A Constrained Growth Method for Procedural Floor Plan Generation

Ricardo Lopes[1], Tim Tutenel[1], Ruben M. Smelik[2], Klaas Jan de Kraker[2], and Rafael Bidarra[1]

[1]Computer Graphics Group, Delft University of Technology, Delft, the Netherlands
[2]Modelling, Simulation and Gaming Department, TNO, The Hague, the Netherlands

Abstract

Modern games often feature highly detailed urban environments. However, buildings are typically represented only by their façade, because of the excessive costs it would entail to model all building interiors of a city by hand. Although there exist automated procedural techniques for building interiors, their application is to date limited. One of the reasons for this is because designers have too little control over the resulting room topology. Furthermore, generated floor plans do not always adhere to the necessary consistency constraints, such as reachability and connectivity.
In this paper, we propose a novel and flexible technique for generating procedural floor plans subject to user-defined constraints. We demonstrate its versatility, by showing generated floor plans for different classes of buildings, some of which consist of several connected floors. It is concluded that this method results in plausible floor plans, over which a designer has control by defining functional constraints. Furthermore, the method is efficient and easy to implement and integrate into the larger context of procedural modeling of urban environments.

INTRODUCTION

Game worlds increasingly often feature highly detailed open worlds. Notable recent examples include Assassin's Creed, GTA IV and Oblivion, where players can explore beautiful cities, of which each building has been modeled by hand. Manual modeling of these city models involves an enormous amount of effort, putting a huge burden on the budget for game development companies. In particular, this is the reason that, in games, city buildings either have no interior (they consist only of a façade) or a fixed set of interiors is repeated all over the city. Urban environments are therefore one of many areas where automated procedural generation methods can excel, by providing a limitless amount of varied content for a fraction of the cost.
The procedural generation of a city entails a number of ingredients, each with its specific procedures and generation techniques: district topology (city center, suburbia), road network (ring road, streets), division of open space into parcels (building lots), building façades, floor plans (room layouts) and interior furniture (chairs, tables). In this paper, we focus on an important if somewhat neglected ingredient: floor plans. We propose a novel and flexible technique for generating procedural floor plans subject to user-defined constraints. Using these constraints, game designers control both the topological layout of the building (*e.g.* which room comes next to which other room) and the areas of the room. Furthermore, the method guarantees reachability of all rooms and over multiple floors.
The remainder of this paper is structured as follows. We first survey previous work on procedural modeling of urban environments, focusing on floor plan generation. Then we present our constrained growth method for procedural floor plans. We show several example building interiors generated by this technique. And last, we discuss our method advantages and limitations, and propose future extensions and potential applications.

RELATED WORK

Procedural modeling techniques have been proposed for almost every aspect of virtual worlds, ranging from landscapes to buildings. An extensive survey of procedural modeling techniques can be found in [1]. Regarding buildings, *L-systems*, previously applied to plant models [2], were among the first automated techniques used for building façades [3]. In recent years, more specialized rewriting systems have been presented for this purpose: the *split grammars* [4] and *shape grammars* [5].
Here, we focus on procedural techniques for building floor plans, *i.e.* the creation of a suitable layout of the different rooms inside a building. Greuter et al. [6] create a floor plan as a combination of 2D shapes. However, this floor plan is only used for extruding building façades. Shape grammars, typically applied to building façades, can also create floor plans, as shown by Rau-Chaplin et al. [7] in their *LaHave House project*. Their system generates a library of floor plans, each of which can be customized and automatically transformed into assembly drawings. It uses shape grammars to create a *plan schema* containing basic room units. Possible groupings of individual units are recognized to define functional zones like public, private or semi-private spaces. After generating the required geometric data, such as the room unit dimensions and the location of walls, a specific function is assigned to each room. The rooms are filled

with furniture, by fitting predefined layout tiles from an extensive library of individual room layouts.

Martin [8] proposes a graph-based method, in which nodes represent the rooms and edges correspond to connections between rooms (*e.g.*, a door). This graph is generated by a user-defined grammar. Starting from the front door, public rooms are added. Each of these public rooms is assigned a specific function (dining room, living room, etc.). Subsequently, private rooms are attached to the public rooms and, finally, stick-on rooms like closets or pantries are introduced. This graph is transformed to a spatial layout, by determining a 2D position for each room. For each node, depending on the desired size of the room, a specific amount of "pressure" is applied to make it expand and fill up the remaining building space.

