Computer-Aided Design 43 (2011) 1391–1401

Contents lists available at SciVerse ScienceDirect

Computer-Aided Design

journal homepage: www.elsevier.com/locate/cad

Inference-based procedural modeling of solids

Keith Biggers [*], John Keyser

Department of Computer Science and Engineering, Texas A&M University, TAMU 3112, College Station, TX 77843-3112, United States

ARTICLE INFO

Keywords:
Procedural modeling
Solid modeling
Virtual environment
Petri net

ABSTRACT

As virtual environments become larger and more complex, there is an increasing need for more automated construction algorithms to support the development process. We present an approach for modeling solids by combining prior examples with a simple sketch. Our algorithm uses an inference-based approach to incrementally fit patches together in a consistent fashion to define the boundary of an object. This algorithm samples and extracts surface patches from input models, and develops a Petri net structure that describes the relationship between patches along an imposed parameterization. Then, given a new parameterized line or curve, we use the Petri net to logically fit patches together in a manner consistent with the input model. This allows us to easily construct objects of varying sizes and configurations using arbitrary articulation, repetition, and interchanging of parts. The result of our process is a solid model representation of the constructed object that can be integrated into a simulation-based environment.

1. Introduction

As the use of simulation increases across many different domains, the need for high-fidelity three-dimensional virtual representations of complex environments has never been greater. As these environments become larger and more complex, there is an increasing need for more automated construction algorithms to support the development process. These constructed environments can then be used within a wide variety of different application areas.

Fig. 1 shows a very complex real-world environment (i.e., a rubble pile) that serves as an inspiration for our work. This environment contains *clutter*, a large collection of objects residing in a small amount of space. Accurately modeling by hand an environment such as that shown would be an extremely difficult and time consuming process. Capturing the full detail of each individual object in a faithful fashion would involve great attention to detail, and an intricate modeling process.

In most cases manually modeling such an environment is not a feasible option due to the underlying cost and complexity required. Thus, more automated methods capable of automatically re-creating an environment to an appropriate level of detail are necessary. The example shown in Fig. 1 has many similar but different objects, and as illustrated, the procedural approach we propose can be used to easily construct a representative environment.

In this paper, we present a novel procedural modeling technique centered on an inference-based construction algorithm for developing diverse models from a set of object templates. Our approach extracts surface patches from a template model, and then fits these patches together in a consistent fashion to fully define the boundary of an object. We use a Petri net to represent the structure of the model's surface, and then use it to guide the model generation process. A characterizing parameterization serves as a "road map" for object construction, and the surface patches extracted from the template are incrementally fit around it to define a new object. Different behaviors can be dynamically incorporated into the construction process, which allows a wider variety of object configurations to be developed. As a result, this approach is capable of generating a rich collection of diverse solid model representations.

Our work provides three main contributions:

- A new algorithm for generating solid models that locally fits patches around a defined parameterization in a globally consistent fashion. This algorithm demonstrates the usefulness of Petri nets as part of such a modeling process.
- Both automated and semi-automated techniques for defining the underlying parameterization used, which allows for easily characterizing many different objects.
- Several extensions of our basic algorithm that allow for more complex object definitions through the use of articulation, repetition of parts, and interchangeable parts.

2. Related work

Our work draws on ideas related to the areas of procedural modeling, parts and example-based modeling, surface reconstruction, and CAD/CAM feature-based modeling.

[*] Corresponding author.
E-mail address: biggers@tamu.edu (K. Biggers).

