Proceedings, The Thirteenth AAAI Conference on
Artificial Intelligence and Interactive Digital Entertainment (AIIDE-17)

Solving for Bespoke Game Assets: Applying Style to 3D Generative Artifacts

Jo Mazeika, Jim Whitehead
Computer Science Department
UC Santa Cruz
Santa Cruz, CA 95064 USA
{jmazeika, ejw}@soe.ucsc.edu

Abstract

In this paper, we present Solus Forge, a system for designing and generating 3D Lego models from a decomposition of the model into pieces and a series of spatial constraints over those pieces. We also include a style specification, which provides a series of transformations to perform on the initial model; adding, removing or modifying various pieces. To generate the models, we use a two-stage constraint solving process in which we first solve for the layout of the final model before then solving for the specific layout of the individual Lego pieces. In this way, we provide a framework for models that incorporates a specific set of criteria but also can be modified to fit a designer's needs.

Introduction

The pipeline for creating assets for 3D games is a time consuming and labor intensive process, primarily because most of these assets are handcrafted by experienced 3D modelers. While some of this has been offset by generative tools (Hendrikx et al. 2013), many hand-designed assets are still required. Because of these, there is a significant benefit in the ability to reuse and repurpose an asset multiple times.

However, this is not always possible; a rock designed for one region of a map may look completely out of place in a different region. There are some simple workarounds; reskinning and re-texturing the assets allows us to alter the surface-level appearance, and assets can be designed to be stretched and scaled in order to fit various spaces. However, any deeper modifications are often as difficult as simply rebuilding the model from scratch.

To this end, we desire a system capable of generating 3D artifacts from a basic specification that can be procedurally modified to produce stylized variations of that particular artifact. The stylized versions should feature specific, directed changes to the initial artifact such that a minimal set of changes happen to the initial artifact. However, it is not always trivial to know how a change will affect a given artifact, especially if some of the specification calls for generative components on its own.

In this paper, we present Solus Forge, a system that allows for the the generation of styled 3D artifacts within a simplified domain, namely the domain of Lego models. Here, we specify our models as hierarchical decompositions with spatial constraints, and then use a two-stage constraint solving approach to generate the output models. In this way, we manage to not only provide generation of models that can be modified by an arbitrary set of rules, but also generate in a way such that we allow for generative components in the specification. Figure 1 shows some sample models generated by the system; the two top models are the defaults, and the bottom two are the same models, but incorporating a "Pirate" style.

Figure 1: Solus Forge Example Outputs

This paper continues with an overview of related work, followed by an overview of object and style representations. We continue with an overview of the Solus Forge system itself, diving into the each of the two components of the system, and end with discussions and future work.

Related Work

The process of generating stylized artifacts is a well-studied problem, with approaches ranging from shape grammars for Frank Lloyd Wright inspired houses or Harley Davidson motorcycles (Koning and Eizenberg 1981; Pugliese and Cagan 2002) to the recent approach of using Convolutional Neural Networks to recreate an existing image in the style of another (Gatys, Ecker, and Bethge 2015; Pan et al. 2016;

Champandard 2016; Isola et al. 2016). However, the system that has come closest to the goals of this paper is Letter Spirit (Hofstadter and McGraw 1995), a system for generating grid fonts: fonts in which all of the characters are constructed from segments on a three-by-seven grid. Designed to simulate human problem solving, the system develops a style specification that is applied to all twenty-six lower-case letters such that all of the letters have a distinct and legible appearance. In order to ensure this is always possible, the Letter Spirit system allows for the relaxation of the style specification when it would make a letter unreadable or too similar to another letter. The limitations of this system, however, come into play with its evaluation metrics. The system requires a function for each letter which it uses to test if a newly generated glyph resembles the associated letter. This process works well for a system where the output is a fixed set of artifacts with easily defined properties, however, constructing these functions is outside of the scope for arbitrary generated 3D models.

We are able to confidently reduce to our generation down to the space of Lego Models thanks to several key affordances that Legos offer. First, since they are grid-aligned, the rules for how Lego bricks combine are simple as compared to the complexity of possible connections between arbitrary 3D models. Additionally, most bricks can only be attached together by snapping the bottom of a brick to the top of another, further reducing the possible ways they can combine. Due to their popularity, and their relative simplicity as compared to arbitrary 3D models, generation of arbitrary Lego structures has been well explored (Devert, Bredeche, and Schoenauer 2006; Petrovic 2001; Testuz, Schwartzburg, and Pauly 2013). Because of these known properties, it is straightforward to work around the space of physically non-viable models—i.e., models that with properties that would cause them to be unstable or that would violate laws of physics.

