Designer-driven 3D Buildings Generated Using Variable Neighborhood Search

Jose M. Peña, Javier Viedma, Santiago Muelas, Antonio LaTorre CeSViMa Supercomputing Center
Universidad Politécnica de Madrid
Madrid, Spain
{jmpena,smuelas,atorre}@fi.upm.es

Luis Peña
University of Technology and Digital Art
Madrid, Spain
luis.pena@u-tad.com

***Abstract*—This paper presents a mechanism to generate virtual buildings considering designer constraints and guidelines. This mechanism is implemented as a pipeline of different Variable Neighborhood Search (VNS) optimization processes in which several subproblems are tackled (1) rooms locations, (2) connectivity graph, and (3) element placement. The core VNS algorithm includes some variants to improve its performance, such as, for example constraint handling and biased operator selection. The optimization process uses a toolkit of construction primitives implemented as "smart objects" providing basic elements such as rooms, doors, staircases and other connectors. The paper also shows experimental results of the application of different designer constraints to a wide range of buildings from small houses to a large castle with several underground levels.**

I. Introduction

The design of virtual scenarios is a key task in the production process of video games. These scenarios sustain part of the narrative process as the immersible world in which player and non-player characters live and interact. On the one hand, game designers require that these virtual scenarios have particular elements or characteristics. On the other hand, there is a full artistic development behind the proper modeling of attractive scenarios. These two considerations make this particular aspect of game development to require tightly coupled interaction to align narrative and visual components of the game. Additionally, the success of multiplayer on-line games, in genres such as Role-Playing Games (RPG) or First Person Shooters (FPS), has required the massive production of virtual scenarios for the gamers to be engaged and to keep them playing.

This situation has encouraged the adoption of *Procedural Content Generation* (PCG) techniques [1], [2] as a mainstream tendency in the industry. PCG uses computational intelligence approaches to assist in the production of game contents. Some games, for instance Skyrim (Bethesda Game Studios, 2011), have achieved impresive results in the development of PCG and support tools in design-time [3].

The present article introduces the combination of a search-based PCG technique based on a heuristic optimization algorithm, Variable Neighborhood Search (VNS), with modeling primitives and libraries in order to produce 3D virtual scenarios, such as buildings and underground structures (dungeons), according to designer's guidelines and restrictions.

The remainder of this paper is organized as follows: Section II surveys the related work on PCG and PCG applied to buildings and levels. Then, Section III introduces the main contribution of the article, the Procedural Building Generation, divided into the Building Constructor Toolkit (the set of modeling primitives and libraries) and the Building Architect Framework (the modular optimization engine to match designer's guidelines and provided building blocks). Section IV presents a set of experiments carried out to automatically generate 3D buildings according to different designer's guidelines. Finally, in Section V the conclusions of this paper are presented.

II. Related work

Although there are different problems in which computational intelligence techniques have been applied in the context of video games, nowadays PCG is one of the most active [4], [2]. PCG has been applied to the production of many different game elements, such as decorative components [5], maps [6], terrains [7], mazes [8], or players [9].

A. PCG in the Generation of Virtual Worlds

The assisted production of virtual worlds has many different levels of detail in which PCG has successfully been applied.

There have been different contributions to produce maps of large regions. For instance, [10] uses a declarative modeling to provide a designer-driven sketch of a fictional area map in which multiple layers (urban, road, water, and landscape) are generated. Software agent approaches have also been applied to generate realistic terrains [7]. In [11], a real-time generation of floor plans based on design parameters is presented. Additionally, some of these techniques have been applied at the level of cities and groups of buildings, [12], [13], using grammar-based procedural mechanisms to define building structures and city landscapes and organization. Multiobjective optimization [6] has been applied to produce 3D level maps according to an interactive fitness evaluation.

PCG has also been proposed as a mechanism to generate buildings and smaller structures. For example, [14] proposes a procedure to model physical constraints to generate feasible buildings (at the structural level). Shape grammars and other semantic elements have been implemented into interactive visual editors for buildings [15]. Constrained-based approaches have also been applied to construct buildings [16] in combination with Bayesian networks to train with real-world data. In the line of purely generative approaches, techniques such as cellular automata [17] have been used. In general all these processes are designer-centric keeping the balance between authorship and automatic content production [18].

Finally, [19] provides an interesting survey of PCG methods for dungeon generation and [20] is a broad range of references from generation of terrains to interiors.

III. Procedural Building Generation

In this paper we present a PCG mechanism based on the combination of (a) a toolkit of mid-level building components that includes a series of smart primitives and some modeling elements, and (b) a search-based PCG engine based on a VNS algorithm that deals with designer's constraints and objectives.

A. General Architecture

The main characteristics of the two aforementioned modules are:

1) **Building Construction Toolkit** (BCT): This toolkit provides a list of primitives of building elements, such as walls, rooms, staircases, windows and doors that are combined in order to produce a given building.
2) **Building Architect Framework** (BAF): This framework allows designers to define a series of constraints in the format of building sketched layout, list of particular rooms to be included and general design considerations and guidelines.

The current implementation mechanism includes a 3DS Max version of the BCT toolkit that produces a series of Maxscipt files that can be processed by Autodesk 3DS Max Design^{TM}[1].

