in a 3D-world by using geometric constraints [Leroux00] (behind, between, close to, ...) instead of geometric data. Although the platform offers a fixed set of constraints currently, it provides means to let users add more, by sub-classing class Constraint and overriding its method sigma().

1. Introduction

Declarative modeling of virtual environments [Colin97, Donikian93 and Plemenos98] is based on the idea that humans conceive the world by studying properties instead of geometry. This paradigm gives big emphasis in specifying a number of constraints which make things have their appearance, doing without numerical data. The purpose of declarative modeling is to express the mental image of an object in a high-level language. As a result, the system builds a geometric model which fits in with the description.

1.1. Modeling with constraints

Unlike geometric modelers, declarative modelers deal with high-level abstraction concepts. Modeling with constraints is very common in declarative modeling. In fact, constraint formalism allows expressing complex design problems as constraint satisfaction problems (CSP) [Leroux00]. This chapter deals with a particular CSP: space planning using geometric constraints. A geometric constraint is a relation concerning one or more geometric parameters owned by one or more objects [Woodbury87]. In this research, the parameters are coordinates and the objects are virtual entities. [Charman95] introduces shape constraints (object shape) and spatial constraints (object position), and classifies the latter as follows:

- Topological constraints: “the sofa and the love seat overlaps”
- Distance constraints: “the boy is two feet from the dog”
- Angular constraints: “the window forms a 30 degrees angle from the wall”
- Orientation constraints: “ship1 is between ship2 and the main door”

This work deals with floor-planning, that is, object positioning in free spaces inside a one-level building. Therefore, spatial constraints are used.

Given a bounded planning 3D-space, a set of virtual entities, boundaries and user-defined constraints, the problem of interest in this work consists in finding a location in the space which best satisfies constraints. Such a problem is called space planning problem [Eastman73]. Seeing space planning as a constraint satisfaction problem allows the use of numerous techniques developed to solve CSP [Baykan91]. Stochastic methods represent a major approach to compute solutions. One of the most popular stochastic methods is Genetic Algorithms.

1.2. Genetic Algorithms (GA)

Genetic algorithms (GA) were formally introduced in the US in the 1970s by John Holland at University of Michigan [Holland75]. A Genetic Algorithm is an adaptive heuristic search algorithm based on the evolutionary ideas of natural selection [Darwin64] and genetics. [Reeves93] defines a heuristic as a technique which finds good solutions (quasi-optimal) with a
reasonable computational cost but with no promise of optimality or feasibility.

A typical GA creates a population of solutions—called individuals—and applies genetic operators to evolve the solutions in order to find the best one(s). Three major aspects to consider when implementing a problem-specific genetic algorithm are the following:

1. Definition of the objective function to evaluate each solution. This function is used to calculate the fitness of each individual. Greater fitness values denote nice solutions. Individuals with great values of fitness tend to preserve along generations.

2. Definition and implementation of the genetic representation. Every individual has a phenotype and a genotype. The phenotype stores the values used in the objective function. The genotype is represented by a chromosome. A chromosome is a chain of values or genes. Typically, a chromosome is in the form of a chain of bits or an integer array. A GA-implementation should include a method to translate genotypes into phenotypes.

3. Definition and implementation of the genetic operators: selection, crossover and mutation. Genetic Algorithms give big significance to crossover operator. These operators work with genotypes, instead of phenotypes. There exist several techniques for each of these operators.

Once these three aspects have been defined, the genetic algorithm should work fairly well.

A genetic algorithm consists of the following steps:

1. Create a random population of solutions.
2. Calculate fitness of each individual.
3. Repeat
   a. Perform Selection
   b. Perform Crossover
   c. Perform Mutation
   d. Translate genotypes into phenotypes
   e. Calculate fitness of each individual
   f. Increase generation index in 1 until best solution is good enough or generation index exceeds a given maximum.

2. System overview

The main goal of this research is to render an animated 3D–scene after a human-like description provided by expert or inexpert users. To accomplish this goal, we have proposed the architecture [Piza04] of a framework useful to create virtual scenes (see Figure 1). All the modules from this architecture are connected through a middleware, called GeDA-3D [Ramos02].

The Virtual Scene Creator comprises an interpreter of GOL, a goal-oriented language [Piza05], an interaction definition reader [Piza06] and a constraint solver. It has permanent communication with agents during the evolution of a scene.

GOL language provides a high-level mechanism for scenarists to set up their own virtual scenes where intelligent creatures interact to fulfill predefined goals. Four sections build up a GOL-description:

1) Declarations. Create and assign physical properties to virtual creatures (entities).

2) Behavior. Assign behaviors to creatures. A behavior is an agent connected to the system.