For our purposes, we leverage the power of Answer Set Programming (ASP), a declarative problem solving paradigm where programs represent problem specifications with solutions that the answer-set solver can locate (Eiter, Ianni, and Krennwallner 2009). ASP has strong generative uses, having formed the backbone of several procedural generation systems in recent years (Antonova 2015; Smith and Bryson 2014; Smith and Mateas 2011), including specifically in 3D spaces (Antonova 2015). ASP, of course, is by no means the only constraint solving paradigm used in generation, as evidenced by the vast body of literature focused on using constraint solving for space layout problems (Medjdoub and Yannou 2001; Regateiro, Bento, and Dias 2012).

Representation of Models

In Solus Forge, 3D objects are modeled as a decomposition hierarchy, where the children of a node are its proper parts (parts that are not equal to the whole). As such, any non-leaf node has two or more children. However, we then need to map the leaf nodes onto some set of Lego pieces in order to be able to generate the model itself.

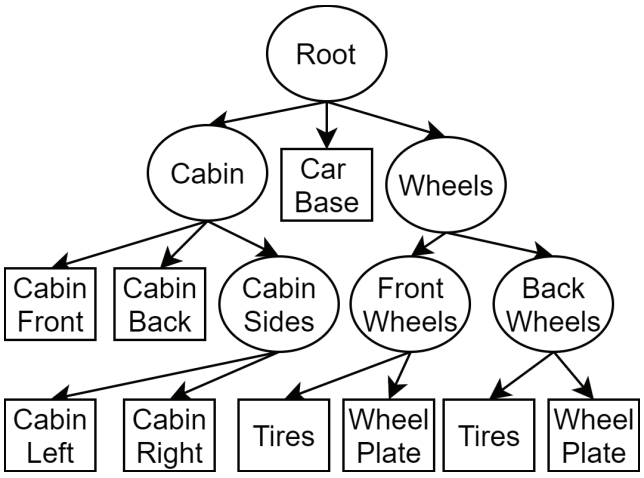

Figure 2: Semantic decomposition of the car model

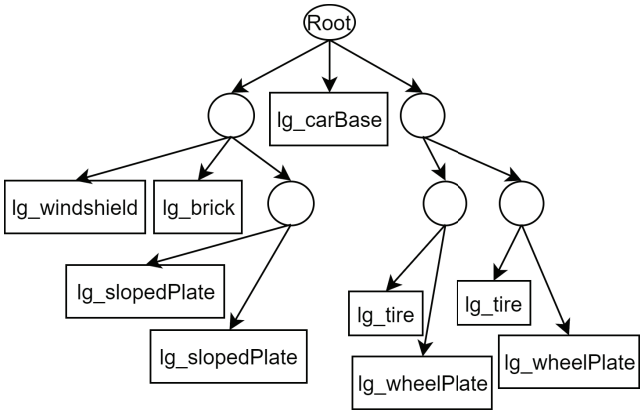

Figure 3: Decomposition of the car model with group labels

To refer to the Lego pieces, Solus Forge has a library of Lego groups, which are simply named collections of pieces. There are two major classes of Lego group: piece groups, which refer to a particular set of Lego pieces, arranged in a predefined way (such as a Windshield piece, or a Flame piece held by a plate) and volume groups, which give a specification of how to fill in a particular volume of space (ranging from filling in a volume with standard bricks to doing a complex spiral staircase made of particular curved pieces). In addition to their base specification, groups also feature some metadata labels (all of the groups that involve wheels are labeled as such) as a way of assisting in searching for groups with a particular property. And so, the hierarchy is an organization of these Lego groups into clusters based on a human designers intuition about a model.

Consider the diagram shown in Figure 2, which presents the hierarchical decomposition for the car shown in the upper left of Figure 1. Each node in the tree is labeled with its semantic meaning (Cabin, Wheel Plate, etc.), to show how this decomposition of the object looks. This structure also includes information about which Lego groups the different nodes of the tree represent, as shown in Figure 3. For in-

stance, in this model, we want the front of the cabin (”Cabin Front”) to map to a windshield piece and so we specify that that leaf represents ”lg_windshield” (the name of the Lego group that represents a windshield piece). The semantic labels have no significance within the system itself other than forming unique identifier for different parts of the object; all of the information needed for generation comes from the Lego groups themselves.

