Imperial College of Science, Technology and Medicine

Automated Scene Generation

Term 1, 2008/09
ISO_1

Michael Nicolaou
michael.nicolaou08@imperial.ac.uk

Supervisor: Dr. Simon Colton

2008

Contents

Chapter 1

Introduction

Automatic Scene Generation (ASG) can be defined as the process of automatically formulating a scene, in order to serve a specific goal. For example, the goal can be the generation of an artistic scene. Visual art (VA) applications have emerged in the past decades, which attempt to generate novel artistic scenes by exploiting the possibilities offered by methodologies and technologies of computer science, such as constraint satisfaction and evolutionary computing. Other possible applications lie in generating scenes for architectural purposes, designing architectural plans or even generating scenes for video games or for virtual 3D environments.

In Fig. 1.1, the pipeline usually followed in applications that automatically generate scenes is presented. Firstly, the application receives some type of input, which can be explicitly supplied by the explicitly by the user for example in the form of desired relations between elements in the scene, by supplying characteristics required in the final scene or by defining general constraints that should hold. This kind of information can be received implicitly from the user, for example by attempting to derive his intentions for the outcome of the final scene by allowing him to form a partial scene. By defining the first step of the pipeline, we can relate this step to many fields such as human machine interaction, psychology and interface design.

The second step of the pipeline is to formulate the problem itself. The formulated problem will be depended on the application that is targeted by the system, and should incorporate the specifics of this application with the input that it has received during the first step of the pipeline. Generally, the problem the system has to deal with is the generation of a scene. The problem formulation can be, defining it as a constraint satisfaction problem, as an optimisation problem, as a problem to be solved by evolutionary algorithms or even as a geometric problem. In some cases, the scene can be generated in a procedural fashion (procedural modelling) for example by following rules defined in formal systems. In this case, the problem formulation step for these systems the problem formulation step can be defined as the actual description of the procedures and rules. Optional interaction by the user in this step is e.g. the user removing some of the constraints, limitations or specifications before the actual algorithm is executed.

After the problem is formulated by taking into account dynamic and static elements, the execution of the methodology adopted by the system takes place, in order to produce a solution which would internally represent the final generated scene. Practically, this can stand for the execution of algorithms which upon termination will reach a description of the output scene. It should be noted here, that a field of research that can be related to the automated scene generation is the problem of spatial planning or object layout, where given a set of objects, the system is required to decide their placements in some limited spaces,

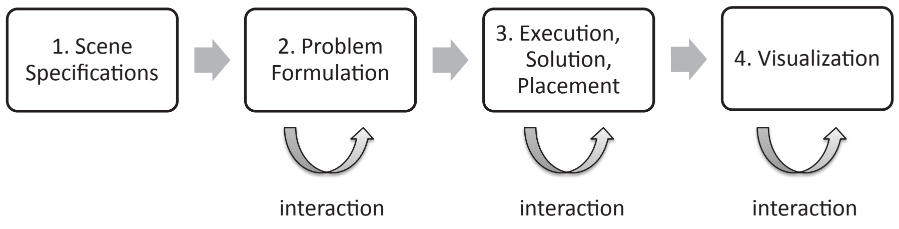


Figure 1.1: The Automated Scene Generation pipeline

taking into account obstacles and other constraints. This problem can be in turn related to many fields of research in computer science and mathematics such as computer graphics and geometry with problems such as dealing with collision avoidance or imitating rules of physics in order to find stable placements of objects in 2D or 3D spaces. The optional user interaction at this level are for example the refinement of some of the specifications if the final scene description is not satisfying, or even if a possible solution can not be reached.

The final step of the pipeline is the visualisation of the scene. The method exploited in this step depends again on the targeted application. Applications that target the creation of artistic scenes use methods such as non-photorealistic rendering, a computer graphics area which is inspired by artistic styles (e.g. painting with acrylics), in order to generate their final visualisation. Systems that are targeted in architectural 3D models or general 3D scene composition take advantage of modern 3D rendering techniques. This is a pure computer graphics area, since it is virtually the visualisation of scenes which exist internally in the system and given as input to a graphics system. User interaction at this level might give the user the possibility to change the rendering specifications or even rearrange objects in the final scene. The later step again relates to areas such as graphics and human machine interaction, and changes would lead to backward steps in the pipeline in order to reformulate the problem or even invoke some algorithms that would build on top of the original problem formulation, thus incorporating the dynamic changes.

It is noted, that throughout the survey we will base our descriptions on the first three steps of the pipeline: How and what kind of input the system receives, how the problem of the generation of the scene is formulated, and how that problem is solved.

Having described the typical steps an application in this field takes, we can infer that automated scene generation can relate to many other areas of research and exploit methodologies from many diverse fields of computer science, mathematics, physics, psychology and even art. The diversity that the field displays is one of reasons why we should clearly state the motivation behind this project as well as the goals that we target to achieve.

1.1 Motivation & Goals

The motivation behind this survey lies in recent research that has been done in the area of automated scene generation. Systems such as The Painting Fool [1], which produces artistic paintings and CityEngine [2], software which specialises in modelling 3D cities for video games and movies, have motivated us to survey the methodologies and techniques used in systems which belong to the broader field of generating scenes.

[1] www.thepaintingfool.com
[2] www.procedural.com

Also, we have been motivated by the research opportunity to integrate Abductive Logic Programming (ALP), a reasoning technique which can be used to generate explanations of observations in automated scene generation systems that base the final scene on the intentions of the user. In more detail, the goals of this project are the following:

- To provide a detailed description of systems that automatically generate artistic scenes for visual arts applications, referring to recent approaches to the problem as well as providing information on some older but innovative systems in the field.
- To describe systems that automatically generate architecture. The automatic generation may refer to producing actual 3D models of cities for use in computer games and movies, or even floor plans that can be used by architects. The systems will be compared based on the versatility, efficiency and weaknesses they have.
- To survey a selection of methodologies used in object placement systems, which confront the problem of placing objects in a usually limited scene which can contain obstacles. The selection will be based on covering characteristic examples from groups of methodologies, as well as on describing methods that do generate scenes automatically.
- To propose the extension of a Constraint Logic Programming automated scene generation system, by introducing a new type of constraint as well as introducing hierarchical constraints
- To refer to possible expansions and further research that can be done in the described fields.

1.2 Structure

The structure of the project is as follows:

- In Chapter 2 we describe some fundamental concepts on which the following chapters rely on
- The description of automated scene generation techniques for visual arts is presented in Chapter 3, while the description of techniques targeted to architecture are presented in Chapter 4.
- In Chapter 5, we present a selection of approaches to solving the spatial planning / object layout problem.
- Proposed expansions to a constraint logic programming automated scene generation system are presented in 6.
- Finally, in Chapter 7 we present a summary of the report, noting future tasks and research that can be based on the work presented in this project.

Chapter 2

Fundamental Concepts

In this section, we will briefly introduce some of the theoretical background required for the coming Chapters. In general, this chapter includes some theory on constraint satisfaction problems, genetic algorithms, the minkowski addition and decomposition as well as an introduction to shape grammars and L-systems. In more detail, in Section 2.1 we will present some simple concepts regarding constraint satisfaction problems, such as notions of satisfiability and consistency. In Section 2.2, we give a short description of evolutionary algorithms, describing genetic operators and various techniques, while in Section 2.3 we present the Minkowski addition and subtraction operators. In Section 2.4, we formally describe shape grammars, a special type of grammars which was formally defined for design. Finally, in Section 2.5, we present the L-systems formality, a system of rewriting rules which is well known for its application in flower and plant modelling.

2.1 Constraint Satisfaction Problems

A Constraint Satisfaction Problem (CSP) is a mathematical framework which can be formally defined as a triplet (X, D, C), where X is a set of variables, D is a set of domains and C, a set of constraints. Each domain d_i corresponds to a variable in X

$$X =< x_1, ..., x_i, ...x_n >, D =< d_1, ..., d_i, ...d_n >, i \in [1, n], x_i \in d_i$$

which means that the variable x_i is limited to the values that belong to the domain d_i. Each constraint $c_i \in C$, involves variables from the set X and imposes some limitations on them, thus defining the legal combinations of values that these variables may obtain in a problem solution instance. An assignment of values to some number of i variables is called *consistent* if it conforms with the set of constraints C, while it is called a *solution* to the CSP if $i = |X|$, that is when all the variables are assigned a value that conforms to the constraints. If the set of *solutions* to a problem is the empty set, then the problem is considered to be *unsatisfiable* or *inconsistent*. Furthermore, a CSP is considered to be *consistent* if at least one solution exists.

Often in related literature, constraints that are separated into the following categories: A *primitive* constraint, is a relation between some variables using a set of operators. For example, $X * 2 - 2 \leq Y$ is a primitive constraint. A conjunction of primitive constraints is simply called a constraint, for example $X * 2 - 2 \leq Y \wedge Y = Z$ is a constraint. A *global* constraint, is a constraint who has a pattern of a variable length and can be represented with a set of simpler constraints. The reason that a global constraint is preferred over a set of simpler constraints in most applications, is that this improves the performance of solving the problem, as we will see later in this section. The most typical example of a global constraint is the constraint *alldiff* which implies that all the assignments of values to variables in the *CSP* can constitute a solution only if no value is assigned to more than one variables, that is all variables have different values.

2.1.1 Local Consistency

An important notion that relates to CSP, is local consistency. It is called local, because it relates to subsets of variables and constraints of the entire CSP, thus narrowing down the definition of consistency we have defined for the entire CSP. These notions of local consistency can be enforced by certain transformations, which will transform (but not change) the problem into a simpler form, narrowing down the search space. These transformations which enforce local consistency are called *constraint propagation.* A popular constraint propagation algorithm is the AC-3 algorithm [71].

The simplest notion of local consistency is *node consistency* which ensures that any *unary constraint*, that is a constraint that relates only to a variable, is satisfied by all values in the domain of the variable. For example, if we have $X \geq 3$ and domain $D_x = \{3, 4\}$ then the domain is trimmed to $D_x = \{4\}$.

Arc consistency expands the previous definition to two variables. To consider the CSP as a graph, we can visualise the variables as the nodes of the graph and the constraints between two variables as arcs between the two nodes. The formal definition of arc consistency states that a variable x_i with domain D_i is consistent with another variable x_j with domain D_j if for every value $c_k \in D_i$ there exists a value $c_l \in D_j$ that satisfies the binary constraints between x_i and x_j. If all variables of a CSP are arc-consistent then the problem itself is arc-consistent.

A third notion of local consistency is path consistency. It expands arc-consistency in the following way: Assuming that we have a pair of two variables (x_i, x_j), then this pair is path-consistent with a variable x_l if for every consistent pair of values assigned to x_i and x_j, v_i and v_j, there is a unique value in the domain of x_l (v_l), which satisfies the binary constraints between x_i and x_l, and between x_j and x_l (e.g. the pairs (x_i, x_l) and (x_j, x_l) are legal values).

The notion of local consistency can be generalised to k-consistency as follows: [38] For any consistent assignment of values to $k-1$ variables, there exists a value for any kth variable such that all k values are consistent.

2.1.2 Constraint Logic Programming

Constraint Logic Programming (CLP) is a form of constraint programming which appears like a regular logic program, which uses special notation (constraint predicates) to impose constraints on some variables.

In a constraint logic program, the user must define a domain for each of the variables that will take part in the constraint solving. For example, *domain(*$[X, Y]$*, [1,2])*[1] limits the domain of variables X and Y to 1 or 2. Numerical constraints can be posed, using the special operators defined usually by including the # symbol. For example, X #< Y constraints the values of X to be strictly less than the values of Y. Integer equality can be posed as X #= $Y + 1$. An important advantage of some CLP libraries, is the addition of propositional constraints, for example X #=> Y, which is validated to true if Y is true or X is false. To attempt to solve the problem, a *labelling* predicate is called, which assigns values to variables.

[1] we will use the sicstus prolog fd library in this project

We have described the basic notions of CSPs, defined some formal concepts that we will require later on and also referred to the constraint logic programming extension of logic programming. For a detailed study on constraints, the reader can refer to [71, 70], while information on CLP can be found in the sicstus prolog user manual [2].

2.2 Evolutionary Algorithms

Evolutionary Algorithms are artificial intelligence algorithms which are inspired by biological evolution and make use of biological mechanisms to optimise a certain problem (meta-heuristic optimisation algorithms).

In evolutionary algorithms, a population of individuals is initially selected, sometimes in a completely random fashion. Each individual is a candidate solution to the optimisation problem. To evaluate the quality of the candidate solution, a *fitness function* is incorporated. After the evaluation, stronger individuals (i.e. with highest fitness [2]) of the population are reproduced using recombination operators (which combine features from parents and produce an offspring) or mutation operators (which transform a parent and produce an offspring), thus forming the next generation.

One of the evolutionary algorithm techniques are *genetic algorithms*, the most common method, which typically use strings to represent problem instances and optimise them. In more detail, the initial set of solutions (population) contains string representations of candidate solutions (chromosomes or genotype of genome). Other methods include *genetic programming* which solutions are programs and these programs are evaluated on their ability to solve problems, *evolutionary programming* which differs from genetic programming by evolving numerical parameters and not programs and finally *evolution strategy*, which represents solutions with vectors of real numbers.

We will describe some of the most common genetic operators used in the evolution of solutions. Assuming that we are using genetic algorithms, and the solution is a 5 bit string. Assuming the parents are 11001 and 10101.

One point crossover is selecting a position in the n-bit string and swapping the 2 parts generated for each parent to create the two children:

$$p1 : 11|001\ p2 : 10|101$$

$$c1 : 10|001\ c2 : 10|001$$

Two point crossover selects two positions and then swaps the bits between the two positions:

$$p1 : 11|00|1\ p2 : 10|10|1$$

$$c1 : 11|10|1\ c2 : 10|00|1$$

A mutation operator generates small random changes in parents in order to encourage diversity.

[2]based on the fittest principle

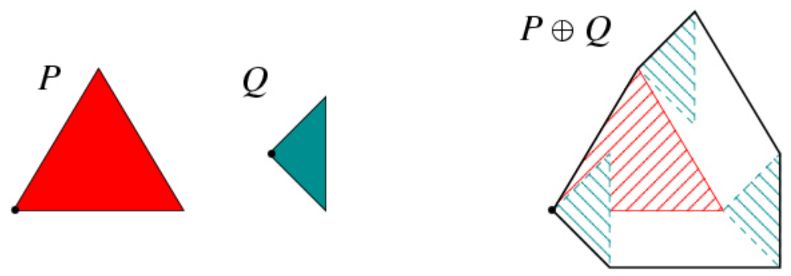


Figure 2.1: The figure shows the Minkowski addition of two convex 2D-simplices (triangles), following the definition [7]. The result is by a convex shape, as a result of the minkowski sum properties. The bullet marks the origin (0,0) for each shape.

In other techniques, these operators are adjusted in order to fit the problem representation. For example, in *genetic programming* where a tree representation is used, to implement crossover some nodes of the tree can be switched. The description we have presented is brief and intends to inform the reader of some very basic concepts, detailed descriptions of this area may be found in [71, 61, 5, 3].

2.3 Minkowski Addition and Decomposition

In this section, we will describe the Minkowski addition and decomposition, operations which play an important role in mathematical morphology and except research in geometry, have also found application in areas such as motion planning and path planning [66, 29].

The Minkowski addition operation is defined for two sets, B and T which belong to an r-dimensional space (R^d). The resulting set S is defined as follows:

$$S = A \oplus B = \{a + b : a \in A, b \in B\}$$

while the definition can also be posed as:

$$S = A \oplus B = \bigcup_{b \in B} A^b$$

where A^b stands for object A translated by b. An example of this operation can be seen in Fig. 2.1.

A simple example that is given in order to grasp the intuition behind the Minkowski addition, is to consider shape A as a brush, and imagine the brush following the trajectory defined by shape B, "thickening" the shape in this way. It is also true, that $S = A \oplus B = B \oplus A$. This is why the operation *grows* the shape.

The Minkowski Decomposition is the inverse of the Minkowski addition operation, and is defined as follows:

$$D = A \ominus B = \bigcap_{b \in B} A^{-b} = \bigcap_{-b \in \bar{B}} A^{-b}$$

where $\bar{B}$ is the symmetrical set of B with respect to the origin: $\bar{B} = \{-b : b \in B\}$. The operation is considered to *shrink* the object. Intuitively, the Minkowski decomposition is to find the $\bar{B}$ set, then place A at every point of $\bar(B)$ and their common intersection is the Minkowski decomposition. An example can be seen in Fig. 2.2.

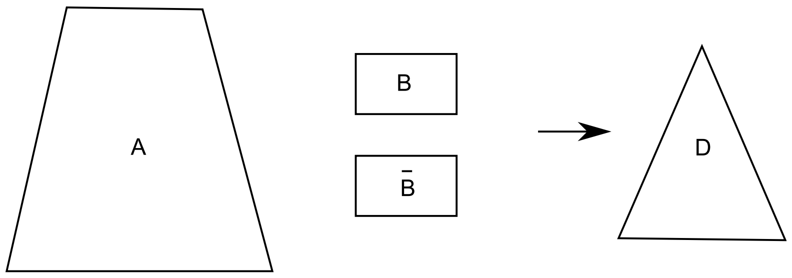


Figure 2.2: Minkowski Decomposition: $D = A \ominus B$

2.4 Shape Grammars

Shape grammars, which were inspired by the work of Noam Chomsky (phrase structure grammars [15]), were invented in 1971-72 by Georgy Stiny and James Gips [39]. A general definition of grammars (generative grammars) can be found in [15]: "a computational device able to generate all the grammatical sentences of a language and only those". In turn, shape grammars introduce a formality, with which the design process can unambiguously be represented. In a philosophical aspect, shape grammars can represent the *continuity of matter* [63], since at each step the evolution of the design can be observed, as well as the ability to split the final design into parts. In the original paper by Stiny and Gips [39] the shape grammars were introduced for *painting and sculpture* but as we will see in later sections, it has found application in architecture and design. In this section, we will formally define shape grammars, give examples of productions and also describe the way rules are selected in a shape grammar production.

2.4.1 Formal Definition

In [39], a shape grammar is formally defined as group of 4 sets: $SG = (V_T, V_M, R, I)$, where:

1. V_T is a finite set of shapes
2. V_M is a finite set of shapes that $V_T^* \cap V_M = \phi$
3. R is a finite set of ordered pairs (u,v) such that u is a shape consisting of an element of V_T^* combined with an element of V_M and v is a shape consisting of (a) the element of V_T^* contained in u or (b) the element of V_T^* contained in u combined with an element of V_M, (c) the element of V_T^* contained in u combined with an additional element of V_T^* and an element of V_M. These cases define the three possible types of rules, $r : u \rightarrow v$.
4. I is a shape consisting of elements of V_T^* and V_M.