Hahn et al. [9] present a method tailored for generating office buildings. Starting from the building structure, they first position all elements that span across multiple floors, *e.g.* elevator shafts, staircases. Then, the building is split up into a number of floors. On each of them a hallway subdivision is applied, adding straight hallway segments or rectangular loops. Next, the remaining regions are subdivided into rooms, for which geometry is created and appropriate objects are placed. A notable feature of this system is that each step is executed just in time: based on the player's position, floors and rooms are generated or discarded. A discarded room can be restored exactly in its previous state, by re-using the same random seed in the procedure, followed by re-applying all changes to its objects, caused by the player.

Marson and Musse [10] introduce a room subdivision method based on *squarified treemaps*. Treemaps recursively subdivide an area into smaller areas, depending on a specific importance criterion. Squarified treemaps simultaneously try to maintain a length to width ratio of 1. Their methods input is the basic 2D shape of the building and a list of rooms, with desired dimensions and their functionality, *e.g.* social area, service area and private area. First these functionality areas are added to the treemap. These areas are again subdivided to create each room. Possible connections between room types are pre-defined; based on this, doors are placed between the rooms in the generated floor plan. Using the A* algorithm, a shortest path is determined, visiting all rooms that need a connection with the corridor. This path is transformed into a corridor, and all rooms are adjusted to make room for it.

Tutenel et al. [11] applied a generic semantic layout solving approach to floor plan generation. Every type of room is mapped to a class in a semantic library. For each of such semantic classes, relationships can be defined. In this context, constraints typically define room-to-room adjacency, however, other constraints can be defined as well, e.g. place the kitchen next to the garden, or the garage next to the street. For each room to be placed, a rectangle of minimum size is positioned at a location where all defined relation constraints hold, and all these rooms expanded until they touch other rooms.

A general overview of the application of constraint solving in procedural generation can be found in [12]. Particularly, several approaches define the creation of room layouts as a space planning problem. Charman [13] gives an overview of constraint solving techniques that, although not specifically focused on space planning, can be applied to these problems. He compares several space planners and discusses the efficiency of these solvers. Due to many recent improvements on constraint solving techniques, his efficiency concerns are no longer relevant. However, the discussed planners are still suitable for layout solving. For instance, the planner he proposed [13] works on the basis of axis-aligned 2D rectangles with variable position, orientation and dimension parameters, for which users can express geometric constraints, possibly combined with logical and numerical operators. These constraint solving techniques create rectangular shapes, subject to dimension and adjacency constraints. However, they are typically quite complex and can still be time consuming. Moreover, many of them cannot handle irregular shapes, *e.g.* L-shaped or U-shaped rooms, which incidentally also holds for many of the other techniques discussed above. This is one of the main drawbacks of all these approaches. Our growth-based approach generates more flexible results, and does that faster than other constraint solving techniques.

FLOOR PLAN GENERATION METHOD

This section discusses a novel method for procedurally generating floor plans. The problem of generating floor plans is primarily a problem of generating the appropriate layout for rooms, *i.e.* their location and area. We believe that a proper procedural floor plan is one that follows the same principles (and, as a consequence, the appearance) of real building architecture. Our method has drawn its inspiration from real life architectural floor plans, where geometric grids are used as a canvas for hand drawing building interiors. Fig. 1 illustrates a real architectural floor plan example, where a point grid is visible. Our method uses a grid-based algorithm for placing and growing rooms within a building layout. We describe how this algorithm hierarchically creates a floor plan by generating building zones followed by its room areas, both of which are constrained by adjacency and connectivity relationships.

Hierarchical layout

The building layout is defined in a hierarchic manner. Certain types of rooms can be grouped, allowing the building to be subdivided in zones for these room groups, followed by subdividing these zones into sub-zones, or into specific rooms. This approach is similar to the method of Marson and Musse [10], which was discussed above. An example of a simple hierarchic building layout