In this way, we have a structured representation of the model, where the Lego groups are organized into a hierarchy of parts and wholes. However, this part-whole information paints an incomplete picture of how to construct the object. Without positional information, relative or absolute, the system would have no way of knowing, for instance, that the wheels of the car described should be at the bottom of the model.

To solve this issue, the structure tree also incorporates a number of positional constraints between pairs of Lego groups, allowing the designer to specify the bounds on the spatial relations between pairs of groups. These constraints form a graph, showing how the different components of the model relate to each other. This graph is not necessarily dense, but must be fully connected over the set of groups.

These constraints allow us to define where a group is located relative to another. Typically, a group will be located on the surface of another (because of the connections defined by Legos, this is almost always that one group is immediately above or below the other), with constraints on where on that surface the group can be located. A common example of this is placing the wheels of a car not only underneath the main body of the car, but also specifically at the front and back of said car.

Additionally, if two groups are on the same surface, constraints can specify a range for how far apart the groups are allowed to be. Typically this is expressed as ways of keeping distance between two pieces (the couch must be three pegs back from the television), but these constraints also cover containment operations (the couch and television must both be inside the walls) and embedded pieces (a window embedded into a wall).

These positional constraints are not required between every pair of groups, and in some structures, the graph of these relations can be quite sparse. For example, in a given scenario it might be fine if the furniture of a living room is arranged any which way, but only as long as the couch faces the TV. In that case, a single constraint would specify the relative positions of the couch and the TV, with the positions of everything else left unconstrained and free.

In particular, in the ASP file that represents our model, we have the following:

- A list of the individual Lego groups that make up the model
- The tree structure that organizes the groups
- The spatial constraints that exist between the different groups

So, in our car example, we have the following set of positional constraints, as shown in Figure 4:

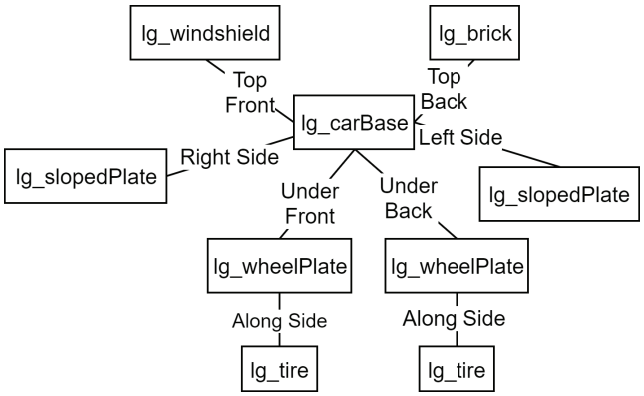

Figure 4: Car Part Adjacency Graph

1. The cabin front and back (represented as simple bricks in the final model) are placed at the top of the front and back of the car base, respectively.
2. The wheels and the plate they connect to are also placed on the bottom of the front and back of the car base, and the wheels are attached to the pegs on the side.
3. The sides of the car are placed on their respective sides of the model.

Representation of Style

Style is a property of design that is hard to identify or quantify. It can be defined as “a replication of patterning. . . that results from a series of choices made within some set of constraints” (Meyer 1979), which is a useful framework for us to work with in. Under this definition, in order to describe a style in an operationalized manner, we first need to describe the constraints that the style places upon the system. With those in place, we claim, the space of choices that the generator can make will necessarily feature the patterning that would make up a style.

While the qualities, choices and insights that lead to the creation of a particular style are an unknown quality, we choose to sidestep that by assuming that the user of Solus Forge has a particular set of constraints that expresses their style already in hand. In order to ensure that any particular style can be represented, we want a representation that errs on the side of being overly general.

To this end, the style is included as a series of ASP rules that are resolved during the first phase of generation. We incorporate the following classes of rules into our system:

- Adding additional Lego groups to the model, specifying at least one additional constraint to incorporate them into the model
- Removing Lego groups from the model, essentially nullifying any constraints that they are involved with.
- Replacing constraints or groups with another constraint or group.
- Modifying the color of a groups pieces.

