Solusforge: Controlling the Generation of the 3D models with Spatial Relation Graphs

Jo Mazeika
Univ. of California Santa Cruz
Dept. of Computer Science
jmazeika@soe.ucsc.edu

Jim Whitehead
Univ. of California Santa Cruz
Dept. of Computational Media
ejw@soe.ucsc.edu

ABSTRACT

In this paper, we propose Solusforge, a system for automatically generating Lego models from a graph of the components' spatial relationships. The system uses a two step constraint solving approach in which the spatial layout is solved for first, followed by the specific pieces that make up the model, thereby allowing us to explore two separate solution spaces independently. This technology has many uses, including in games featuring a system of snap-together pieces, including Kerbal Space Program, Beseiged, and Spore. While many of these games involve procedurally augmenting human generated design, none of them feature a fully procedural system for generating the artifacts within that space.

CCS CONCEPTS

•Computing methodologies → Shape modeling; •Theory of computation → *Constraint and logic programming;*

KEYWORDS

Lego; Procedural Generation; Constraint Solving

ACM Reference format:
Jo Mazeika and Jim Whitehead. 2017. Solusforge: Controlling the Generation of the 3D models with Spatial Relation Graphs. In *Proceedings of FDG'17, Hyannis, MA, USA, August 14-17, 2017,* 4 pages.
DOI: 10.1145/3102071.3106348

1 INTRODUCTION

Procedural content generation techniques are often used in games to create the spaces where gameplay takes place, such as planets, terrain, caves, and levels. Less common is the use of procedural content generation techniques for creating in-game 3D objects, which is primarily focused on vegetation and buildings, though with occasional use outside these domains (e.g., the procedural weapons in the Borderlands series) [7]. Generation of 3D objects in games usually takes the form of a *one-way* generation process: the algorithm begins in one location, and builds out content from there. For example, the typical approach for tree or building generation is to use generative grammars (or L-systems). These systems start at a specific location, and then build out the content in space following the grammar rewrite rules.

FDG'17, Hyannis, MA, USA
 978-1-4503-5319-9/17/08…$15.00
DOI: 10.1145/3102071.3106348

While grammars and other one-way constructive algorithms are effective at creating 3D objects, they have several shortcomings. First, due to the directionality of generation, any change made to geometry created early in the process will have significant ripple effects on the remainder of the object. A further drawback of one-way generation approaches is their limited support for modularity. Ideally we would like to generate objects via the composition of several sub-pieces, each of which may themselves be generated. Grammar based approaches support this to some degree via specific rewrite rules which represent the construction of a particular sub-piece. However, grammar-based approaches are limited by the difficulty of predicting all of the ways a sub-piece may geometrically interact with an emerging model. For example, a sub-piece placed via a grammar may accidentally intersect other parts of the model. In order to support generation of 3D objects out of compositions of sub-parts, it is necessary to express spatial relationship constraints (e.g., above, beside, next to, etc.) among the sub-pieces, which are then evaluated using a constraint solving system. This permits sub-pieces to be defined independently of one another, and then combined to create a spatially consistent complete model.

This paper presents our first steps at creating a system for procedurally generating 3D objects modeled as a composition of multiple sub-pieces. Our system, Solusforge, is a two stage generative system for the generation of 3D structures. Solusforge takes as input a hierarchical specification of the various parts of the final product, as well as the relative positional constraints that exist between each of these parts. In the first stage, Solusforge generates a layout where the spatial positions of each of the parts are fixed. In the second stage, this layout is converted into the specific pieces used in the underlying representation, which are then used to build the final intended artifact.

To reduce the overall complexity of the problem, we focus on generation of a class of useful connected block objects, specifically Lego models. Lego models have appeared in a wide range of Lego-themed video games. As well, snap-together piece models appear in other games, such as Kerbal Space Program, Besieged, and Spore. As compared to the full complexity of 3D object generation, Lego models have the benefit of being created from a set of voxel-like blocks created on a grid, and tend to have less complex geometries than the full set of possible 3D models. However, the large number of existing Lego models demonstrates the wide expressive range possible even under the constraints of the Lego block system.