B. Building Construction Toolkit

The Building Constructor Toolkit (BCT) is a set of 3D components for the procedural building construction implemented on top of Autodesk 3DS Max Design^{TM}. These components are object-oriented programmed in Maxscript. Unlike the work of Whiting et al. [14], the components and their combination do not consider dynamic concepts for the stability of the architectures or the underlying physics. The generated buildings have an objective that is purely visual and not physically feasible in structural terms.

BCT toolkit offers the possibility to generate a diversity of constructions attending to the exterior components as well as the interior ones. The BCT is based on the modular construction approach that has been successfully used for design levels in games such as Skyrim [3].

1) Building components: The BCT components are implemented as parameterized 3D primitives, managed and modeled by Maxscript objects using Boolean and BREP (Boundary Representation) operations. These components are "smart objects", publishing methods to manage the relationships among them, such as alignment, positioning and connectivity (similar to [21]). Additionally, the BCT components are divided into two main types: *primitives* and *nexus.*

[1]http://www.autodesk.com/products/autodesk-3ds-max

Primitives are classes that are contained in the *room class*, which is the cornerstone of the toolkit. *Room class* primitives also operate as a node containing other primitives and connecting different components (e.g., walls, floor, ceiling and columns). These secondary components become the binding elements when a room is connected with another. In other words, rooms are connected by their walls, ceilings and floors along with those belonging to other rooms (Figure 1.A is an example of a basic room).

In this version, BCT *room class* supports only basic prism shapes with either rectangular or regular polygonal base (Figure 1.B). It does not limit the possibilities to generate a wide range of building constructions as the combination of these basic components together with the adaptive nature of the primitives allows to remove, merge or combine basic rooms to produce complex room shapes.

On the other hand, nexus are objects that model the primitives to create spaces to navigate through them. The BCT includes nexus for doors, windows, ramps, "wall-less" (removing one of the walls), staircases, "ceiling-less" (removing room ceiling). Additionally, in the case of nexus assigned to ceilings and floors, they create objects enabling the transition between rooms on distinct stories according to some parameters and using 3DS object library, such as those shown by Figure 1.C.

2) Smart Object Combination: As stated above, the room is the key component of the construction workflow in the BCT, whereas the nexus are the components that connect one rooms with the others. Anyway, if the correct combination of these elements relayed on the appropriate placement of these components (either by a human designer or the BFA framework in our case) that would require an exhaustive low-level definition on where these components should be located.

Instead of that, the BCT provides an intelligent behavior for the components. This behavior enforces context-aware modifications when the components are combined. In that sense, for instance, if we want to create a door on one of the walls in a given room, this room will interact with the room next to it by this wall in order to negotiate the appropriate nexus to be included. This facilitates, for example, that if a room removes one of the walls to another room one or more levels below, the nexus will insert some banisters, or if we would like to place a staircase to an upper floor, the room above will open a hole corresponding with the dimensions of the staircase, and will put some upper frame if there is no wall close to the stairs.

All these BCT *smart object* features are provided by a *story class* component, which interfaces between room interactions in order to achieve a coherent integration of room components and nexus. The *story class* component issues 2D/3D transformations to ensure the integration of room components: geometric validation, linear transformations, etc. In a similar manner, the relationships between stories and their nexus (floors, ceilings, staircases, etc.) are delegated to a *building class* component.

In a general sense, primitives work at the topological level and the *story* and *building class* components perform the translation from topological relationships (close to, next to, etc.) into geometrical operations.

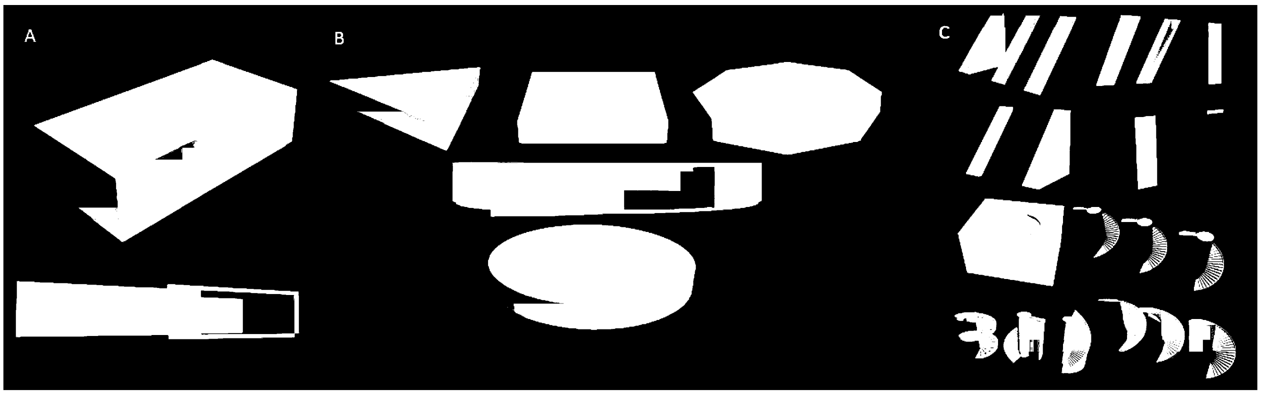

Figure 1. (A): Example of a simple room, formed by four walls, a ceiling and a floor. The BREP modeling technique is necessary to manipulate the vertices and the edges of the polyhedrons. (B): The number of walls of a room is variable. The ceiling as well as the floor are generated with *n* lateral faces whose vertices adjust themselves according to the position of the walls. (C): The image shows distinct types of staircases, ramps, and spiral staircases implemented as BCT nexus (these components have exploited the library of predefined objects included in 3DS Max).