3) Arrangement. Places creatures in the world using geometric constraints [Leroux00]:
   Pencil1: on top of Table1 and in front of Pencil2

4) Sketch. Assigns goals to creatures using sentences of the form <entity, skill, target>.
   Target is present if skill involves moving the creature toward a place:
   Smith walks outside the kitchen
   Ship1 flies toward a place around two meters from Ship1

3. Design of a GA-based Constraint Solver

Figure 1. System Architecture
If scene-description is errors-free, GOL-Interpreter generates low-level commands to be sent to different modules of the system. This command-sending can be appreciated in Figure 2. Four kinds of commands are created:

a) Create-Entity. Lines read from Declarations section generate commands with the form:

\[
\text{Create-Entity(entity-name, avatar-name)}
\]

These commands are sent to the Rendering module.

b) Request-Agent. Lines read from Behaviors section generate commands with the form:

\[
\text{Request-Agent(entity-name, behavior-name)}
\]

These commands are sent to the Agents Control who runs an instance of a register agent-class called behavior-name; the running agent will be identified as entity-name; direct communication between the agent created and the Scene Creator is established.

c) Arrange-Entity. Lines read from Arrangement section generate commands:

\[
\text{Arrange-Agent(entity-name, 3D-position)}
\]

These commands are sent to Rendering module.

d) Assign-Goals. Lines read from the Sketch section generate the following command:

\[
\text{Assign-Goals(list of goals)}
\]

This command is sent to the Agents Control who redirects it to the appropriate agent. A goal is a 3-tuple \(<\text{character, skill, } R^3>\), where the third parameter is a target’s coordinate.

Figure 2. Commands sent from Scene Descriptor

Figure 3. Translating an Arrange-Entity Command

Commands Arrange-Entity and Assign-Goals involves to previously computing a 3D-coordinate that best satisfies all the constraints defined in the description. GOL-Interpreter has a module called Constraint Solver (CS) in charge of resolving 3D-coordinates from arrangement requests.

Figure 4. Translating an Assign-Goals Command

The main operation of Constraint Solver can be appreciated in Figures 3 and 4. In the case of an Assign-Goals command, Constraint Solver reads the goals-specification and, for each goal having a place-skill it replaces both constraint and reference data with a 3D-coordinate. Before this replacement every goal has the form: \(<\text{Entity, Skill, [Target]}\) and target has the form: [Constraint] (Entity | Zone).

Arranging ships in the scene:

\[
\Rightarrow \text{Ship3: in the Kitchen, on the left side of Ship1}
\]

\[
<\text{Ship3, [(in, Kitchen), (left, Ship1)]}>
\]
\[ CS \rightarrow \langle \text{Ship2, (97.25, 0.0, -32.5)} \rangle. \]

⇒ Ship2 flies toward a place in the Dining Room
\[ \langle \text{Ship2, flies, in, Dining Room} \rangle \rightarrow CS \rightarrow \langle \text{Ship2, flies, (375.0, 0.0, -51.75)} \rangle. \]

**Figure 5. Two-point crossover**

The implementation of Constraint Solver consists of a GA who computes the best position of a virtual creature, given a list of constraints, and a list of coordinates and dimensions of every constraint’s reference object. Currently, the dimensions of a virtual entity are given by a radius value, that is, the 3D-space occupied by an entity is defined by a surrounding sphere. The boundaries of a room are actually bounding boxes. The dimensions of a boundary are given by width, height and depth values.

Characteristics of the GA Implementation:

a) Genotype has *Binary Representation*. 16 bits for X-value, 16 bits for Z-value [Section 3.1].

b) *Roulette Selection*. Elitist: the best individual always preserves [See Figure 6].

c) *Two-Point Crossover* [See Figure 5].

d) *One-Point Uniform Mutation* [See Figure 7].

e) Fitness \( \in [0.0..1.0] \) is in terms of how close a solution is from the target. [Section 3.2].

f) Constraints are subclasses of abstract class Constraint overriding method \( \text{sigma():double} \). This method returns 0.0 if the constraint was fully satisfied and big values if the solution was far from being acceptable. [Section 3.2].

Notice in Figure 6 that orange individual (the best solution) passes to the next generation at the same position because of the elitism of the algorithm. Moreover, orange, blue, red and yellow individuals had great odds to be selected (even more than once) for being good solutions.

Crossover is performed between pairs of adjacent individuals \( (k, k+1) \), where \( k \) is an even index \( (0, 2 \ldots) \), and \( k, k+1 \neq \) best individual index. Crossover rate is set to 0.8. Mutation is performed only if \( k \neq \) best individual index. The bit to change is selected randomly. Mutation rate is set to 0.1.

