EUROGRAPHICS 2014 / B. Lévy and J. Kautz
(Guest Editors)

Volume 33 (2014), Number 2

Game Level Layout from Design Specification

Chongyang Ma[*‡] Nicholas Vining[*] Sylvain Lefebvre[†] Alla Sheffer[*]

[*] University of British Columbia [†] ALICE/INRIA [‡] University of Southern California

Abstract

The design of video game environments, or levels, aims to control gameplay by steering the player through a sequence of designer-controlled steps, while simultaneously providing a visually engaging experience. Traditionally these levels are painstakingly designed by hand, often from pre-existing building blocks, or space templates. In this paper, we propose an algorithmic approach for automatically laying out game levels from user-specified blocks. Our method allows designers to retain control of the gameplay flow via user-specified level connectivity graphs, while relieving them from the tedious task of manually assembling the building blocks into a valid, plausible layout. Our method produces sequences of diverse layouts for the same input connectivity, allowing for repeated replay of a given level within a visually different, new environment. We support complex graph connectivities and various building block shapes, and are able to compute complex layouts in seconds. The two key components of our algorithm are the use of configuration spaces defining feasible relative positions of building blocks within a layout and a graph-decomposition based layout strategy that leverages graph connectivity to speed up convergence and avoid local minima. Together these two tools quickly steer the solution toward feasible layouts. We demonstrate our method on a variety of real-life inputs, and generate appealing layouts conforming to user specifications.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Curve, surface, solid, and object representations

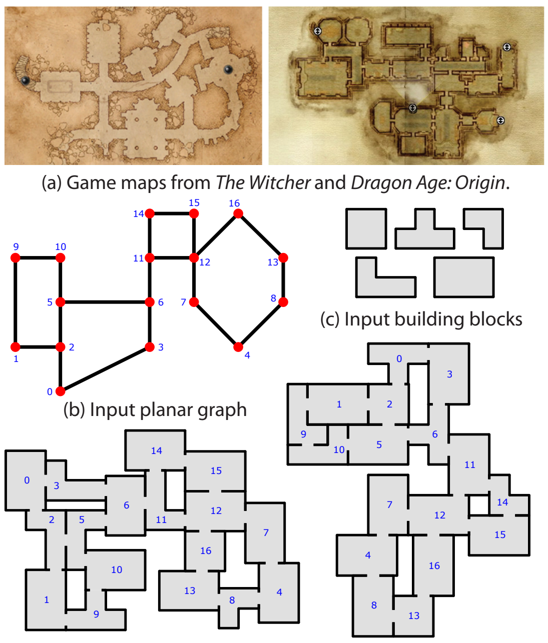

(a) Game maps from *The Witcher* and *Dragon Age: Origin*.
(b) Input planar graph
(c) Input building blocks
(d) Output level layout 1
(e) Output level layout 2

Figure 1: *(a) Game levels. (b-e) Level layout synthesis.*

1. Introduction

Typical video games contain complex virtual environments, or *game levels*, that players must traverse in order to advance through the story. Each level is a series of spaces, or rooms, with connections [Bar03]. Designers typically enforce a restricted level organization with linear passages that steers players to progress through a number of specific challenges: find a treasure, fight a monster, etc [Aar05]. Between each of these steps, players may freely explore their environment subject to the designer's impediments upon their progress [SZ03]. Rich, interesting game levels are a key component of a successful game (Figure 1a). Our work automatically computes visually engaging, complex, and diverse game levels that conform with designer specifications (Figure 1b-e).

Level layouts are typically manually constructed by game designers [Bar03]. Starting from a desired gameplay flow, designers typically construct each level from a set of artist-created building blocks, or spaces, some of which are unique while others are reused from level to level [Per02, BP13]. Since levels are created manually, they typically remain static from one gameplay session to another and the player experience never changes; consequently, a player forced to replay a level repeatedly experiences what is referred to as 'grinding'. Some games (e.g. so-called 'roguelikes') offer randomly generated levels, but the cost of random level generation is that designer control is lost and they can no longer steer the flow of gameplay [Cra04].

We propose a novel approach for game level layout generation, capable of automatically producing a variety of distinct game levels for replay while still allowing the designer to define the flow of gameplay. The input to our method consists of a planar connectivity graph that reflects the designer-intended gameplay flow (Figure 1b) and a set of polygonal building blocks (Figure 1c). Each graph node corresponds to a space, or room, in the desired layout, and the edges define the connectivity between them. We use the graph to assemble the building blocks into diverse possible layouts which satisfy the gameplay-flow encoded by the graph (Figure 1d,e).

The technical challenge we face is to layout the blocks such that two key constraints are satisfied: the layout must be *intersection-free*, i.e. we require that blocks within each level do not overlap, and must also satisfy all pairwise *contacts* - each pair of blocks connected by a graph edge must share a boundary segment long enough to place a doorway through. The interplay between the need to move blocks together to enforce contacts and the need to keep them apart to avoid intersections make game level layout a challenging problem, distinct from those addressed in other contexts (Section 2).

To obtain layouts that satisfy these constraints we develop an optimization strategy that leverages two geometry processing tools. First, we exploit graph connectivity to design a divide-and-conquer layout strategy, speeding up convergence to valid solutions. Second, instead of exploring the set of all possible building block positions, our algorithm considers reduced, continuous *configuration spaces* [LP83] defined for pairs of adjacent blocks. We use these configuration spaces to quickly evaluate contacts and to instantaneously improve them for individual blocks. We use these two ingredients to efficiently explore the solution space using a stochastic optimization process. We leverage partial solution caching to facilitate quick, on demand, generation of new distinct levels from the same graph whenever a player repeats a given level.

Our contribution is two-fold. We introduce the first game level layout algorithm for general 3D maps that provides high-level designer-control of gameflow while enabling the use of existing game building blocks. Our algorithm supports complex planar graph layouts, including graphs with multiple interconnected cycles that are common in game design, and arbitrarily shaped polygonal building blocks. Designers can enforce additional requirements on the generated levels, such as associating graph nodes with particular blocks and specifying appropriately connected multi-floor layouts (Section 4). Our algorithm can easily be integrated into existing game development workflows, and is suitable for a wide range of genres. Our technical contribution is a practical solution to a challenging layout problem, variants of which had been shown to be PSPACE-complete [HSS84]. Our experiments have shown our algorithm to robustly handle a large spectrum of typical game layout inputs (Section 4).