In a similar way to context free grammars (CFG) [51], elements in V_T^* which appear in some (u,v) pair of R are considered as *terminal shape elements* or *terminals*. Elements in the V_M set are called *non-terminals* or *markers*, while elements of R stand for the (production) rules of the system. The * symbol in V_T^* stands for the *Kleene Star* or *Kleene Closure* [51], and in the context of shape grammars it stands for a finite arrangement of elements of V_T, which appear in the arrangement any number of times with any scale or orientation. The *production* of a shape is quite similar to the production of *strings* in CFG and proceeds as follows:

1. Find part of left hand side (LHS) of rule $r \in R$ which is similar to the current shape, considering both terminals & non-terminals.

2. Find the geometric transformations[3] that make the LHS rule identical to the similar part in the shape.
3. Apply the transformations to the RHS of rule r and replace in the shape

This production method is very similar to the production of strings in CFG, excluding the geometric transformation parts. Of course, once a terminal shape has been placed in the shape, it can not then be replaced since for any terminal $t \in V_T^*$, the available rule is $t \rightarrow t$ (rule type (a)). The language $L(SG)$ of a shape grammar SG is defined as the set of all shapes that are produced and include only terminals. A noticeable difference between shape grammars and CFG, is that shape grammars include both terminals and non-terminals on the LHS of a rule, while rules for context free grammars consist only of non-terminals on their LHS. This is because shape grammars have the option of being context-sensitive, that is a production rule can consider not only a non-terminal but also its neighbours.

2.4.2 Shapes in Shape Grammars

Since shape grammars are dealing with shapes, it is useful at this point to formalise the concept of shapes in shape grammars. The rules of shape grammars operate on points, lines, planes and solids or any combination of the previous. In fact, Stiny [82] has introduced formalised ways to unambiguously represent shapes which in turn fall into specific sets of shape algebras. This does not only offer an unambiguous way to represent a shape but also a way to symbolically describe the shape itself.

They can also be linked to labels, which can carry some information on the shapes themselves, for example weights or information which relates to the production process. For example, a shape might have a label which points the position where a new rule could be applied. An example can be seen in Fig. 2.3.

2.4.3 Selection Rules

Selection rules provide a way to select the shapes produced, since the shapes produced by a shape grammar can be infinitely many. The rules assign *levels* to terminals during the generation of shapes, while more than one level can be assigned. The rules are as follows:

1. Initialise terminals in initial shape to zero
2. Assuming that shape rule r is applied, $r_{lhs} \rightarrow r_{rhs}$. Assume that $N = max_level(r_{lhs})$, the highest level assigned on the LHS of the rule r.
   (a) If *rule* type $= (a)$, any part of terminal enclosed by marker (non-terminal) in the r_{lhs} is assigned the level N. The marker must be a closed shape.
   (b) If *rule* type $= (b)$, the previous assignment is executed and also any part of the terminal enclosed by the marker in the r_{rhs} is assigned a level of $N + 1$. The marker must be a closed shape.
   (c) If *rule* type $= (c)$, the terminal added is assigned the *level* $N + 1$.

After *levels* are assigned to terminals, *selection rules* can be applied, in order to filter the levels that are required for a shape to have, in order to belong to a specific class of shapes. A *selection rule* has the form of a tuple (*min*,*max*) where *min* is the minimum *level* allowed, and *max* the maximum.

[3]In the original paper, the allowed Euclidean transformations mentioned were scale, translate, rotate, mirror - along with their combinations

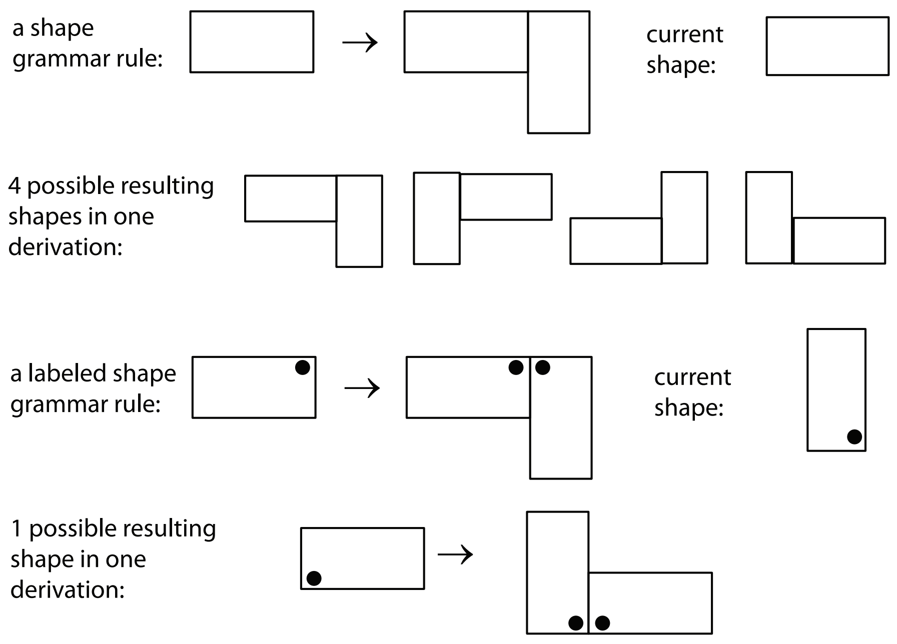

a shape grammar rule:
current shape:
4 possible resulting shapes in one derivation:
a labeled shape grammar rule:
current shape:
1 possible resulting shape in one derivation:

Figure 2.3: Example of a shape grammar rule. A simple shape grammar rule can produce four different shapes on application, since the original shape can match the rule shape in four different ways, by applying rotations by 0,90,180 and 270 degrees. Using a labelled rule constraints the possible output shapes to one, determining the derivation unambiguously.

2.5 L-systems

An L-system (Lindenmayer system, parallel rewriting system) is a variant of a formal grammar which has been widely used in modelling the growth and development of plants [68]. Also with L-systems, it is quite natural to model fractal structures - that is structures which are recursively self-similar.
An L-system is formally defined as a 4-tuple $G = V, S, \omega, P$, where:

- V is a set of symbols called variables, which can be replaced during the production.
- S is a set of symbols called constants, that remain fixed
- ω is a string of variables (symbols in V) that define the start state. It is often called *initiator, start* or *axiom.*
- P is a set of production rules r that define how variables are replaced by combinations of variables ad constants.

Comparing L-systems to regular production grammars, such as context free grammars, we can relate $constants$ and $variables$ to $terminals$ and $non-terminals$, as they do have the same role in the system. The characteristic that discriminates L-systems is that *all* applicable rules are applied simultaneously, per iteration. Symbols in L-systems can refer to drawing commands, and in this way the resulting string of an L-system can be visualised. Most applications that visualise L-systems, use *turtle graphics* to produce images, while following the drawing interpretation of symbols. Turtle graphics work in a quite simple way: a turtle is located at a point on the screen when the application initiates, usually facing north. Then, follows the drawing commands (e.g. move forward 10 pixels, turn left 90 degrees) and while moving to conform with the commands, the actual drawing occurs as well. For example, after following the turn left by 90 degrees command, the turtle will be facing west.

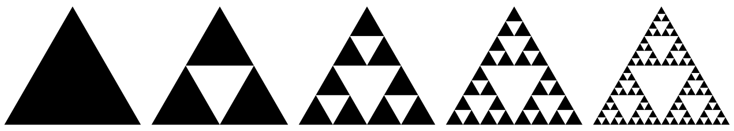


Figure 2.4: Steps in the recursive algorithm for constructing a Sierpinsky triangle

Common variations of the system described above, which are described in detail in [68], are:

- DOL systems: A DOL system is a deterministic context-free L-system. An L-system is considered to be context-free when each production rule refers to an individual symbol and not to its neighbours. Also, it is considered to be deterministic if there exists exactly one production rule for each symbol.
- Stochastic L-systems: Stochastic L-systems are systems in which when more then one rules exist for a given symbol, then each one of these rules are selected with a given probability.

2.5.1 L-Systems & Fractals: The Sierpinsky Triangle

To give an example of an L-system's actual production and also display the natural ability of L-systems to capture the structure of fractals, we will present an example of an L-system which produces a very well known fractal: the Sierpinsky Triangle. The algorithm for creating a Sierpinsky triangle is, starting with an equilateral triangle, create three triangles with half the width - length of the original triangle and fit them in each of the corners of the initial triangle (Fig. 2.4).

An equivalent L-system is the following:

$$\begin{aligned} V &= \{A, B\} \\ S &= \{-, +\} \\ \omega &= \{A\} \\ P &= \{(A \rightarrow B - A - B), (B \rightarrow A + B + A)\} \\ angle &= 60\ deg \end{aligned}$$

Where A and B stand for *draw forward*, while $-$ and $+$ stand for turn right by *angle* and turn left by *angle* respectively. In Fig. 2.5, we can see 2, 4, 6 and 9 steps of the derivation of the above mentioned system.

2.5.2 Parametric, Environmentally Sensitive & Open L-Systems

In [67], Prusinkiewicz et al. present an extended version of L-systems, which takes into account the *environment* that exists while the drawing procedure is taking place and also the current position or orientation that the drawing turtle is at. To offer this kind of functionality, environmentally sensitive L-systems use *query modules*, which are executed when a rule is used, and the results are returned as parameters to the *query module.*

We will now present the notation followed in [67], which is slightly different to the general grammar-style notation we introduced in Sec. 2.5. The approach described is focusing on parametric L-systems, which focus on *modules* instead of just symbols. *Modules* consist of a *symbolic name* and a number of associated

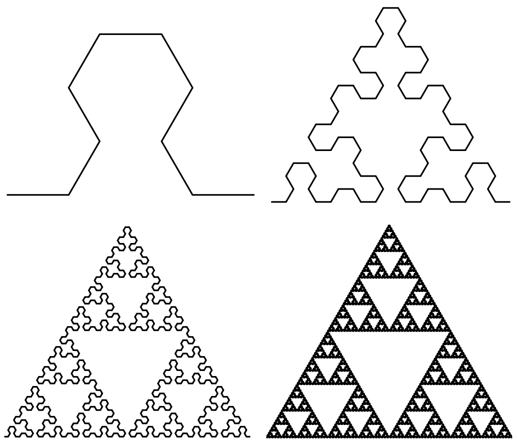


Figure 2.5: Steps in the production of a Sierpinsky triangle with an L-system

parameters, something similar to a programming *function* or *method*, which can then be evaluated and called, in order to interact with the *parameters*. The production format as presented in [67] is:

$$id : lc < pred > rc : cond \rightarrow succ : prob$$

where *id* is the production *id* - or the production *label*. The left hand side rule is composed out of $lc, pred$ and rc where $pred$ is the strict predecessor and lc, lr are modules. The previous description is for context-sensitive parametric L-systems, and for the rule to match, it is required that module $pred$ is preceded by lc and followed by lr. On the right hand side, we have the *cond*, which is the condition to check, *succ* is the successor and *prob* is the probability of applying the rule, in the context of *stochastic* L-systems. We proceed with an example of such systems, presented in [67]:

$$\begin{aligned} \omega :& A(1)B(3)A(5) \\ p1 :& A(x) \rightarrow A(x+1) : 0.4 \\ p2 :& A(x) \rightarrow B(x-1) : 0.6 \\ p3 :& A(x) < B(y) > A(z) : y < 4 \rightarrow B(x+z)[A(y)] \end{aligned}$$

This system, initiates with three modules, as seen in the first line (ω). The first two productions, state that $A(x)$ can be replaced with either $A(x+1)$ or $B(x-1)$ with probability 0.4 or 0.6 respectively. The third production ($p3$), is a context sensitive production. It denotes that for the rule to apply, the $B(y)$ module should be preceded by $A(x)$ and followed by $A(z)$. Furthermore, the rule $y < 4$ is used as a guard, and must be valid for the rule to apply. The rule replaces the $B(y)$ module with the $B(x+z)$ module, and adds the supporting branch $A(y)$ with the bracket notation. This bracket notation refers to the work of Lindenmayer in [53], where the notion of bracketed strings in order to represent branching in certain organisms is introduced. A left bracket represents the node of a parent branch, where the child branch will be attached. This process is terminated by the right bracket, forming in this way a recursive definition of trees.

Environmentally sensitive L-systems, provide functionality for the system to query about the state of the environment, e.g. checking if colliding with objects or finding out the orientation of the turtle (or the imagined turtle if the system is still on theoretical base) used for the drawing procedure. During a production step, the query modules bear parameters that remain undefined and are assigned values during interpretation. Syntactically, these query modules are defined as:

$$?X(u_i, ..., u_n)$$

where, as originally defined X can be P, H, U or L and there are three arguments, x, y, z so $n = 3$ and x, y, z stand for a 3D vector or 3D point. P stands for position, H, U, L stand for the orientation vectors in a 3D coordinate system. Examples and methodology are described in depth in [67], while any further description is outside the scope of this survey.

Another extension of L-systems, are *open* L-systems [59], which further extend parametric environmentally sensitive L-systems: they augment the previously described functionality, by introducing bilateral communication with the environment. The parameters can be set by the organism/system and received by the environment, or set by the environment and received by the system. The *query modules* as used in environmentally sensitive L-systems have the same syntax and are now called *communication modules*, since they are used both to send and receive values.

2.6 Discussion

In this chapter, we have described some prerequisite knowledge, which is required for the understanding of the following chapters. Having described the basic notions of CSPs, shape grammars, L-systems, evolutionary algorithms and Minkowski operations, we will now describe applications to scene generation in fields such as visual arts and architecture that exploit these concepts.

Chapter 3

Automated Scene Generation in the Visual Arts

In this chapter we will present methodologies for automated scene generation which have been designed for visual arts applications. We will refer to the AARON [20] system designed by Harold Cohen, which is reported to be the first actual software application that generates original artistic images [90]. We will then refer to The Painting Fool project, and cover two methodologies in ASG for visual arts applications: A machine learning and evolutionary approach, which uses fitness functions and correlations to construct a scene, and a constraint based approach which extracts constraints from a partial scene, forms a CSP and uses the solution set of the CSP to construct a scene. This chapter's sections are as follows:

In Section 3.1, we will present an overview of the AARON system, while trying to compare the methods used in the system to modern approaches in computer graphics and modelling. We will cover information relating the way knowledge is represented in the system, how the system uses this information to produce scenes and finally how these scenes are coloured.

In Section 3.2, we describe the machine learning approach to automated scene generation for visual arts. More specifically, we will describe how a fitness function can be formed based on the desired relations that the scene elements of the final scene should adopt. Experiments in attempting to optimise the fitness functions are described, while an approach to generating fitness functions is presented.

Continuing in Section 3.3, we present the constraint based approach to generating visual art scenes. We will describe a system that can infer intentions of the user by extracting information from a partially completed scene, forming a CSP and then attempting to solve that CSP in order to build the final scene.

Finally, in Section 3.4 we will give a brief description of some evolutionary approaches to visual arts.

3.1 AARON

AARON is a software program that generates original artistic images, designed by the artist Harold Cohen. The timeline of the application begins in 1973. Since then, the application has evolved from just drawing rocks to drawing plants and finally to drawing people. Initially, the system was very simplistic and it was not until the early 1980s when a primitive sense of perspective was introduced, while in 1989 a completely 3D knowledge base was used in the drawing process. During the 1990s, interior and complex scenes were produced, while Cohen begun experimenting with colour in 2000. In this section, we will attempt to review the system, as described in [20, 56]. It is noted though, that no actual technical description of the source code and methodology used in the system is available, only information in the form of essays. Although the source code of AARON has not been publicly released, it is known that the application was originally written in the C programming language, and then completely rewritten in Lisp.

3.1.1 How Knowledge is Represented in AARON

As Cohen says, "AARON can make paintings of anything it knows about, but it actually knows about very little – people, potted plants and trees, simple objects like boxes and tables, decoration" [21]. The knowledge is presented to AARON in a procedural way, for example by using a lot of if then else statements to determine the posture of a human figure to be drawn. An example of such a fragment follows:

```
...
if (left-arm-posture is "hand-on-hip")
    (add-upper-arm left -.3 .5 .65)
else
if (left-arm-posture is "arms-folded")
...
```

To be precise, the AARON painter has two basic knowledge kinds, which it in turn uses to produce a painting [19]:

- The knowledge about external-world objects, which is declarative.
- Knowledge about the representation building process, which is internal and procedural.

To give an example of the internal knowledge representation in AARON, we will present the knowledge that AARON maintains about human figures. The AARON system considered the human figure in a hierarchical way, a way that is commonly used in computer graphics and skeletal animation. The internal knowledge of the system held each part of the body, e.g. the head, as an array of points, with its origin at the articulation of the next part, for example the head to the neck, the hand to the forearm. This is the same way that bones are organised in skeletal animation in graphics, allowing the rotation of the following part in terms of the previous part (e.g. rotating the forearm around the arm) and thus creating believable animation. An example of a way the system maintains this information is presented in [17]: "Tree" expands to "big-tree/small-tree/shrub/grass/flower", "big tree" is expanded into "oak/willow/avocado/wideleaf" until a *leaf* node is found, that is a node than can not be further expanded. We can relate this tree form to the parse tree of a simple grammar[1], where the *leaf* nodes stand for a terminal symbol in the grammar.

3.1.2 Drawing with AARON

Cohen, based on his observation of the way children draw images (starting from scribbles and then expanding them to capture enclosing forms), devised the way that AARON would create drawings. In particular, after selecting from a predefined (and rather narrow) range of poses, AARON would form the enclosure of the figure, with the nose defining the direction of the head. The system, knew from its procedural knowledge (if then else statements) how to draw a figure in relation to the posture presented. This was all the knowledge the system maintained about perspective. This approach produced certain limitations to the paintings actually produced. As posed by the author, "AARON had no way of placing an arm across the body". In 1985-86, the version of AARON implemented incorporated a full 3D knowledge

[1]we will introduce grammars specific to drawing shapes in Section 2.4 of this survey, called *shape grammars*

base. The system then worked in two phases. The first phase incorporated the knowledge of the system in the 3D world aspect, and the other phase included the rules that actually generated the projection of the 3D imagined scene in 2D. The equivalent method in computer graphics is called *perspective projection* of a 3D set of points, although the AARON approach was more hardcoded in the system itself rather than applying a generic mathematical approach to a 3D scene.

As we already mentioned, the system maintained sets of points that corresponded to each part of the body as a conceptual representation of the part. Cohen faced two problems, relating to the occlusion of objects: Firstly two parts of an object's body might partially overlap each other, for example the hand of a figure might overlap with the other hand and vice-versa. Secondly, Cohen states that a part of the drawing could not have actual endpoints which could be found in a simple way before they were actually drawn in 3D [19]. Cohen found no other way to impose rules of occlusion and solve these two problems than implement a set of procedural *if-else* rules that determined the occlusion and the depth, in other words in what order the drawing should be drawn. A sample of the pseudocode, which is presented in [19] follows:

```
if  left wrist  closer (in 3-space) than the left elbow,
        and  left elbow is closer (in 3-space) than  left shoulder,
        and  left wrist is to die right (in 2-space) of  left shoulder,
        and  left wrist is not higher (in 2-space) than  right shoulder:
then left arm will obviously obscure the torso, and will have to be drawn
        before the torso is drawn
```

Following this fragment of code we deduce that the *closer* conditions refer to the distance from the view point of an observer. The first two rules, succeed if the left arm is extended (e.g. wrist in front of elbow in front of shoulder). The third rule succeeds if the wrist of the extended arm is towards the right of the left shoulder and the fourth that it is not higher than the right shoulder and thus obscures the torso - is in front of the torso, and has a lower depth in the image, thus should be painted first.