In the remainder of the paper we begin with an overview of related work in Section 2. Section 3 presents an overview of the Solusforge system, before diving into details of the 3D object representation in Section 4. Section 5 then details the two-phase process

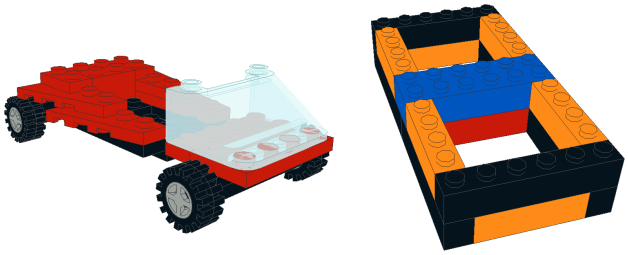

Figure 1: Two sample outputs; a car model and a twin-squares model

(sketch, then realization) for realizing 3D objects such that they are consistent with a set of spatial constraints.

2 PRIOR WORK

Game asset generation is a rich field of research, and work has been done in a wide array of domains, including trees [2], caves [8], buildings [11, 17] and even entire cities [12]. These systems all use a variety of techniques, but most of them lean heavily on L-system grammars, which allow for the organic evolution of the artifacts.

However, in this paper, our technology of choice is constraint solving, and more specifically Answer Set Programming. Answer Set Programming (ASP) is a declarative problem solving paradigm in which the programs represent the specification of problem, which is solved by executing the program in an answer-set solver [4]. In particular, we use Gringo (version 5), the chosen language of the Potassco ASP project.[1]

In recent years, several Procedural Generation systems have used ASP as the backbone of their generative process [1, 15]. For generation tasks, a core strength of ASP is the ability to carve away undesirable portions of the expressive range of a generator without having to redefine the base generative space.

However, ASP is not the only constraint solving paradigm used for procedural generation. Merrell in [10] describes a system that extrapolates out a larger area from a much smaller model, and there is an entire literature devoted to using constraint solving for space layout problems [9, 14].

Finally, there are a number of papers on generative Lego structures, meaning that the physical properties of Lego bricks are well modeled. Several systems have been implemented to generate arbitrary free-standing Lego structures [3, 5, 13, 16]. The baseline structure of Legos, rectangular bricks that can connect top-to-bottom, is very simple to represent and reason over within a computational system (similarly to voxels), and because of these well-known physical properties, we can worry less about a model being nonviable.

3 SYSTEM OVERVIEW

Solusforge is a system for generating Lego models from a high-level hierarchical representation of the structure, which we call the *specification* of the structure. In the specification, we incorporate all of the positional constraints of the model, as well the hierarchy of the different base-level components of the model.

[1] https://potassco.org/

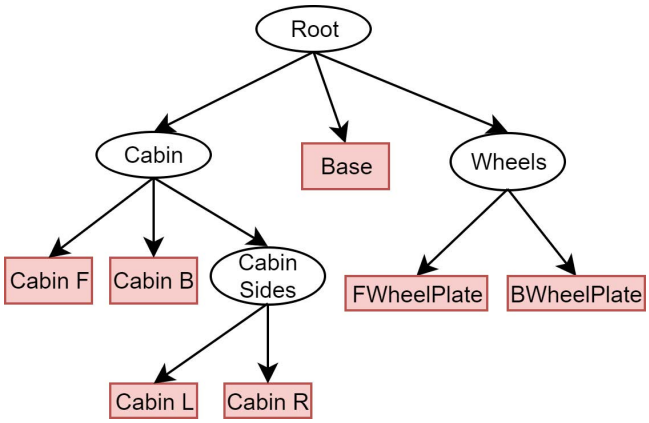

Figure 2: Hierarchical decomposition of a Lego car, with the node representing Lego groups highlighted

From the specification, we generate a *sketch*. A sketch fixes the particular sizes and positions of the components contained in the specification. Depending on the constraints and components, multiple sketches can be created from a single specification. The differences between the sketches all are visually distinct, which allows a user to sift through the sketches to find ones that meet particular criteria, or to provide alternate options for the realization. Then, we have the realization step. In this, we take the sketches and decide on a configuration of Lego bricks to fill out the sketch so that the final model can be realized. This step mainly focuses on solving for the sets of pieces that fill in specified volumes of spaces. There can be multiple different layouts produced from the same sketch, representing different possible realizations into Lego bricks.