2. Background and Related Work

Industry Practices. Virtual worlds in games can be decomposed into a series of areas or levels [Bar03, Aar05, HC12, Ash11], Figure 1a. Games often include hundreds of levels, each typically containing up to a few dozen connected spaces [Bar03, Aar05]. Game designers use the connectivity between the spaces to control the flow of gameplay and to tell a compelling story [Cra04, Bar03, SZ03]. In typical game design, each level is constructed by hand, in a time-consuming and repetitive process. A widespread industry practice [Per02, BP13] is to create a set of templates, or building blocks, defining possible space shapes, and then reuse those multiple times when assembling a level. This approach allows for easy assembly of the final 3D realizations of the layouts. These practices guide our choice of user input - a graph defining the space connectivity and a set of 2D space templates or building blocks.

Automating Level Generation. Shaker et al. [STN14] discuss three methods for level generation. Recursive spatial partitioning is commonly used for architectural floorplan layouts and is not suitable for arbitrary game levels (see below). Agent-based layout methods, such as the one employed by Dungeons of Dredmor [Gas11], connect building blocks to each other using a greedy strategy that maintains a queue of "open positions" where a new door connecting two spaces can be placed. To ensure contact and avoid intersections they handle only acyclical layouts and use blocks quantized to and aligned with a fixed grid. Methods inspired by cellular automata [JYT10, ALM11] generate 2D layouts for organic environments such as caves and do not support the predefined blocks used to quickly assemble 3D levels.

Academic Research. The computer graphics community has so far focused on placement of objects within levels. e.g. [YYW*12] which places sand, water traps and holes on virtual golf courses, or on automatically adapting building blocks extracted from game levels [CLDD09], and has not addressed actual level layout. The artificial and computational intelligence community also investigates level design; see [TYSB11] for a comprehensive survey. The focus is on designing engaging level structures, or quest graphs. This is complementary to our goal as the generated quest graphs can serve as inputs to our method. A number of approaches produce levels for two-dimensional "side-scroller" environments [STY*11], in which the X direction represents "progress" and the "Y" direction represents vertical movement; we solve the harder problem of generating two-dimensional floorplans for 3D levels, with "progress" described by an arbitrary planar graph.

Graph Drawing. 2D planar graph drawing aims to generate intersection-free layouts of graphs that satisfy specific user requirements [NR04]. Level layout requires placing a polygonal building block at each vertex such that the blocks do not intersect and such that each pair of blocks corresponding to adjacent graph vertices share a common boundary segment. While it is easy to extend traditional graph-drawing algorithms to generate intersection-free block layouts (one can simply scale a 2D graph embedding till the blocks associated with each vertex no longer touch), it is the requirement for contact that makes our problem significantly harder. To the best of our knowledge this constraint had not been addressed before in the graph drawing community.

Procedural Geometry Layout. A variety of procedural methods have been successfully used for the design of cities [PM01, MM08, CEW*08, VKW*12] and buildings [WWSR03, MWH*06, LCOZ*11]. These approaches focus on the plausibility and aesthetics of the outputs, and do not aim to control contacts or adjacencies between individual buildings or interior spaces.

Floorplan Layout. Our work has strong links to the generation of floorplans for architectural spaces, also known as the spatial allocation problem [MS74, Sha87, Lig00, M-SK10, LYAM13, BYMW13]. As research in this area has focused on real-world architectural spaces, floorplans are typically generated for predefined envelopes, rooms are composed of axis-aligned walls, and no unoccupied voids are allowed between rooms. The main degree of freedom in this setting comes from the ability to scale rooms in the axial directions and to relax contact constraints. Contrary to these expectations, in a game setup we require the outputs to strictly satisfy contacts and operate on user-specified arbitrarily-shaped polygonal building blocks. At the same time, we have no constraints on envelope shapes or on the location of voids. In fact, flexible envelopes and interior voids are often used to add visual interest to game levels (Figure 1a) and make the layout feasible even for complex user-specified space graphs.

VLSI Layout. Our problem has some similarity to VLSI circuit layout [She98]. VLSI layout algorithms search for the best placement of axis-aligned cells and wires connecting them on a compact chip such that wire length is minimized and crossovers are avoided. This layout problem is known to be NP-complete, and algorithms for solving it focus on generating the best solution with considerable computational effort; our work concentrates on quickly providing multiple and varied layouts of differently shaped blocks strictly satisfying contacts, modifying the layout envelope at will.

Configuration Spaces. In motion planning for robotics, the configuration space of an object is the set of all possible transformations that can be applied to the object while avoiding intersections with other objects [LP83, LaV06]. Operating on the space of transformations converts complex geometric problems into simpler ones by spatial collapse. We use configuration spaces to assist in our space layout by formulating contacts and intersection avoidance as restrictions on configuration spaces. It has been observed that the general configuration space form of our layout problem, expressed solely for rectangular regions in a rectangular envelope, is PSPACE-complete [HSS84].

3D Game Levels. 2D game levels and floorplans can be converted into 3D architectural structures using a variety of methods [YWR09, KW11, MM08]. Given a game level layout assembled from a finite set of building blocks, Cabral et al [CLDD09] generate seamless 3D levels by deforming the blocks to match openings and appropriately resizing corridors, doors and stairs. Our outputs can directly be used as the input to their method.

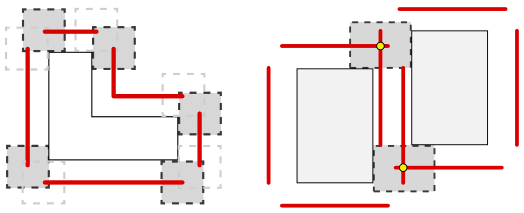


Figure 2: *Configuration spaces. Left: Configuration space (red) of the square block (dark) with respect to the L-shaped (light) one defines all the locations of the center of the square in which the two blocks are in contact and do not intersect. Right: Intersection of configuration spaces (yellow dots) of the moving dark block with respect to the two light ones. Pairwise configuration spaces shown in red.*

3. Algorithm

Our method takes a planar graph and a set of 2D polygonal building blocks as input data and generate various corresponding *valid* level layouts by arranging the blocks such that each graph node corresponds to an admissible building block, no two blocks intersect, and each pair of blocks corresponding to adjacent graph nodes share a common boundary segment (see Figures 1 and 3).

Combined together, these requirements define a high-dimensional, mixed continuous-discrete problem, where the continuous degrees of freedom are the block positions and the discrete ones are the node-to-block associations. A standard approach to such mixed problems would be to define an energy term encapsulating our requirements, and then to attempt to find a layout that minimizes this energy term using state-of-the-art stochastic optimization. However, naively doing so fails to take advantage of domain specific knowledge. Instead we use a tailored solution mechanism that leverages key level layout properties to quickly generate diverse layouts with high convergence rates. We compare our solution to more standard approaches in Section 4.