What we can infer from the example code fragment, is that the difficulties arising in occlusion are because a simple version of the Painter's algorithm [36] is applied in this system. The Painter's algorithm is a computer graphics algorithm which attempts to solve the geometric *Visibility* problem (in this case, which points or objects are visible from another given point). The algorithm assigns depths to the polygons in the scene and then draws the most distant ones, progressing to the closest ones. This is exactly what is implemented here and problems arising in this algorithm also occur in the AARON program. There exist generic algorithms that can determine what happens in cases where problems such as the ones mentioned occur, for example, when two polygons are occluding each-other, there are algorithms that cut pieces of each polygon and determine how to draw them. Various problems with this algorithm led to the development of algorithms that decide on a pixel-by-pixel basis, assigning a depth value to each pixel, which is stored in a *depth* buffer. A well known algorithm that implements this, is the *z-buffer* [12] algorithm.

3.1.3 Colouring in AARON

One of the last problems addressed in the AARON system, was the selection of colours [18]. During the selection, Cohen decided to base the procedure on the brightness of a colour, since "the eye is used predominantly as a brightness discriminatory" [18]. This choice can be supported by the fact that brightness alone when selected properly can produce a proper black and white scene. The colour model that the system was based on is the Hue, Saturation and Lightness colour model (HLS). This was because

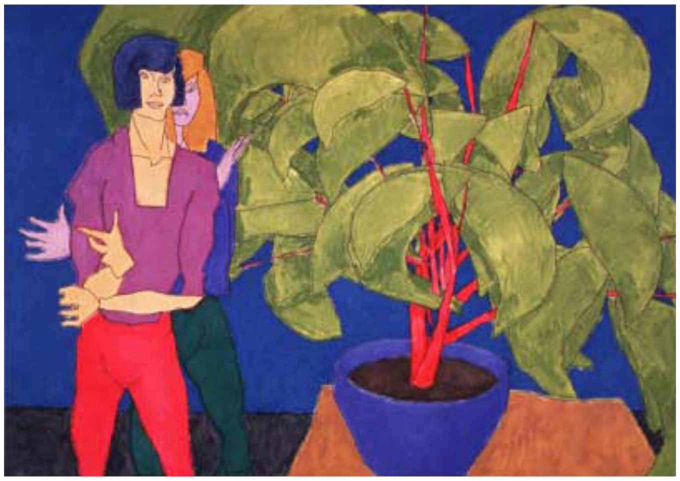


Figure 3.1: A painting produced by the AARON system

both RGB (cathode ray monitors) and CMY (printer and plotter) colour models, which use additive and subtractive [2] mixing respectively, do produce the same hue. This was important, because Cohen wanted the mixing of colours that appeared on screen, also to appear on paper when the final scene was put on paper. More on colour models and colour can be found in [93].

Implementation-wise, Cohen produced around 12 000 samples, narrowed them down to 800 samples and then distributed them arbitrary to 180 around a circle. Cohen then sorted them in decreasing saturation (purity). Then, these samples were coded into the system and were available as instructions to it. Cohen experimented with the way colours were selected in the system, by selecting from these samples and matching the brightness required from the scene specifications to the brightness of the samples available.

In summary, the goal of this system is to represent a narrow and specific range of objects and place them in a predefined way in scenes. The variety of scenes produced by the system are very limited, and knowledge is in a way hardcoded into the system. In the next two sections, we will describe more versatile systems where this kind of hardcoded information does not exist but the information for the constructed scene is derived based on the intentions of the user.

3.2 Machine Learning Approach

In the system described by Colton in [23], a method is discussed for creating a novel scene from a high level description input of the scene, which can be provided for by the user or even extracted from a picture. This high level description provides a general specification to the system, which in turn constructs a fitness function, turning the problem of generating the final scene into a search problem. In addition, a method is proposed for inventing fitness functions by forming a theory and then optimising that fitness function using evolutionary algorithms in order to produce the final scene.

[2] subtractive mixing can be thought of as starting with white light and then subtracting wavelengths of colours, while additive mixing can be considered as starting with black light and then adding light sources with colours

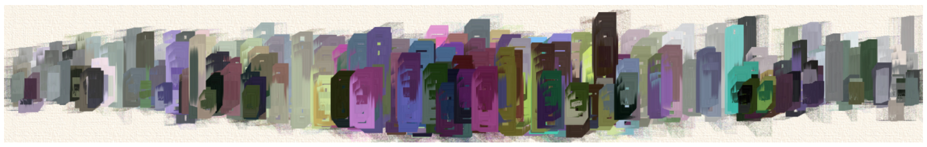


Figure 3.2: Cityscape scene rendered with acrylic paints [23].

3.2.1 Problem Specification

As input to the system, a high level description of the scene is provided to the application, while the scene is considered as an ordered list or scene elements, $L_{scene} = el_1, el_2, ..., el_k$. The description consists of miscellaneous scene elements, as the dimensions of the scene and the number of elements that the final scene should contain. A set of numerical attributes and their range is supplied to the system, in this way specifying the scene elements (e.g. y-coordinate is of range 0 to 300, x-coordinate is of range 150 to 250), while other attributes that can be calculated at run time can be provided. Since the system will eventually render the scene, a rendering specification is also provided. The essence of the system though, lies in the supply of a set of desired correlations, which the attributes that have been specified should conform to. We will discuss the notion of correlation in the following section.

3.2.2 Correlation Coefficient

This method uses the *correlation coefficient* to determine the relation between attributes of elements if the scene. In simple words, the correlation coefficient will be 1 if the variables are positively correlated and thus they both increase or they both decrease, can be -1 when the variables are negatively correlated (the increase of one of the variables will cause the decrease of the other and vice-versa). The coefficient can be 0 if the variables are not correlated, for example if the increase of the one is accompanied sometimes by a decrease and sometimes by an increase of the other.

$$\rho_{X,Y} = \frac{\mathrm{cov}(X,Y)}{\sigma_X \sigma_Y} = \frac{E((X-\mu_X)(Y-\mu_Y))}{\sigma_X \sigma_Y},$$

In the above equation, we can see the *Pearson product moment correlation coefficient* which is used in this paper to supply the correlation coefficient. The X and Y stand for two certain scene elements, σ_X and σ_Y stand for the standard deviations of the corresponding elements, μ_X and μ_Y stand for the expected values of the elements, E is the expected value operator and $\mathrm{cov}(X,Y)$ represents the covariance of the two variables. The covariance can be a measure of how much the two variables are changing together, in respect to their mean value.

To fully comprehend how this system works, a simple example is when we want to produce a particle scene, where the particles are travelling from left to right and loosing energy while they travel. We would then associate the x coordinate with the brightness or/and the saturation of each *scene element* with a -1 value. In this way, the particles who have greater x coordinate will be less bright than the ones with less x coordinate. An example presented in [23], is a cityscape scene with a correlation of 1 between the edge distance (minimum distance to the left or right edge) of the scene and the width, height, y-coordinate and saturation of the element. Also, the depth of an element (number of elements earlier which also overlap with the element) has a correlation of 1 with saturation, -1 with the height and -0.5 with the brightness. These correlations are targeted to represent distance by modifying size, saturation and brightness. An evolved scene is presented in Fig. 3.2.

3.2.3 Searching the Scene Solution Space

In order to perform a search and assign values to the attributes of a scene, a fitness function is required to evaluate the attributes in relation to the correlations required, considering the weight of each correlation. The function proposed in [23] is:

$$f(S) = \sum_{i}^{n} w_i \left(1 - \frac{|r_i - p(u_i, v_i)|}{max\{|1 - r_i|, |-1 - r_i|\}} \right)$$

where n is the number of correlations between variables defined by the user, with the i-th correlation described as a quadruple $< u_i, v_i, r_i, w_i >$. We consider $p(u_i, v_i)$ to be the result of the Pearson product-moment calculation which we compare against the required value for this correlation (r_i), while the w_i specifies the weight for each correlation. It is assumed that the sum of all weights should be 1. Taken the previous into consideration, the fitness function increases between 0 and 1 as the target correlation is satisfied the scene, and sums to 1 if the scene is perfectly correlated.

Algorithm 1 Evolutionary approach to Scene Generation

```
Construct random population of scenes $P$
while Terminating criteria not met do
    for $S_i \in P$ do
        calculate $f(S_i)$
    end for
    $avg_f \leftarrow \sum_i^{size(P)} f(S_i)/size(P)$
    for $S_i \in P$ do
        $e(S) \leftarrow f(S_i)/avg_f$
    end for
    Place $\lfloor e(S) \rfloor$ copies of $S$ in intermediate population $I$
    Add another copy of $S$ with probability $r \leftarrow e(S) - \lfloor e(S) \rfloor$
    while size($newPopulation$) ¡ size($requiredPopulation$) do
        Create offspring scene $off$ from parents $\in I$, using crossover
        Mutate $off$ depending on the mutation rate
        add $off$ to $newPopulation$
    end while
    $P \leftarrow newPopulation$
end while
```

The evolutionary approach described is summarised in Algorithm 1. We just note that the random population P is constructed by randomly selecting element attributes within user specified range, while the total number of elements fits the number provided by the user (n). Also, the crossover method is implemented by swapping continuous blocks of scene elements from the parents ordered list, while mutation occurs by replacing a scene element with a new one. In the same paper, we can see a hill climbing method, where a scene is randomly generated and altered at each step, by changing the numeric value of an attribute for a specified number of repetitions. Also in the same paper, a hybrid approach to the problem is proposed, which initially runs the evolutionary algorithm and uses the best population produced as the starting point of hill climbing search. The later approach appears to perform better.

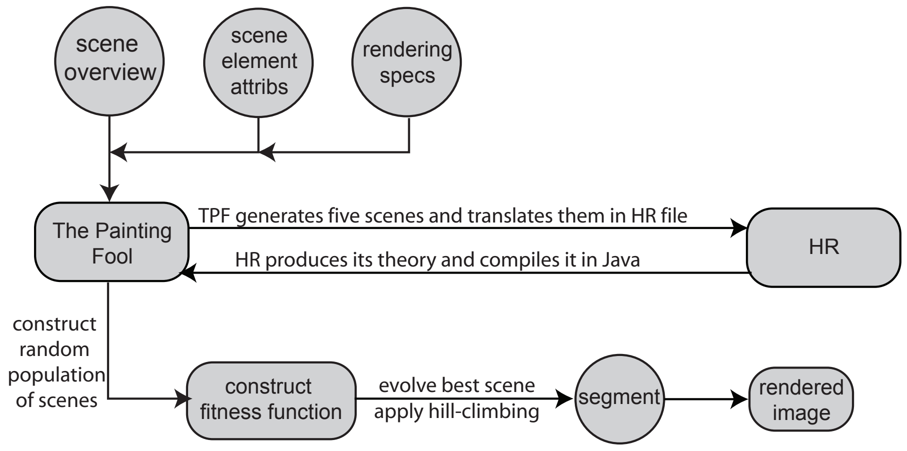

scene overview
scene element attribs
rendering specs
The Painting Fool
TPF generates five scenes and translates them in HR file
HR produces its theory and compiles it in Java
HR
construct random population of scenes
construct fitness function
evolve best scene
apply hill-climbing
segment
rendered image

Figure 3.3: Interaction between the HR system and TPF

3.2.4 Automatic Invention of Fitness Functions with the HR system

In this section we will give a brief description of the HR system, and describe how it is exploited in order to invent fitness functions, which would be then used for producing scenes.

3.2.4.1 The HR System

The HR system [22], which is named after the mathematicians Srinivasa Ramanujan and G. H. Hardy[3], is a system that can formulate mathematical theories from basic concepts of a domain, for example addition and multiplication for the field of number theory [25]. The HR system is supplied with initial concepts (e.g. integers), objects of interest, ways to decompose the previous objects into sub-objects and relations between the sub-objects. The HR system uses production rules in order to combine old concepts into new ones. For example, the compose production rule produces a concept by *identifying objects or sub-objects with properties from 2 concepts*, the forall concept identifies objects for which all sub-objects share a common property and so on. An example presented in [25, 22] where a detailed discussion of the system can be found, is the following: The HR system is given the concept of divisors, having integers as objects of interest. It uses the size rule to determine how many divisors (sub-objects) an integer has and then uses the split rule, to keep the values which resulted from the previous rule and are equal to 2. In this way, the HR system reinvents the *prime numbers*. The system uses *heuristics* to build new concepts and uses *metrics* to evaluate how important they may be, by assessing their *complexity* and *novelty*.

3.2.4.2 The HR System & The Painting Fool

In the case of Automated Scene Generation and The Painting Fool, as described in [23], the objects of given to the HR system are the scenes themselves and in essence, the input consists of the attributes defined by the user in the first step. In addition, the HR system was enriched with the *correlation rule*, which was able to produce the correlation concept between two other concepts, and the *float-summary* production rule which produced useful details about a floating point concept (minimum maximum and average value, smallest difference between the values and range of values).

[3]These two mathematicians are linked in an interesting taxi story

In the experiments described in [23], given the scene specifications, the element attributes and the specifications on rendering, five random scenes are created randomly. Then, they are translated into input for the HR system, which actually invents correlation concepts. Then, the hybrid evolutionary / hill-climbing approach is used, with an invented fitness function which includes correlation concepts produced by the HR system. The chosen scene(s) are then rendered. The interactive process between TPF and the HR system is presented in Fig. 3.3.

3.3 Constraint Based Approach

In this section, we will describe a constraint based approach presented by Simon Colton [24]. The system, introduces a novel approach in visual arts applications, which attempts to understand the intentions of the user from a partially drawn scene, express them as constraints and then attempt to complete the scene based on the solution set of the constraints.

This method addresses some issues from [23], like giving the user an interface easier to work with instead of defining the correlations and also being able to guarantee the satisfaction of certain constraints. That was not the case in the fitness function approach, since the fitness function was not always maximised (i.e. satisfying all correlations) [4]. There is also a substantial amount of interaction between the user, who reviews the constraints and rearranges the partial scene in order to satisfy the true intentions of the user. Also, the user can select some variables to have random value ordering, in order to produce different scenes each time the program completes a run.

3.3.1 Interpreting the Partial Scene & Forming Constraints

In this section we will refer to the constraint formulation as implemented in the system using a CLP schema (Section 2.1.2), specifically the CLPFD [11] library. The description will follow the assumption that all shapes in the partial scene which will be examined are rectangles, and that the rectangles will express the intentions of the user regarding the object placement in the scene.

The first type of constraints implemented in the system, regard the *domain* of each CLP variable. For example, if the scene contains rectangles with a characteristic a which has the same value v for all the rectangles, then this means that the variable that will correspond to the variable a will be assigned the value v for all scene objects. Narrowing down the domain of other variables can occur by detecting the *minimum* and *maximum* of certain values that appear in the partial scene. For example, if the user inserts rectangles with height $\in [3, 10]$, then the domain of variables that represent the height for new scene objects will be limited to that range. For the previous example we would have: $domain(HeightVar, 3, 10)$.

Another type of constraints detected by the system, are collinearity constraints. The system searches for rough collinearity between a locus of points at the center of each rectangle and around its edges. The line equation can be used to represent the constraint, although it should be rearranged if it is in a form that includes division in order to keep the variables integers. This is because the CLPFD library deals only with integers. The example presented in [24] regards the line equation $8x + 28y - 11476$ and is the following:

```
unary_constraint(line0,S):-
        get_x(S,X), get_y(S,Y),
```

[4]of course there is always the case of CSPs which are unsatisfiable, in this case since since the user already forms a partial scene there is only the case that the partial scene formed by the user is the only possible scene

```
    (X*8) + (Y*28) - 114476 #> -150, (X*8) + (Y*28) - 114476 #< 150
```

The above code segment, takes the rectangle object S for which the constraints are to be applied, and performs a rough collinearity constraining, allowing 150 pixels of freedom - so that the system does not end up to be over-constrained.

Other constraints produced by the system, are constraints that relate the shape of rectangles in the scene, in order to produce similar shapes to the ones of the partial scene. For example, there can be a constraint imposed on the aspect ratio of the rectangles (the width:height ratio). The system can retrieve the maximum and minimum aspect ratio of each object in the partial scene, and limit the aspect ratios of all other shapes to this range. Since these constraints relate properties from a rectangle (e.g. width and height), they are called intra-rectangle constraints.

The system is also able to notice global relations between pairs of rectangles. For a given property P, the system calculates the set of *pairs* of rectangles which have equal value v for this property ($R_{P,=}$), the set of pairs of rectangles where the first rectangle has *less* value for the property than the second one ($R_{P,<}$), and the set were the first rectangle has *higher* value for the property ($R_{P,>}$). Then constraints are produced. The example presented [24] is the hypothesis $R_{(bottommost-y,<)} \subseteq R_{(width,<)}$. This relationship states that if the first rectangle has a base (bottommost) higher than the second, then it also has less width than the second rectangle. A code fragment that portrays the implementation of this constraint which is also presented in [24] is shown in the next code fragment.

```
binary_constraint(bottom_above_implies_less_width,S1,S2) :-
        get_y(S1,Y1), get_y(S2,Y2),
        get_height(S1,H1), get_height(S2,H2),
        Bottom1 #= Y1 + H1, Bottom2 #= Y2 + H2,
        get_width(S1,W1), get_width(S2,W2),
        (Bottom1 #< Bottom2) #=> (W1 #< W2).
```

In the above example, the first rectangle is $S1$ and the second is $S2$. The bottom y coordinates for each one of the rectangles are calculated and stored in the variables $Bottom1$ and $Bottom1$. Also, the width of each rectangle is stored in the variables $W1$ and $W2$. Then, a *propositional constraint* is imposed, in order to express the constraint. The constraint reads if $Bottom1$ is less than $Bottom2$ then $W1$ should be less than $W2$. One might think that an analogous way to define this property in the fitness function system (Section 3.2, [23]), is to define that the *bottom* and *width* attributes of an object have a correlation of 1, e.g. when the bottom decreases width will decrease as well. This is not exactly the case, since this also implies that:

$$(W1\# < W2)\# => (Bottom1\# < Bottom2)$$

The versatility of the CLP appoarch, is that the user can rule out one of the previous two constraints if both are detected.