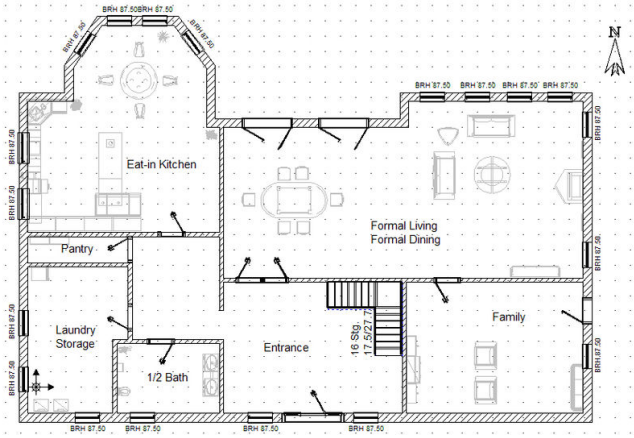


Figure 1: Example of a real architectural floor plan, using a geometric point grid

is shown in Fig. 2. In this example, we subdivide a house into two zones: a public zone and a private zone. The public zone is then subdivided into a dining room, a kitchen and a living room and the private zone into a hallway, two bedrooms and a bathroom. This entails that the room generation algorithm is applied twice, once for each level in the hierarchy. Note that this hierarchy can of course have more than two levels.

Because of the hierarchic approach, we do not allow adjacency constraints to be defined between two rooms from different zones. We can however put adjacency constraints between a room and a parent zone, *e.g.* in the previous example, we can demand that the hallway from the private zone is adjacent to the public zone to allow a connection with this public zone through the hallway.

Room placement

Our method uses a grid as the basis for the room placement and expansion process. There are two advantages in using this representation. Firstly, it maps closely to

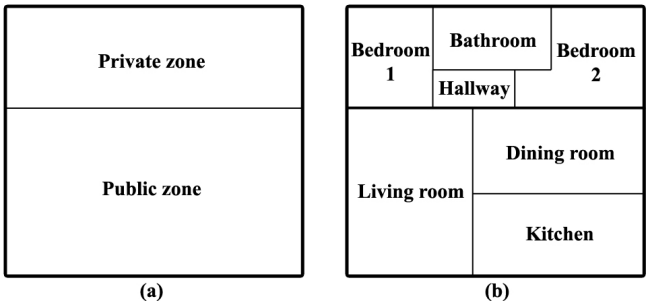


Figure 2: This is an example of a hierarchic subdivision of a building. (a) shows a building subdivided into two areas, (b) shows these areas subdivided into their required rooms.

the way architects design floor plans. Secondly, it is efficient to traverse and expand in a 2D grid, which allows our floor plan generation method to execute fast. Our grid representation (a matrix) does not limit a floor to a rectangular shape. We fit non-rectangular buildings in a grid that includes space marked as outdoor.

The method starts with a building footprint polygon, which we rasterize into the 2D grid. Additional input includes:

- the grid cell dimension in meters (*e.g.* 0.5m);
- a list of room areas to generate;
- for each room area its corresponding house zone (public, private, hallway) and its preferred relative area (or size ratio);
- adjacency and connectivity constraints.

The output it produces is a floor plan, consisting in room areas, interior and exterior walls, and doors.

In the room placement phase, the initial positions for each room are determined. For this, we create a separate grid of weights of the same dimensions as the building grid. Initially, cells inside the building get a weight of one and cells outside get a weight of zero. Many of the rooms in a building are preferably placed next to an outer wall. Therefore, to apply this constraint, the weights are altered. A straightforward way to do this would be to set weights of cells not connected to the outside to zero. However, placing initial positions adjacent to a wall does not always result in plausible results, as they tend to cause less regular room shapes. Therefore, we use a different approach. Based on the size ratio of the room and the total area of the building, we can estimate a desired area of the room. Cells positioned at least a specific distance (based on this estimated area) away from the walls are assigned a weight of 1. This results in much more plausible room shapes.

This phase also deals with the defined adjacency constraints. The adjacency constraints are always defined between two rooms, *e.g.* the bathroom should be next to the bedroom, the kitchen should be next to the living room, etc. When selecting the initial position of a room, we use the adjacency constraints to determine a list of rooms it should be adjacent to. We check whether these rooms already have an initial position. If there is, we alter the weights to high values in the surroundings of the initial positions of the rooms it should be adjacent to. This typically results in valid layouts; however there is a small chance that the algorithm grows another room in between the rooms that should be adjacent. To handle this case, we reset the generation process if some of the adjacency constraints were not met.

Based on these grid weights, one cell is selected to place a room, and the weights around the selected cell are set to zero, to avoid several initial positions of different rooms to be too close to each other.