We first note that given a layout where all but one node's blocks and positions are fixed, we can directly compute the set of positions for the free node that best satisfy our constraints locally (i.e. with respect to its adjacent graph nodes) by computing the configuration space [LP83] of the block associated with this node with respect to the neighboring blocks (Section 3.1, Figure 2). Given a building block shape associated with the node, this configuration space defines a (possibly empty) set of line segments or points, such that placing the center of the block at any point in the space locally satisfies contacts and ensures that no local intersections occur (Figure 2). We cannot formulate the entire level layout problem solely as a configuration space computation, as even a restricted version of such a computation is PSPACE-hard [HSS84]. Using local configuration spaces within a randomized optimization setup (Section 3.3), however, not only lets us drastically speed up convergence but also supports our variability goal as placing a block at any location inside the configuration space locally satisfies our constraints.

A classical approach for speeding up optimization in high-dimensional spaces is to reduce dimensionality by breaking the input problem into smaller easier to solve sub-problems, appropriately communicating constraints between sub-problem solves. Following this strategy, we break the layout problem into a set of smaller ones, where at each step we only process a portion of the input graph, thus reducing the dimension of the search space (Section 3.2). Our challenge is to appropriately define these subgraphs and the communication strategy. A standard divide-and-conquer approach would recursively split the problem into equal smaller subgraphs, performing a bottom-up merge of partial solutions. However, this leads to merging two large partial solutions together, an operation as difficult as laying out the entire graph at once. We instead propose an incremental processing order, adding sequences of blocks (*chains*) to a partial solution. The added chains have roughly the same complexity, preventing the problem from growing out of proportion. We ensure that the generated partial layouts are both valid and connected at all times, increasing the odds that they can be extended into valid full layouts of the entire input graph. Our control mechanism allows for backtracking in the rare cases that this extension process fails.

Our basic algorithm can be extended to support a number of features desired by game level designers, showcased in Section 4. These include fixed or restricted building blocks, rotation and scaling of rooms, fixed positions, and multiple-floor layout generation. In particular we automatically enrich input building block sets by precomputing scaled and rotated variations of the basic blocks, then allowing the optimizer to select them as new blocks.

3.1. Configuration Space

In general, finding a high-dimensional configuration of objects that satisfies a set of hard non-linear constraints is known to be a difficult problem. However, we observe that for level layouts we can use geometric tools to directly compute local solution spaces, such that positioning a block inside these spaces best satisfies both intersection and contact constraints for this block with respect to its immediate neighbors in the input graph. Specifically given a pair of blocks, one fixed and one free, this set of valid positions is a union of 1D line segments and can be computed analytically (Figure 2, left). By leveraging these valid position sets, which in robotics literature are referred to as *configuration spaces* [LP83, LaV06], we dramatically reduce the size of the space that we must search.

We compute the configuration space for a pair of blocks, one of which is fixed and the other one of which is allowed to move, as follows. We fix a reference point on the moving block, e.g. its center, and consider all locations in R^2 such that, if the moving block is translated so that our reference

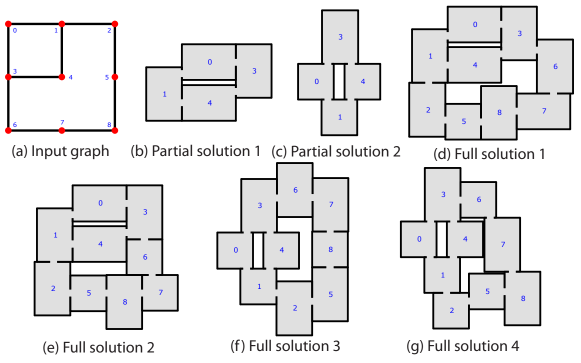


Figure 3: *Incremental level layout. Here (b) and (c) show two partial solutions after laying out the first chain; (d) and (e) show two full solutions after extending the partial layout in (b) to include the second chain; (f) and (g) show two full layouts extending the partial layout in (c).*

point is placed at this location, the stationary block and the moving block contact each other but do not intersect. Note that in order for two polygons to contact each other along a non-zero length segment, allowing for a doorway between them, they must touch along a common parallel edge. As the moving block slides along this edge, the fixed point on the moving block traces a line segment in Euclidean space. Repeating this process for every pair of parallel edges on the two blocks yields a collection of line segments, the *configuration space* of the two blocks (Figure 2, left). For each pair of parallel edges these segments can be computed efficiently using any number of geometric computing engines. The configuration space of a moving block with respect to two or more stationary ones, is simply the intersection of the individual configuration spaces placing the moving block in valid contact with each of its neighbors - or, in other words, the intersection of collections of line segments (Figure 2, right). These are usually points, but can be line segments if the block can contact both of its neighbors along two parallel or collinear edges.

Since our block geometry is fixed during optimization we precompute the configuration space for each pair of block shapes, facilitating quicker processing later on.

3.2. Incremental Layout

Configuration spaces let us dramatically reduce the local search spaces for individual graph nodes; however, the overall search space we consider remains too large for a solution to be located reliably, within a practical amount of time. To speed up processing we break the layout problem into, smaller, easier to solve ones. We note that chains, or graphs where each node has at most two neighbors, are relatively easy to lay out as the number of local contact constraints is always smaller than the number of block edges. Since a game layout is dominated by linear progression, we anticipate our input graphs to be dominated by a small set of long chains. We decompose our input graphs into chains, as described below, taking particular care to address graphs with cycles where the layout is more constrained.

Given a set of chains, we recall that our final output layout has to be a single connected component; hence there is no benefit in processing chains independently and then trying to join them into a single layout. We therefore process the set of chains incrementally (Pseudocode 1) using their adjacencies to guide us. We first compute a set of possible layouts for one chain, and then extend these partial layouts by adding neighboring chains, i.e. those sharing graph edges with the partial layout, one at a time, generating multiple extended layouts and storing them. The process terminates once all chains are processed and a sufficient number of layouts for the entire input graph have been found, or when no more distinct layouts can be computed. If an extension step fails, we backtrack to the last previously computed and stored partial layout and continue the extension process from it. An extension typically fails if the partial layout it starts from surrounds the blocks that need to come into contact with the new chain (Figure 4). The goal of generating multiple partial layouts instead of just one is twofold. First, we increase the likelihood that at least one of these layouts can be extended to a full layout, i.e. a layout of the entire graph. Second, by pre-caching partial layouts we facilitate quick, on-demand, creation of additional full layouts.