3.3.2 Solving the CSP

After the constraints were inferred by the system and the user refined them by selecting out some constraints or selecting some for random value ordering, the CSP is passed to the constraint solver in order to obtain the solution set. Two approaches have been tested, an approach where the entire CSP was passed to the solver and returned a solution for all rectangles (*single pass*), and a second *sequential* approach, which *incrementally* builded the scene by solving for one rectangle per go. A parameter b was imposed, which affected the way back-tracking worked: If solving for a given rectangle failed more than *b* times then the previous two rectangles where removed from the solution set. This is a method of *backjumping* in constraint solving, and has been confirmed experimentally to give good results. In experimenting with these two methods, the results obtained proved that the single pass method performed worst than the sequential when the scene was becoming complex, with more rectangles. The approach of placing objects one by one is found to best in another similar problem we will examine in Section 5.1.1.1.

3.4 Evolutionary Art

Evolutionary Art can be defined as a way of generating artwork through the process of biological evolution, usually using evolutionary approaches, such as genetic algorithms and genetic programming. Evolutionary art is connected to the subject of this ISO because of the motivation behind this project, which lies in automatic scene generation in visual arts and the relation of *generating* a scene to *evolving* a scene. In the sense of considering every application of evolutionary approaches with an artistic result as evolutionary art, then the machine learning approach to scene generation [23] can fall into the broader category of evolutionary art, although in this project we try to discriminate between evolution of scenes and systems that generate novel scenes, especially when based on user desires.

Experiments in combining computer graphics and evolution begun in the early 1990s, while the research in this field by both researches and artists has grown over the next years. In this Section, we will briefly describe some of the approaches to evolution in visual art and design, referring to the detailed survey presented in [69].

In 1991 Karl Sims described an expression-based technique to evolve 2D images [75], which inspired many artists to generate artwork from mathematical expressions, such as [88, 92], while many websites display visual art scenes which were created in an analogous way. Mathematical expressions and evolution has also been used to create textures for graphics such as [76]. An interesting application of evolution in artistic images is the evolution of fractals. One of the most well known evolutionary art projects is the electric sheep project, which is a form of a distributed open source screen saver [5] which evolves primitives which are typically fractals. Evolutionary approaches in image processing have also been researched, such as image colouring, morphing and dissolving [54, 43], while work has been done in evolving images of faces [10].

Another field is the evolution of 3D art, with famous results such as *Mutation Art* by William Latham, who was part of one of the earliest efforts in evolving 3D geometry along with Stephen Todd [87]. Their approach included evolution and branching of organic forms. Other work in evolving and then rendering 3D geometry can be found in [28, 47]. In the field of computer graphics, there has been work in evolving plant geometry. We have already described L-systems (Section 2.5), a production system widely applied to the generation of plants and simulation of their growth [68]. Sims and Jon McCormack evolved animated plant life in landscapes [57, 58]. Other applications in visual arts include evolving images in time (animations and videos), while work has been done in forming the solution spaces for each problem, selecting the fitness functions and controlling the diversity of the solutions (whether the visual difference is obvious between different solutions).

We have briefly described some of the example applications of evolutionary art in visual arts. A much more detailed survey can be found in Chapter 1 of [69], while in other chapters of [69], evolution in fields such as photography and music is explored.

[5]`http://www.electricsheep.org/`

3.5 Discussion

In this chapter, we have described some approaches to generating novel and artistic scenes for visual art applications. We have given a small description of the AARON system, which is the first automated computerised system that draws artistic paintings; an innovative idea from the 1970s. Noting the limitations and the drawbacks of AARON (has a hardcoded and limited knowledge, knowledge specific to the objects that have been coded into it‘s knowledge base, limited variety) we have moved on to describe two more sophisticated approaches to generating novel artistic scenes. The evolutionary/machine-learning approach presented, offers a way for the user to describe his intentions for the outcome of the scene and then evolve the scene in order to satisfy those requirements. Also, the HR system was exploited in order to produce correlations for some objects in the scene and evolve them adjusting the fitness function to these correlations. A step further was to derive the intentions of the user from a partially completed scene, instead of having the user input correlations. This was implemented in the constraint logic programming approach, where a partial scene was created by the user and an attempt was made to extract his intentions from the partial scene. Randomness was introduced to some variables, while maintaining interaction with the user who decided which inferred constraints to keep for a final scene, while the solution was found by backtracking.

Since this is a novel area, other approaches can be tested for these methods, while there is still space for new constraints to be inferred from the user partial scene. In Chapter 6, we propose a procedure to detect equidistant objects in the partially generated user scene and then impose an equidistant constraint. Also, we discuss introducing soft constraints in the constraint logic programming method, in order to create a hierarchy of constraints and hopefully add to the variety of the generated scenes.

In experimentation, other methods can be used in searches. For example in the machine learning approach, other methods can be used for searching the scene space such as simulated annealing. In the constraint based approach, while solving the CSP, other methods of backtracking can be used such as conflict directed backjumping [71], which when a rectangle fails to be inserted in the scene, would take into account the rectangles that have constraint the one that failed and remove them from the scene instead of removing the chronologically previous rectangles. As mentioned in [23], research can be done on inventing more sophisticated fitness functions as well as improving the variety of the scenes. In [24], future work includes attempts to generate new constraint schemes, allowing more flexibility and combining evolutionary and machine learning approaches with the machine learning approach. Also, a possible application of formal methods such as shape grammars (Section 2.4) and L-systems (Section 2.5) can be explored in scene generation applications for visual arts.

Chapter 4

Automatic Generation of Architecture

This chapter will be devoted to presenting some applications that target the automatic generation of architecture. Applications described in this chapter, find application in fields such as the reconstruction of cities (e.g. a historical representation of Pompeii), the construction of cities, virtual models of cities generated for video games or movies or even as a tool for architects themselves. It is noted that this chapter makes use of fundamental concepts such as shape grammars and L-systems, which have been covered in Sections 2.4 and 2.5 respectively.

We will refer to automatically generating architectural floor plans, for example houses of a particular style as well as automatically generating actual 3D models of cities and buildings. We will describe the application of L-systems to generating cities, by taking into account geographical and statistical data. We will look at how streets are modelled, separating urban areas and how factors like population density effect the layout of streets and how the zones formed are then transformed into urban areas. We will then describe applications of shape grammars into architecture mostly for generating architectural floor plans or even models themselves. We will look at work inspired from shape grammars, such as discursive grammars, split grammars and CGA grammars, covering work that has been done in this field since the early 1970s. In more detail, in this Chapter we will cover the following:

In Section 4.1, we will present an approach to modelling cities with L-systems [64]. We will describe how roads are generated using extended L-systems (Section 2.5), how constraints are imposed to drive the production process while conforming with local requirements and how the buildings are eventually modelled.

In Section 4.2 we will give a brief overview of applications and use of the original shape grammar formality (Section 2.4), mostly for generation of architecture of a particular style by creating specific shape grammar rules.

Discursive grammars, which expand the notion of shape grammars by adding a special type of grammars called description grammars, will be described in Section 4.3. The description grammar along with heuristics are here used to drive the derivation process.

In Sections 4.4 and 4.5, we will describe two different forms of grammars, which were heavily inspired by the initial work on shape grammars: Split Grammars and CGA Grammars. We will see how these last forms of grammars expand the possibilities of shape grammars while producing more detailed and flexible models.

Finally, in Section 4.6 we will give a brief description of some work that has been done in the field of evolutionary architecture.

4.1 L-systems in City Modelling

The system described in [64] by Parish and Muller, presents a way to construct urban environments by using a small set of data as input, which are geographic and statistical data. This method takes advantage of the previous success that L-systems demonstrated with the modelling of growth in plants [68, 77] and relates this methods to the way that streets appear to branch in urban environments. In order to achieve this functionality, the authors of [64] have extended the L-system production system, as we will describe in this section.

The described in [64] system, takes the following steps in order to produce and visualise the resulting urban environment:

1. The road generation software receives the input geographical and statistical data, and the roads are constructed with the use of an extended L-system
2. Areas between the roads are subdivided in order to decide where the buildings will appear
3. Buildings generated as string representations resulting from the application of another L-system. The grammar represents Boolean operations on simple shapes.
4. A parser interprets the resulting scene in a suitable format, which is then passed to the visualisation software.

To define the actual input data, this consists of geographical information, as elevation maps and maps of resources (land, water, vegetation). Also sociostatistical information maps are provided, which relate to population density, zone maps (residential, commercial, mixed), street patterns (the style of the structure of streets) and height maps which relate to the desired height of buildings in different regions. The system allows user interaction for modification of these values. It is also noted, that the statistical information interact with the generation process, since for example the density of the population is related to the road structure and network in the area.

4.1.1 The Street Map & Extended L-Systems

Due to the high complexity that would result from an effort to embody all the rules for the generation of street maps into the L-system productions and the need for rewriting rules to represent new constraints, the L-system is actually used to produce a generic template (called the *ideal successor*) at each step of the derivation and the modification and control of the actual parameters and constraints that are imposed are left to two external functions, *globalConstraints* and *localConstraints*. The extended L-system used, is quite similar to the *open L-system* described in Section 2.5.2. It is reminded that open L-systems systems use the notion of *modules* with parameters (which appear like function calls with parameters). If the module is a query module (using the ? symbol in front), the parameters will be assigned outside the actual L-system. Also, curly brackets ({,}) are used to call external methods which exist in the namespace of the application. We will also remind the syntax of these rules: $id : lc < pred > rc : cond \rightarrow succ : prob$, where id is the production *label*, the left hand side to be rewritten is $lc < pred > rc$, where $pred$ should be preceded by lc and followed by rc, $cond$ should be true for the rule to fire, $succ$ is the successor and $prob$ is the probability of applying the rule.

The L-system presented for road generation has modules, which consist of symbolic names and parameters. At each step of the production, the L-system returns the *ideal successor*, which is a generic template. The *globalGoals* function is then called, and sets the parameters according to higher level tasks, such as the target street pattern and the population density. Then the *localConstraints* function is invoked, to adjust the parameters so that they are consistent with the local constraints (such as resources and street elevation) posed by the environment. If the parameters can not be made consistent, then this production step fails and is deleted.

In the system presented in [64] by Parish and Muller, the L-system is initialised as follows:

```
ω : R(0, initialRuleAttr)?I(initRoadAttr,UNASSIGNED)
```

The above states the initialisation of the system with a road module (R) and a query module I.

```
p1 : R(del, ruleAttr) : del < 0 → ϵ
```

The above production is used as a break flag. If the globalGoals function sets the *del* parameter to a negative value, then this module is deleted (ϵ usually stands for the empty string in grammars).

```
p2 : R(del, ruleAttr) >?I(roadAttr, state) : state ==SUCCEED
{globalGoals(ruleAttr, roadAttr) creates parameters for pDel, pRuleAttr, pRoadAttrarraysxx}
→ +(roadAttr.angle)F(roadAttr.length),
B(pDel[1], pRuleAttr[1], pRoadAttr[1]),// create a branch module
B(pDel[2], pRuleAttr[2], pRoadAttr[2]),// create a branch module
R(pDel[0], pRuleAttr[0]),// create a road module
?I(pRoadAttr[0],UNASSIGNED)// create a query module
```

Production $p2$, controls road and branch creation. Two branch modules (B) and a road module (R) are created, while the attributes that each module receives are initialized by a call to the *globalGoals* method.

```
p3 : R(del, ruleAttr) >?I(roadAttr, state) : state = FAILED → ϵ
```

Production $p3$ causes the deletion of any road module which has a state set to FAILED. As we have already mentioned, this happens when there are no parameters available to comply to local constraints (this is handled by the *localConstraints* function).

```
p4 : B(del, ruleAttr, roadAttr) : del > 0 → B(del − 1, ruleAttr, roadAttr)
```

This production ($p4$) decreases the *del* flag of a branch (B), while:

```
p5 : B(del, ruleAttr, roadAttr) : del == 0
→ [R(del, ruleAttr)?I(roadAttr,UNASSIGNED)]
```

Production 5 creates a new road module and inserts a query module, after the delay counter has reached zero (*del*). The *globalGoals* method can prohibit the branches from expanding by setting the *del* counter to a negative value, something that is noticed in $p6$. Production 7 is also similar, since it notes the deletion of a query module if the *del* parameter for an R module is set to a negative value by *globalGoals*.

```
p6 : B(del, ruleAttr, roadAttr) : del < 0 → ϵ
p7 : R(del, ruleAttr) <?I(roadAttr, state) : del < 0 → ϵ
p8 :?I(roadAttr, state) : state ==UNASSIGNED
{localConstraints(roadAttr) adjusts the parameters for state and roadAttr}
→?I(roadAttr, state)
p9 :?I(roadAttr, state) : state! = UNASSIGNED → ϵ
```

Production $p8$ adjusts the parameters of a query module as long as the state is unassigned. The *localConstraints* method attempts to adjust the parameters in order for them to be consistent with the local constraints. In case this is not possible, the state variable is assigned the value $FAILED$, instead of the $SUCCEED$ value that is assigned if the parameters have been adjusted. In this way, productions $p8, p9$ decide if the road segment is going to be created or not.

4.1.2 Global Goals

The global goals are used to drive the production procedure and assign values to parameters of modules in the L-system described above. These goals depend on the input of the system. For example, highways connect centres of population, so rays are casted from every highway road end in order to sample the population along those rays and decide upon connecting them. When no population exists, then roads stop expanding. Goals are also set depending on the street pattern to be followed. In [64], 4 different patterns are specified:

- A basic rule where no special pattern is used and the roads just follow the population growth. This can usually be found in older cities.
- The checkers (New York) rule where a global angle is followed, enclosing the roads in a rectangular pattern. This is a very common street structure.
- The radial (or Paris) rule, where the streets are distributed along radiuses of a circle with a given centre.
- The San Fransisco rule, where streets follow the least elevation rule (least height from sea level). This means that roads on different heights are connected by short and small streets.

If more than one patterns are active then they are weighted and blended in the resulting city structure.

4.1.3 Local Constraints

As we have already mentioned, local constraints modify the parameters of the modules (which have been assigned by *globalConstraints*) in order to keep them consistent with certain local constraints, which correspond to the local environment. They function in two steps:

- Check if road intersects (ends inside or crosses) with an illegal area. This illegal area can be e.g. water or sea. In this case, the system attempts to prune or transform the rode segment by rotation, in order to overcome the obstacle of the illegal area. Some roads (highways) can cross illegal areas for up to a certain length and then be replaced by, for the example of water, bridges.
- Searches for intersections with other near-by roads in order to form crossings and intersections. If suitable roads are found, then:
    1. If two streets intersect, generate crossing
    2. End segment of road near an existing crossing then extend road to reach crossing
    3. If two roads are close to intersecting, then they are extended to form an intersection

4.1.4 Modelling Buildings with Parametric Stochastic L-Systems

In this system, areas between the roads are separated using simple recursive algorithms until they are small enough to fall under a certain threshold. Taking into account other parameters (such as the maximum height of a building in a city), the buildings are modelled using a *parametric, stochastic l-system*, which has modules for translating, scaling, extruding, branching and terminating. The generation starts

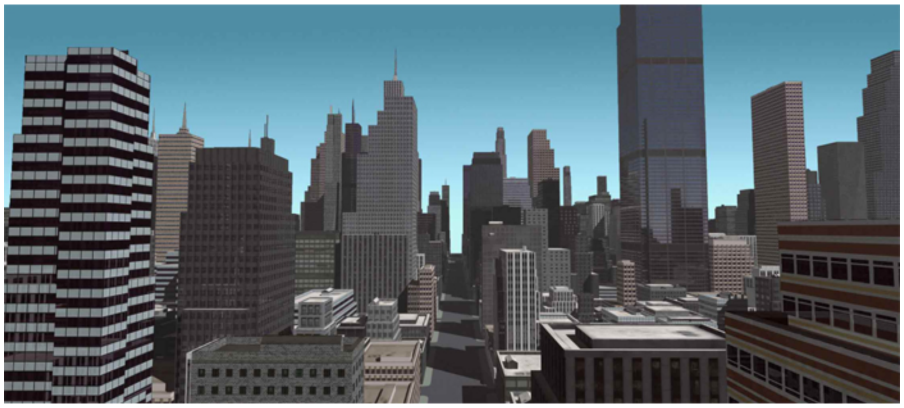


Figure 4.1: A visualization of Manhattan, from [64], using L-systems

with the bounding box[1] and it is refined step by step in order to achieve the desired result. The output of the L-system is then translated and visualised.

The approach used for texturing the facades of the buildings, is based on overlaying and compositing textures. This approach is based on the assumptions that facades of buildings usually have a grid like structure. The hierarchical grid structure is based on interval groups (set of non-overlapping, ordered intervals). One dimensional grids are used, and are combined in order to form a 2D grid. This 2D grid is then evaluated by an evaluation function and colours are assigned to each cell. Layers can be superimposed in order to produce a more complex texture.

An image from the use of L-systems in [64] to construct cities, can be seen in Fig. 4.1.

4.2 Shape Grammars in Architecture

The first approach to the use of shape grammars in architecture was published in 1977 by Stiny [80], where with five simple shape grammar rules (as described in Section 2.4.1), Stiny captured the creation of Chinese ice-ray lattices which were constructed in China, with a structure which imitated the ice-lines which were formed during ice-formation. In fact, the work of Stiny has been expanded in [85], for the generation of these latices onto 3D surfaces. The second publication in 1978 by Stiny and Mitchell [83], where a more complex and detailed approach is presented with a goal of constructing Palladian style architecture, that is the style of the Italian architect Andrea Palladio, influenced by ancient Greek and Roman temples. In [83], the authors provide the shape grammars used to produce this kind of architecture, from producing the grid on which the buildings would be based on to the production of the walls, rooms, doors and windows. From there on, many architectural style shape grammars emerged, such as other work by Stiny [84] , work on Taiwanese traditional housings [14], work by Flemming on styles such as Queen Anne houses and the bungalows of buffalo [33, 30, 34], on prairie houses [50], on modelling the style of architect Glenn Murcutt [44], city church design [9], floor plans of Japanese tearooms [48] or even the modelling of designs on ancient Greek pottery [49]. We will not describe these approaches, since they are in essence applications of the same formalism in different case studies and would be more interesting to a purely architectural survey. Having referred to the research work, case studies and applications that

[1]a bounding box in 2D is defined as the smallest rectangle that covers the shape, in 3D this the rectangle is just generalised into a 3D rectangular parallelepiped

the original shape grammars found, we will now describe some methods that extend the original approach to shape grammars described in Section 2.4.