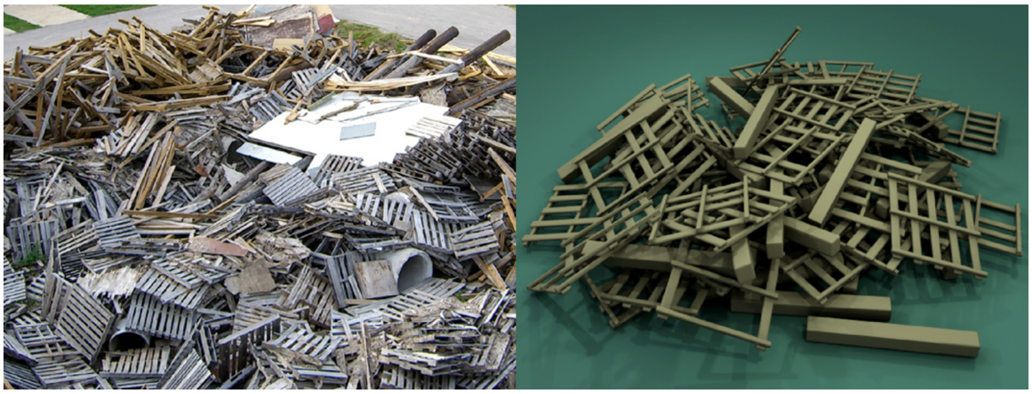

Fig. 1. A real-world environment that serves as an inspiration for our work and a representative environment created using our approach.

2.1. Procedural modeling

Procedural modeling methods have been used in a wide range of applications. Many different approaches have been proposed for automatically generating textures and environmental effects, and for modeling [1]. Modeling methods have been developed for automatically generating buildings and roads [2–6], trees and plants [7–11], and terrain [1,12–14]. The output from these different methods can be used in real-time games and simulations, and incorporated into animation and movies. These techniques are being used to develop entire mathematically based worlds [15].

Procedural approaches leverage techniques such as fractals [16], Perlin noise [17], L-Systems [18], tiling [19], and shape grammars [20] to drive the underlying generation process. These methods can work with provided examples to help define the construction process (e.g., [4]), work with no provided information at all (e.g., [12]), or develop this knowledge on the fly through machine learning or other strategies (e.g., [5]). Structural symmetry has also been used for procedurally constructing models using shape operations on parts extracted from provided examples (e.g., [21]).

Our approach is, to our knowledge, the first to leverage Petri nets for procedural model generation.

2.2. Parts- and example-based modeling

Parts-based and example-based modeling methods commonly use information from known/high quality models for a variety of operations with other unknown/lower quality models. These approaches can work with both models as well as point cloud data, and they can be broken down into several categories.

The first category fits simple primitives (i.e., planes, cylinders, spheres, etc.) to point cloud data [22,23]. An extension of this idea uses constrained graphs of primitive shape configurations to describe features, and works well with point clouds from architectural domains [24].

A second category goes beyond simple primitives and relies on a database of object models for extracting the fitting elements. These approaches usually focus on fitting more functional items (i.e., an arm, leg, handle, wheel, etc.). They have made use of ideas such as interchangeable parts [25,26], salient parts [27], and hierarchical analogies between parts [28]. They typically deal with meshed models as opposed to point clouds. Gal et al. propose using a database of local shape priors to augment point clouds during reconstruction [29]. Their approach matches and fits shape priors extracted from provided high quality models to similar regions in a point cloud. These fitted patches are then integrated during the reconstruction process. Their approach produces higher quality results by using the fitted priors to smooth out noisy data and fill small gaps.

A third category assumes no prior knowledge and simply uses data available from the model itself. These approaches focus on the identification of symmetry and regular geometry within models (i.e., reoccurring parts/features) [30]. Extensions of this work use graphs of salient features [31] and feature lines [32] to help with the process. In general, these approaches are feature oriented and deal with structural regularity across a dataset (e.g., finding regular patterns of window facades across the surface of a building). These features can then be iteratively transformed across the dataset to define a regular pattern. These approaches have been shown to work with both point cloud data and meshed models, and can be used for model repair, compression, and geometry synthesis [30].

Our approach uses a parts-based methodology to select and fit together model segments to fully define the boundary of a newly defined object.

2.3. Surface reconstruction

We build on ideas from the area of surface reconstruction. Surface reconstruction techniques commonly take a point cloud description of an object or environment as input, and attempt to construct a continuous surface representation from this discrete sampling. These techniques are categorized based on those that generate meshed (e.g., [33,34]) and implicit (e.g., [35–38]) surface representations.

Our approach uses a reconstruction-based approach to integrate fitted point sampled surface patches together to construct a solid model representation.