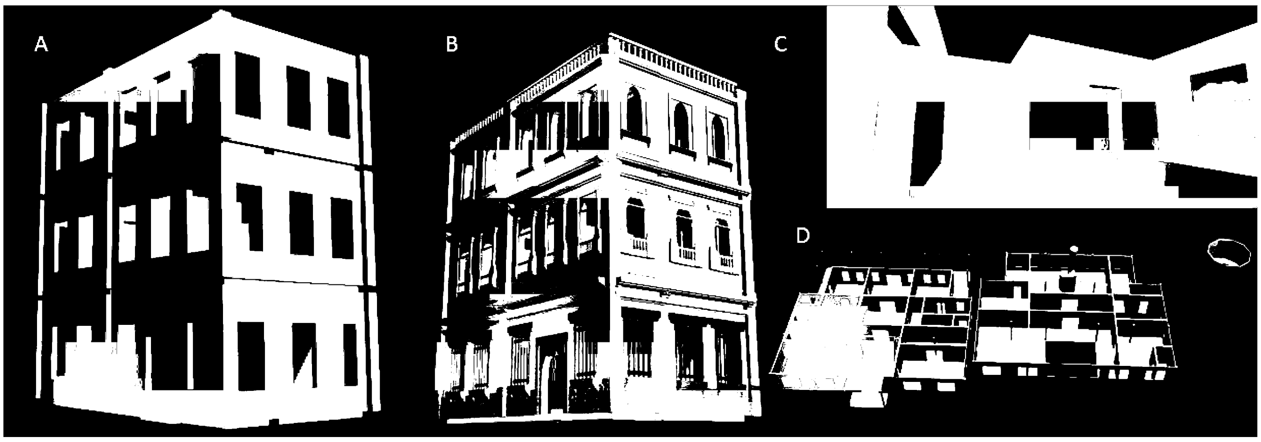

Figure 2. (A): In this sample building, a slot is detected between the distinct stories. This is the place to situate a cornice decorator. (B): Same building once decorated and texturized. (C): A screenshot of the combination of multiple rooms by means of different nexus (although it seems to be a complex room shape it is implemented by the combination of rooms parameterized with the appropriate "wall-less" nexus). (D): Sample of how to configure the interior of a building using different room combinations.

3) Materials, Decorators and Placeables: The BCT includes a decoration and texturing kit based on the *decorator pattern* supported by the the modular construction methodology. Both the room and the story level, the BCT allows to define a configuration file describing the desired decoration theme. These files define materials for the 3D objects, mapping them, and adding other decorative elements to both primitives and nexus (after properly scaling them), such as, for example: friezes, baseboards, cornices, moldings, door and window frames, banisters, etc. Figure 2.B shows the result after the application of a decoration theme to Figure 2.A. This BCT feature establishes the difference between artistic work addressed by graphic designers and modelers and the creative process of level design. The BCT provides an extensible library of materials, decorators and placeable elements, for the modelers to contribute.

Additionally, the BCT also implements a set of parameterized *placeable components* (such as tables, chairs, chests, torches and many others). The location is relative to positions on the floor, the ceiling, above other placeables, on the walls, ...One particular type of placeable components are those located at the doors and windows. The related decorators provide the functional operations (such as closing, opening, and so on) to these elements.

C. Building Architect Framework

The Building Architect Framework (BAF) is based on a sequence of optimization algorithms that are linked (the output of one of the algorithms becomes the input of the following one). The inputs of the complete BAF process are designer-level guidelines (objectives to optimize) and constraints (mandatory conditions to be satisfied):

1) Building layout sketch (constraint): The general shape of the desired building. This shape is provided by the designer with a interactive drawing tool. The designer defines the external boundaries of the building as a composition of rectangular blocks. For each of the blocks the designer should indicate the height in stories. (An automatic method, such as [12], can be used for fully automatic building generation).
2) Ground level (constraint): The designer specifies the desired ground level. The building entrances will be set at this level. The levels below this one are considered underground levels.

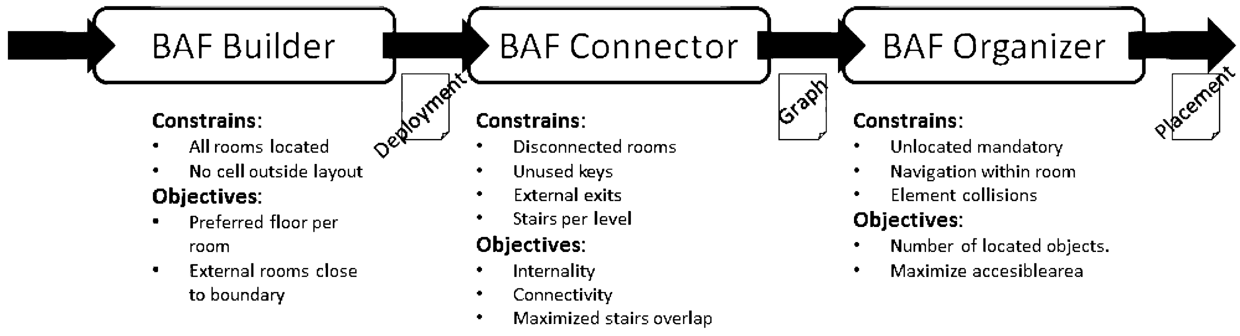

Figure 3. Chain of VNS optimizers conforming the Building Architect Framework (BAF): *Builder*, *Constructor* and *Organizer*. The list of constraints and objectives are indicated below each of them.