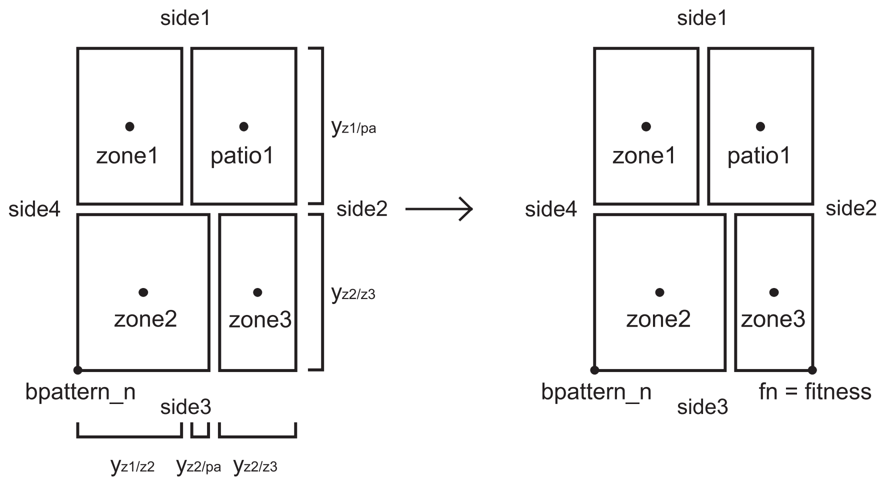
4.3 Discursive Grammars

A discursive grammar [31] consists of a shape grammar [39], a description grammar [81] and a set of heuristics. Description grammars augment the approach to using shape grammars, by dealing with the semantics of the production, while the heuristics employed are used for refining the production rules that will be used at each step of the production, while taking into account possible problems that could result from the application of a rule. We can say that in a way, this approach expands the initial selection rules and labelling approaches used in shape grammars, in order to achieve a more expressive and accurate system. In [31], the goal of the author is to generate an architectural solution to match a set of given user requirements in an interactive way, encoding the rules of an architectural style. This is posed as an optimisation problem, assigning weights to the requirements and taking cost into account. From a technical point of view, the authors consider a programming grammar, that encodes the specifications and requirements of the user and produces a symbolic description of them and a designing grammar which actually produces the housing solution. The programming grammar consists of both the description and shape grammar, while the designing grammar is essentially the shape grammar. Labels attached to the shape grammar express the function of the room being designed, as well as markers (•) are used to denote special shape areas, while the description part can describe symbolically the rules and produce additive information, for example counting the remaining number of rooms to be designed according to the requirements. Heuristics are used to drive the derivation in order to match the produced design with the goals set by the user, for example by evaluating the fitness of a configuration and selecting the one which matches best to the requirements. An example of shape rules along with description rules is shown in Fig. 4.2.

4.4 Split Grammars (Instant Architecture)

Wonka, Wimmer, Sillion and Ribarsky present in [91] a new type of grammars for procedural architectural model. These grammars are called *split grammars* and have been designed in order to overcome some other difficulties presented in procedural modelling by other production systems, such as L-systems and shape grammars. L-systems have found application in plant modelling [68] as well as in street modelling [64] and terrain - landscape generation, but L-systems have characteristics (such as simulating growth) which can not be easily adjusted to the strict spatial and geometric constraints that would be required in modelling cities and buildings. The authors of [91] produced a system that differentiated their work from shape grammars, in terms of having a complete set of production rules and use the same set to produce a variety of designs, instead of shape grammars, which as we have seen are specific to a certain design. The extension is similar to the extension of discursive grammars described in Section 4.3 in the way that it maintains some goals and then uses the rules to generate architecture that satisfies them. We will now proceed to formally define split grammars. We will then describe control grammars (Section 4.4.1), which assign values to attributes of shapes and then we will describe the attribute matching process (Section 4.4.2), which helps drive the derivation of shapes based on attribute values assigned by control grammars.

Split grammars, similarly to shape grammars, manipulate sets of *basic shapes*, that is shapes which can have attributes, parameters and labels. Split grammars defined over alphabet B of basic shapes, have the following two rules [91]:



zone1, zone2, zone3 $\in$ {living, sleeping, service}.
number of assessed patterns = number of assessed basic patterns + 1
number of existing basic patterns = number of existing basic patterns + 1

Figure 4.2: Shape grammar rules along with description grammar rules from [31]. The rule is for assessing a basic pattern, which later may be removed if below requirements. We can observe how the description grammar rules below the shapes give symbolic descriptions of the types of zones as well as increment counters. This rule is for assessing because of the calculation of the fitness function that is causes (fn=fitness)

- Split Rule: $a \rightarrow b$, where a is a connected subset of shapes in B, while b contains one element less than a, the element on which the split is applied.
- Conversion Rule: $a \rightarrow b$, where again a is a connected subset of B containing just one basic shape, while b is the same as a but contains a different basic shape, replacing in this way the basic shape in a, while the new basic shape in b is limited to be contained in the volume of a.

Similarly to all grammars, a derivation of a split grammar halts when the set of shapes derived are all labelled as terminal shapes. The resulting set is called a *building design*. An example of a split rule, along with a possible derivation is presented in Fig. 4.3.

Attributes in split grammars are propagated during the production and encode information that relates to the current shape. This information can regard information about texture or colour, but can even go deeper in driving the derivation itself by filtering the applicable rules to satisfy the goals of the production. This is part of the attribute-matching process for split grammars. Since the attributes are very important to the production process, it is also important to be precise and accurate on assigning attributes and values. In [91], the following ways are presented with which attributes are assigned:

- Assign an attribute to the start shape, and propagate the attribute accordingly during the derivations
- using Control Grammars, which will be described in the following section

4.4.1 Control Grammars

A control grammar is defined as a regular context free grammar (CFG) [91]. A control grammar rule is based on a grammar, where the non-terminals are descriptive symbols or strings and the terminals stand

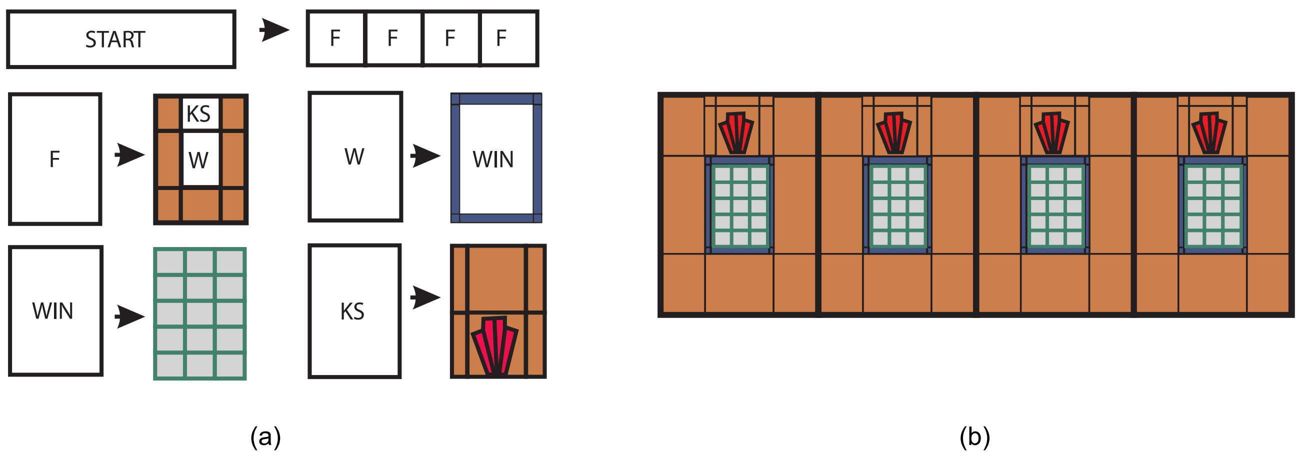


Figure 4.3: (a) a simple set of split grammar rules. White shapes represent non-terminals while coloured stand for terminals. (b) A derivation of the split rules in (a). Images from [91]

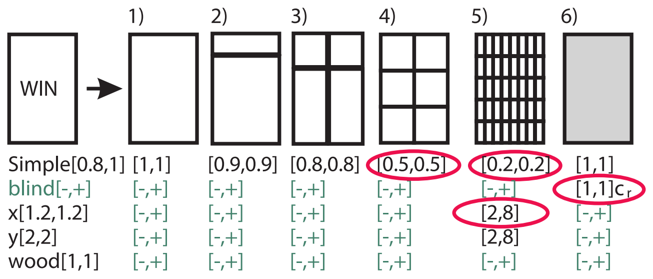

WIN
1)
2)
3)
4)
5)
6)
Simple[0.8,1] [1,1] [0.9,0.9] [0.8,0.8] [0.5,0.5] [0.2,0.2] [1,1]
blind[-,+] [-,+] [-,+] [-,+] [-,+] [-,+] [1,1]c_r
x[1.2,1.2] [-,+] [-,+] [-,+] [-,+] [2,8] [-,+]
y[2,2] [-,+] [-,+] [-,+] [-,+] [2,8] [-,+]
wood[1,1] [-,+] [-,+] [-,+] [-,+] [-,+] [-,+]

Figure 4.4: In this figure from [91], we can see how the attribute matching process disqualifies some of the 6 possible rules that can be applied for the non-terminal left hand shape. The circled attributes are the attributes that disqualify a rule. [-,+] stands for $[-\infty,+\infty]$. Rules 4 and 5 are disqualified because there is no interval overlap while the 6th rule is disqualified due to the containment flag (c_r), due to the fact that [-,+] $\not\subseteq$[1,1]

for commands, which in turn modify an attribute. More specifically, a non-terminal in a control grammar is specified as a triplet $< c, a, v >$, were c is a spatial locator and actually defines the target shapes of this rule while a stands for attribute name and v for the value that should be assigned to the attribute. In other words, the terminal command is interpreted as: if(c) then a=v. There is a special attribute which exists in shapes and when found, invokes the control grammar with a specific start symbol.

4.4.2 Matching Attributes and selecting rules

In the framework presented for split grammars, attributes are associated with production rules themselves. In this way, whenever more than one rules can be applied at one step of the derivation, the attributes that are attached to the rule are compared with the attributes that are associated with the current symbol in the production, and so the best matching rule is selected. It is noted, that this type of matching occurs in both control and split grammars. The matching procedure has a deterministic and a stochastic step, both of which we will now describe. The interaction between the elements of the system can be seen in Fig. 4.5, while an example of attribute matching can be seen in Fig. 4.4.

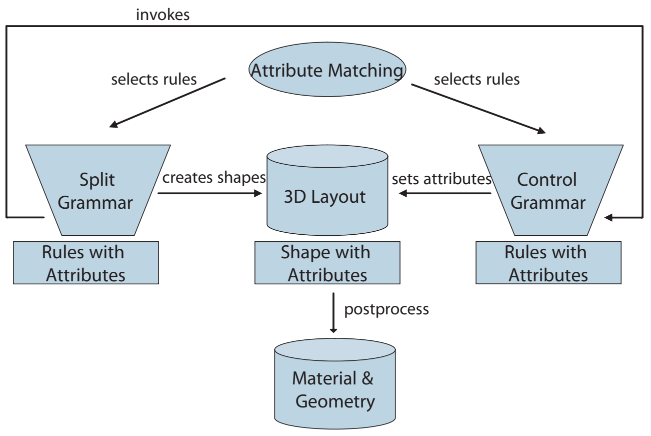


Figure 4.5: Interaction between the elements of the instant architecture system, as presented in [91]

Assuming that we have a symbol s, with attributes $a_{1s}, ..., a_{ns}$ and a set of rules R, with each rule $r \in R$ being related to the attributes $a_{1r}, ..., a_{nr}$. Rule and symbol attributes are defined as follows:

$$a_{ir} =< I_r, c_r >, a_{is} =< I_s, c_s, f, SD >$$

where I_j is an interval, c_j is a containment flag (which means that the corresponding rule/symbol which is being matched with this one has to have an interval which is a subset of the interval of this rule/symbol), f is a positive priority factor and SD is the statistical distribution of the symbol. The function used to evaluate the matching is actually a *similarity function*, and for a symbol and a rule attribute is defined as follows:

$$m_{DV}(a_s, a_r) = f + E((I_s \cap I_r \neq 0) \wedge (c_s \rightarrow I_r \subseteq I_s) \wedge (c_r \rightarrow I_s \subseteq I_r))$$

From above, we can see that the function returns a metric which corresponds to the matching of the two attributes. Firstly, the f value of the symbol attribute is added to the value. Then, three logical rules in a conjunction ensure that:

- that the intervals of the two attributes overlap
- if the containment flag for the symbol c_s is true, then the interval of the rule attribute should be a subset of the symbol attribute interval, and
- if the containment (c_r) flag for the rule is true, then the interval of the symbol attribute should be a subset of the interval of the rule attribute.

The body of the function E used is:

$$E(P) = \begin{cases} 0 & if P = true \\ -\infty & if P = false \end{cases}$$

so if any of the logical rules fail, the input to the function is false and negative infinity is returned. The total metric of a rule against a symbol is then summed:

$$M_{DV}(S, r) = \sum_{i=1}^{n} m_{DV}(a_{i,s}, a_{i,r})$$

The above, conclude the deterministic step in choosing a rule to use during a split grammar production. Wonka et al. [91], also provide a stochastic way to choose a rule, which is only used if the previous deterministic procedure returned more than one rules with a maximum value. The stochastic matching function takes the statistical distribution defined for the attribute of the symbol S and returns its value in the middle of the interval of the rule attribute, I_r. The function is as follows:

$$M_{SV}(S,R) = p_r \prod_{i=1}^{n} f_{SD,a_{i,s}}(\frac{u_r - l_r}{2})$$

To provide consistency in the final scene or building, the result of this stochastic function is always the same for the same contesting attributes. It is suggested [91] that pre-computing the value of this function is the way to deal with these cases.

4.5 CGA grammars

CGA (Computer Graphics Architecture) grammars are an expansion of the original shape grammars [39], which was proposed by Mueller et al. [62]. This method is an iterative method, which increments the detail of designed buildings, evolving a shape to more and more detail, implicitly providing a hierarchical procedure. CGA grammars are targeted at the production of architectural models for computer games and movies at a low cost.

The goal of Mueller et al. in [62] is a combination of the methods the urban environment generation with L-systems [64] (Sec. 4.1) and the geometric detail of buildings described with the use of *split grammars* in [91] (Sec. 4.4), which will overcome weaknesses of both the previous methods, for example the simplicity of the buildings produced with L-systems [62] and the excessive amount of splits required for split grammars [91] to generate complex models.

4.5.1 System Overview

The CGA grammar described in [62], works with shapes in the following way: Each shape, has a symbolic description, a set of geometric attributes (position P, coordinate system of shape X, Y, Z, size vector S) and some numeric attributes. The symbolic description of the shape can be a terminal or a non-terminal. The geometric attributes define a shape oriented bounding box called scope.

The production procedure is as follows. Initially, the *axiom* concept (which stands for the start state) is adopted from L-systems (Sec. 2.5) and is defined here as an arbitrary configuration (a finite set of basic shapes). The production is as follows:

- Select an active shape with symbol B in the set (all sets are considered active in the first step)
- Compute a new set of shapes, B_{new} by applying a valid production rule (match the left hand side). Rules are assigned a priority and the selection is determined based on that.
- Mark shape B as inactive
- Add shapes B_{new} to configuration

The definition of a production rule is as follows:

$$id : predecessor : cond \rightarrow successor : prob$$

which as we can see is identical to the L-system rules defined in Sec. 2.5. Here, the *predecessor* is a non-terminal (variable) to be replaced with successor, *cond* is a logical expression that must be evaluated to true and *prob* is the probability for the application of the rule (just like stochastic L-systems).

Translation (T), rotation (R_x, R_y, R_z) and scaling (S) rules (commands) are also used for the shapes. The brackets [and] are used to push and pop the current scope (bounding box) on the stack. The current scope is applied to every non terminal created, while the command $I(objId)$ generates a geometric primitive.

A basic split rule is used to split the current scope along one dimension. The example given in [62] is the following:

$$fac \rightarrow Subdiv("Y", 3.5, 0.3, 3, 3)\{floor|ledge|floor|floor|floor\}$$

The context of the above command is that the *Subdiv* command is invoked to perform the splitting. The first argument stands for the axis while the rest for the split sizes. In curly brackets, the shape corresponding to each split size is positioned. Similar splits are used for more than one dimensions. Placing the r character instead of a split size, a relative split occurs, with the relative value being calculated as $r_i * (Scope.sizex - \sum abs_i)/\sum r_i$.

The repeat command allows for an arbitrary number of splits. Again, the example given in [64] is:

$$floor \rightarrow Repeat("X", splitLength)\{B\}$$

which splits the scope along the x axis as many times as there is space (which can be calculated by $\lceil Scope.sizex/splitLength \rceil$), and adds so many elements of type B.

A final grammar command is called *component split*, and allows to split into shapes of less than 3D:

$$a \rightarrow Comp(type, optionalparam)\{A|B|..|Z\}$$

An example command is $Comp("faces")\{A\}$ which creates a shape A for each face of the original 3D face. Vertices, edges and other types can also be used. Optional parameters are used to access selected components of a type, for example $Comp("edge", 3)\{A\}$ creates a shape A which is aligned with the third edge. The reverse is also possible, i.e. to go from lower to higher dimension by extruding a shape, by using the *scale* command and scaling across a dimension which is already zero, to represent the lower dimension.

4.5.2 Occlusion & Snapping

Another important aspect of the system is that the system can obtain information about the occlusion of shapes. The grammar rules can query if a shape is occluded or note, and furthermore if it is partially or fully occluded. In this way, the grammar can contain rules which can test if a shape is occluded and then apply some transformations (e.g. if tile is not occluded then it can be transformed into a door). This avoids placing objects on the intersection of other shapes. Another improvement is to use snap lines to guide the splitting, i.e. where the split will occur in the shape.

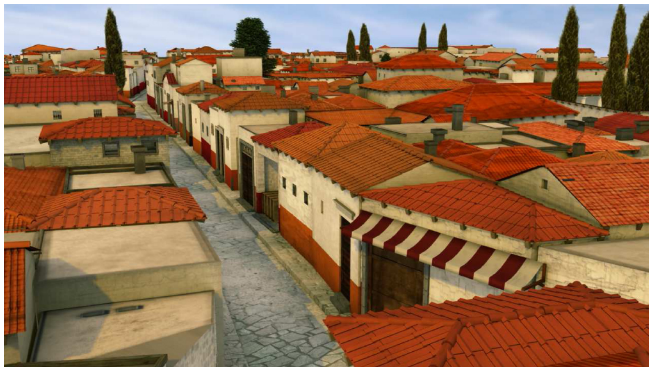


Figure 4.6: Pompeii modelled with a CGA grammar, from [62]

Figure 4.7: Evolutionary Architecture by Architect Makoto Watanabe (metro station in Japan)

4.6 Evolutionary Architecture

There has been work in combining evolution and architecture by architects such as Gero, Tsui, Frazer, Soddu and Watanabe. Frazer has published a book (*An Evolutionary Architecture* [37]), in which he describes some of his research work. He used concepts of evolution and growth in his work, revolving around concepts of forming seeds for evolution and evolving them. Frazer has used genetic algorithms, evolutionary concepts as well as other machine learning notions such as neural networks. Gero's work is around recognising creativity and novelty, representing style and generally in intelligent design [40, 41, 73], while other work includes the research of Paul Coates in evolving structures based on some performance fitness criteria, also using L-systems and shape grammars [8, 16]. Tsui [2] and Watanabe mostly look into nature and evolution as inspiration to architecture, while Soddu[3] has experimented in emulating and controlling the evolution of architectural and design processes.