2.4. CAD/CAM feature-based modeling

Finally, our work is partially inspired by the large area of feature-based modeling from the CAD/CAM communities. These approaches commonly use a set of defined primitives (i.e., the data) along with a set of rules for primitive interaction (i.e., the relationships between data) to construct solid model representations. There are many survey papers that provide overviews of the different feature-based techniques that have been developed (e.g., [39,40]).

Petri nets have been used previously for geometric modeling [41], but in a very different fashion. Their approach uses a Petri net for modeling constrained geometric objects using a message propagation method. Our approach uses a Petri net to define and guide the process of fitting surface patches together to form a complete object.

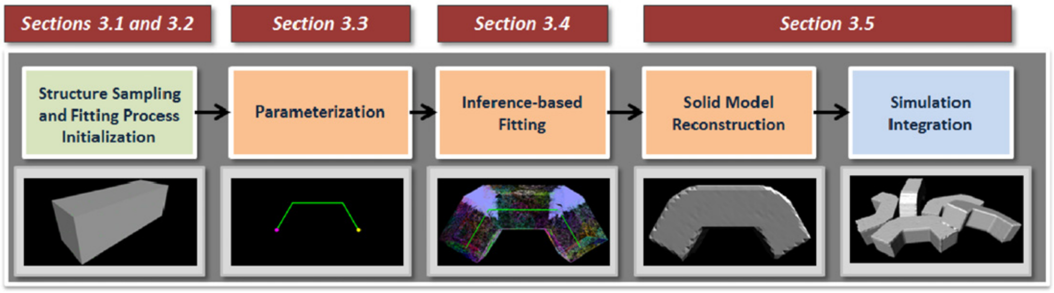

Fig. 2. Overview of our inference-based procedural modeling process.

3. Inference-based modeling details

Our process for constructing objects and incorporating them into a virtual environment is decomposed into five stages (illustrated in Fig. 2). Our approach loosely builds on the augmentation idea of Gal et al. [29].

The steps of our algorithm are as follows, and will be described in detail within each of the defined sections:

– Take as input a set of object templates (i.e., polygonal meshed models), sample and extract a set of sub-sampled surface patches along with their relationships across the structure of the object, and generate and initialize the data structures used by the patch fitting process (Sections 3.1 and 3.2).
– Obtain a defined parameterization of the object to be constructed using one of several different automated/semi-automated methods (Section 3.3).
– Take the sub-sampled patches, the developed data structures, and the characterizing parameterization, and use an inference-based fitting process to construct an object based on these defining elements (Section 3.4).
– Combine the fitted patches and reconstruct a solid model representation of the constructed object (Section 3.5).
– Integrate the constructed object into the overall virtual environment being developed (Section 3.5).

3.1. Structure sampling and annotation

Our approach takes as input one or more annotated polygonal meshed models that represent templates for the objects to be constructed. The first stage in our process takes these input models and samples them, constructing *surface patches* that capture the local surface properties of each object. These patches are then sub-sampled with a set of points, and these points will serve as the underlying fitting elements used during the later construction process. A neighbor graph that defines the interconnectivity of these patches, thereby capturing the underlying structure of the object, is then generated. Finally, the data structures used during the fitting process are constructed.

Each model is first uniformly sampled using a random area weighted scheme, followed by a relaxation procedure that ensures an even distribution of point samples across the model's surface [42]. For each sample, the faces and vertices that intersect the volume of a support sphere (i.e., of a fixed defined size, centered at that sample, and containing a normal that resides inside a defined angle around the sample's normal) are captured and stored as the defining elements for the patch. An example sampling of a model is shown in Fig. 3 where the samples are shown in green.

Each identified surface patch is then independently sub-sampled with a collection of points (again using the process as defined by Turk [42]) with the additional constraint that these sub-samples must reside inside the support sphere. Fig. 3 shows an example sub-sampling of a patch where the sub-samples are

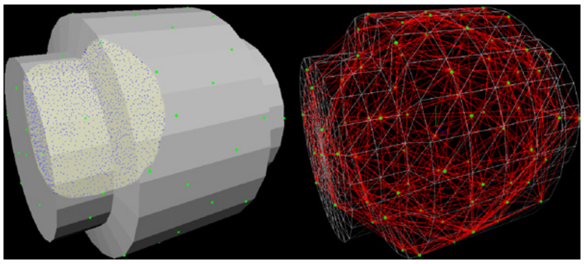

Fig. 3. Example model sampling (green points), surface patch (yellow region) and patch sub-sampling (blue points), and neighbor graph (nodes in green and edges in red). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

shown in blue. The sub-sampled patches are then normalized (using the process defined by Elad et al. [43] which allows for easier transformation) and stored. These sub-samples provide a characterization of the local behavior of an object's surface and will serve as the basis for the later construction process.