Pseudocode 1 Incremental level layout

```
Input: Planar graph $\mathcal{G}$, building blocks $\mathcal{B}$, layout stack $\mathcal{S}$
 1: procedure IncrementalLayout($\mathcal{G}$, $\mathcal{B}$, $\mathcal{S}$)
 2:     Push empty layout into $\mathcal{S}$
 3:     repeat
 4:         $s \leftarrow \mathcal{S}$.pop()
 5:         Get the next chain c to add to $s$
 6:         AddChain(c, $s$)   //extend the layout to contain c
 7:         if extended partial layouts were generated then
 8:             Push new partial layouts into $\mathcal{S}$
 9:         end if
10:     until target # of full layouts is generated or $\mathcal{S}$ is empty
11: end procedure
```

Our strategy when decomposing the graph into chains and when deciding on chain processing order is guided by two factors. First, we observe that cyclical chains are significantly more constrained than open-ended ones, as in the former case the blocks must form a cycle, with the last and first ones touching one another. We first decompose the input graph into a set of cycles and trees, using a planar embedding of the input graph generated using a standard algorithm [CP89], then find all the faces in the embedding. This embedding serves only for decomposition into chains and chain ordering, and the positions are discarded. We form the first chain using the edges of the face with the minimum number of edges in the embedding. We then iteratively consider neighbouring planar faces, picking a face and grouping all edges on that face that are not already in a cycle into a new cycle. Given multiple neighboring chains we select the shorter ones first. We repeat the process until all faces are processed. The remaining

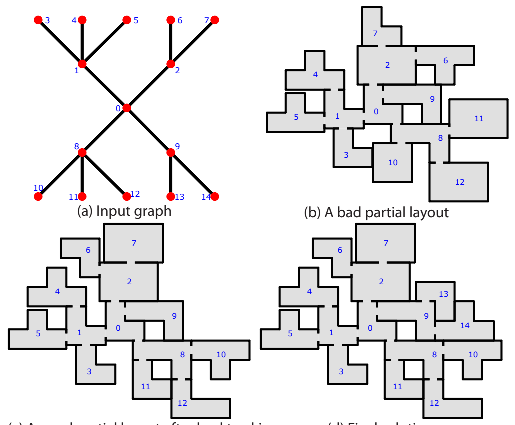


Figure 4: *A partial layout (b) of the input graph in (a) cannot be easily extended – rooms 13 and 14 cannot reach room 9. Backtracking to a different partial layout (c) facilitates a full layout generation (d).*

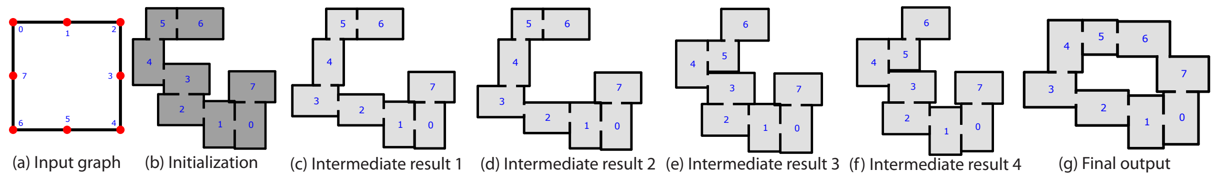
acylical graph components are then processed in a breadth-first order. The incremental layout processes the chains in creation order (Pseudocode 1). Our motivation for trying to place smaller cycles before larger cycles whenever possible is that small cycles impose less constraints on subsequent extension then larger ones. The breadth-first-order is chosen due to similar considerations.

3.3. Chain Layout

The goal of the chain layout step is to extend a, possibly empty, valid partial layout to include an additional chain that is connected to this layout via one or more graph edges (Figures 5 and 3). In other words, we need to find blocks and positions for the nodes of the added chain such that the extended layout is valid. To assist the layout of this new chain, we allow changes to the input partial layout, but keep the probability of those low. As earlier noted, instead of searching for one extended layout, we need to search for multiple ones to increase the chances of success in subsequent iterations of the chain-by-chain layout process.

To achieve this goal we use a simulated annealing framework, as its built-in randomization process is, in our experience, especially well suited for exploring multiple solution alternatives (Pseudocode 2). Simulated annealing operates by iteratively considering local perturbations to the current configuration, or layout, and moving to these configurations with some probability based on the energy of each new configuration and the current annealing temperature. Configurations that lower the energy have a higher probability of acceptance. As it proceeds, it keeps track of the best solution encountered, or, in our case, of the different valid layouts encountered.

Energy Function. Our energy function is designed to heavily penalize both intersections and missing contacts, and uses exponential terms designed to drop dramatically whenever



(a) Input graph (b) Initialization (c) Intermediate result 1 (d) Intermediate result 2 (e) Intermediate result 3 (f) Intermediate result 4 (g) Final output

Figure 5: *Chain layout. While some updates, e.g. (f) to (g) lower the energy, others support exploring the configuration space by accepting states that keep the energy constant or even increase it, e.g. (c) to (d).*

Pseudocode 2 Extend partial layout s adding the chain $\mathbf{c}$

```
 1: procedure ADDCHAIN($\mathcal{G}$, $\mathcal{B}$, $\mathcal{S}$, $\mathbf{c}$, $s$)
 2:     $t \leftarrow t_0$                                   // Initial temperature
 3:     for $i \leftarrow 1, n$ do                           // $n$: # of cycles in total
 4:         for $j \leftarrow 1, m$ do                       // $m$: # of trials per cycle
 5:             $s' \leftarrow$ Locally perturb $s \cup \mathbf{c}$
 6:             if $s'$ is valid then
 7:                 if $s \cup \mathbf{c}$ is full layout then output it
 8:                 else if $s'$ passes variability test
 9:                     Push $s'$ into $\mathcal{S}$
10:                     Return if enough extended layouts computed
11:                 end if
12:             end if
13:             if $\Delta\mathbf{E} < 0$ then                  // $\Delta\mathbf{E} = \mathbf{E}(s') - \mathbf{E}(s)$
14:                 $s \leftarrow s'$
15:             else if rand() $< e^{-\Delta\mathbf{E}/(k*t)}$ then
16:                 $s \leftarrow s'$
17:             else
18:                 Discard $s'$
19:             end if
20:         end for
21:         $t \leftarrow t \times ratio$                       // Cool down temperature
22:     end for
23: end procedure
```

an intersection is removed or a contact is achieved.

$$\mathbf{E} = e^{\frac{A}{\sigma}} \cdot e^{\frac{D}{\sigma}} - 1 \tag{1}$$

A is the total area of intersection between two blocks in the layout and D is the sum of squared distances between the center of pairs of blocks that are connected in the extended subgraph, but which are not in contact (or share a segment shorter than the user specified doorway width). The choice of σ impacts how frequently the annealing is allowed to move to higher-energy configuration. A smaller value of σ produces a higher success ratio, and a larger value of σ produces a quicker convergence. Empirically we found that setting σ to one hundred times the average block area provides a reasonable trade-off between the two.