Room expansion

We use a growth-based method for determining the precise shape of rooms. Our algorithm gradually grows rooms from initial room positions until the building interior is filled. The expansion process is done by appending adjacent grid cells to rooms.
Algorithm 1 outlines the expansion of rooms in our method. It starts with a grid m containing the initial positions of each room. It then picks one *room* at a time, selected from a set of available *rooms* (`SelectRoom`), and expands the room shape to the maximum rectangular space available (`GrowRect`). This is done until no more rectangular expansions are possible. At this point, the process resets *rooms* to the initial set, but now considers expansions that lead to L-shaped rooms (`GrowLShape`). In a final step, the algorithm scans for remaining empty cells and assigns them to a nearby room (`FillGaps`).

Algorithm 1: Room expansion algorithm

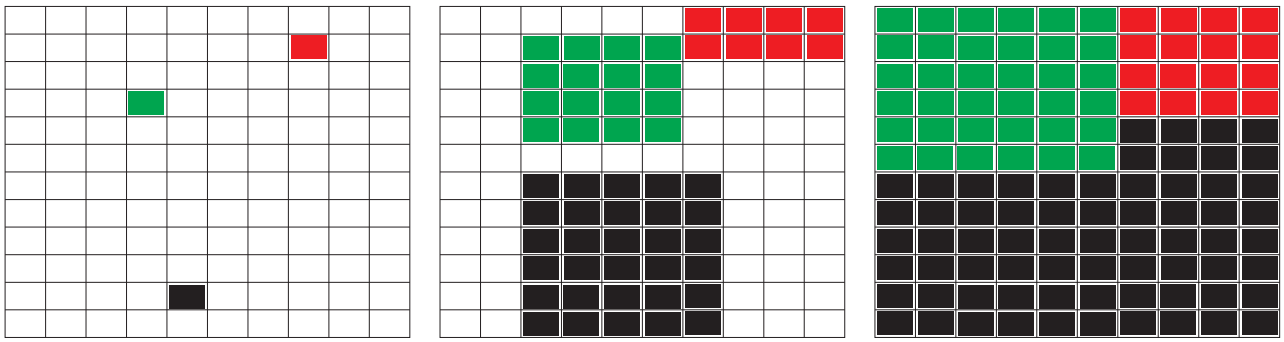
in : A list of rooms to be placed l
in : A list of room area ratios r
in : A building grid m
out: A floor plan defined in m

```
rooms ← BuildRoomSet(l);
while rooms ≠ ∅ do
    room ← SelectRoom(rooms, r);
    canGrow ← GrowRect(room, m, r[room]);
    if ¬canGrow then
        rooms \ {room};
    end
end
rooms ← BuildRoomSet(l);
while rooms ≠ ∅ do
    room ← SelectRoom(rooms, r);
    canGrow ← GrowLShape(room, m);
    if ¬canGrow then
        rooms \ {room};
    end
end
if HasEmptySpaces(m) then
    FillGaps(m);
end
```

In `SelectRoom` the next room to be expanded is chosen on the basis of the defined size ratios r for each room. The chance for a room to be selected is its ratio relative to the total sum of ratios, defined in r. With this approach, variation is ensured, but the selection still respects the desired ratios of room areas.
The first phase of this algorithm is expanding rooms to rectangular shapes (`GrowRect`). In Fig. 3 we see an example of the start situation (a) and end (b) of the rectangular expansion phase for a building where rooms black, green and red have size ratios of, respectively, 8, 4 and 2. Starting with rectangular expansion ensures two characteristics of real life floor plans: (i) a higher priority is given to obtain rectangular areas and (ii) the growth is done using the maximum space available, in a linear way. For this, all empty line intervals in the grid m to which the selected *room* can expand to are considered. The maximum growth, *i.e.* the longest line interval, which leads to a rectangular area is picked (randomly, if there are more than one candidates). A room remains available for selection until it can not grow more. This happens if there are no more directions available to grow or, in the rectangular expansion case, if the room has reached its maximum size. This condition also prevents starvation for lower ratio rooms, since size ratios have no relation with the total building area. In Fig.3 (b), all rooms have reached their maximum size.
Of course, this first phase does not ensure that all available space gets assigned to a room. In the second phase, all rooms are again considered for further expansion, now allowing for non-rectangular shapes. The maximum growth line is again selected, in order to maximize efficient space use, *i.e.* to avoid narrow L-shaped edges. In this phase, the maximum size for each room is no longer considered, since the algorithm attempts to fill all the remaining empty space. Furthermore we included mechanisms for preventing U-shaped rooms. Fig. 3 (c) illustrates the result of the L-shaped growth step, continuing the previous example. The final phase scans the grid for remaining empty space; this space is directly assigned to the room which fills most of the adjacent area.
With the described mechanism, our growth algorithm generates valid floor plans. Even with all the constraints, each room layout problem has many valid solutions. Therefore, there is quite some variation in generated floor plans. Since our grid-based algorithm is fast enough, we can collect multiple alternative solutions for each situation. We execute our growth algorithm N (parametrizable) times, obtaining N different but valid floor plans. We can either randomly select one of alternatives, or we can evaluate and rank the solutions (by *e.g.* preference for smaller hallways or the least amount of corners in room), and pick the best solution. Post-processing will convert the grid to a geometric representation of rooms, and ensures the creation of doors and windows or extrusion of walls.