4 OBJECT REPRESENTATION

In order to generate 3D Lego objects, we need a way of encoding our models that allows for expressivity both from the human author and the generative system itself.

To that end, we developed a hierarchical object representation using a tree structure. The format is a decomposition hierarchy, where the children of a node represent the proper parts—or parts that are not equal to the whole—of that node. For instance, consider the decomposition of a Lego car shown in Figure 2. At the bottom of the tree are indivisible leaves, the portions of the whole that cannot be subdivided further, and that ground out into actual collections of Lego bricks. These collections are called "Lego groups," and the object representation can be seen as a way of organizing these Lego groups. Referring back to our car example, Figure 2 shows the different Lego groups within the decomposition, highlighting the nodes in question.

Lego groups are divided, broadly, into two categories: *piece groups* that represent a finite and fixed collection of pieces, and *volume groups* that instead represent a volume of space to filled in by some combination of pieces. This distinction will be important in the generation process, as the system treats these separately for two reasons.

First, volume groups are allowed to intersect and overlap arbitrarily with other groups, while the intersection of two piece

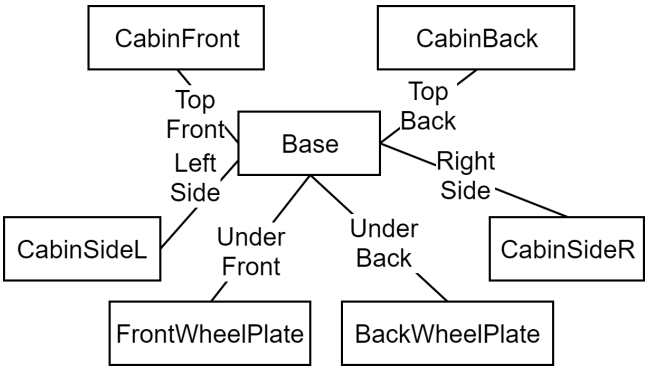

Figure 3: The graph of constraints on the car model

groups is strictly prohibited. This allows for expression of concepts like "A window is placed in the middle of a wall" and "Two rooms, which have their walls represented as a volume group, share a wall" without having to break up the representations in counterintuitive ways. Because of these, we have our second reason, namely that the generator separates out from the rest of the spatial constraint solving, the step in which it solves for the precise configuration of bricks that make up a volume group. This will be detailed later, in the description of the generation process.

Of course, in order to say where two groups are relative to each other requires additional information in the object representation beyond the simple hierarchy. So, in addition to the decomposition hierarchy above, we also include a general graph of constraints that exist between the different nodes. These constraints are positional information, such as a piece group inset to a volume group, or a piece group atop of another piece group. This allows us to constrain the relative positions of Lego groups as much or as little as we need, barring some fixed properties. For our car model, Figure 3 shows the graph of constraints.

Formally, an object representation is as follows:

First is a list that describes the tree. The first line always refers to the root, and each subsequent line has the ID of the node (typically a human-readable name, but it can be any unique string), the name of its parent, and the name and parameters of the associated Lego group, if any. For instance, consider the example of the car shown in Figure 1. The representation of the hierarchy looks like:

Root.
Base (lg_CarBase) → Root.
Wheels → Root.
FrontWheelPlate (lg_PlateWithWheels) → Wheels.
BackWheelPlate (lg_PlateWithWheels) → Wheels.
Cabin → Root.
CabinFront (lg_WindshieldWithBase) → Cabin.
CabinBack (lg_BrickVolume) → Cabin.
CabinSides → Cabin.
CabinSideL (lg_SplayedPlatesL) → CabinSides.
CabinSideR (lg_SplayedPlatesR) → CabinSides.

Next is the set of constraints required to ensure that all of the pieces are in their correct place, forming the graph shown in Figure 3. In the file, these are represented in the following way:

FrontWheelPlate → Base (UnderFront).
CabinFront → Base (TopFront).
...

5 OBJECT REALIZATION

With our object specification in place, we now need to generate a Lego model from it. Clingo, an ASP solver, is the primary technology [6] use for both the Sketch and Realization steps in Solusforge. In this way, our system flow is simply to setup the constraint solving problems, pass them to Clingo, and interpret the returned answer set or sets. While the order of statements in ASP files doesn't matter, different chunks of the files are grouped together for readability and comprehension[2].