4.7 Discussion

In this chapter we have presented an overview of research that has been done in the area of automatic generation of architecture. We have described systems that make use of grammars and L-systems in order to model architecture with certain requirements. We have seen how L-systems and their ability to depict growth and branching has been used in modelling roads and cities. Then we proceeded to describe methods which were related to shape grammars, and expansions to the original description of shape grammars and L-systems that were implemented, in order to better adjust to the generation of architecture and cities. Specifically, an example is discursive grammars which included descriptions and heuristics in order to drive the generation process. Split grammars were introduced due to the weakness of L-systems to model buildings in great detail. Also, the approach of split grammars allowed modelling many designs with just one grammar, differing thus from shape grammar approaches where each set of rules was limited to one object, while the introduction of stochastic procedures increased the variety of the generated scenes. Finally, CGA grammars compared to L-systems are more shape-oriented instead of working with strings, implementing shape-rules not available in L-systems and better adjusting to architectural requirements. Comparing to split grammars, CGA grammars allowed more target-specific grammars to be created, instead of the strict hierarchy imposed by the split grammars.

[2]`http://www.tdrinc.com/`
[3]`http://www.artegens.com/`

Chapter 5

Spatial Planning, Scene Layout & Composition

In this chapter we will present some approaches which solve the problem of spatial planning and the problem of scene layout. The problem of spatial planning (Section 5.1), is generally defined as safely placing objects in a certain space taking into account obstacles, such as holes or other objects. This problem often occurs in areas such as architecture (for example, planning buildings), circuit design and of course, the problem of laying out a virtual scene. We will refer to approaches that form a CSP (Section 2.1) out of a spatial planning problem and attempt to solve it [1], as well as a geometric approach to solving the problem by making use of the minkowski operators (Section 2.3). The motivation is that the techniques used for solving this problem may be transferred and used in automatic scene generation systems.

In Section 5.2 we will describe some 3D systems which deal with an analogous problem: Of laying out 3D objects in 3D scenes. We describe an approach that is based on geometry and uses semantic constraints in order to layout a scene and an approach that uses genetic algorithms in order to evolve and optimise a scene configuration.

It is important to note that in this chapter we do not attempt to exhaustively cover all the systems and approaches to this problem, such an extensive study is outside the scope of this project. We will though, attempt to give characteristic examples from each group of methodologies, as well as present methods that actually generate and output scenes, mostly in the section relevant to object placement.

5.1 Automated Space Planning

The problem of space planning, is by definition quite similar to to a problem faced when automatically generating scenes. It is defined as "the problem which consists in finding the locations and the sizes of several objects under geometric constraints in a location space" [13].

There have been quite a few approaches to solving the spatial planning problem. Initially in the 1960s, exhaustive search was used [35, 46]. Then, in the 1970s, heuristics were incorporated [32, 65] in order to shrink the search space. Heuristics used, included sorting by size of object and preferring locations to placing objects, for example corners of rooms. The result was order dependent and completeness was not

[1]We will not go in depth in CSP solving methodologies in this Chapter, as we are focusing on the actual formulation of the problem and not to the specifics of constraint solving

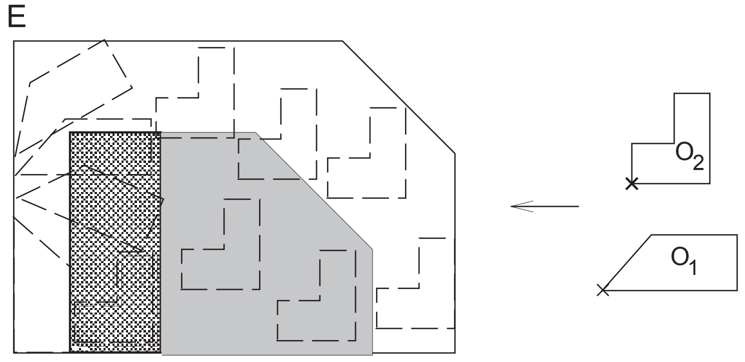


Figure 5.1: Two shapes O_1 and O_2 are to be placed in location space conforming to a set of constraints. Before arc-consistency is imposed, the two reference points of the two shapes (marked as X) are just constrained to be inside the location space. After arc-consistency is imposed, the reference point of O_2 should fall inside the gray regions. [13]

guaranteed. In this section, we will focus on methodologies presented in by Charman [13] and Baykan [6], which in general transform the spatial planning problem into a CSP and attempt to solve it by using constraint solvers, as well as a geometric approach presented by P. K. Ghosh [42]. Other interesting approaches are presented in [45] by Honda and Mizoguchi, where again the problem is specified as a CSP and in [60] by Michalek et al., where the constraints are posed in equations (similarly to approaches that we will describe) and then the entire problem is posed as an optimisation problem, that is a problem of optimising a cost function.

5.1.1 Geometric CSP and Space Planning

In [13], a Geometric Constraint Satisfaction Problem (GCSP) extends the notion of CSP (Section 2.1) by placing geometric constraints on objects (which correspond to CSP variables) which are to place and size in a scene. Attributes of each shape are the orientation, the shape itself, the possible sizes as well as the location of the shape with respect to a reference point. In the approach described in [13], position variables belong to multidimensional and geometric domains, while the orientation variables are discrete.

In [13], experiments were done by enforcing local consistency (arc-consistency) on objects which are part of a GCSP. In Fig. 5.1, we can see an example from [13] where arc-consistency is imposed on a GCSP. We have a location space E, where two objects O_1 and O_2 are to be placed, conforming to the next three constraints: O_1 must touch the left side of E, O_2 must be right of O_1 and O_2 must have a fixed orientation.

5.1.1.1 Placement of Objects and Backtracking

Experiments in placing objects in a constraint based way have shown that the best way of placing the objects in a given space is by placing them one by one, as argued in [13] by Charman. This result agrees with the results of the constraint based approach to automatic scene generation for visual arts (Section 3.3, [24]). The backtracking approach in [24], proposed that when an object could not be positioned in a way to satisfy the constraints then the previous two determined objects would be removed. In Charman's system [13], propagation directed backtracking is used, which is a form of intelligent backtracking. More specifically, for an object O_i, the objects which have reduced its configuration (domain) are noted. When an object O_i can not be placed, then the search returns to the last located and placed object which reduced the configuration of that object or one of the non-placed objects for which the configuration has been reduced by the positioning of an object. This is very similar to conflict directed backjumping (CDBJ) [71]. Problematic configurations of objects can also be recorded in order to avoid repeating them.

The detection of symmetrical geometric constraints (defined as geometric interchangeability in [13]) is quite important to limit the search depth. A symmetric geometric constraint is a constraint of the form $no_{overlap}(A, B)$. When two objects have exactly the same constraints and the constraints between them are symmetrical then it is said that the objects are interchangeable. The method proposed also takes advantage of other heuristics, such as removing redundant constraints (e.g. if object A is supposed to be entirely to the left of object B then there is no point in a no-overlap constraint) and removing incompatible constraints, i.e. constraints that can not be both true in the context of the scene. There is no standard set of heuristics for limiting the search space which are problem independent, but heuristics can be used that are specific to a certain problem, for example every room must have at least one door. The system, in order to solve the CSP, first detects incompatible and redundant constraints, then applies heuristics as previously described, imposes arc-consistency and finally uses backtracking.

5.1.2 Spatial Planning by Disjunctive Constraint Satisfaction

In [6], Baykan and fox present an approach for configuring rectangles in 2D space, where the sides of these rectangles are parallel to an orthogonal coordinate system. The layout problem defined is as follows: "Given a set of rectangles, desired spatial relations between them, limits on their dimensions, areas and aspect-ratios, generate feasible alternatives". The WRIGHT system which is described in [6], solves this problem. In the terminology of the system, the objects are called *design units*, while the relations (topological and geometrical) between them are called *spatial relations*. Also, general limitations and requirements for the objects, such as range of allowed width, length and area are given as *performance constraints*. Another set of constraints, called *realizability constraints* are specific to the problem and can be for example a limitation that objects which stand for interior spaces should not overlap. Finally, the *style* of the design is defined by constraints that e.g. allow holes in the final design or which objects should be attached to the perimeter.

The first set of constraints, that define locations, dimensions and inequalities of the design units (topology and geometry) are sets of *atomic constraints* [2], which are here defined as algebraic equations and inequalities. These form a CSP on their own. The rest of the constraint sets, are compiled into sets of *disjunctive constraints* i.e. atomic constraints (c_i), conjunctions of which (d_i) are combined in the form of $d_i \vee d_j \vee ...d_n$. These disjunctive constraints in turn form a *disjunctive* CSP which is then solved by instantiating one of the disjuncts taking into account a priority imposed by *textures*, which are here defined as rules or heuristics which are based on features of the constraints and generally impose some backtracking ordering heuristics [3].

The system uses interval domains (i.e. minimum and maximum values) to define the range of values variables such as width, height, area and aspect ratio can take. Orientation is a discrete variable that defines south, east, north and west orientations with the set $\{0, 90, 180, 270\}$, which corresponds to the degree of rotation.

[2]In constraint logic programming, an atomic constraint usually defined as an atomic formula with a constraint predicate symbol[79], here it is just defined as a constraint that defines a relation between two variables

[3]Heuristics for value ordering and variable ordering are typically used in constraint solving, the reader can refer to Chapter 5 of [71]

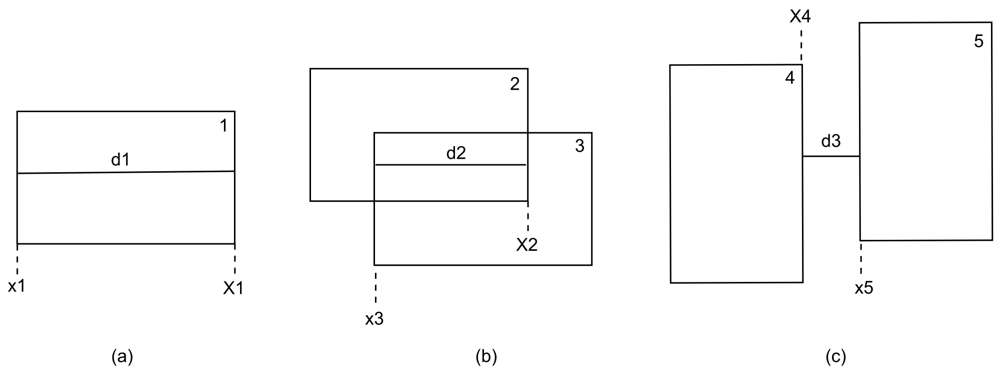


Figure 5.2: Examples of using bounded difference constraints for spatial layout [6]. (a) In design unit 1, its length $d1$ can be described as $(X_1 - x_1 \in [d_{1min}, d1max])$ (b) The length of overlap between design units 2 and 3 can be described as $X_2 - x_3 \in [d_{2min}, d_{2max}]$ (c) Distance between two design units (4 and 5 here) can be described as $x_5 - X_4 \in [d_{3min}, d_{3max}]$

5.1.2.1 Atomic Constraints

There are several forms of atomic constraints which are allowed in the WRIGHT system, as described in [6]. These types are as follows:

- Bounded difference constraint: The interval of allowed distances between two lines and can be expressed as $x_i - x_j \in [d_{min}, d_{max}]$. In this way, constraints can be imposed that can define the length of a design unit the length of overlap between two units, the distance between two areas, force adjacency and so on. Examples can be seen in Fig. 5.2.
- Area constraints connect the variable that corresponds to a design unit with an interval that defines the range of possible areas, for example $A_i \in [100, \infty]$.
- Aspect ratio constraints impose bounds constraining the aspect ratio of the design unit, which here is defined as: $\frac{|a|}{|b|}$ where a is the long side and b is the short side.
- Orientation constraints simply limit the possible angles of rotation.

These constraints form a CSP of their own, and possible solution algorithms are extensively described in [6]. In essence, each bounded difference constraint $x_i - x_j \in [d_{min}, d_{max}]$ is decomposed into two canonical form constraints, $x_i - x_j \leq d_{max} \wedge x_j - x_i \leq -d_{min}$. One of the methods used is formulating a distance graph and then detecting inconsistencies using the Floyd-Warshall shortest path algorithm (Section 26.2, ”The Floyd Warshall algorithm”, [26]). These notions as well as the methods used are explored in [6].

5.1.2.2 Spatial Relations

In a spatial planning system, it is, as previously noted, important to represent relations between objects. These constraints can involve properties such as adjacency, enclosure, distance and more. Since the x and y components of an object in this system are actually intervals, then adjusting from Allen‘s qualitative temporal relations[4] between two time intervals [1], we conclude that there are 13 mutual exclusive (no more than one can hold at a time) and exhaustive (at least one must hold) relations between two intervals.

[4]This relations were adjusted here from time to space, i.e. overlaps stands for the space interval and not the time interval, as was originally in [1]

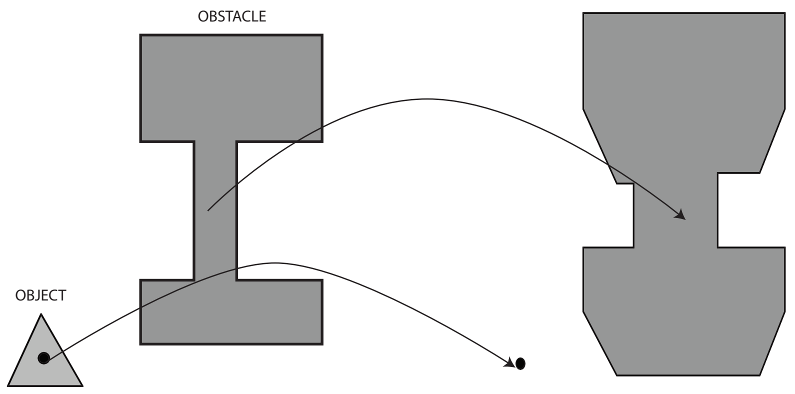


Figure 5.3: Transforming from real space to configuration space

Given that we have two intervals here, for the two coordinates (x, y) then we have 13*13 relations. Taking into account the 4 possible orientations for each design unit, the final number of possible spatial relations is 2704 [1]. Since this is quite a large number and a lot of the relations where too specific (e.g. 192 adjacency relations), a small subset of those relations was maintained, implemented in the WRIGHT system and is listed in [6]. Some examples of relations are spatial overlap relations (e.g. inside, complete-overlap), adjacency, position, alignment, distance. Again, the approaches and algorithms which the authors experimented on are extensively described in [6] and consist of backtracking search in the search space defined by the disjunctive constraints.

5.1.3 Minkowski Operations for Space Planning

A geometric approach for solving the spatial planning problem, is moving from real space to *configuration space*. This transformation is based on the notion that it would be easier to solve a problem of placing an object O_1 within another object O_2, if the object O_1 to be placed was shrunk to a reference point, while O_2 was transformed accordingly to O_2', so that if the reference point is safely placed in the transformed O_2' object this would be equivalent to object O_1 being placed inside O_2, while if the reference point is not inside the O_2' object then O_1 is not completely inside O_2. This means that if O_2 has holes inside (or obstacles for the space planning problem) the holes should grow accordingly [5] (Fig. 5.3).

An approach presented in [42], solves the spatial planning problem by using the Minkowski operations (addition and decomposition, Section 2.3) to achieve this. More specifically, the Minkowski decomposition is used to shrink the object, while the Minkowski addition is used to grow holes or grow the obstacles.

Assuming that the entire space is the set S, the obstacle (or hole) is the object H and a placement is required for object B. The authors of [42], show how to find the set of *safe positions* to place B as well as the *forbidden region* where B can not be placed. If S' is the entire allowed space except the obstacles then:

$$S' = S \cap H^c$$

[5]This approach has also found application in robot path planning, since it determines the possible locations of a robot avoiding obstacles

where H^c is the complement of H, it is proven [42] that the feasible region F is:

$$F = S' \ominus B = (S \cap H^c) \ominus B$$

The complete proof can be found in [42], while a complete scene layout system which builds on this method is presented in the next section.

5.2 Object Placement for Scene Composition

The area of scene layout and object placement has found application in the field of 3D virtual environments and modelling, where the user positions objects in the environment where some constraints exist. It is quite relevant to the area of space planning while at some occasions this problem is degenerated to a spatial planning problem. There has been a lot of research work on topics such as avoiding collisions in environments, placing objects in virtual environments and constraining the user in interactive user interface systems. Almost all the systems form constraint problems and use constraint solvers to solve the problem. Mainly, these systems differ on implementation level, for example on the type of algorithms implemented and the freedom of constraints they allow. ‘In this ISO, we are mostly interested in work that directly relates to automatically generating a scene. Maintaining this scope, we will look at some methods that actually layout a scene automatically: A constraint based method that uses geometry to place objects 5.2.1, and an evolutionary method 5.2.2.

5.2.1 Constraint Based Object Placement for Scene Composition (CAPS)

The system presented by K. Xu in [94], uses a combination of methods to layout a scene based on some user preferences. The scene uses notions described in Section 2.3 (the Minkowski operators) and Section 5.1.3 [42] where an approach to spatial planning with the Minkowski operations is presented.

5.2.1.1 Placement Constraints

In the CAPS system [94], the user can place some constraints on where he wants some objects to appear or not. The constraints defined are the following:

- Surface Constraints: A surface constraint which is related to an object, defines how that object will be positioned. The constraint is formed by a supporting surface and a flag, which determines if the placement of the object is exact. If the placement is exact then the actual coordinates are given. If not, at least one container polygon is given, so that the object is placed inside. Optionally, some areas can be defined that are forbidden.
- Proximity Constraints: They are defined with respect to another polygon (b), and can have the value NEAR if the associated polygon is to be placed within b, or AWAY if the polygon is to be placed outside of b.
- Support Constraints: Which define if a given polygon can support other polygons (other objects can appear above), or if the given polygon can be supported by other polygons (other objects can appear below), using two boolean flags. The flags are CanSupport, CanBeSupported.

5.2.1.2 Constraint Spatial Planning

The system, uses geometric 2D spatial planning (Section 5.1.3, [42]), incorporating the constraints mentioned above. Assuming that S is the surface, F is the forbidden area and O is the object to be placed, then the problem is reduced to finding all the translations of O, O^t such that:

$$O^t \subseteq S \wedge O^t \cap F = \emptyset$$

That is, all the translations which are completely inside the surface and do not intersect with the forbidden area. It is also important to note that the system described is a system which deals with 3D objects, but the spatial planning solution it incorporates deals with only with 2D elements. The system actually uses the *footprints* of the actual objects, which are defined as the convex hull[6] of the projection of the object onto the ground. To take into account the constraints, the system proceeds as follows:

- Surface Constraints define container polygons and optionally forbidden regions. O is considered to be the current object under placement, S the union of container polygons, F the union of forbidden areas.
- Proximity constraints define proximity polygons. If a proximity constraint (for a proximity polygon P) is set to NEAR, then the S set is now defined as $S \cap P$, while if it is set to AWAY, then the forbidden area F becomes $F \cup P$.
- Support Constraints: If an object can not support other objects (CanSupport flag is false), then its footprint is forbidden area for other polygons. If the CanBeSupported flag is false, then during placement of this object the footprints of other objects are included as forbidden areas. This can be seen as a weakness of the system, since this disallows placing objects under other objects, since the footprints would overlap (e.g. placing shoes under bed).