3) Number of building entrances (optional guideline): The designer may indicate the minimum and maximum number of building entrances (external doors).
4) Preferred entrance location (optional guideline): The designer may indicate (using the drawing tool) the area on the building boundary to locate the building entrances.
5) A list of special rooms to be included (optional constraint): The designers may specify any number of specific rooms they want to include in the final building. For these special rooms, they may specify dimensions, preferred floor level, and the particular designed model or decoration theme.
6) A list of room templates (constraint): Instead of selecting specific rooms the designer may indicate a series of room templates (generic rooms randomly created within a range of sizes and decoration parameters).
7) Preferred number of rooms (guideline): The produced design will have a minimum number of rooms according to this value.
8) Room *internality* (guideline): For each room the designers may define (if they wishe) the level of *internality*, which is the minimum numbers of rooms to cross to reach this room from the nearest building entrance. In the case of room templates, this value may be generated for a user-defined interval. This value ranges from -1 (outermost) to 1 (innermost). In addition, some rooms may also specify the number of preferred external doors.
9) Room connectivity (guideline): The designer may also specify the *connectivity* grade of the room, defined as the number of other rooms directly connected to it. In the same way as in the case of *internality*, room templates may define this according to a random interval of values. This value ranges from -1 (single connection) to 1 (maximum number of connections).
10) Number of *keys* (optional constraint): *Keys* are special objects located at specific rooms in the design that allow the player to cross a particular door in the building. The designer may indicate the number of required *keys* to visit all the rooms in the building. The BAF algorithm will automatically assign where the *key* and the corresponding *key door* are located.
11) Placeable elements (optional guideline): The designer may select any number of Placeables from the BCT library. The optimizer will try to locate as much as possible of these optional placeables.

Instead of solving the building generation as a single optimization problem we have divided it into three different subproblems to optimize. The chain of optimizers, shown in Figure 3, includes three different modules, each of them tackling the corresponding subproblem: (a) BAF *Builder* (in charge of the placement of rooms within the layout boundaries), (b) BAF *Connector* (responsible of the definition of the connectivity graph), and (c) BAF *Organizer* (that places elements at the exact positions of the building, ensuring navigation and no overlapping elements).

D. Base Optimizer Structure

All the three BAF modules are implemented as configurable variants of a modified Variable Neighborhood Search (VNS) metaheuristic algorithm [22]. VNS is an effective mechanism to solve complex combinatorial problems in which both domain constraints and knowledge can be easily implemented.

VNS algorithms use a single solution that is continuously improved by means of a series of neighborhood operators. Each of the possible operators (named *shakers*) define a different neighborhood of candidate solutions for a given solution providing alternative methods to explore the fitness landscape.

The version implemented in our solution includes the following three modifications: (1) temperature-based non-improvement movement, (2) biased shaker selection based on past performance, and (3) constraint handling.

1) Temperature-based non-improvement movement: Although the success on the application of any search-based PCG depends on the objectives of the fitness function [23], the landscape of candidate solutions could be extremely multimodal, and thus it is necessary to equip the search algorithm with some mechanisms to escape from local optima.

Our VNS algorithm implements a mechanism similar to the one existing in Simulated Annealing (SA) [24] in which a *temperature function* defines a probability of the solution to transit to a worse candidate solution depending on an exponential decaying function that reduces this probability as the number of iterations increases. In particular, we have used the following *temperature function* $T(i)$ and the corresponding transition probability $\pi_{a,b}$:

$$T(i) = 1 - e^{\frac{i}{NIter}-1} \quad \pi_{a,b} = \begin{cases} 1.0 & \text{if } fit(b) < fit(a) \\ \frac{T(i)\frac{fit(a)}{fit(b)}}{2} & \text{otherwise} \end{cases}$$

Where i is the current iteration and $Niter$ the total number of iterations the algorithm will execute and $fit(x)$ is the fitness function of the minimization problem to optimize for the solution x.

2) Biased shaker selection based on past performance: Traditionally, shakers are selected according to an unbiased uniform sampling among the list of shakers or, in some variants (such as Variable Neighborhood Descent) in a deterministic way. In our case we have implemented a biased selection with memory reset. The probability for a shaker to be selected depends on a weight factor defined as:

$$W(s) = Base + Hits - \frac{Fails}{NShakers} - \frac{Tries}{10}$$

Where $NShakers$ is the number of available shakers, $Base$ is 10 times the number of available shakers, $Hits$ is the number of cases in which the shaker has produced a better solution (5 hits) or a solution to which the algorithm has transited according to the temperature-based probability (1 hit), $Fails$ is the number of cases in which the shaker has failed to produce a feasible solution, and $Tries$ are the number of times the shaker has been used.

The probability for a shaker to be selected is determined as:

$$\pi(x) = W(x) \sum_{s \in S} \frac{1}{W(s)}$$

The weight factors are reset ($Hits$, $Fails$ and $Tries$ are set to 0) after a number of iterations (1.0% of the overall number of iterations, in our case) to avoid shakers selection to be too heavily biased by any early stage performance (under the assumption that some shakers perform better in specific phases of the optimization process). Finally, Table I shows the list of shakers implemented for each VNS optimizer.