Initialization. To speed up processing we aim to start the annealing in a low energy configuration. To this end, we generate a BFS ordering of the chain blocks starting with those adjacent to the input partial layout if one exists, or with a random root block otherwise. We place the blocks one at a time, sampling their configuration space with respect to their already laid out neighbors and selecting the sampled position with the lowest energy.

Local Perturbation. We perturb the current layout by changing the block position, or the node-to-block association, of one node in the extended graph containing both the nodes corresponding to the input partial layout and those of the currently processed chain. When deciding on the node to perturb, we only consider nodes in the currently added chain and nodes in the previous layout with non-zero energy (thus minimally perturbing the input partial layout). In our implementation, we set the ratio of position perturbations to node-to-block association changes to 7 to 3. When computing a new block position, instead of considering the Euclidean space around the current position, we directly consider the local configuration space of the given block with respect to its adjacent blocks in the extended graph. To do so, we compute the intersection of all configuration spaces of the block with respect to each individual adjacent block. We then randomly sample this intersection space to obtain a new position. If this intersection is empty, we compute the maximal non-empty intersection of spaces, and sample it instead. Using this approach we locally maximize the number of contacts satisfied at each iteration, pushing the layout closer to a desirable global solution.

Processing Valid Layouts. After each local perturbation, we check to see if the extended layout is now valid. We note that having each block within the configuration space of its neighbors is not sufficient for validity, and that we must also check for intersections between pairs of non-adjacent blocks. Each valid layout, which differs sufficiently from previously located ones, is placed on the stack for further processing by the incremental layout framework. The similarity between two partial layouts is defined as the sum of squared distances between corresponding block centers, after centering the two layouts around the origin. We terminate the algorithm when enough newly valid extended layouts have been found. For partial layouts we generate up to 15 variations. When processing the last chain, i.e when the output is a layout of the entire input graph, we use the user-specified target number of layouts.

Parameters. Pseudocode 2 uses $n = 50$ and $m = 500$ as the numbers of cycles and trials per cycle, $t_0 = 0.6$ and $t_1 = 0.2$ as the initial and final temperatures. The coefficient k in line 15 is computed using $\Delta\mathbf{E}$ averaging [Hed13].

4. Results

Throughout the paper we demonstrate the layouts generated by our method on nine different input connectivity graphs and multiple sets of building blocks. For all these inputs our method was able to generate multiple diverse layouts,

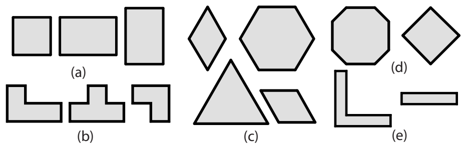


Figure 6: *Building block sets used throughout the paper.*

with some representative examples shown throughout the paper. As demonstrated in Figure 1 our input graphs are representative of the type of layouts commonly used in games, which contain multiple cycles and use up to a dozen differently shaped blocks.

Figure 7 showcases our ability to handle different types of building blocks, from rectangles, frequent in architectural layouts, through more complex axis-aligned shapes, to rich sets of diversely shaped polygons. Our method is able to painlessly process the different block sets, generating visually interesting and often surprising layouts from complex graphs with multiple cycles. It successfully tackles large graphs such as the one shown in Figure 8 which has forty vertices and leverages backtracking (Figure 4) to process graphs with high valence vertices (Figure 9), where contacts are particularly challenging to satisfy.

Figure 10 demonstrates the application of our method to multi-floor layouts. Here we use positional constraints on layouts generated for the second and third floors, forcing the highlighted blocks to match their first floor counterparts. In Figure 11 we generate layouts with walk-through corridors by constraining the configuration spaces of the corresponding blocks to allow contacts only at the two corridor ends. This example is typical of commonly used dungeon level designs. Figure 12 showcases another type of constraints - preventing the layout from intersecting user provided obstacles.

Statistics. Since we use a stochastic framework our method's behavior depends on the seeding of the different randomized components. To evaluate speed and robustness we ran the algorithm 50 times with different randomization seeds for each combination of input planar graph and building blocks. We record the rate of successfully computed valid solution, as well as the runtime required to find the first solution, and also the time and iteration count (i.e. trials to locally perturb a layout) necessary to compute ten solutions, see Table 1. Our success rate is very high, a perfect 50/50 on simple graphs and well over 90% for others. The only exception is the graph in Figure 7 which contains multiple interconnected cycles and is therefore particularly challenging to process. The layout cost is quickly amortized: For Figure 1, the first solution is obtained after 4.9 seconds, but the cost *per–solution* drops to 1.1 second after generating ten *different* solutions (as guaranteed by the variability test). Even on the most complex graphs the amortized cost is less than 10s per layout. This makes our scheme perfectly suited for our goal of generating many layout variations from a level specification.

Impact of Design Choices. We evaluated our algorithm design choices by comparing our results to those generated using alternative, more generic approaches (Table 2). We tested our solution framework without configuration spaces,

	success rate first/ tenth	#sol.	first solution	ten solutions	
			time avg/med	time avg/med	iter. # avg/med
Fig 1	1/0.94	9.8	4.9/2.3	10.9/6.8	51k/33k
Fig 7,top	1/1	10	1.1/0.4	1.8/1.2	7k/4k
Fig 7,bot	0.94/0.84	9.3	23/18	48/40	229k/187k
Fig 8	0.98/0.98	10	80/55	94/73	385k/295k
Fig 9	1/1	10	1.7/0.3	2/0.6	22k/6k

Table 1: *Algorithm statistics. We provide success rates for generating one or ten layouts, the average number of layouts created when ten were asked for (includes the unsuccessful attempts), times (sec) for generating one or ten layouts and iteration counts (thousands) for the later. We provide both average and median values as the average can be heavily influenced by outlier runs.*

	success rate first/tenth	#sol.	first solution	
			time avg/med	iter. # avg/med
NO config. space				
Fig 7,top	0.4/0	1	64/64	9k/4k
Fig 9	0.7/0	1	51/38	9k/1.5k
NO inc. layout:				
Fig 7,top	0.7/0	3.5	1.4/0.8	8k/1k
Fig 9	0.76/0	3.8	1.6/1.2	8k/5k