Room connectivity

Our method ensures connectivity between rooms by procedurally placing inner doors. This process is more elaborate than simply making all rooms reachable from *e.g.* the entrance of the building. For plausible results, room connectivity needs to influence and be influenced by the topology of the building. We achieved this using connectivity constraints combined with a specialized connectivity algorithm.
Connectivity constraints, declared as input, state that



(a) (b) (c)

Figure 3: Example for the room expansion algorithm: (a) initial room positions (b) rectangular growth (where rooms reached their maximum size) (c) L-Shape growth.

two rooms should be directly connected. Stating that room A should be connected to room B will create a door that opens from room A to room B. These constraints influence the room layout, as they are also interpreted as adjacency constraints. After placement and expansion of rooms, we proceed to a post-processing phase, in which, among other things, our connectivity algorithm is executed. For this, the grid representation is no longer appropriate, as it is more natural to consider inner walls rather than room areas. Therefore, in this post-processing phase matching inner walls for the rooms are created first.

The connectivity algorithm places doors in these walls, influenced by the building topology and connectivity constraints. It should ensure the common connectivity principles of a building (*e.g.* hallway always leads to adjacent public rooms), while the constraints should only be used to declare specific cases. Our aim is to emulate real life architecture in its basic connectivity concepts, independent of style or type of buildings. Therefore, we use the notion of private and public rooms to decide on door placement. The goal is to create full connectivity, while respecting room privacy.

Our algorithm starts by placing doors between rooms for which connectivity was explicitly declared. Next, it connects any hallway to all of its adjacent public rooms. Unconnected private rooms are then connected, if possible, to an adjacent public room. Publics rooms with no connections are connected to an adjacent public room as well. Finally, our last step is a reachability test. We examine all rooms and if any is not reachable from the hallway, we use the adjacency relationships between rooms to find a path to the unreachable room, and create the necessary door(s).

With this algorithm, public rooms (and in particular the hallway) have priority to become focal points, in terms of connectivity to private rooms. We wanted to bring about the situation where, for example, two adjacent private rooms are connected through a common public room, instead of having a direct door between them. The hierarchical layout ensures rooms distribution in such a way that this setup is possible.

The creation of an inner door is implemented in a simple manner. First, we select a shared wall between the two rooms to connect, in which the door fits. Next, we randomly select a segment of that wall and place the door in its midpoint. Finally, doors are assigned to open towards the destination room with its hinge towards the closest corner of the wall.

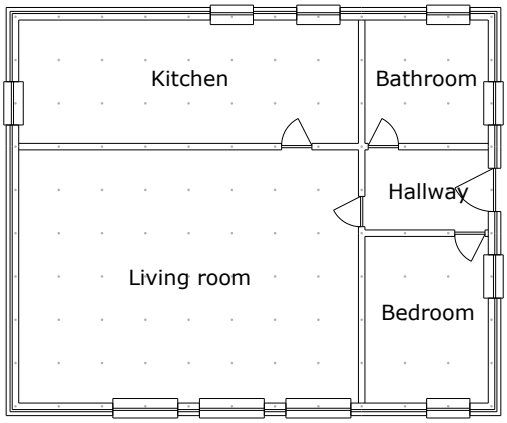


Figure 4: Example of a relatively simple floor plan including four rooms and a hallway.