3) Constraint handling: An important element to incorporate along the design of a PCG process is the handling of unfeasible contents defined in terms of constraints and domain semantics [**?**].

There are different mechanisms to handle constraints in VNS algorithms. We have implemented an incremental combined weight function. The constraints unsatisfied by the solution are identified with $unsat$ a continuous value that indicates the ratio of unsatisfied constraints and with the objective value as obj also defined as a real number to minimize. These two values are combined to get the effective fitness function:

$$fit(a) = obj(a) + unsat(a)\frac{\frac{i}{NIter}}{(1 - \frac{i}{NIter})^2}$$

E. BAF Builder

The BAF *Builder* is in charge of the deployment of all special and template rooms within the building layout, as well as to fill in the holes within the layout with any number of extra rooms. These holes are cells not occupied by any room. This module initially rescales the sketched layout to enclose a volume enough to include all the requested rooms plus an additional margin (configurable by the designer, we have used a 25% additional volume in our experiments). Then, the BAF *Builder* solves the 3D packing problem of fitting all the rooms in the volume. This is a discrete version of the general problem in which all the elements to pack are multiples of a basic cell unity.

The BAF *Builder* solves the problem attending to two constraints:

BC1	All the input rooms must be located within the volume.
BC2	The boundaries of the packing volume must be preserved (no room partially located outside of the volume).

In addition, the BAF *Builder* also tries to minimize two objectives:

BO1	Distance between the preferred and the final floor level from those rooms indicating this value.
BO2	The number of rooms with preferred external door not placed at the volume boundary.

Once the builder has deployed all the rooms, it generates additional rooms to fill in the empty areas of the volume taking them from the template room library. Figure 4 (left hand) shows the output generated for the case of a small building with 10 rooms (rooms 11 to 15) are additionally deployed by the BAF *Builder* to complete the building. In this example, room 10 has been configured as a special room with 0.9 internality but with a preferred external door. At this stage the Builder managed to deploy this room in the boundary of the building structure.

F. BAF Connector

The BAF *Connector* takes the room deployment provided by the previous optimizer, then the *Connector* produces the connectivity graph for the building. The optimizer considers the maximum connectivity graph with all the possible connections between every pair of rooms sharing a common wall or floor/ceiling. The *Connector* has to select which of the edges from this maximum connectivity graph to maintain (and which ones to remove) in order to deal with the following four constraints:

CC1	There must be no disconnected rooms.
CC2	All the *keys* must be used (which means that every key is required to visit the entire building and the *key* must be located in an area accessible from outside or by means of other accessible *keys*).
CC3	There must be the indicated number of building entrances (external doors).
CC4	There must be a number of stairs between floors indicated by the designer.

Table I. LIST OF SHAKERS USED BY EACH OF THE VNS OPTIMIZERS

BAF Module	Shaker Name	Description
Builder	UnlocatedToHole	Deploys a room not yet located on a hole (an unoccupied cell in the volume). The shaker will check rotation and small displacements to fit the room in the map (inside or outside the volume).
Builder	ExternalToHole	Moves a room that is partially outside of the volume to any available hole (rotation and displacements are checked).
Builder	SwapExternalInternal	Swaps the positions of two rooms if one of them is partially outside of the volume (rotations and displacements are checked).
Builder	MoveRoomToBoudary	Selects a room, with preferred external doors, to the closest boundary of the volume (rotations and displacements are checked)
Builder	SwapRooms	Any two rooms are swapped (rotations and displacements are checked).
Connector	SolveDisconnection	Selects a room with no connections and assigns one of them randomly
Connector	ReduceOverconnectivity	Finds a room with more than one connection that has a *connectivity* value higher than the preferred value and removes one of the connections.
Connector	ReduceUnderconnectivity	Finds a room that has a *connectivity* value lower than the preferred value and adds one additional connection (if possible).
Connector	ReduceUnderInternality	Finds a room with more than one connection that has an *internality* value lower than the preferred value and removes one of the connections. Removing the connection representing the shortest path to the nearest entrance may increase the *internality*.
Connector	ReduceOverinternality	Finds a room that has an *internality* value higher than the preferred value and adds one additional connection (if possible).
Connector	ExternalConnection	Finds a room with a preferred number of external connections higher than one and if it is in the boundary of the volume, then it opens a connection outside.
Connector	InternalityBalancer	Selects a random room and removes or adds any additional connection depending on the internality values of the nearby rooms.
Connector	MoveKey	Changes the location of one of the *keys*.
Connector	MoveKeyDoor	Changes the connection closed with a given *key*.
Connector	RandomConnection	Adds or removes a random connection
Organizer	LocateElement	Locates a mandatory element not yet placed in the building. If there is no mandatory elements in this situation then it selects an optional element.
Organizer	SolveCollision	Detects a position occupied by a placeable element which some other elements have also declared either occupied or open. It selects one of the conflicting elements and changes the location of the placeable.
Organizer	RotateElement	Detects an element (typically a staircase) in which the placeable orientation collides with one of the walls (the intended entry point is outside of the room) and rotates the placeable.
Organizer	SolveNavigation	Detects the rooms in which there is a navigation problem (there is at least one pair of entry points that are not mutually accesible) and then moves one of the placeables assigned to this room.