Note that we make no claim about this being an all encompassing list; merely that this is the list of functionality that we have incorporated into our system thus far.

However, at this point, we run headlong into a problem: In order for our system to make directed style choices to a 3D model, we first need to be able to identify the different parts of that model—after all, the style for the wheels of a car is going to be vastly different that the style for the windows. However, in a generic 3D model, it is not clear where one part of a model ends and the next begins, nor what those parts should be.

To solve this, we leverage that the model is broken up into the well-specified Lego groups mentioned before. Each of these is labeled with a number of properties and the style rules can leverage these built-in labels to indicate whether or not a given rule should apply to a given group.

For the styles, we include the following affordances:

- Add in an additional group: The style can add additional Lego groups to the model. In order to do so, it must also include a relational constraint tying the group to another that already exists in the model, some picking a group either arbitrarily or by using the metadata embedded into the groups.
- Remove a group: The style can remove Lego groups from the model, identifying the groups to remove either by the name of the group or by a given metadata tag. This necessarily invalidates any constraint that the group is involved with.
- Replace a group: The style can, instead of removing a group, can replace it with a different group instead, allowing the constraints associated with the old group to carry over to the new group.

For this paper, we present the pirate style shown on the bottom two models of Figure 1. This style is made of the following rules:

- The inclusion of the treasure chest, the cannon, and the flag, the last of which is comprised of three separate pieces (the flag itself, the pole and the connector plate).
- The chest is placed either inside a volume group, or at the back of the model. The flag and the cannon are placed on top of some part of the model.
- Sloped-piece groups are removed (more as a point of demonstration than any innate pirateness)
- The color scheme is updated to reflect a more wood-and-stone look, focusing on the use of brown and grey.

As ASP rules, the rules for manipulating the pieces is as follows:

```
nodeRef(name("Cannon")).
pieceRef(size(8,6,2, "Cannon1"),
   parent("Cannon")).
focusGroup("Cannon","Cannon1").

nodeRef(name("Flag")).
pieceRef(size(1,10,1,"Flagpole"),
   parent("Flag")).
pieceRef(size(2,1,1,"Flagholder"),
   parent("Flag")).
pieceRef(size(1,5,3,"PirateFlag"),
   parent("Flag")).
attachRef("Flagholder",
   "Flagpole", (1,1,0)).
attachRef("Flagpole",
   "PirateFlag", (0,5,1)).
focusGroup("Flag", "Flagholder").

nodeRef(name("Chest")).
pieceRef(size(6,6,2,"TreasureChest"),
   parent("Chest")).
focusGroup("Chest", "TreasureChest").

styleAdd("Cannon", onTop,1).
styleAdd("Chest", inside,1).
styleAdd("Flag", onTop, 1).
styleAdd("Flag", onSide, 1).

removeGroups("slope").
```

With this style in place, we are ready to begin the process of generating our models.

System Overview

Using the 3D modeling approach described above, comprised of a structural decomposition combined with a positional graph, and the style specification, it is now possible to describe the Solus Forge system for applying a style to a 3D model and generating a realized artifact. Figure 5 shows the system diagram for this process.

To perform this process, we use a two-step ASP approach by which we first solve for the positions of each of the Lego groups in the model in the Sketching phase, and afterwards solve for the individual Lego pieces in the model in the Realization phase. By separating out the stages like this, we can gain a number of benefits. Firstly, we are able to swap out either part of the system with an alternate implementation without impacting the other. This allows not only for incremental progress (or bug fixing) on each part independently, but also leaves room to rebuild one of the parts in a completely separate paradigm without impacting the other. Secondly, it gives the affordance of being able to look at and analyze different sketches without realizing them, which allows us to better see how the different parts of the model are interacting with each other.

Sketching

First, we use the positional constraints between the groups, along with the size data encoded into the groups themselves, to build a sketch of the final output. In this sketch, none of the pieces that actually make up the final model are in place, but the sizes and positions of all of the Lego groups are finalized. Because our constraints can be somewhat loose, there are often a number of different sketches that can be produced from a single model, each with the property that they have a different arrangement of the Lego groups. These differences can be subtle, but they will always be noticeable to an observer.