RESULTS

This section provides examples of floor plans generated with our method. The first example is a relatively simple rectangular building (11 x 9 m), for which no hierarchy is defined, and which should contain four rooms (a bedroom, a bathroom, a kitchen and a living room) and a hallway. We defined a connectivity constraint between the kitchen and the living room. An example of a generated floor plan for this building is shown in Fig. 4. Notice that the connectivity constraint is satisfied, and that all rooms

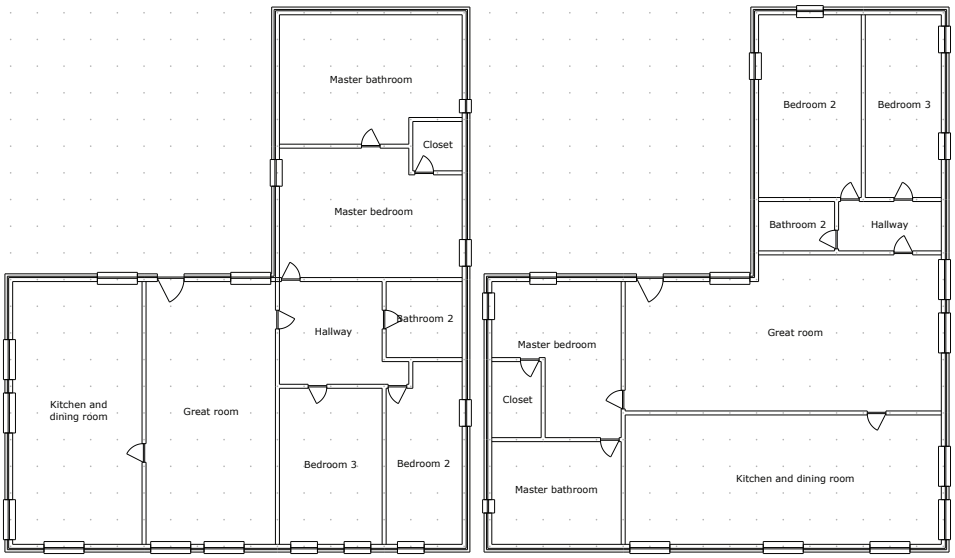


Figure 5: Two examples of a more complex floor plan, built up of a public zone with a great room and kitchen and dining room, a private zone with a master bedroom and walk-in closet, master bathroom and a private zone with a hallway, a bathroom and two bedrooms.

are reachable. Generating these kind of floor plans takes less than 100 ms.

In our second example, we show more complex floor plans, inspired from real floor plans of North-American style villas. The building (which is L-shaped, measuring 17 x 20 m with a top part of 7 x 9.5m and a bottom part of 17 x 10.5 m) is subdivided into three areas: one public and two private zones. There are adjacency constraints defined between the public area and the two private areas. The public area includes a great room and a room for a kitchen and a dining area. These rooms have a connectivity constraint defined. The first private area includes a master bedroom with an adjacency constraint to the public area, a master bathroom and a walk-in closet both connected to the master bedroom. The second private area includes a hallway with an adjacency constraint to the public area and three other rooms (a bathroom and two additional bedrooms), all connected to this hallway. Two floor plan examples for this building are shown in Fig. 5.

One clearly notices L-shaped areas not only for the individual rooms but also for the different areas within the floor plan. This creates more varied and less restrictive results than methods that are limited to rectangular rooms. Because of the increased complexity these floor plans take longer to generate: on average around one second per floor plan.

In our method, we included a simple mechanism for placing exterior elements, *i.e.* outer doors and windows. These wall elements are declared as requirements for each room. They are placed using the same method as for inner doors. The results show that these elements are placed correctly, if there is space available. In the future, it would be interesting to consider generating these elements automatically.

Multiple floors are also supported by our method. When generating the ground floor, a staircase (or elevator) room is generated. In the subsequent floors, this room is duplicated, fixed to the same position as the floor below. Adjacency and connectivity constraints for staircase rooms still hold as in normal rooms. Our results show that room expansion in higher floors is not affected by the initial duplication of the staircase. Rooms expand naturally around the staircase, respecting all constraints. An example of a two floor house is shown in Fig. 6.

Discussion

We discussed how our floor plan generation method is suitable for procedural building generation, since it provides fast and plausible floor plans. However, there is some room for improvement. Firstly, our connectivity algorithm identifies between which two rooms a door needs to be placed, but the actual location of the door is at random within the selected wall segment. This sometimes creates situations where the walk paths inside the building are not optimal. A smart approach for positioning these doors would be a helpful extension.