Table 2: *Statistic for algorithmic alternatives measured on graphs in Figure 7,top and 9. Both have significantly lower success rate for generating one result and never even come close to generating ten results. The non incremental method, when successful generates at most 3.5 results (on average). Without using configuration spaces we ar enot able to ever generate more than one result on even the simplest graphs.*

using a purely energy-based formulation. Using constant or randomized changes in position, the method was never able to satisfy contacts exactly. We were more successful when using a combination of random walk and line search for locating lower energy close-by configurations (Table 2). However even on simple examples, the success rate remained very low and when convergence did occur, runtimes were fifty times worse than ours. Additionally, we were able to find at most one solution during a trial run (Table 2). We also examined the impact of skipping the incremental layout process and optimizing the entire layout at once using the framework in Section 3.3. Without the incremental layout, even for our simplest inputs we were only able to achieve valid solutions 70% of the time, and were only ever able to find at most 5 solutions when searching for 10 (Table 2). While more advanced stochastic methods may in theory perform better, our approach as-is quickly achieves the desired solution.

To perform the comparison, we ran each algorithm 50 times with different seeds using the same input planar graph and building blocks, on the graphs in Figures 7, top and 9. As shown in Table 2 using either of the simplified methods drastically reduces the percentage of successful runs (ones where a layout is successfully obtained), and were unable to generate multiple solutions when requested.

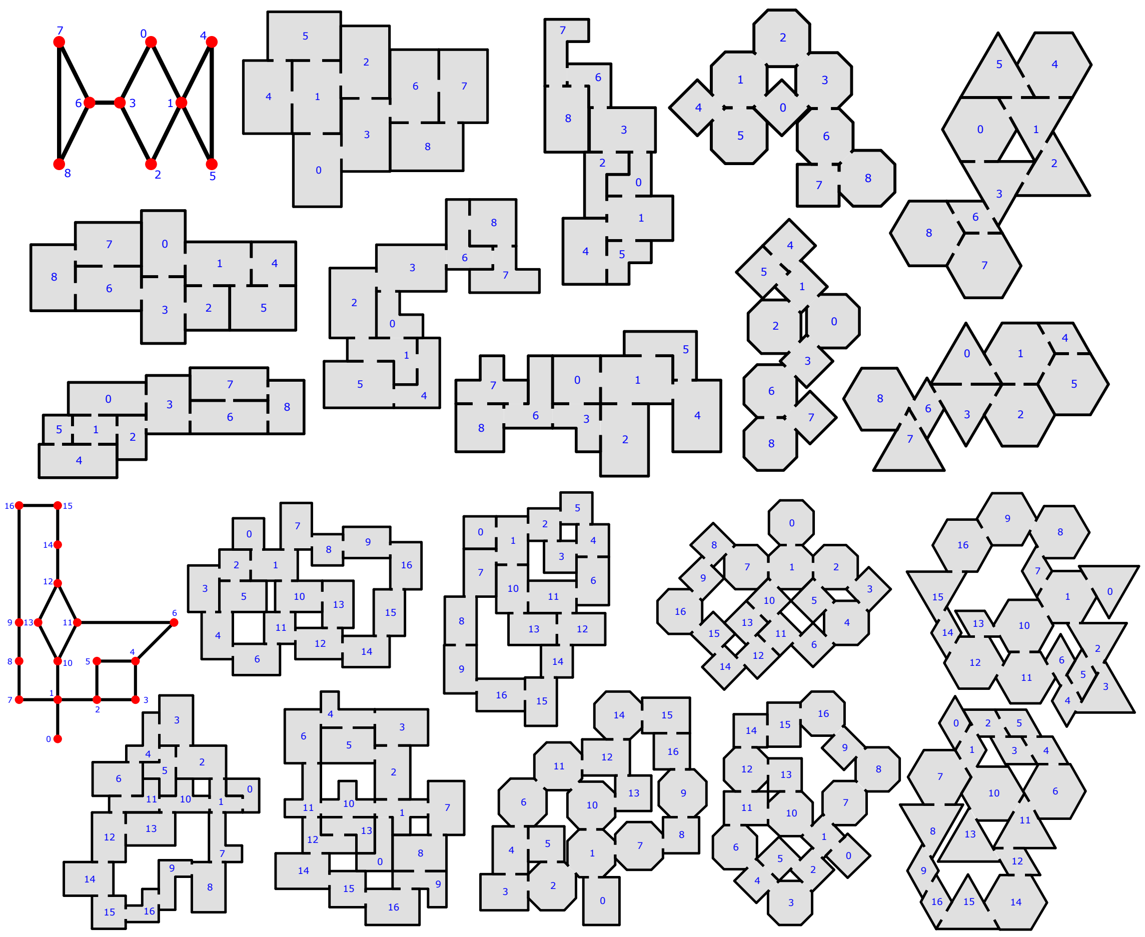


Figure 7: *Layouts generated from two input graphs using different sets of building blocks from Figure 6.*

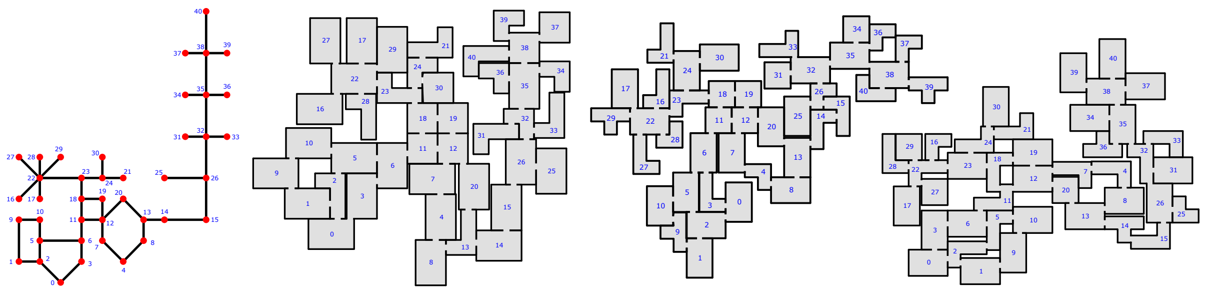


Figure 8: *Diverse layouts created using the same graph and block set.*

We asked ten professional game developers from independent and AAA studios to review our generated layouts. Feedback was uniformly positive; results were rated as being "nice and natural", "more complex than can easily be made by hand", and "layout and packing feel good". One designer suggested our work as a brainstorming tool for exploring level ideas. One designer asked for doors to not be placed directly opposite each other, and we can account for this by adjusting the configuration spaces at each step of the algorithm.