For our pirate car, we first take our new graph, as shown in Figure 6, and set up our constraint problem to solve for

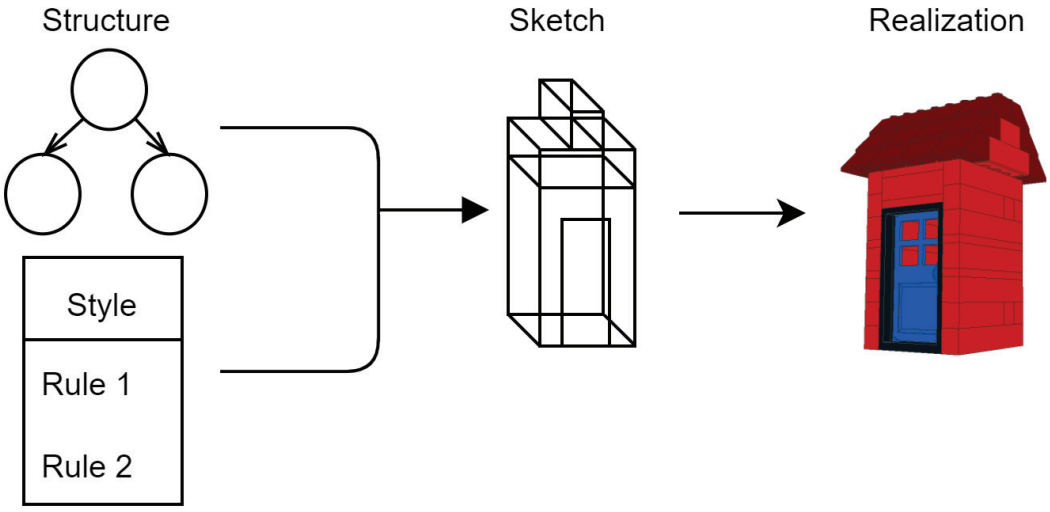

Figure 5: Solus Forge System Diagram

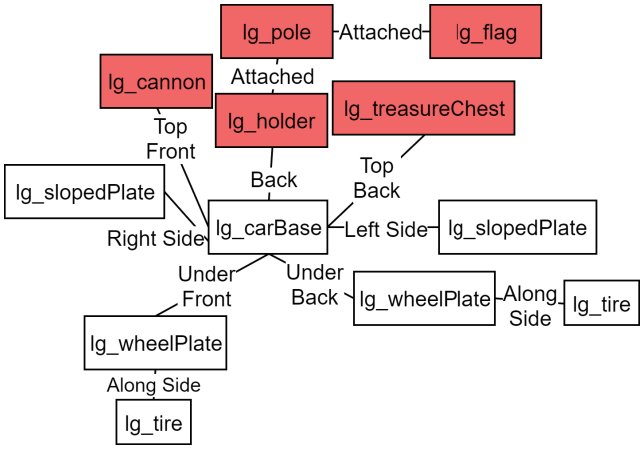

Figure 6: Final Car Graph after Pirate Style (Red are newly included groups)

the positions and sizes of each of the groups in space. Our coordinate system is based on the fact that, in general, Legos have a grid structure – one unit in the X or Z axes is equal to moving one peg, and one unit along the Y axis is equal to the height of a single plate (a brick is exactly three plates tall).

Each of the constraints specified has an underlying logical representation - for instance, saying that group A is "Under Front" relative to Group B means that group A is under group B (ie, group A's Y-position is such that it ends adjacent to the bottom of B) and that group A is towards the front of B (ie, group A's X position is as far forward as possible, while still being directly underneath B). From these, we able to create a 3D layout (the sketch) of all of the groups.

To that end, our underlying engine for the sketching step takes on a few parts. First, we have some definition references:

- Space definition - the space in which the pieces can be placed is constructed as a finite range in 3 dimensions
- Piece group definitions - Each piece group has a fair bit of associated data - each piece group needs to have its size specified, and all have metadata that is leveraged by the styles.
- Volume group functions - Since volume groups can be of any size, we include rules for choosing their size here. Some of these choices are straightforward (a rectangular wall can simply pick a size in each dimension), but others can be more complex, such as the sloped roof we see in the building model. For that, we specify the position, size and orientation of each of the rows of bricks based on the size and location of the footprint of the group overall. This turns one simple group into a large number of individual groups for the purposes of further generation.