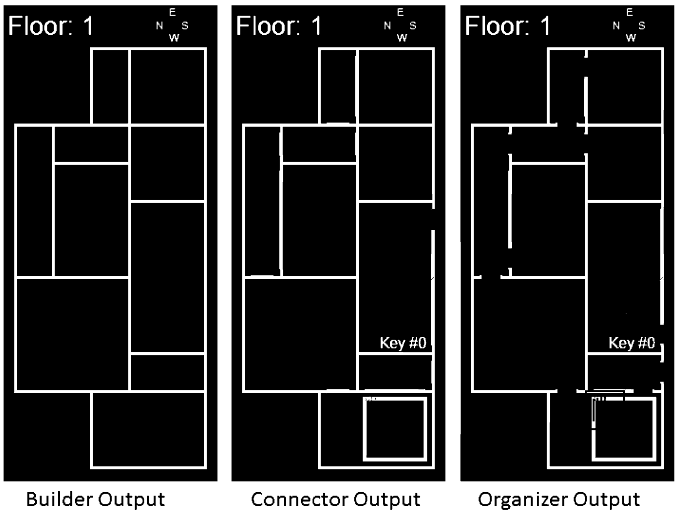

Figure 4. Example of an small building structure, results of the BAF *Builder* (left hand), BAF *Connector* (center), and BAF *Organizer* (right hand). The structure is based on a minimum of 10 rooms, 3 of them (8,9 and 10) provided by the designer and the rest described by a basic room template. Room 10 is defined as 0.9 *internality* and must have an external entry.

In addition the BAF *Connector* should optimize the following objectives:

CO1 The correlation between the preferred *internality* values and the actual *internality* (obtained as a ranking of the rooms according to the minimum number of other rooms the player must cross to reach the given room). The connections closed by a *key* cannot be transited unless the player has that *key*, which means that the *internality* value of these rooms is the path from the outside to the room in which the *key* is found and back to the closed connection.

CO2 The correlation between the preferred *connectivity* values and the actual *connectivity* (defined as the ranking of rooms according to the number of connections with other rooms).

CO3 The overlap area of the inter-floor connections (stairs). This objective tries to ensure that the volume to place staircase nexus allows different nexus alternatives.

In Figure 4 (center), the reader can see how the BAF *Connector* defines the connectivity graph (the actual walls to have a connection nexus). This optimizer also places keys and key doors. In our case, the optimizer placed an external door to room 10 (as indicated by the designer) but, in order to reach the 0.9 internality value, this door is closed with Key #0, which is located in one of the rooms in the upper floor. We can also see how the optimizer assigned room 9 three connections (this room has a preferred connectivity value of 0.8), this room also places the inter-floor connector (where a staircase will be latter placed).

G. BAF Organizer

This optimizer locates the placeables associated with the mandatory elements (doors and stairs) plus as many as possible from the optional ones selected by the designer. The BAF *Organizer* must ensure the correct navigation of the produced building, which means that all the areas must be accessible and there must be a path between all the entry points for every room (doors or stairs). This optimizer decides the type of nexus to be used (single or double rooms, remove the entire wall or all the types of staircases, ramps and spiral stairs). All these elements, as well as the optional placeables, define a given set of positions that must be open and accessible to prevent, for example, that a door is blocked by a staircase going up in another direction. In order to do that the BAF *Organizer* must satisfy the following constraints:

OC1 All mandatory placeables (associated with connections) must be positioned.

OC2 All the room must have paths to travel between every pair of entry points.

OC3 There must be no collisions between the occupied positions of any element and the required open or occupied positions of the rest.

In addition, the BAF *Organizer* should maximize:

OO1 The number of optional placeables located in the building.

OO1 The accessible area within the building (the number of cells the player may move across).

Figure 4 (right hand), shows how the three entry points of room 9 are located to avoid collisions and maximize navigation area.

IV. Experimental Results

In order to test the flexibility and scalability of this approach we have conducted a series of experiments. The experiments are defined by the designer input (rooms, layout sketch, number of stories high, ground level floor, number of keys and external doors, and desired decoration theme). Detailed information is described in Table II and Figure 5 shows some screenshots of the generated structures.

Table II. Description of the experiments

Id	Rooms	Description
Rural	10	3 special rooms, 2 stories high and cottage theme.
Villa	20	2 special rooms, 2 stories + 1 tower and stone brick theme.
Mansion	30	4 special rooms, 3 stories high and renaissance theme.
Temple	40	2 special rooms, 3 stories high + 1 underground level and brick theme.
Castle	125	5 special rooms, 4 stories high + 1 story more for the 4 towers + 3 underground levels of dungeons, and castle theme.

For each scenario we have defined 4 sets of designer inputs (different specific rooms or *internality/connectivity* values) and performed 25 executions of the pipeline sequences with $500 \times NRooms$ iterations limit ($NRooms$ is the minimum number of rooms provided by the designer). For all these experiments, the following information has been computed:

- **Final number of rooms**: The original number of rooms provided by the designer plus the average of those included in the optimization process to fill in the gaps.
- **Succeed ratio**: Percentage of executions that obtain successful results (all design constraints satisfied).
- **Deployment diversity–average**: The ratio of original rooms deployed at the same position two or more times out of the 25 executions (for the same Id and designer input set).
- **Deployment diversity–highest**: The largest ratio (out of the 25 executions) that the room is deployed at the same position.
- **Connectivity diversity–average**: The ratio of original room pairs connected two or more times out of the 25 executions (for the same Id and designer input set).
- **Connectivity diversity–highest**: The largest ratio (out of the 25 executions) that a pair of rooms is connected.