Having taken into account the constraints, the resulting sets are then posed as a spatial planning problem. To reach the desired result, we take into account that given two point sets A and B, the set of *forbidden translations* T_f can be computed using the Minkowski sum [52], such that $B^t \cap A \neq 0$ for $t \in T_f$, while the set of *allowed translations*, T_p such that $B^t \subseteq A, t \in T_p$ can also be calculated [42]. In order to solve the space planning problem, the system first determines the point sets O, S and F and then set of safe translations is computed at follows: The system first calculates the set of permitted translations, for which $O^t \subset S$, $t \in T_p$, then the set of forbidden translations, $O^t \cap \neq \emptyset$, $t \in T_f$. The set of safe translations is now $T_p - T_f$. It is noted that since the system is dealing with convex 2D shapes (footprints) the calculation of the Minkowski operations is much more efficient.

5.2.1.3 Semantic Constraints

To organise the application of semantic constraints, the system uses classes under which objects fall. The semantic constraints are independent of the geometry of the objects. Each class in the system, holds a set of parent classes (which can appear under objects of instances if this class) and a set of child classes (which appear on top of instances of this class). Also included are the CanBeSupported and CanSupport constraints, which are valid only for objects which do not belong in the parent or child classes.

5.2.1.4 Scene Layout Procedure

The procedure of generating a scene, starts by determining a surface S on which to place an object O. The surface is chosen from the parents of S class. The forbidden regions are identified, and the spatial planning problem is solved. If the safe translation set is empty (i.e. no positions exist for the placement), then a different surface is chosen, although it must be noted that this system does not backtrack, i.e.

[6]A convex shape is a shape for which any line that begins from one point of the shape and ends up in another does not pass outside the shape itself. A convex hull is the smallest such shape for a given set of points

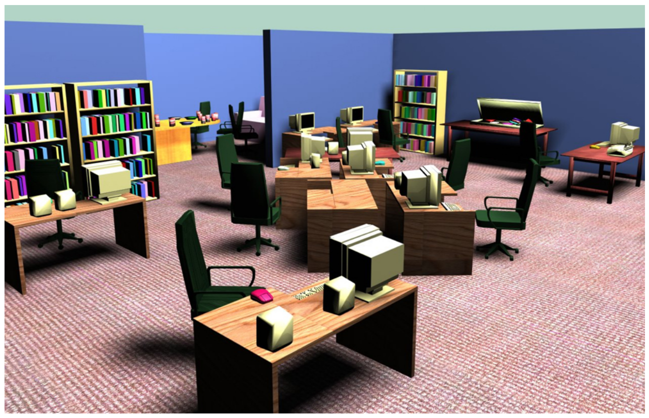


Figure 5.4: A scene layout using the CAPS system [94], using L-systems

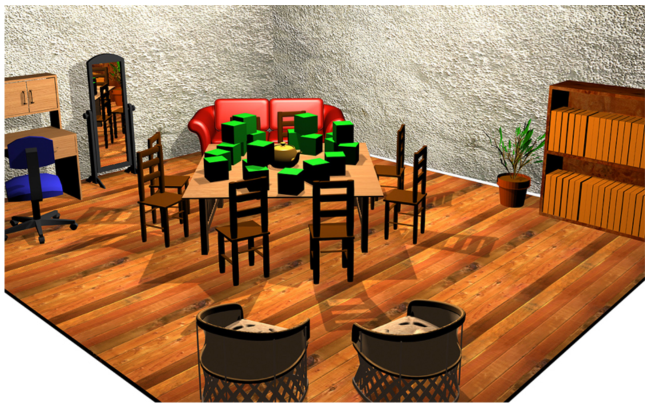


Figure 5.5: Scene layout using the genetic algorithm based system presented in 5.2.2

no previously placed objects will be removed if no possible positioning is found. If a position is found, then the object is positioned above the surface using pseudo-physics (emulate the drop of the object). An example of a scene layout is presented in Fig. 5.4.

5.2.2 Object Layout using Genetic Algorithms

In [72], Sanchez et al. presented an approach where genetic algorithms are used to solve the 3D object layout problem. The system allows topological constraints (object is on another object), metrics constraints (distance of one object from another), angular constraints (object around another object), orientation constraints and size constraints, while maintaining a realistic scene. Hierarchies are also used in order to express hierarchical constraints. Logical operators are also used in order to combine the constraints. Objects are placed in a limited 3D space, and are identified using a hierarchical set of bounding boxes. A bounding box is defined by its 3D position, the orientation angle (around the height axis) and the size parameters.

5.2.2.1 The Constraints

To ensure the realism of the final scene, the authors of [72] have imposed *physical constraints*, which consist of ruling out objects that are not on a surface (ground or other object), and forces the objects not to overlap. The geometric constraints in the system, are imposed between two objects: the object under placement (target object, O_t) and another object, which is used as a landmark (O_l). The two objects are related with a spatial preposition S_P and some parameters, as follows:

$$O_t\ S_P(parameters)\ O_l$$

The logical operators that combine the constraints are AND, OR, XOR and NOT. Since the NOT operator is included in the logical operators permitted by the system, and so negative constraints can be imposed. The allowed geometric constraints include:

- Zone constraints, which define the zone of the object to be placed in relevance to another, e.g. LEFT, ON.
- Orientation constraints which limit the orientation of the object to be placed, for example if it is constraint to be facing another object

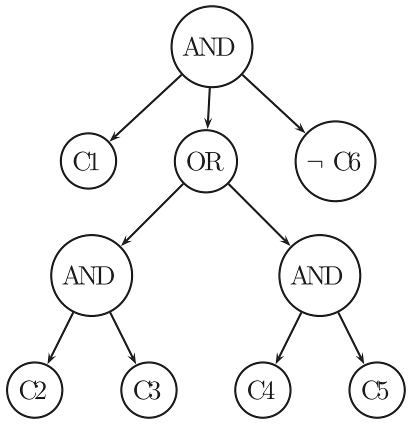


Figure 5.6: Tree form of constraints

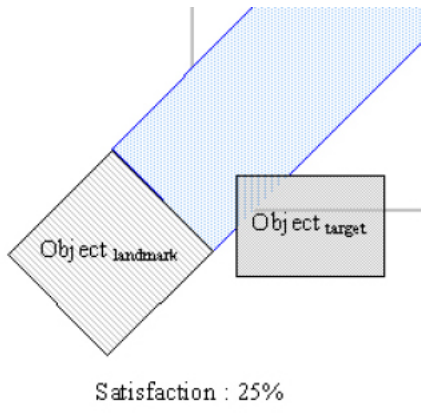


Figure 5.7: Satisfaction returned by the fitness function for a zone constraint, where the object target should be on the object landmark

- Distance constraints define the distance between two objects and the size constraint, which is the only constraint that is unary, defining the permitted size of an object.

5.2.2.2 Problem Formulation

To give an example of how a problem is formulated in this system, we can take an example of a set of constraints. For the textual description ”the plate on the table facing one of the two chairs but not on the vase” can be translated to the following six constraints:

- C1: Plate ZONE(on) table
- C2: Plate ORIENTATION(pi) chair #1
- C3: Plate ZONE(in_front) char #1
- C4: Plate ORIENTATION(pi) chair #2
- C5: Plate ZONE(in_front) chair #2
- C6: Plate NOT ZONE(ON) vase

The constraints are then translated into a tree structure, as seen in Fig 5.6. The fitness function that determines the amount of satisfaction for each one of the constraints is then weighted according to the priority of each constraint. For the example above, the fitness function for the object plate would be of the following form:

$$f_{object} = AND(p_{c1} * C1, OR(AND(p_{c2} * C2, p_{c3} * C3), AND(p_{c4} * C4, p_{c5} * C5)), p_{c6} * NOT(C6))$$

where Ci represents the fitness function for each constraint for this specific problem, and p_{ci} represents the priority factor of the constraint i.

5.2.2.3 Solution & Layout

To solve the problem, each instance of a possible solution is evaluated by a fitness function. Members of the population are basically objects which are to be placed in the scene and the constraints that they have between them. The *gene*, or the members of the individuals which are part of the population, are objects which belong to the scene and are related to a constraint (e.g. the plate is the gene in a constraint which reads plate on table). A fitness function for every geometric constraint is applied to the objects, and a percentage of satisfaction is returned which is normalised by the priority of the constraint. The final fitness is the sum of the fitness function multiplied by the priority, averaged by the sum of priorities:

$$F_{scene} = \frac{\sum_{object \in scene}(priority_{object} * fitness_{object})}{\sum_{object \in scene}(priority_{object})}$$

5.3 Discussion

In this chapter, we have described systems that generally lay out objects in a certain space, given a set of constraints to conform to. Some of the approaches described formulate CSPs and then attempt to solve them, such as the GCSP and Disjunctive CSP solutions we have described. It is noted that the general CSP problem is NP-complete, and the complexity of each approach depends on the limitations of possible constraints imposed. The same is true for a general optimisation problem, where the constraints would be described as equations and a cost function would be optimised. We have also seen methods that rely on geometry in order to determine safe areas for placement of objects, such as the Minkowski sum approach. In the approach described above, the convex hulls of projections of objects were used. This means that the Minkowski operations were executed on convex shapes, something that greatly reduces the complexity (the Minkowski sum for two convex shapes is $O(m+n)$ where m,n are the vertices of the two polygons, while for two non-convex shapes it takes $O((mn)^2)$). We have also described an approach which uses fitness functions and evolutionary algorithms in order to optimise the placement of objects in scenes, according again to sets of constraints. The reader could be interested in other relevant research, such as generating scenes from natural language recognition [74, 27], modelling with constraints [55], user interfaces with constraints [86] or other uses of genetic algorithms in design automation [55, 89].

Chapter 6

Extending the Constraint Logic Programming approach for Automated Scene Generation in the Visual Arts

6.1 Equidistant Constraint

The system described in Section 3.3 [24], attempts to infer the intentions of the user, and form constraints in order to layout an artistic scene. A constraint that could be inferred from the scene, is an equidistant constraint, where some of the rectangles placed in the scene share equal distance from a certain point in the scene. The definition of equidistant points, actually defines that the points we are looking for all lie in the circumference of a circle. In this context, the problem lies in detecting points that do belong in the circumference of a circle, or satisfy the equation:

$$(x - x_c)^2 + (y - y_c)^2 = r^2$$

where (x, y) are the coordinates of the point, (x_c, y_c) are the centre coordinates of the circle and r is the radius. An initial thought for dealing with this problem, was constructing a Voronoi diagram [29]. A Voronoi diagram (in 2D) is a diagram which given some input points, separates the 2D space into regions called cells. Each cell (c_i) contains one of the input points (i) called *sites*, and the 2D points that belong to the area of the cell have the following property:

$$\forall p = (x, y), (x, y) \in c_i iff \ \nexists x_j, j \neq i, dist(x_j, p) < dist(x_i, p)$$

That is, if the distance of a point p to the input point x_i is smallest than the distance of p to any other input point, then point p belongs in the cell of x_i. If a point p shares equal distance with two different input points, then a Voronoi edge is formed. If a point p equal distance with three or more different input points then a Voronoi point is created. The last property seems useful for detecting equidistant points, but the problem lies in the fact that the Voronoi point which is created for three or more equidistant input points is destroyed when other input points are inserted between the previous input points. The property of the Voronoi diagram is that a Voronoi point shares equal distance from three or more input points, but it is not the case that a Voronoi point will exist for all equidistant points (See Fig. 6.1). A Voronoi diagram would be suitable for detecting only such relations between neighbouring input points. That is why we will use the Hough transform, a technique which has been primarily used in Computer Vision [78, 4], in order to detect equidistant points.

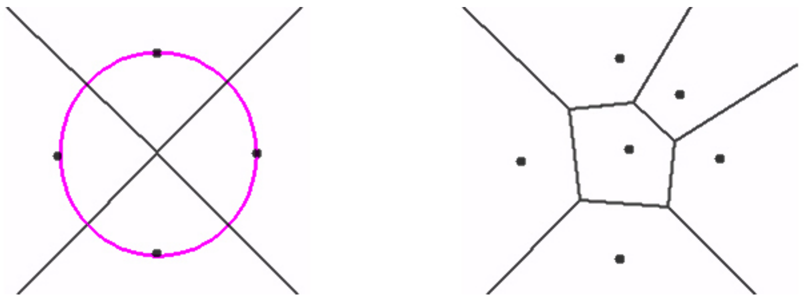


Figure 6.1: In the left image, we can see how a Voronoi point is created, where a point p shares equal distance from the four input points (the circle is added just to display that these four points are indeed equidistant). On the right, the image presents how the diagram changes with the insertion of new sites and how this does not allow us to capture the relation between the four originally inserted points.

6.1.1 Detecting Equidistant Points with the Hough Transform

The Hough transform was initially used to detect lines in images, while it was later generalised to detect circles and arbitrary shapes. To detect an arbitrary circle of radius r, which may be formed by some subset of a set of given points, the Hough transform for circles uses the circle equation:

$$(x - x_c)^2 + (y - y_c)^2 = r^2$$

For each of the input points $p_i = (x_i, y_i)$, a circle of radius R is constructed using the point p_i as a centre. The equation is now:

$$(x - x_i)^2 + (y - y_i)^2 = r^2$$

The result is shown in Fig. 6.2. We can see that the point where the constructed circles intersect is the actual centre of the original circle. To detect this point, a voting scheme is used in the Hough transform. The space is quantified into a 2D grid. Then, for each of the constructed circles from the points, a vote is given in each cell of the grid if that circle intersects with the cell (Fig. 6.3). Usually, a 2D accumulator array is used in order to hold the votes for each cell and then peaks are found that are credited as circle centres. If many circles are present then false peaks may be created, which can be filtered out by matching to points in the original image. If the radius R is not known, then a 3D array should be used.

In Algorithm 2, we can see how the pseudo-code of a possible implementation of the Hough transform. The *intersects* method checks for an intersection of a cell with a circle, and can be implemented to allow a tolerance factor (because the user can not be 100% precise). The result can be filtered as desired, for acquiring for example more than 3 equidistant points, and then verified against the original points to make sure that the actual circles have been detected. The complexity of the Hough transform depends on the number of radii examined, the number of cells in the grid and the number of input points. The constraint as would be implemented in the original system is presented in the next code fragment:

```
% Constraint for circle: (X - 5)^2 + (Y - 6)^2 = 5^2
% limiting the placement of object S around the circumference
% of the circle, allowing a tolerance e
unary_constraint(circle0,S):-
        get_x(S,X), get_y(S,Y),
        (X-5)**2 + (Y-6)**2 - 25 #> -e, (X-5)**2 + (Y-6)**2 - 25 #< e
```

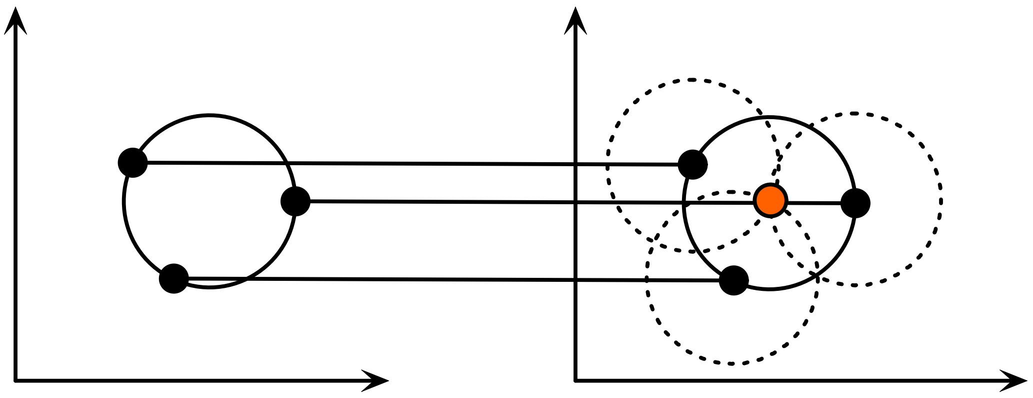


Figure 6.2: Figure on the right, shows three points which are equidistant from another point, i.e. belong in the circumference of a circle centred at that point. On the right, we can see the Hough transform of those points.

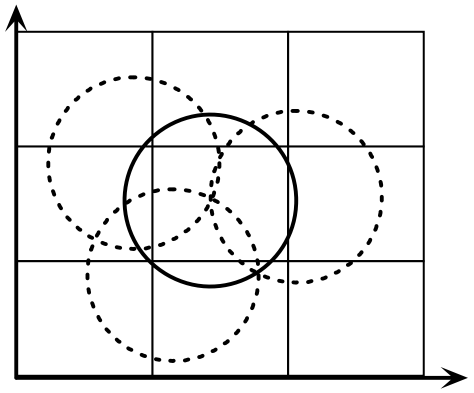


Figure 6.3: Simple quantification of the space resulting from the Hough transform in Fig. 6.2. The middle cell will acquire three votes since three circles intersect with it, while the lower left cell will only have one vote.

Algorithm 2 Apply the Hough Transform for Circles

```
Require: $R$ is a list of radii to search for, $P$ is the set of points in the original image
    H ← 0 {H is a 3D array}
    for $r = 1$ to length($R$) do
        for $p = 1$ to length($P$) do
            $(x_p, y_p) \leftarrow P[p]$
            for all $c$ where $c$ is a cell in 2D grid do
                if intersects($(x - x_p)^2 + (y - y_p)^2 = R[r], c$) then
                    H[c.i,c.j,R[r]]++
                end if
            end for
        end for
    end for
    res ← findPeaks(H)
    validate&return(res)
```

6.2 Using Soft CLP Constraints

Since the target application of the system is a visual art applications, experimenting with hierarchical constraints or soft constraints, which impose a priority on the satisfaction, could result in more variety in the resulting scenes.

In general, a CLP program can be posed as an optimisation problem by calling the predicate *minimize* in order to pose the problem as an optimisation problem. For example, a constraint can impose a cost in case of not satisfaction, and then minimize a weighted some of certain constraints.

```
optim(X,Y,Z):- X in 1..3, Y in 3..5, Z in 0..15,
        constr(X,Y,Z),labeling([minimize(Z)],[Z|[X,Y]]).
constr(X,Y,Z):-pcons_1(X,Y,C1), pcons_2(X,Y,C2),
        Z #= 10*C1+5*C2.
pcons_2(X,Y,C):- X#=Y #<=>CT,C#=1-CT.
pcons_1(X,Y,C):- X#\=Y #<=>CT,C#=1-CT.
```

In the above code fragment, a combination of propositional constraints and the minimise predicate is used, in order to construct an optimisation problem. More specifically, the predicate *optim* first calls the *constr* predicate, which imposes the constraints and returns a variable Z which is actually the cost function. Then, the *labeling* predicate, which is the predicate the constraint library uses to assign values to variables, while a request is made that the value of Z should be minimised. The propositional constraint used is <=>, which is synonymous for $\Leftrightarrow$ (iff) and is true iff both the constraints on left, right hand are true, or both are false. In this way, if for example the constraint $X\# = Y$ is satisfied, then CT gains the value 1, else it acquires the value 0. The variables used in the final cost formula are the cost variables subtracted from 1, in order to have the value 0 if the constraint is satisfied and 1 if not. Each of the cost variables is multiplied by the weight or cost of not satisfying the constraint. The solution the sicstus prolog fd solver returns is $X = 1, Y = 3, Z = 4$, that is the solver chose that the inequality constraint is imposed, since it would cost 10 to fail to impose it, while it costs only 5 not to impose the equality rule.