A *neighbor graph* is then constructed from the samples/patches of the model. Nodes in this graph correspond to the patch samples, and two nodes are connected with an edge in the graph when their respective support spheres overlap in space. An example of such a graph is shown in Fig. 3. This graph serves as a definition of relationships between different patches, and will be utilized during the later inference-based fitting process to guide the creation of the underlying data structures used during fitting.

Our algorithm makes the assumption that each template model is provided in an axis-aligned pose and that its primary orientation is defined along a specific fixed axis. A one-dimensional *parameterization* is then inherently defined along each axis line. A parameterization is simply defined as the specification of a curve that maps the structure of an object from 0.0 to 1.0 along that particular axis line. This parameterization provides a simple topological skeleton of the object along that axis direction, but does not incorporate branching as with a medial axis. This approach works well for objects that are very regularly defined along an axis (e.g., a block, pipe, etc.) and may not work well for other objects that are more irregular/organically defined (e.g, a rock, chair, etc.).

For each model, a user may manually annotate special regions used during the fitting process. Fig. 4 shows an example of an annotated model. The region shown in yellow corresponds to a part that can be repeated iteratively along a parameterization, and regions in blue can be interchanged with each other. Any *annotated sample* that resides inside one of these regions, and any graph edges that cross its boundary, are identified and marked for later use.

Any sample not residing inside one of these regions is referred to as an *anchor sample*. Anchor samples serve as base elements to begin the construction process with, and the remaining *annotated samples* are then used to dynamically build the object around an arbitrarily defined parameterization. The details behind how these items are used within the construction process will be described in the next few sections.

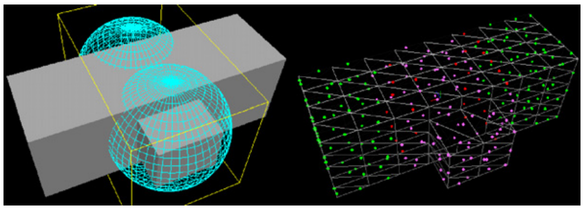

Fig. 4. Example manually annotated model (left) and corresponding annotated/anchor samples (right).

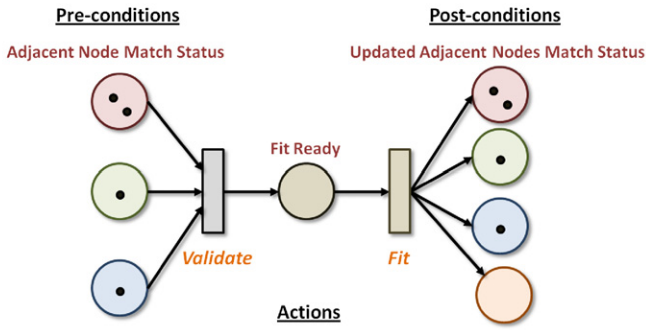
Pre-conditions
Adjacent Node Match Status
Post-conditions
Updated Adjacent Nodes Match Status
Fit Ready
Validate
Fit
Actions

Fig. 5. A colored Petri net structure defining the process for fitting a given node in the neighbor graph. Places are illustrated as circles and transitions as rectangles.

3.2. Petri net initialization

The goal of our approach is to define the boundary of a new object by consistently fitting together a set of extracted sub-sampled patches from the template model in an alternative configuration. This patch fitting process is driven by our inference-based algorithm which leverages a *colored Petri net* data structure. This process allows patches to be locally fit in a stepwise fashion while ensuring consistency between adjacent items. It also allows the incorporation of different extensions for the generation of a wider variety of different constructed objects. The next step in our process generates and initializes the Petri net data structure used during the fitting process.