Table III. Analysis of the experiments

			Diversity Measures			
	Final	Succeed	Deployment		Connectivity	
Id	Rooms	Ratio	Average	Highest	Average	Highest
Rural	15.20	100%	8.75%	12%	14.0%	16%
Villa	34.75	100%	8.50%	12%	1.25%	12%
Mansion	53.45	98%	6.50%	12%	0.05%	8%
Temple	72.20	90%	5.00%	8%	< 0.01%	8%
Castle	278.35	81%	2.50%	8%	< 0.01%	8%

Table III shows that the diversity of the generated buildings is quite high, only in those small scenarios in which the combinations are rather limited, few rooms were placed at the same positions (twice or three times among the 25 executions of each configuration), while the diversity in the large scenarios is very large (only 2.5% of the rooms was ever located at the same position in two out of the 25 executions). As an example, a series of 6 different generated buildings of the smallest example (*Rural*) are depicted in Figure 5.G. While the designer's constraints are the same, there are different solutions satisfying all these constraints and equivalent fitness values. For example, the largest special room (room #2) is deployed at different location (either in floor 1 or 2), and sometimes used as a connector element. The number of building entry points or the number of stairs (both restricted to a minimum of 1 up to a maximum of 2 by the designer) is different in these designs, as well as the location of the keys and the locked doors.

V. Conclusions

In this paper, we have presented an integrated approach that uses "smart object" primitives for building components and a search-based procedural content generation. This approach allows a coordinated development of the procedural mechanisms to design buildings according to designer constraints, while the modeling of attractive structures and elements is managed by an extensible library in which 3D modelers and digital artists may contribute. Finally, we have conducted a series of experiments with this integrated workflow that shows that our proposal is flexible yet powerful to create complex 3D buildings.

Acknowledgments

The authors thank the *Centro de Supercomputación y Visualización de Madrid* (CeSViMa) for the computer resources. A. LaTorre gratefully acknowledges the support of the Spanish Ministry of Science and Innovation (MICINN) for its funding throughout the *Juan de la Cierva* program.

References

[1] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 3, no. 3, pp. 172–186, 2011.

[2] T. Betts, "Procedural content generation," *Handbook of Digital Games*, pp. 62–91, 2014.

[3] J. Burgess, "Modular level design for skyrim," in *Proceedings of the Games Developers Conference (GDC'2013)*, 2013.

[4] J. Togelius and J. Schmidhuber, "Computational intelligence and game design," in *Proceedings of the PPSN Workshop on Computational Intelligence and Games.*, 2008.

[5] J. Whitehead, "Towards procedural decorative ornamentation in games," in *Proceedings of the FDG Workshop on Procedural Content Generation*, 2010.