**Figure 6. Roulette selection**

**Figure 7. Uniform mutation**

### 3.1. Solution representation

An Individual consists of the following:

a) Phenotype: 3D-Coordinate (where \( y = 0 \))

b) Genotype: a chain of 32 bits (an integer value)

c) Fitness: a double-precision number

d) Expected Value: a double-precision number (used for Roulette Selection)

The phenotype is given by the formulas:

\[
\text{Phenotype.X} = \text{(Genotype \& 0xFFFF0000)} \gg 16 / 16
\]

\[
\text{Phenotype.Z} = \text{(Genotype \& 0x0000FFFF)} - 215 / 16
\]

\[
\text{Phenotype.Y} = 0.0 \text{ [Currently, the Constraint Solver is 2D, only horizontal constraints are taken into consideration]}
\]

The following holds:

\[-2048.0 \leq \text{Phenotype.X, Phenotype.Z} \leq 2047.9375\]

Assume the following chromosome was randomly generated at first:

220
Genotype: 1110010111101000100000011101
Phenotype (-472.15, 0.0, 129.8125)

The resulting phenotype is calculated as follows:

\[
X = \text{Individual.Phenotype.X} \\
Z = \text{Individual.Phenotype.Z}
\]

3.2. Fitness evaluation

As said before, fitness is in terms of how far or close a solution is from the target. The target is constituted by one or more tuples having one of the following forms:

a) \(<\text{Entity}-\text{Constraint}, \text{Entity}>\)
   “Ship3: behind Ship1 and far from Ship2”
   <behind, Ship1>, <far-from, Ship2>

b) \(<\text{Entity}-\text{Constraint}, \text{Entity}, \text{Entity}>\)
   “Ship4: between Ship1 and Ship2”
   <between, Ship1, Ship2>

c) \(<\text{Absolute}-\text{Constraint}, \text{Number}, \text{Entity}>\)
   “Ship5: around 3.5 meters from Ship1 and at least 4.0 meters from Ship2”
   <around, 3.5, ship1>, <at-least, 4.0, ship2>

d) \(<\text{Inside}, \text{Room}>\)
   “Ship6: inside Kitchen”
   <in, Kitchen>

e) \(<\text{Room}-\text{Place}, \text{In-Tends}, \text{Room}>\)
   “Ship7: in the right near corner of Room2 and tends to the left wall corner of Room3”
   <right-near-corner, in, Room2>, <left-wall, tends-to, Room3>

The following algorithm calculates fitness of each individual. Notice that positions which collide with an object in the world are not completely discarded, since they may be close to the target desired. At the end of this algorithm, fitness has a value between 0 and 1.
Every constraint should be a subclass of abstract class `Constraint` and override method `sigma()`. That is, every constraint computes sigma in a particular way. The class-diagram from Figure 8 shows the inheritance relation between some constraint classes.

The following algorithms are implementations of abstract method `sigma()` in classes `InFront`, `Behind` and `InRoom`, respectively.

```java
Angle ← Relative Angle between Reference Position and Phenotype
Fitness ← Angle / 180
Denominator ← 180
If Angle is in [-45..45]
    Denominator ← Denominator × 2
If Angle is in [-15..15]
    Denominator ← Denominator × 2
Sigma = |relative angle| / Denominator

-------------------------------
Angle ← Relative Angle between Reference Position and Phenotype
Fitness ← Angle / 180
Denominator ← 180
If relative Angle in [45..135]
    Denominator ← Denominator × 2
If relative Angle in [75..105]
    Denominator ← Denominator × 2
Sigma = |relative angle| / Denominator
```

Notice that the following holds in the three algorithms above: 0.0 ≤ `sigma` ≤ 180.

Assigning big values of `sigma` result in small values of Fitness, but not necessarily 0.0. Using Genetic Algorithms, not-very-good solutions (0.2 < Fitness < 0.5) are not for sure discarded, because these solutions denote coordinates outside but close to the virtual zone desired, and they help the algorithm to converge by finding the best solutions.

3.3. Experimental Results

Figure 9 shows the GUI of the virtual scene creator introduced in Section 1. There, we can appreciate a scene description; it includes the arrangement of two virtual entities: `Ship1` and `Ship2`, where `Ship1` is an imperial ship and behaves as a prey. This scene occurs...
in a virtual world called RedBlue which includes two rooms: Red-Room and Blue-Room.

Figure 10 depicts a screenshot of the scene created in the description from Figure 9. Figure 11 depicts a topview of the same screenshot.

The Genetic Algorithm took less than two seconds to compute both coordinates.

4. Conclusions

In this paper, we have proposed the implementation of a stochastic method Genetic Algorithms, to handle the space planning problem inherent in declarative modeling of virtual scenes. The implementation of this method has been done inside a core module of our Virtual Scene Creator, the Constraint Solver.

An individual denotes a 3D-position in the world. The genetic algorithm uses binary representation, roulette selection, two-point crossover and uniform mutation. Our language (GOL) includes a number of constraints (on the left side, behind, far from,…), but more can be easily attached by extending super-class Constraint and overriding abstract method sigma(). Small values returned by this method denote good solutions: the position represented by an individual is quite close to the target desired.

We have tested the method by arranging virtual creatures (space ships) in a one-level virtual building, like the one depicted in Figure 10. Currently, our method works fine with horizontal constraints, where y-coordinate is always 0. The adaptation of vertical constraints, like over, on top of, below, and multi-level buildings is not a complex process. It suffices to modify the representation of the solution, increase number of generations and population size, and to implement the vertical constraints desired; no changes in the architecture and fitness calculation are required. Naturally, there will be penalization in performance, since the search space increases in one dimension. The response-time of the algorithm is not an essential issue here, since the arrangement of virtual creatures is performed before the scene is started.

5. References