Petri nets are powerful data structures because they provide a natural methodology for modeling stepwise processes that include action, choice, iteration, parallelism, synchronization, and dependency [44]. Each of these properties is very applicable to our construction process because our algorithm attempts to logically fit together pieces of an object in a stepwise, yet consistent fashion. A Petri net is a directed graph containing two types of nodes, places and transitions (an example Petri net is shown in Fig. 5). A colored Petri net allows the storage of values in the *tokens* that are passed through the network, which can be very beneficial for tracking statuses as shown later.

When modeling a system/process with a Petri net, *places* (shown as circles) correspond to the states of the system and *transitions* (shown as rectangles) correspond to actions that the system can perform. In order for the Petri net to define the construction process for a given template model, a set of states and transitions must be defined and joined accordingly to describe the steps necessary for constructing the template object using the generated set of patches.

We begin by taking the samples, patches, and neighbor graph developed during structure sampling, and use these items to construct a Petri net structure. The objective of this structure is to help define, guide, and track the distributed fitting process. A locally defined structure is defined for each node in the neighbor graph, representing the steps necessary to correctly fit a patch. When combined, these structures form a global network that fully defines the process for constructing the template object.

Fig. 5 shows the local structure for a given node in the neighbor graph. For each node a *matching status place*, a *fit ready place*, a *validation transition*, and a *fitting transition* are constructed. The pre-conditions necessary for, and the post-conditions that result from, fitting a patch are modeled as places (i.e., using the matching status places constructed for each node). The first of the two transitions, the validation transition, determines when a patch is capable of being fit (i.e., if all of the adjacent nodes have been marked as possible fits) and after firing moves the item into a fit ready state. The second, the fitting transition, prompts the fitting process to start and updates all adjacent nodes after the actual fitting has been performed. As a node is fit, its adjacent nodes are alerted to this fact. As enough information about the neighbors of an adjacent node becomes available, it is in turn fit.

In this way, a Petri net is automatically constructed that describes the process by which all of the sample nodes can be fit together to form the template model. This approach avoids the requirement of explicitly defining a set of ad hoc rules for generating similar behaviors.

3.3. Parameterization

The construction process begins by obtaining a *parameterization* that defines the basic configuration of the object to be constructed. This parameterization is defined with a piecewise smooth curve, and serves as a "road map" for the patch fitting process. It is somewhat similar to an object's skeleton, but it is a piecewise linear curve only and based on its definition, different behaviors extracted from the template model can be dynamically incorporated into the constructed object. Patches are fit around this curve, but fit relative to each other in a consistent fashion based on their original definition in the template model. This allows a reconfigurable definition of an object that still fully defines a complete boundary.

We have experimented with four different methods for obtaining this parameterization, including methods that function in both a fully automated and semi-automated fashion. These include:

- A sketch-based interface that allows the user to define a configuration using a simple 2D drawing interface.
- A random walk that randomly generates an ordered set of vertices for defining a completely random curve.
- A turtle graphic [45] that samples the defined space using a uniform grid, and a set of simple movement rules are then used for generating a non-intersecting curve.
- A Hilbert space-filling curve [46] that subdivides the space into a set of cells, and then recursively constructs a curve using a defined mathematical function.

All four methods provide an easy means for quickly defining the general configuration of the object to be constructed, and this set of methods can be used to generate a diverse collection of results. The defined parameterization will be used to guide the construction process to follow.

3.4. Inference-based fitting

Given the Petri net structure and a new parameterization, our inference-based construction algorithm incrementally fits surface patches around the defined parameterization. This fitting must be done in a consistent fashion to ensure that an accurate and complete object definition is generated (i.e., adjacent patches must have consistent overlapping areas to define a smooth and complete surface when joined; dynamic behaviors such as repetition and interchanging of parts makes this fitting process more complex). The general flow of the fitting process is described in Algorithm 1, and the details behind each step of the process are explained in the remainder of this section.