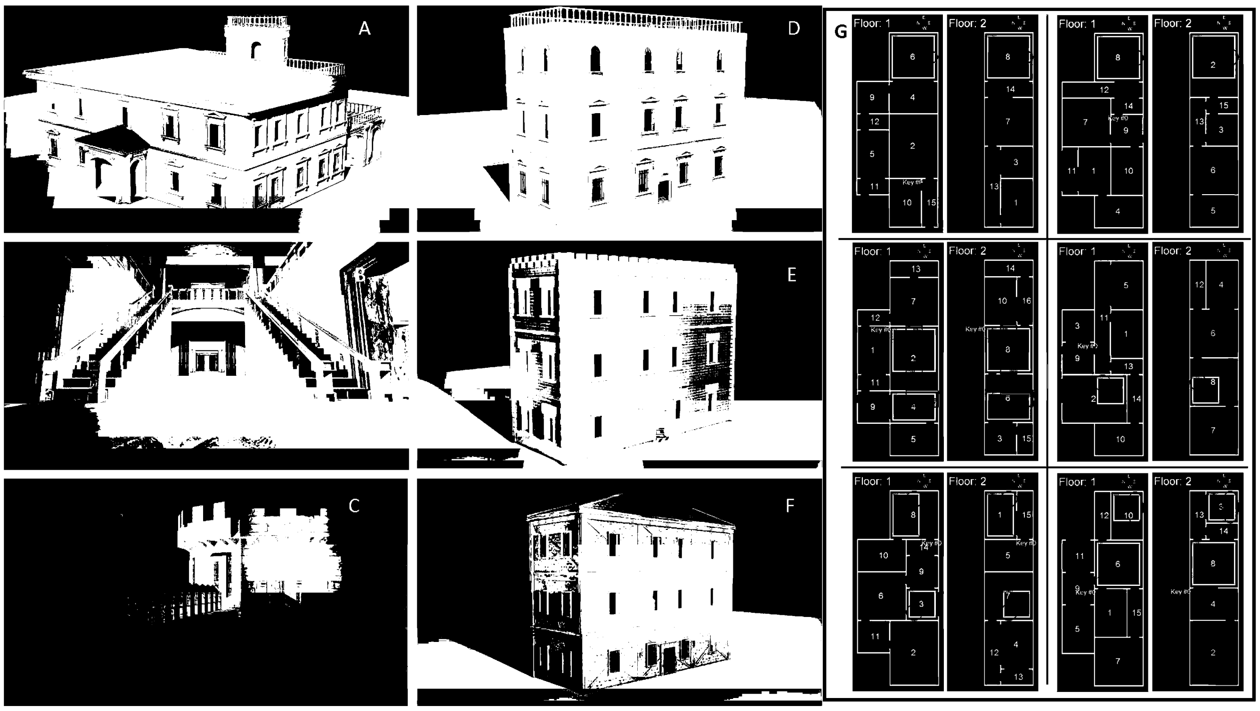

Figure 5. A series of building samples generated by the BCT and the BAF framework: (A): One of the produced results of the *Villa* example case, (B): Internal part of the *Temple* example case, the special room "*main entrance*", a 2-level high chamber with maximum *connectivity*, (C): Internal perspective of *Castle* example case, it can be seen how cornice decorators are changed to produce battlement structures, and (D-F): Same example building with different decoration themes: stone brick (D), red brick (E) and cottage (F). The type of decoration theme implies also a different level of model complexity, for example the same original building (2,285 polygons and 2,148 vertexes) the different models have (D: 61,604 polygons, and 35,498 vertexes; E: 13,850 polygons and 9,742 vertexes; and F: 12,572 polygons and 35,243 vertexes). (G) A series of 6 floor plan outputs produced from the same designer's constraints (Id=Rural) and different random seeds.

[6] J. Togelius, M. Preuss, and G. N. Yannakakis, "Towards multiobjective procedural map generation," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. ACM, 2010, p. 3.

[7] J. Doran and I. Parberry, "Controlled procedural terrain generation using software agents," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 2, pp. 111–119, 2010.

[8] C. McGuinness and D. Ashlock, "Incorporating required structure into tiles," in *IEEE Conference on Computational Intelligence and Games (CIG)*, 2011.

[9] J. M. Peña, E. Menasalvas, S. Muelas, A. LaTorre, L. Peña, and S. Ossowski, "Soft computing for content generation: Trading market in a basketball management video game," in *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, 2013, pp. 1–8.

[10] R. M. Smelik, T. Tutenel, K. J. de Kraker, and R. Bidarra, "A declarative approach to procedural modeling of virtual worlds," *Computers & Graphics*, vol. 35, no. 2, pp. 352–363, 2011.

[11] F. Marson and S. R. Musse, "Automatic real-time generation of floor plans based on squarified treemaps algorithm," *International Journal of Computer Games Technology*, 2010.

[12] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool, "Procedural modeling of buildings," in *ACM Transactions on Graphics (TOG)*, vol. 25, no. 3. ACM, 2006.

[13] J. O. Talton, Y. Lou, S. Lesser, J. Duke, R. Měch, and V. Koltun, "Metropolis procedural modeling," *ACM Transactions on Graphics (TOG)*, vol. 30, no. 2, p. 11, 2011.

[14] E. Whiting, J. Ochsendorf, and F. Durand, "Procedural modeling of structurally-sound masonry buildings," *ACM Transactions on Graphics (TOG)*, vol. 28, no. 5, p. 112, 2009.

[15] M. Lipp, P. Wonka, and M. Wimmer, "Interactive visual editing of grammars for procedural architecture," in *ACM Transactions on Graphics (TOG)*, vol. 27, no. 3. ACM, 2008, p. 102.

[16] P. Merrell, E. Schkufza, and V. Koltun, "Computer-generated residential building layouts," in *ACM Transactions on Graphics (TOG)*, vol. 29, no. 6. ACM, 2010, p. 181.

[17] L. Johnson, G. N. Yannakakis, and J. Togelius, "Cellular automata for real-time generation of infinite cave levels," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. ACM, 2010, p. 10.

[18] T. Roden and I. Parberry, "From artistry to automation: A structured methodology for procedural content creation," in *Entertainment Computing–ICEC 2004*. Springer, 2004, pp. 151–156.

[19] R. van der Linden, R. Lopes, and R. Bidarra, "Procedural generation of dungeons," *IEEE Transactions on Computational Intelligence and AI in Games*, 2013.

[20] R. M. Smelik, T. Tutenel., R. Bidarra, and B. Benes, "A survey on procedural modelling for virtual worlds," *Computer Graphics Forum*, 2014.

[21] M. Kallmann and D. Thalmann, "Modeling objects for interaction tasks," in *Computer Animation and Simulation*, 1998, pp. 73–86.

[22] P. Hansen, N. Mladenović, and J. A. M. Pérez, "Variable neighbourhood search: methods and applications," *Annals of Operations Research*, vol. 175, no. 1, pp. 367–407, 2010.

[23] D. Ashlock, C. Lee, and C. McGuinness, "Search-based procedural generation of maze-like levels," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 3, no. 3, pp. 260–273, 2011.

[24] S. Kirkpatrick, "Optimization by simulated annealing: Quantitative studies," *Journal of statistical physics*, vol. 34, no. 5-6, pp. 975–986, 1984.

[25] T. Tutenel, R. Bidarra, R. M. Smelik, and K. J. D. Kraker, "The role of semantics in games and simulations," *Computers in Entertainment (CIE)*, vol. 6, no. 4, p. 57, 2008.