5.1 Sketch Step

The sketch step solves for the sizes and positions of all of the groups in an object, providing a fixed layout for the final model. This step takes as input the object specification. First, we setup some initial machinery — bounds on the overall size of the model, and the rules for how to handle the data from the specification.

We list all of the Lego groups from the specification, including their name and their size, as well as all of the constraints from the specification's graph. Finally, we include one final set of common sense constraints — no two piece groups can intersect each other in space. Since piece groups represent physical Lego pieces, their intersection would be the equivalent of physical Lego pieces intersecting each other. However, volume groups can be intersected by piece groups and other volume groups. This allows us to represent windows inserted into walls and two rooms sharing walls in a simple fashion.

We can now solve for the sketches, which consist of the positions and sizes for all of the Lego groups. By collecting multiple variations, and adjusting for the models being offset relative to each other, we can construct the complete possibility space for the sketches that can be produced by the specification. While some of the changes between the sketches may be subtle, they are noticeable and non-trivial compared to the difference between results in the next step.

For instance, let's refer back to the car example. Here, we have the constraints and groups shown in Figure 3. We list all of the groups as follows, referring to them not by the name of the node, but the name of the underlying Lego group:

- Piece Groups: "Base", (10,1,4)
  "CabinFront", (3,4,4)
  "CabinSideR", (6,3,2)
  "CabinSideL", (6,3,2)
  "FrontWheelPlate", (1,2,6)
  "BackWheelPlate", (1,2,6)
- Volume Groups: "CabinBack"

The numbers in each of the piece groups represent the sizes of each of the groups. We also include all of the constraints, similarly to how they were referred to in the input file, consisting of the pairs of pieces and the description of how they relate to each other.

[2]See https://github.com/jomazeika/SolusForge for detailed code

From this, we solve for the positions of every group, as well as the sizes for the volume groups.

5.2 Realization Step

Once we have a sketch—either from the Sketching step, or from some other process—we need to realize it into actual Lego bricks. This process is very straightforward for piece groups, which simply get transformed into their respective pieces.

However, this process is less straightforward for the volume groups, as these don't have a specific representation as pieces, both because they don't need to have a specified size in the model, but also because of the arbitrary intersections that can occur with them. So, we need to find the set of bricks that exactly fill in the volumes, and only the space occupied by the volumes.

First we specify the set of Lego bricks that the system can use to fill in the volumes. This choice comes with trade-offs: having more pieces to choose from, and smaller pieces included, leads to greater precision over what kinds of shapes can be filled in but increases the runtime as the space of possible answers increases. So, we only consider a set of eight bricks (ranging from 2x1 to 6x2 in dimension).

Once we have those in place, our next step is to set up the problem. We use a lot of the information from the sketch, including the size of the space as well as the positions and locations of all of the Lego groups. To do this, the system views the volume groups as an empty set of cells (unit volumes) that need to be filled in. Duplicate cells are ignored, i.e., cells that belong to two different groups are only considered once. In this way, all of the volume groups in a sketch are treated as a single mass to fill in, which allows us to handle the intersections of volume groups nicely. Additionally, some volume groups can be hollow, and so we specify cells that are meant to be hollow. Any cells with that label are treated as already filled in by the system.

With those, we have the set of cells that must be filled in. Next, we include the piece groups, using the positions from the sketch to place them. Any of the cells from the volumes that these groups intersect with are treated as filled in (as the piece group is already occupying those spaces), neatly handling the issue of intersections with piece groups.

At this point, all of the requite information is required for specifying the space that needs to be filled in. The last thing to incorporate before starting the solving process is an integrity constraint. One of the main ways that Lego insures that their models are stable is by interweaving the pieces: having one row of bricks face horizontally, and the next vertical. This is accomplished by enforcing the constraint that no two identical bricks can be stacked on top of each other. This is a crude strategy, but an effective one, as it ensures the kind of interweaving seen in the corners of real world Lego models.

Structural integrity is also increased by having the solver optimize for the minimum number of pieces included in the final output. This ensures that each individual piece covers as much area as possible, hence increasing the stability of the final model.