Chapter 7

Discussion & Conclusions

In the previous chapters of this report, we have described automated scene generation methodologies, targeting mostly on the domains of visual arts and architecture, by providing in-depth description of such approaches and systems while comparing some of their features, e.g. in terms of variety for visual arts, or ability to adjust and express the spatial requirements in the architecture domain. We have then described some techniques for solving the spatial planning and object layout problem, considering that such methods may be adapted and used in scene generation applications. Also, we have described some possible expansions to the CLP system for automated scene generation, while noting other possible research in these areas throughout the chapters. As we have noted in Chapter 1, in our description we have focused on how these systems receive their input, how they formulate the problem of generating a scene and how they solve this problem. We will now attempt to categorise the different input methods and the different methodologies used to generate the scene in the systems we have described.

We have seen various ways of feeding input to the systems. In more detail, we have seen systems which receive an implicit form of input, for example by inferring user intentions [24]. We have described systems that attempt to invent relations between the scene objects - thus limiting their input to the form of objects in the scene [23], as well as systems which explicitly receive desired or required relationships that are defined for objects in the final scene, such as city generation based on L-systems [64], which received geographical data, or the other object layout and space planning systems described. There are also systems which are designed to produce a certain result and their input is actually the first symbol or shape that they are initiated with, such as approaches based on shape grammars described in Chapter 4.

Concerning the different methodologies used in order to generate a scene, we can categorise them into the following groups:

- CSP Formulation: These systems formulate a CSP out of the specifications for the final output scene. The problem can be posed as a constraint logic programming problem, or can be solved by constraint solvers, which are applications that apply specific algorithms in order to provide a solution to the problem. Some systems that we have described which fall under this category are the constraint based approach to ASG in VA [24] and two of the approaches we described for spatial planning [6, 13].
- Evolutionary/Optimisation: Approaches that fall into this category can apply evolutionary algorithms in order to evolve a scene which might be randomly arranged into a scene which is acceptable, by optimising a fitness function. Approaches described are the evolutionary approach to scene generation for visual arts [23] and the object layout with genetic algorithms approach [72], while an

approach mentioned that uses numerical optimisation (optimise a certain cost function) is described by Michalek et al. [60]. Evolutionary approaches not only in terms of applying algorithms but also as imitating nature and the evolution of nature have been found in fields such as evolutionary art (Section. 3.4) and evolutionary architecture (Section. 4.6).

- Geometry: We have described a geometric technique which applies to the problem of laying out objects in a 3D scene [94], or solving the 2D spatial planning problem [42].
- Procedural: This common form of generating scenes is usually based on formal systems such as shape grammars and L-systems. Essentially, the methods that we have described in the automated architecture generation all make use of such systems. It is noted though, that these systems do not conform to blind procedural application of rules but can use extensions such as stochastic rules (probability of applying a rule) in order to increase the variety of the produced scene. Also, the AARON [20] system models in a procedural way, only the actual modelling procedure is hardcoded into the source code of the application. As we have mentioned in Section 2.5, L-systems have been extensively applied in modelling plants [68].

7.1 Future Work

In this section we will present some of the possible future work that may build on this survey:

- The most apparent work that can be done, is the implementation of the equidistant constraint and the hierarchical constraints as described in Chapter 6, for the CLP approach to ASG.
- Further experimentation can be done on interchanging methodologies covered in this survey and adjusting them to new applications. An example is the adoption of methods such as shape grammars and L-systems in current research in automated scene generation for visual arts, or even applying alternative search methods in an attempt to increase the variety of the scenes produced by the systems in searching the search space, e.g. applying simulated annealing in the machine learning approach to ASG in VA or other methods of backtracking in the CLP approach (Section 3.5).
- Another possibility for future research is to apply new methodologies in the automated generation of 3D architecture models: The approaches we have seen are basically procedural and are based on shape grammars and L-systems, since they are strict systems that can capture the analogous architectural formalities. There is space for experimentation in incorporating other methodologies such as geometric and constraint based, while building on the foundations of evolutionary architecture and art, evolutionary methods can be added to architecture generation systems.
- Other surveys that may focus on areas such as collision detection, placement of objects using physics, constraint solvers and their optimisation, human machine interaction and user interface design, always from a scene generation perspective.
- One of the long term goals which will build on this survey is the incorporation of Abductive Logic Programming (ALP), a reasoning technique that generates explanations for observations, into automated scene generation applications. That is why a survey on the topic of ALP is already planned.

Bibliography

[1] J. F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, 1983.

[2] J. Andersson, S. Andersson, K. Boortz, M. Carlsson, H. Nilsson, T. Sjöland, and J. Widen. Sicstus prolog user"s manual. Technical report, 1993.

[3] T. Bäck. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms.* Oxford University Press, Oxford, UK, 1996.

[4] D. H. Ballard. Generalizing the hough transform to detect arbitrary shapes. *Pattern Recognition*, 13(2):111–122, 1981.

[5] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming : An Introduction : On the Automatic Evolution of Computer Programs and Its Applications (The Morgan Kaufmann Series in Artificial Intelligence).* Morgan Kaufmann Publishers, November 1997.

[6] C. A. Baykan and M. S. Fox. Spatial synthesis by disjunctive constraints, 1993.

[7] CGAL Editorial Board. *CGAL User and Reference Manual*, 3.3 edition, 2007.

[8] T. Broughton, P. S. Coates, and H. Jackson. Exploring three-dimensional design worlds using lindenmeyer systems and genetic programming. In Peter Bentley, editor, *Evolutionary Design Using Computers*, chapter 14, pages 323–341. Academic press, London, UK, 1999.

[9] H. Buelinckx. Wren"s language of city church designs: a formal generative classification. *Environment and Planning B: Planning and Design*, 20:645–676, 1993.

[10] C. Caldwell and V. S. Johnston. Tracking a criminal suspect through "face-space" with a genetic algorithm. In *Proceedings of the Fourth International Conference on Genetic Algorithm*, pages 416–421. Morgan Kaufmann Publisher, July 1991.

[11] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *PLILP '97: Proceedings of the 9th International Symposium on Programming Languages: Implementations, Logics, and Programs*, pages 191–206, London, UK, 1997. Springer-Verlag.

[12] E. E. Catmull. *A subdivision algorithm for computer display of curved surfaces.* PhD thesis, 1974.

[13] P. Charman. Solving space planning problems using constraint technology, 1993.

[14] S-C Chiou and R. Krishnamurti. The grammar of taiwanese traditional vernacular dwellings. *Environment and Planning B: Planning and Design*, 22:689–720, 1995.

[15] N. Chomsky. *Syntactic Structures.* Mouton, The Hague, 1957.

[16] P. Coates and D. Makris. Genetic programming and spatial morphogenesis. In *AISB Symposium on Creative Evolutionary Systems*, pages 105–114, Edinburgh College of Art and Division of Informatics, University of Edinburgh, 6-9 April 1999.

[17] H. Cohen. Brother georgios kangaroo. `http://www.kurzweilcyberart.com/aaron/pdf/brothergkanga.pdf`.

[18] H. Cohen. Colouring without seeing: a problem in machine creativity. `http://www.kurzweilcyberart.com/aaron/pdf/colouringwithoutseeing.pdf`.

[19] H. Cohen. The further exploits of aaron, painter. `http://www.kurzweilcyberart.com/aaron/pdf/furtherexploits.pdf`.

[20] H. Cohen. Harold cohen, essays on aaron. `http://crca.ucsd.edu/~hcohen/`.

[21] H. Cohen. Saf's ask the scientists: Harold cohen's q & a. `http://www.pbs.org/safarchive/3_ask/archive/qna/3284_cohen.html`.

[22] S. Colton. *Automated Theory Formation in Pure Mathematics.* Springer-Verlag, 2002.

[23] S. Colton. Automatic invention of fitness functions with application to scene generation. In *EvoWorkshops*, pages 381–391, 2008.

[24] S. Colton. Experiments in constraint-based automated scene generation. In *Proceedings of the fifth international workshop on computational creativity*, 2008.

[25] S. Colton, A. Bundy, and S. Bridge. Mathematical reasoning group.

[26] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* The MIT Press, September 2001.

[27] B. Coyne and R. Sproat. Wordseye: an automatic text-to-scene conversion system. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 487–496, New York, NY, USA, 2001. ACM.

[28] S. Das, T. Franguiadakis, M. E. Papka, T. A. Defanti, and D. J. S. A genetic programming application in virtual reality. In *In Proceedings of the first IEEE Conference on Evolutionary Computation*, pages 480–484. IEEE Press, 1994.

[29] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer, 2000.

[30] F. Downing and U. Flemming. The bungalows of buffalo. *Environment and Planning B: Planning and Design*, 8:269– 293, 1981.

[31] J. P. Duarte. Towards the mass customization of housing: the grammar of siza's houses at malagueira. *Environment and Planning B: Planning and Design*, 32(3):347–380, May 2005.

[32] C. M. Eastman. Heuristic algorithms for automated space planning. In *Proc. of the 2nd IJCAI*, pages 27–39, London, UK, 1971.

[33] U. Flemming. The secret of the casa guiliani frigerio. *Environment and Planning B*, 8:87–96, 1981.

[34] U. Flemming. More than the sum of its parts: the grammar of queen anne houses. *Environment and Planning B: Planning and Design*, 14:323–350, 1987.

[35] J. G. Fletcher. A program to solve the pentomino problem by the recursive use of macros. *Commun. ACM*, 8(10):621–623, 1965.

[36] J. D. Foley, A. van Dam, S.K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice in C.* Addison-Wesley Professional, second edition, August 1995.

[37] J. Frazer. *An Evolutionary Architecture.* Architectural Association Publications, 1995.

[38] E. C. Freuder. Synthesizing constraint expressions. *Commun. ACM*, 21(11):958–966, 1978.

[39] J. Gips G. Stiny. Shape grammars and the generative specification of painting and sculpture. In *Information Processing 71, C. V. Freiman, Amsterdam: North Holland*, pages 1460–1465, 1972.

[40] J. S. Gero. *Design Computing and Cognition '06.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[41] J. S. Gero, V. A. Kazakov, Department Of Architectural, and Design Science. An exploration-based evolutionary model of generative design process. In *Microcomputers In Civil Engineering*, pages 209–216, 1996.

[42] P. K. Ghosh. A solution of polygon containment, spatial planning, and other related problems using minkowski operations. *Comput. Vision Graph. Image Process.*, 49(1):1–35, 1990.

[43] J. Graf and W. Banzhaf. Interactive evolution of images. In *Evolutionary Programming*, pages 53–65, 1995.

[44] N. L. R. Hanson and A. D. Radford. On modelling the work of the architect glenn murcutt. *Design Computing*, pages 189–203, 1986.

[45] K. Honda and F Mizoguchi. Constraint-based approach for automatic spatial layout planning. In *CAIA '95: Proceedings of the 11th Conference on Artificial Intelligence for Applications*, page 38, Washington, DC, USA, 1995. IEEE Computer Society.

[46] W. E. Howden. The sofa problem. *The Computer Journal*, 11(3):299–301, November 1968.

[47] M. W. Jones. Direct surface rendering of general and genetically bred implicit surfaces. In *Proc. 17th Ann. Conf. of Eurographics (UK Chapter*, pages 37–46, 1999.

[48] T. W. Knight. The forty-one steps. *Environment and Planning B: Planning and Design*, 8:97–114, 1981.

[49] T. W. Knight. Transformation of the meander motif on greek geometric pottery. *Design Computing*, 1:29–67, 1986.

[50] H. Koning and J. Eizenberg. The language of the prairie: Frank lloyd wright"s prairie houses. *Environment and Planning B*, 8:295–323, 1981.

[51] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation (2nd Edition).* Prentice Hall, 2 edition, August 1997.

[52] Z. Li. *Compaction algorithms for non-convex polygons and their applications.* PhD thesis, Cambridge, MA, USA, 1995.

[53] A. Lindenmayer. Mathematical models for cellular interactions in development ii. simple and branching filaments with two-sided inputs. *Journal of Theoretical Biology*, 18(3):300–315, March 1968.

[54] P. Machado. Giving colour to images. In *Proc. AISB 2002 Symposium on AI and Creativity in the Arts*, 2002.

[55] T. Masui. Graphic object layout with interactive genetic algorithms. In *In Proceedings of the 1992 IEEE Workshop on Visual Languages*, pages 74–80. IEEE Computer Society Press, 1992.

[56] P. McCorduck. *Aaron's code.* W. H. Freeman & Co., New York, NY, USA, 1991.

[57] J. McCormack. Interactive evolution of L-system grammars for computer graphics modelling. In D. Green and T. Bossomaier, editors, *Complex Systems*, pages 118–130. IOS Press, 1993.

[58] J. McCormack. Impossible nature: The art of jon mccormack, 2004.

[59] R. Mech and P. Prusinkiewicz. Visual models of plants interacting with their environment. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 397–410, New York, NY, USA, 1996. ACM Press.

[60] J. J. Michalek, R. Choudhary, and P. Y. Papalambros. Architectural layout design optimization, 2001.

[61] T. M. Mitchell. *Machine Learning.* McGraw-Hill, New York, 1997.

[62] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. V. Gool. Procedural modeling of buildings. *ACM Trans. Graph.*, 25(3):614–623, 2006.

[63] M. Özkar and S. Kotsopoulos. Introduction to shape grammars. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, pages 1–175, New York, NY, USA, 2008. ACM.

[64] Y. I. H. Parish and P. M&#252;ller. Procedural modeling of cities. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 301–308, New York, NY, USA, 2001. ACM Press.

[65] C. E. Pfefferkorn. A heuristic problem solving design system for equipment or furniture layouts. *Commun. ACM*, 18(5):286–297, 1975.

[66] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction (Monographs in Computer Science).* Springer, August 1985.

[67] P. Prusinkiewicz, M. James, and R. Měch. Synthetic topiary. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 351–358, New York, NY, USA, 1994. ACM.

[68] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants (The Virtual Laboratory).* Springer, October 1991.

[69] J. Romero and P.l Machado, editors. *The Art of Artificial Evolution: A Handbook on Evolutionary Art and Music.* Natural Computing Series. Springer, 2008.

[70] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming.* Elsevier, 2006.

[71] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1995.

[72] S. Sanchez, O. Le Roux, H. Luga, and V. Gaildrat. Constraint-based 3d-object layout using a genetic algorithm. In *Interactive Computer Graphics and Artificial Intelligence*, 2003.

[73] T. Schnier and J. Gero. From frank lloyd wright to mondrian: Transforming evolving representation. In *Adaptive Computing in Design and Manufacture*, pages 207–219. Springer Verlag, 1998.

[74] L. M. Seversky and L. Yin. Real-time automatic 3d scene generation from natural language voice and text descriptions. In *MULTIMEDIA '06: Proceedings of the 14th annual ACM international conference on Multimedia*, pages 61–64, New York, NY, USA, 2006. ACM.

[75] K. Sims. Artificial evolution for computer graphics. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 319–328, New York, NY, USA, 1991. ACM.

[76] K. Sims. Interactive evolution of equations for procedural models. *The Visual Computer*, 9(8):466–476, 1993.

[77] A. R. Smith. Plants, fractals, and formal languages. *SIGGRAPH Comput. Graph.*, 18(3):1–10, 1984.

[78] M. Sonka, V. Hlavac, and R. Boyle. *Image Processing, Analysis, and Machine Vision.* Thomson-Engineering, 2007.

[79] Springer-Verlag. Amast'98. In A. M. Haeberer, editor, *Proceedings of the 7th International Conference on Algebraic Methodology and Software Technology*, London, UK, 1999.

[80] G. Stiny. Ice-ray: a note on chinese lattice design. In *Environment and Planning B4*, pages 89–98, 1977.

[81] G. Stiny. A note on the description of designs. *Environment and Planning B: Planning and Design*, 8:257● 267, 1981.

[82] G. Stiny. The algebras of design. *Research in Engineering Design*, 2(3):pp. 171–181, 1991.

[83] G. Stiny and W. J. Mitchell. The palladian grammar. *Environment and Planning B*, 5:5–18, 1978.

[84] G. Stiny and W. J. Mitchell. The grammar of paradise: on the generation of mughul gardens. *Environment and Planning B: Planning and Design*, 7:209–226, 1980.

[85] R. Stouffs and M. Wieringa. The generation of chinese ice-ray lattice designs on 3d surfaces. In *Communicating Space(s) (eds. V. Bourdakis and D. Charitos)*, pages pp. 316–319, 2006.

[86] I. E. Sutherland. Sketchpad a man-machine graphical communication system. In *25 years of DAC: Papers on Twenty-five years of electronic design automation*, pages 507–524, New York, NY, USA, 1988. ACM.

[87] S. Todd and W. Latham. *Evolutionary Art and Computers.* Academic Press, Inc., Orlando, FL, USA, 1994.

[88] T. Unemi. SBART 2.4: breeding 2D CG images and movies and creating a type of collage. In *Third International Conference on Knowledge-Based Intelligent Information Engineering Systems, KES 1999*, pages 288–291, Adelaide, Australia, 31 August-1 September 1999. IEEE.

[89] N. Vassilas, G. Miaoulis, D. Chronopoulos, E. Konstantinidis, I. Ravani, D. Makris, and D. Plemenos. Multicad-ga: A system for the design of 3d forms based on genetic algorithms and human evaluation. In *SETN '02: Proceedings of the Second Hellenic Conference on AI*, pages 203–214, London, UK, 2002. Springer-Verlag.

[90] R. West. Genetics and culture: From molecular music to transgenic art. UCLA, Design and Media Arts Course `http://www.viewingspace.com/genetics_culture/pages_genetics_culture/gc_w05/cohen_h.htm`.

[91] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky. Instant architecture. *ACM Transactions on Graphics*, 22(4):669–677, july 2003. Proceeding.

[92] L. World. Aesthetic selection: The evolutionary art of steven rooke. *IEEE Computer Graphics and Applications*, 16(1):4–5, 1996.

[93] G. Wyszecki and W. S. Stiles. *Color Science: Concepts and Methods, Quantitative Data and Formulae (Wiley Series in Pure and Applied Optics)*. Wiley-Interscience, 2 edition, August 2000.

[94] K. Xu. Constraint-based automatic placement for scene composition. In *In Graphics Interface*, pages 25–34, 2002.