Discussion. There is an intriguing link between the shape of the blocks and the existence of a solution. Consider a cycle of four blocks all connected to a single, enclosed block. If the only available shape is a rectangle twice as long as it is high, then the cycle does not offer enough space to fit the block inside. This is a very simple instance of a more subtle issue: whenever the graph is made of nested cycles, using shapes elongated along the same axis will make the layout more challenging, as their aspect ratio limits the available area enclosed by a cycle of a given number of blocks. Similarly,

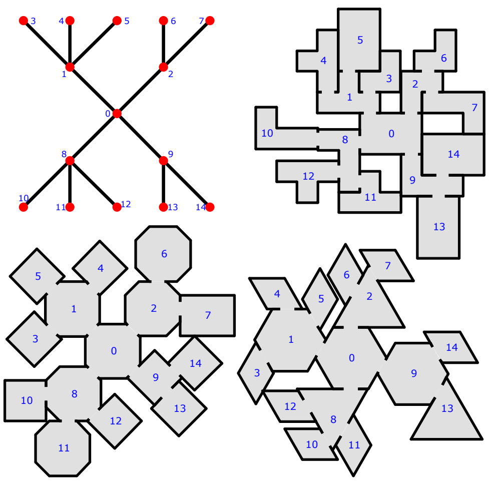


Figure 9: *Level layouts for a tree with high valence vertices.*

some shapes may induce a directivity in the layout because they offer a simpler or longer edge interface on one side. Therefore, a good guideline for designers is to provide a sufficient variety of shapes, globally having wall interfaces in all directions. This limits biases during the layout process. Note that this issue is unique to our setting: Approaches for architectural floor planning [MSK10, LYAM13] do not consider this constraint and let the shape of the rooms emerge from the optimization.

5. Conclusions

We presented the first method for game level layout able to preserve designer-intended gameflow. Given a user-specified layout connectivity graph our method is fast enough to on-demand generate different diverse layouts that satisfy the contact constraints imposed by the graph, supporting an enjoyable replay experience. Our experiments show that our method can robustly handle complex graph representative of typical game levels.

Our current algorithm is designed to satisfy hard constraints such as intersection-avoidance and contact. It would be interesting to investigate additional criteria to guide level synthesis based on design goals in production scenarios. While the speed of our method is acceptable for most gameplay setups, it would be interesting to explore future speedups, e.g. by replacing the simulated annealing with more sophisticated stochastic search techniques. The key here would be to achieve speedup, without sacrificing the output variability we currently achieve. Finally, we may explore ways to increase output variability, e.g. by allowing blocks to deform during layout computation.

Acknowledgements We thank anonymous reviewers for their valuable comments. This research was partly supported by NSERC, GRAND NCE and ERC ShapeForge (StG-2012-307877).

References

[Aar05] Aarseth E.: From hunt the wumpus to everquest: Introduction to quest theory. In *Proceedings of the 4th International Conference on Entertainment Computing* (2005), Springer-Verlag, pp. 496–506. 1, 2

[ALM11] Ashlock D., Lee C., McGuinness C.: Search-based procedural generation of maze-like levels. *IEEE Trans. Comput. Intellig. and AI in Games 3* (2011), 260–273. 2

[Ash11] Ashby A.: *Legend of Zelda: Skyward Sword (Prima Official Game Guide)*. Prima Games, 2011. 2

[Bar03] Bartle R.: *Designing Virtual Worlds*. New Riders Games, 2003. 1, 2

[BP13] Burgess J., Purkeypile N.: Skyrim's modular approach to level design. Game Developers Conf., 2013. 1, 2

[BYMW13] Bao F., Yan D.-M., Mitra N. J., Wonka P.: Generating and exploring good building layouts. *ACM Trans. Graph. 32*, 4 (2013), 122:1–122:10. 3

[CEW*08] Chen G., Esch G., Wonka P., Müller P., Zhang E.: Interactive procedural street modeling. *ACM Trans. Graph. 27*, 3 (2008), 103:1–103:10. 3

[CLDD09] Cabral M., Lefebvre S., Dachsbacher C., Drettakis G.: Structure-Preserving Reshape for Textured Architectural Scenes. *Computer Graphics Forum 28*, 2 (2009), 469–480. 2, 3

[CP89] Chrobak M., Payne T.: A linear-time algorithm for drawing a planar graph on a grid. *Information Processing Letters 54* (1989), 241–246. 5

[Cra04] Crawford C.: *Chris Crawford on Interactive Storytelling*. New Riders Games, 2004. 2

[Gas11] Gaslamp Games, Inc.: Dungeons of dredmor, 2011. URL: http://www.gaslampgames.com. 2

[HC12] Hodgson D., Cornett S.: *Elder Scrolls V: Skyrim Revised and Expanded: Prima Official Game Guide*. Prima Games, 2012. 2

[Hed13] Hedengren J.: Simmulated annealing tutorial, 2013. URL: http://apmonitor.com/me575/index.php/Main/SimulatedAnnealing. 6

[HSS84] Hopcroft J., Schwartz J., Sharir M.: On the complexity of motion planning for multiple independent objects; pspace hardness of the "warehouseman's problem". *International Journal of Robotics Research 4*, 3 (1984), 76–88. 2, 3

[JYT10] Johnson L., Yannakakis G. N., Togelius J.: Cellular automata for real-time generation of infinite cave levels. In *Proc. 2010 Workshop on Procedural Content Generation in Games* (2010), pp. 10:1–10:4. 2

[KW11] Kelly T., Wonka P.: Interactive architectural modeling with procedural extrusions. *ACM Trans. Graph. 30*, 2 (2011), 14:1–14:15. 3

[LaV06] LaValle S. M.: *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. 3, 4

[LCOZ*11] Lin J., Cohen-Or D., Zhang H., Liang C., Sharf A., Deussen O., Chen B.: Structure-preserving retargeting of irregular 3d architecture. *ACM Trans. Graph. 30*, 6 (2011), 183:1–183:10. 3

[Lig00] Liggett R.: Automated facilities layout: past, present and future. *Automation in Construction 9*, 2 (2000), 197–215. 3

[LP83] Lozano-Pérez T.: Spatial planning: A configuration space approach. *IEEE Trans. on Comp. 32* (1983), 108–120. 2, 3, 4

[LYAM13] Liu H., Yang Y.-L., AlHalawani S., Mitra N.: Constraint-aware interior layout exploration for pre-cast concrete-based buildings. *The Visual Computer 29*, 6-8 (2013), 663–673. 3, 9