For this step, our car model is not the best example, considering that the only volume group is the small piece at the end. So, instead consider the Twin Squares model shown in Figure 1. Here, we have two 6x2x6 volume groups of equal size, sitting next to each other, one at position (1,1,1) and the other at position (7,1,1).

From this, we also have that each has a 4x2x4 empty area in their centers, in this case starting at position (2,1,2) and (8,1,2) respectively. With this information in place, we are ready to solve for the set of pieces that fill in the areas.

Notice that, as shown in Figure 1, in our resulting model we have a central area with bricks of width 2. This comes from the fact that we meld together the volume areas, and as such, unite the two areas that are touching each other to form the whole presented. At the end of this step, we have our final model, and it suffices to render it out.

6 CONCLUSION

In this paper, we describe an ASP-based system for solving for Lego models, starting with a specification of the model and producing a sketch of the model, followed by the final model itself. In this way, we have provided the groundwork for future endeavors into generative 3D structures for domains that feature an existing set of pieces, prominently games with snap-together models or voxel systems.

REFERENCES

[1] Evgenia Antonova. 2015. *Applying Answer Set Programming in Game Level Design.* Master's thesis. Aalto University, Espoo, Finland.

[2] Armando de la Re, Francisco Abad, Emilio Camahort, and M Carmen Juan. 2009. Tools for Procedural Generation of Plants in Virtual Scenes. In *International Conference on Computational Science.* Springer, 801–810.

[3] Alexandre Devert, Nicolas Bredeche, and Marc Schoenauer. 2006. Blindbuilder: A New Encoding to Evolve Lego-like Structures. In *European Conference on Genetic Programming.* Springer, 61–72.

[4] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. 2009. Answer Set Programming: A Primer. In *Reasoning Web. Semantic Technologies for Information Systems.* Springer, 40–110.

[5] Pablo Funes and Jordan Pollack. 1999. Computer Evolution of Buildable Objects. In *Evolutionary Design by Computers*, Peter J. Bentley (Ed.). Morgan Kaufmann, Chapter 17, 387–403.

[6] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. 2014. *Clingo* = ASP + Control: Preliminary Report. *arXiv preprint arXiv:1405.3694* (2014).

[7] Mark Hendrikx, Sebastiaan Meijer, Joeri van der Velden, and Alexandru Iosup. 2013. Procedural Content Generation for Games: A Survey. *ACM Transactions on Multimedia Computing, Communications, and Applications* 9, 1 (February 2013), Article 1.

[8] Benjamin Mark, Tudor Berechet, Tobias Mahlmann, and Julian Togelius. 2015. Procedural Generation of 3D Caves for Games on the GPU.. In *FDG.*

[9] Benachir Medjdoub and Bernard Yannou. 2001. Dynamic Space Ordering at a Topological Level in Space Planning. *Artificial Intelligence in Engineering* 15, 1 (2001), 47–60.

[10] Paul Merrell. 2007. Example-based Model Synthesis. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games.* ACM, 105–112.

[11] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. 2006. Procedural Modeling of Buildings. In *Acm Transactions On Graphics (Tog)*, Vol. 25. ACM, 614–623.

[12] Yoav IH Parish and Pascal Müller. 2001. Procedural Modeling of Cities. In *Proceedings of SIGGRAPH 2001.* ACM, 301–308.

[13] Pavel Petrovic. 2001. Solving LEGO Brick Layout Problem Using Evolutionary Algorithms. In *Proceedings to Norwegian Conference on Computer Science.*

[14] Francisco Regateiro, João Bento, and Joaquim Dias. 2012. Floor Plan Design using Block Algebra and Constraint Satisfaction. *Advanced Engineering Informatics* 26, 2 (2012), 361–382.

[15] Adam M Smith and Michael Mateas. 2011. Answer Set Programming for Procedural Content Generation: A Design Space Approach. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (2011), 187–200.

[16] Romain Testuz, Yuliy Schwartzburg, and Mark Pauly. 2013. Automatic Generation of Constructable Brick Sculptures.. In *Eurographics (Short Papers).* 81–84.

[17] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. 2003. Instant Architecture. In *SIGGRAPH 2003.* 669–677.