Another potential issue arises from the fact that we currently can not constrain the dimensions of a specific room. Our approach often creates L-shaped rooms, which might not be desirable in some particular cases. An example of this is a garage, which needs to have specific dimensions such that a car can be efficiently parked. An L-shaped garage makes not much sense in that respect. Constraints on the width-to-height ratio of a particular

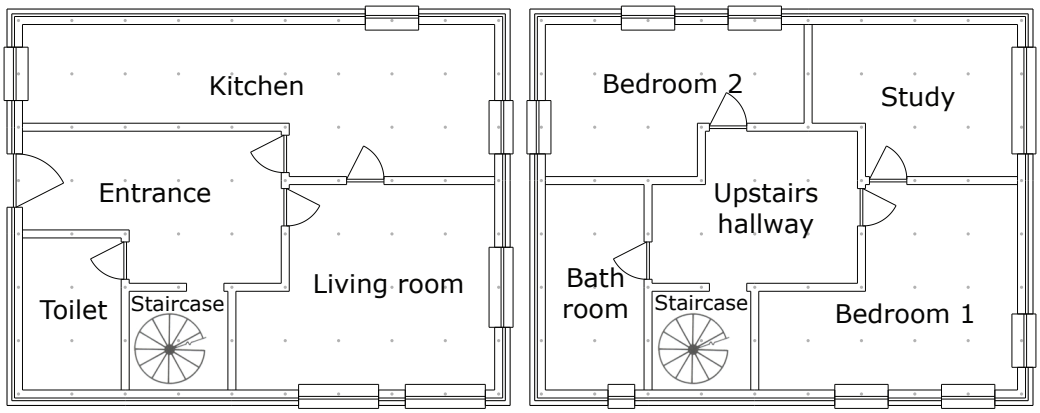


Figure 6: An example floorplan of a two floor house. Downstairs (left of the figure) an entrance, connected to a toilet, a staircase and a kitchen and living room, upstairs, the staircase is repeated and is connected to a hallway which in turn is connected to two bedrooms (one of which is connected with a study) and a bathroom.

room could improve the methods ability for handling these cases.

Since we use a grid-based algorithm, buildings with arbitrary angles are not supported. However, there are straightforward extensions for such cases as well. As in the technique used by Greuter et al. [6], a building's outer shape can be created by combining multiple geometric shapes. If all of these basic shapes would be constrained to axis-aligned polygons (with an arbitrary rotation in reference to the buildings orientation), these parts could be hierarchically subdivided into separate areas with our approach, and later combined into one complete building.

CONCLUSIONS

We can conclude that the method presented here efficiently generates valid and plausible building floor plans. We validate it in two ways. First, we visually compared floor plans generated by our method with many real-world plans of North-American houses, with which they typically matched well. However, although the method is very suitable for generating floor plans of houses, for highly regular structured spaces, such as some types of office buildings, more simple and specialized subdivision methods would be more efficient and effective. Second, we have consulted with architects and other experts on the output generated, and received valuable feedback. This has already resulted in further tuning of the method's parameters and constraints, for example on plausible adjacencies and desirable floor topologies. In the near future, we also expect this feedback to lead to the introduction of additional consistency tests.

An area which we could improve upon is the scoring mechanism for rating the set of alternative layouts generated by the method. This score could be a combination of the adjacency constraint satisfaction with a number of heuristics, *e.g.* minimum amount of internal doors, minimum amount of internal wall segments, etc.

Our implementation of the method performs well, due to its low complexity and efficient data structures. Although it is by far not optimized, it is still suitable in the context of dynamic generation of virtual worlds, where, *e.g.*, the interior of procedural buildings is generated within a small frustum around the player.

As future work, we would like to embed this method in the broader context of generating fully featured procedural cities. For this, we would have, among other things, to integrate the method with a façade generation system, resulting in complete buildings. Also, by using our semantic layout solving approach [11], we could fill the procedural interiors with appropriate furniture.

In our view, the constrained growth floor plan generation method is a small but significant step towards the goal of realistic and full-featured procedural cities.

References

[1] R. M. Smelik, K. J. de Kraker, T. Tutenel, R. Bidarra, and S. A. Groenewegen, "A Survey of Procedural Methods for Terrain Modelling," in *Proceedings of the CASA Workshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS)*, Amsterdam, The Netherlands, June 2009.

[2] P. Prusinkiewicz and A. Lindenmayer, *The Algorithmic Beauty of Plants*. New York, NY, USA: Springer-Verlag, 1990.

[3] Y. I. H. Parish and P. Müller, "Procedural Modeling of Cities," in *SIGGRAPH '01: Proceedings of the* 28^{th} *Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: ACM, 2001, pp. 301–308.

[4] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky, "Instant Architecture," in *SIGGRAPH '03: Proceedings of the* 30^{th} *Annual Conference on Computer*

Graphics and Interactive Techniques. New York, NY, USA: ACM, 2003, pp. 669–677.