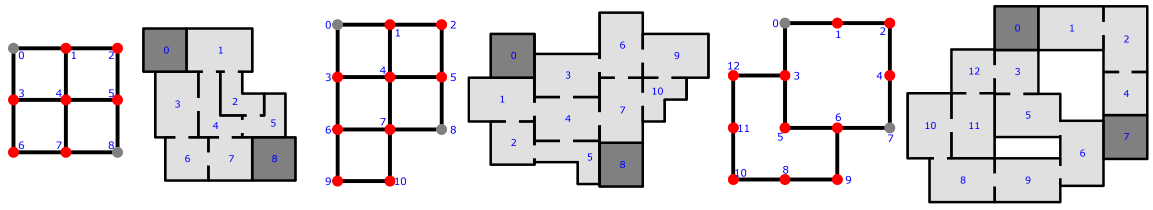


Figure 10: *Multi-floor layout generated by constraining stairwells (dark gray) to match on all floors.*

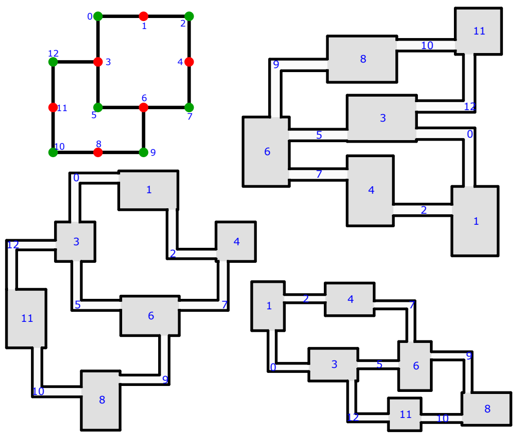


Figure 11: *Layout with restricted door positions. Here we use two set of building blocks for different types of nodes in the input planar graph (top-left). The red nodes correspond to room blocks, while the green nodes correspond to narrow corridors with door positions restricted to be at the two ends.*

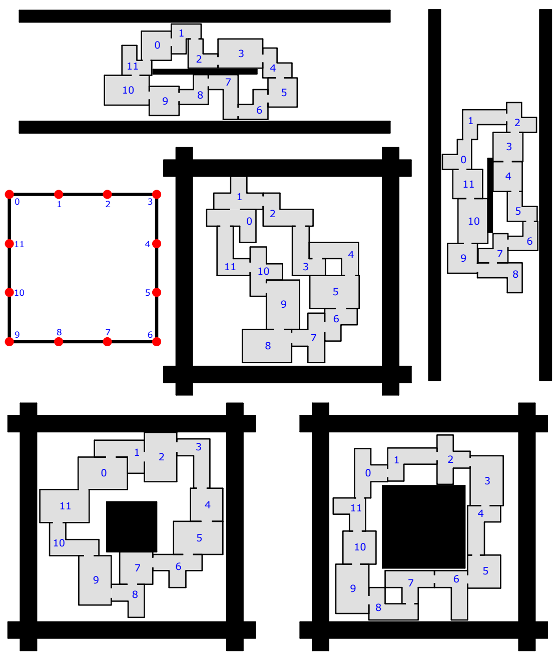


Figure 12: *Constrained levels designed to avoid different polygonal obstacles. These types of constraints are frequent in games, e.g. designing a palace to wind around a lake.*

[MM08] Merrell P., Manocha D.: Continuous model synthesis. *ACM Trans. Graph. 27*, 5 (2008), 158:1–158:7. 3

[MS74] March L., Steadman P.: *The geometry of environment: an introduction to spatial organization in design*. M.I.T. Press, 1974. 3

[MSK10] Merrell P., Schkufza E., Koltun V.: Computer-generated residential building layouts. *ACM Trans. Graph. 29*, 6 (2010), 181:1–181:12. 3, 9

[MWH*06] Müller P., Wonka P., Haegler S., Ulmer A., Van Gool L.: Procedural modeling of buildings. *ACM Trans. Graph. 25*, 3 (2006), 614–623. 3

[NR04] Nishizeki T., Rahman M.: *Planar Graph Drawing*. Lecture notes series on computing. World Scientific, 2004. 2

[Per02] Perry L.: Modular level and component design. *Game Developer Magazine* (2002). 1, 2

[PM01] Parish Y. I. H., Müller P.: Procedural modeling of cities. In *Proc. SIGGRAPH* (2001), pp. 301–308. 3

[Sha87] Shaviv E.: Principles of computer-aided design: computability of design. Wiley-Interscience, New York, NY, USA, 1987, pp. 191–212. 3

[She98] Sherwani N. A.: *Algorithms for VLSI Physcial Design Automation*, 3rd ed. Kluwer Academic Publishers, Norwell, MA, USA, 1998. 3

[STN14] Shaker N., Togelius J., Nelson M. J.: *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2014. 2

[STY*11] Shaker N., Togelius J., Yannakakis G., Weber B., Shimizu T., Hashiyama T., Sorenson N., Pasquier P., Mawhorter P., Takahashi G., Smith G., Baumgarten R.: The 2010 mario ai championship: Level generation track. *IEEE Trans. Comput. Intellig. and AI in Games 3*, 4 (2011), 332–347. 2

[SZ03] Salen K., Zimmerman E.: *Rules of Play: Game Design Fundamentals*. The MIT Press, 2003. 1, 2

[TYSB11] Togelius J., Yannakakis G. N., Stanley K. O., Browne C.: Search-based procedural content generation: A taxonomy and survey. *IEEE Trans. Comput. Intellig. and AI in Games 3*, 3 (2011), 172–186. 2

[VKW*12] Vanegas C. A., Kelly T., Weber B., Halatsch J., Aliaga D. G., Müller P.: Procedural generation of parcels in urban modeling. *Comp. Graph. Forum 31* (2012), 681–690. 3

[WWSR03] Wonka P., Wimmer M., Sillion F., Ribarsky W.: Instant architecture. *ACM Trans. Graph. 22*, 3 (2003), 669–677. 3

[YWR09] Yin X., Wonka P., Razdan A.: Generating 3d building models from architectural drawings: A survey. *IEEE Comput. Graph. Appl. 29*, 1 (2009), 20–30. 3

[YYW*12] Yeh Y.-T., Yang L., Watson M., Goodman N. D., Hanrahan P.: Synthesizing open worlds with constraints using locally annealed reversible jump mcmc. *ACM Trans. Graph. 31*, 4 (2012), 56:1–56:11. 2