We then have a set of rules for handling the constraints. Each of these is handled differently (as all of the constraints are distinct); however, positional constraints follow a common pattern in that they all limit the space of choices for a group to a range specified by another groups position and size. For example, assume that we have group A on top of group B. In this case, we fix group As vertical position to be exactly adjacent to the top of group B (requiring group Bs vertical position and height to be known, of course), and restrict group As position on the other two axes to intersect with group Bs. In the system, the rule for deriving positions in an "onTopOf" relations is represented as:

```
groupPos(y,Y+J, B) :-
   onTopOf(name1(B), name2(A)),
   groupSize(y,Y,A),
   groupPos(y,J,A).
{groupPos(D,X1..X1-1+I1-I,B)} = 1 :-
    groupPos(D,X1,A), groupSize(D,I1,A),
    groupSize(D,I,B), notY(D),
    onTopOf(name1(B), name2(A)),
    I <= I1.
```

```
{groupPos(D,X1-1+I1-I..X1,B)} = 1 :-
   groupPos(D,X1,A), groupSize(D,I1,A),
   groupSize(D,I,B), notY(D),
   onTopOf(name1(B), name2(A)),
   I >= I1.
```

where "notY(D)" matches with any dimension other than the y-axis.

Finally, for each group, we pick its location in space. Because of the restrictions given to us by the constraints, we know that all of these positions are going to be valid and that our model fit all of our specifications, but, for any underconstrained parts of the model, we allow completely free choice. It is here where our possibility space of models is created.

These rules[1], when combined with a model and style specification, produce a sketch, which is captured both in a human readable JSON file, as well as a long list of facts that is handed off to the the Realization step.

Realization

Once all of the positional information is in place, the system then needs to realize the sketch. For our Pirate car, this step is trivial, since all of the Lego groups included are piece groups. As such, the only step that this step takes is to specify the particular pieces that make up the different groups in the locations determined by the previous step. However, for the house model, we need to a fair bit of solving to construct the walls of the model, and the roof of the base model.

For this, we specify a list of possible pieces—for the walls, this is a set of basic Lego bricks and plates, and we also include a set of sloped pieces for the roof—as a list of `legoCollection/3` facts, specifying the size of each piece. We also include our information from the sketch, transforming into a set of voxel coordinates that need to be filled as a list of `target/3` facts—including the space of each volume group, then subtracting out the space encapsulated by the piece groups.

We then solve for a collection of pieces (and their locations) such that the following criteria are met:

- The target volume, and only the target volume, is covered with pieces.

```
pos((X,Y,Z),X,Y,Z) :- target(X,Y,Z).
{ blank(N);
  legoPiece(N, loc(I,J,K), size(X,Y,Z)) :
  legoCollection(X,Y,Z),
  target(X+I-1,Y+J-1,Z+K-1)} = 1
  :- pos(N,I,J,K).
coveredBy(N,I1,J1,K1) :-
  legoPiece(N, loc(I,J,K), size(X,Y,Z)),
  I1 = I..I+X-1,
  J1 = J..J+Y-1,
  K1 = K..K+Z-1.
covered(X,Y,Z) :- coveredBy(N,X,Y,Z).
:- target(X,Y,Z), not covered(X,Y,Z).
:- covered(X,Y,Z), not target(X,Y,Z).
```

- No two pieces used intersect with each other

```
:- pos(N,X,Y,Z),
  2 { coveredBy(Ni,X,Y,Z) }.
```

- No two identical pieces are directly on top of each other (decreasing the potential for free-standing sections of the model)

```
:- legoPiece(N1, loc(I,J,K), size(X,Y,Z)),
 legoPiece(N2, loc(I,J+Y,K), size(X,Y,Z)).
```

While a large number of solutions exist that meet the above criteria, we (from previous Lego literature) know that using the fewest number of Lego pieces in our final model will help increase the overall stability. So, we include a statement to optimize our model to use the fewest pieces possible:

```
#minimize {1@1,N,L,S: legoPiece(N,L,S)}.
```

Our final model is pictured in Figure 1.

Discussions and Future Work

While SOLUSForge produces good results thus far, it is not without its limitations. First, the style descriptions are very formalist, focused solely on describing the changes in terms of physical components of the Lego model. While deep changes are certainly possible, the modifications presented here can be described as more of aesthetic and construction choices. To break away from this formalist approach, a much richer representation of both the structures and the styles would be required. However, this work has yet to even fully explore the depths of the styles as proposed here.