[5] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. V. Gool, "Procedural Modeling of Buildings," in *SIGGRAPH '06: Proceedings of the* 33^{rd} *Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: ACM, 2006, pp. 614–623.

[6] S. Greuter, J. Parker, N. Stewart, and G. Leach, "Real-time Procedural Generation of 'Pseudo Infinite' Cities," in *GRAPHITE '03: Proceedings of the* 1^{st} *International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia*. New York, NY, USA: ACM, 2003, pp. 87–94.

[7] A. Rau-Chaplin, B. Mackay-Lyons, and P. Spierenburg, "The LaHave House Project: Towards an Automated Architectural Design Service," in *Proceedings of the International Conference on Computer-Aided Design (CADEX)*, Hagenberg, Austria, September 1996.

[8] J. Martin, "Procedural House Generation: a Method for Dynamically Generating Floor Plans," Research Poster presented Symposium on Interactive 3D Graphics and Games, 2006.

[9] E. Hahn, P. Bose, and A. Whitehead, "Persistent Realtime Building Interior Generation," in *Sandbox '06: Proc. of the ACM SIGGRAPH Symposium on Videogames*. New York, NY, USA: ACM, 2006, pp. 179–186.

[10] F. Marson and S. Musse, "Automatic Generation of Floor Plans Based on Squarified Treemaps Algorithm," *IJCGT International Journal on Computers Games Technology*, 2010, accepted for publication.

[11] T. Tutenel, R. Bidarra, R. M. Smelik, and K. J. de Kraker, "Rule-based Layout Solving and its Application to Procedural Interior Generation," in *Proceedings of the CASA Workshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS)*, Amsterdam, The Netherlands, June 2009.

[12] T. Tutenel, R. Bidarra, R. Smelik, and K. J. de Kraker, "The Role of Sematics in Games and Simulations," *ACM Computers in Entertainment*, vol. 6, pp. 1–35, 2008.

[13] P. Charman, "Solving Space Planning Problems Using Constraint Technology," in *Nato ASI Constraint Programming: Students' Presentations, TR CS 57/93, Institute of Cybernetics, Estonian Academy ofSciences, Tallinn, Estonia*, 1993, pp. 80–96.

ACKNOWLEDGEMENTS

This research has been supported by the GATE project, funded by the Netherlands Organization for Scientific Research (NWO) and the Netherlands ICT Research and Innovation Authority (ICT Regie).

BIOGRAPHY

Ricardo Lopes is a PhD student from the Delft University of Technology. His current research interests include: adaptivity in games, player modeling, interpretation mechanisms for in-game data and (on-line) procedural generation techniques. Ricardo got his master's degree in Information Systems and Computer Engineering at the Technical University of Lisbon, in Portugal.

Tim Tutenel is a PhD candidate from Delft University of Technology, participating in a research project entitled "Automatic creation of virtual worlds". His research focus is on layout solving, object semantics and object interactions. In particular, he is researching how semantics can improve procedural generation techniques. Tim got his master's degree in Computer Science at the Hasselt University in Belgium.

Ruben Smelik is a scientist at the TNO research institute in The Netherlands. He is a PhD candidate on a project entitled "Automatic creation of virtual worlds", in close cooperation with Delft University of Technology. This project aims at developing new tools and techniques for creating geo-typical virtual worlds for serious games and simulations. Ruben holds a masters degree in computer science from the University of Twente.

Klaas Jan de Kraker is a member of the scientific staff at the TNO research institute. He holds a PhD in computer science from Delft University of Technology. He has a background in computer-aided design and manufacturing, collaboration applications, software engineering (methodologies), meta-modeling and data modeling. Currently he is leading various simulation projects in the areas of simulation based performance assessment, collective mission simulation, multifunctional simulation and serious gaming.

Rafael Bidarra is associate professor Game Technology at the Faculty of Electrical Engineering, Mathematics and Computer Science of Delft University of Technology, The Netherlands. He graduated in electronics engineering at the University of Coimbra, Portugal, and received his PhD in computer science from Delft University of Technology. He leads the research line on game technology at the Computer Graphics Group. His current research interests include: procedural and semantic modeling techniques for the specification and generation of both virtual worlds and game play; serious gaming; semantics of navigation; and interpretation mechanisms for in-game data. He has published many papers in international journals, books and conference proceedings, and has served as member of several program committees.