For instance, one obvious extension of the work here would be to blend and merge different styles together to form new styles that contain the properties of both. The simplest approach would be to take subsets of the rules from each initial style, but other, more sophisticated approaches could lead to more interesting and expression results. One technique that would likely be promising is conceptual blending, since the spaces around the style rules is easy to manage.

Secondly, more work on the structure models would allow us to greatly increase the expressive range of the models innately, which would in turn allow us more expressiveness in the styles we can represent. The next immediate improvement would be the inclusion of non-deterministic nodes into the models, which would allow for an explosion of expressivity in the Sketching step.

Conclusions

In this paper we prevent a pipeline for using constraint solving to generate 3D Lego models, in a way that allows for the style of the models to be modified to fit a designer's needs. In this way, we have provided one method for generating 3D objects that meet a particular design criteria while keeping their primary attributes consistent.

References

Antonova, E. 2015. Applying Answer Set Programming in game level design. Master's thesis, Aalto University, Espoo, Finland.

[1]For a full description of the rules, please view the source at https://github.com/jomazeika/SolusForge

Champandard, A. J. 2016. Semantic style transfer and turning two-bit doodles into fine artworks. *arXiv preprint arXiv:1603.01768*.

Devert, A.; Bredeche, N.; and Schoenauer, M. 2006. Blindbuilder: A New Encoding to Evolve Lego-like Structures. In *European Conference on Genetic Programming*, 61–72. Springer.

Eiter, T.; Ianni, G.; and Krennwallner, T. 2009. Answer Set Programming: A Primer. In *Reasoning Web. Semantic Technologies for Information Systems*. Springer. 40–110.

Gatys, L. A.; Ecker, A. S.; and Bethge, M. 2015. A neural algorithm of artistic style. *arXiv preprint arXiv:1508.06576*.

Hendrikx, M.; Meijer, S.; van der Velden, J.; and Iosup, A. 2013. Procedural Content Generation for Games: A Survey. *ACM Transactions on Multimedia Computing, Communications, and Applications* 9(1):Article 1.

Hofstadter, D., and McGraw, G. 1995. Letter spirit: Esthetic perception and creative play in the rich microcosm of the roman alphabet. In *Fluid concepts and creative analogies*, 407–466. Basic Books, Inc.

Isola, P.; Zhu, J.-Y.; Zhou, T.; and Efros, A. A. 2016. Image-to-image translation with conditional adversarial networks. *arXiv preprint arXiv:1611.07004*.

Koning, H., and Eizenberg, J. 1981. The language of the prairie: Frank lloyd wright's prairie houses. *Environment and Planning B: Planning and Design* 8(3):295–323.

Medjdoub, B., and Yannou, B. 2001. Dynamic Space Ordering at a Topological Level in Space Planning. *Artificial Intelligence in Engineering* 15(1):47–60.

Meyer, L. B. 1979. Toward a theory of style. In Meyer, L. B., and Lang, B., eds., *The Concept of Style*. University of Pennsylvania Press. 3–44.

Pan, Y.; Burnap, A.; Liu, Y.; Lee, H.; Gonzalez, R.; and Papalambros, P. 2016. A quantitative model for identifying regions of design visual attraction and application to automobile styling. In *Proceedings of the 2016 Internation Design Conference*.

Petrovic, P. 2001. Solving LEGO Brick Layout Problem Using Evolutionary Algorithms. In *Proceedings to Norwegian Conference on Computer Science*.

Pugliese, M. J., and Cagan, J. 2002. Capturing a rebel: modeling the harley-davidson brand through a motorcycle shape grammar. *Research in Engineering Design* 13(3):139–156.

Regateiro, F.; Bento, J.; and Dias, J. 2012. Floor Plan Design using Block Algebra and Constraint Satisfaction. *Advanced Engineering Informatics* 26(2):361–382.

Smith, A. J., and Bryson, J. J. 2014. A Logical Approach to Building Dungeons: Answer Set Programming for Hierarchical Procedural Content Generation in Roguelike Games. In *Proceedings of the 50th Anniversary Convention of the AISB*.

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

Testuz, R.; Schwartzburg, Y.; and Pauly, M. 2013. Automatic Generation of Constructable Brick Sculptures. In *Eurographics (Short Papers)*, 81–84.