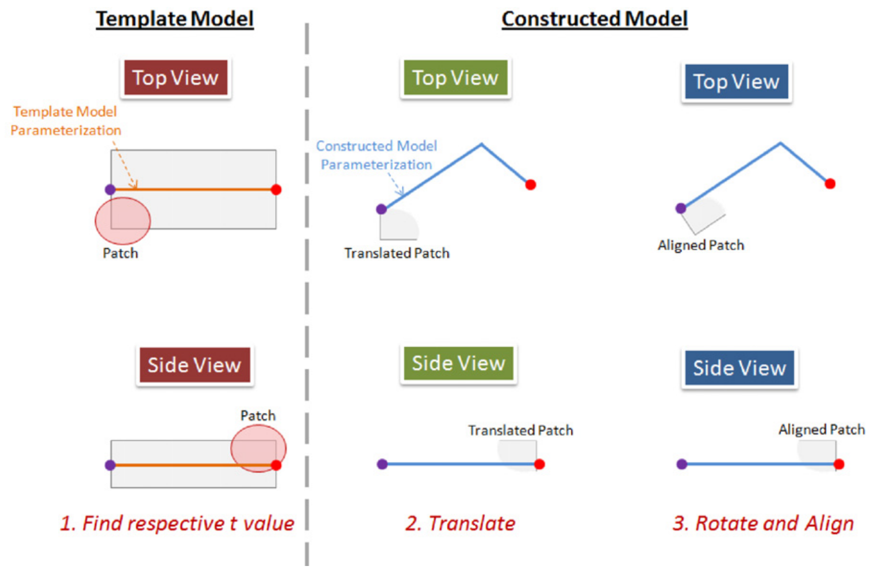

Fig. 6. An illustration of the patch mapping process used to fit a patch from the template model to its correct position and orientation along the defined parameterization. The extracted patch is translated along the parameterization to the determined t value, and then rotated such that its principal axes are aligned with the curve.

Algorithm 1

```
1.  $node = FindStartingNode();$
2.  $AddStartingTokenToNode(node);$
3.
4.  repeat
5.      $FirePetriNet();$
6.      $PerformPostFireCleanup();$
7.      $anyFittingsReady = CheckPetriForFittings();$
8.
9.      if $anyFittingsReady$
10.         then $HandlePetriFittings();$
11.         else $SearchNewPetriFitting();$
12. until $CheckIfFittingProcessFinished() == $ true;
```

3.4.1. *Initialization and stepwise fitting*

We first determine the starting sample/patch with which to begin the fitting. This element is found by analyzing the samples with respect to the parameterizations defined in the original model space (i.e., referred to as *template space*), and identifying a sample with a minimum value along the parameterization of the longest primary axis. The selected sample must also be an anchor sample to ensure a correct start to the overall process.

We then make use of the Petri net to iteratively fit adjacent patches. An initial matching token is placed into the *fit ready place* for the initial patch, triggering the fitting process on the next firing of the Petri net. A colored Petri net (which we use) allows each token to store values. We store the current state (i.e., "match", "trial", or "invalid"), the location of the node the token is set for, the current repetition iteration and/or interchangeable part index, and the current offset (i.e., as a result of repetition of regions in the template). Each of these values will be used during later steps of the fitting procedure and will be explained later.

Firing the Petri net involves firing transitions, in our case either a validation or fitting transition. For either transition to fire, all the incoming places must contain a "match" or "trial" token. In addition, a check is performed to ensure that the patch has not already been fit to that particular location in construction space. If these conditions hold and a validation transition fires, then the tokens are removed and a token is placed into the item's fitting place. This triggers the start of the fitting process for the patch. Upon the next fire of the network, the fitting transition will fire, during which the patch is fit and the post-conditions are handled. As a result, all adjacent elements (including the item being fit itself) are given a new "match" token. These new tokens imply that the adjacent patches may be ready for fitting based on the information obtained from previously handled items.

If any iteration results in no new patches to fit, we must select a "best" node to fit. To do this, we iterate over all of the validation transitions to identify the transition with the highest number of valid incoming places (i.e., with the most supporting information). In this case, a "trial" token is added to each element's matching status place that does not currently have a valid "match" token. This allows a test fitting to be performed upon the next fire of the Petri net.

The overall fitting process continues incrementally, fitting patches across the surface of the object based on the progressive fitting of neighboring patches. This continues until all items in the original template object have been used, and there are no longer any items left to be handled. The propagation across the parameterization, fitting patches along the way, ensures that all elements of the constructed object are handled and a complete object is defined around the parameterization.

3.4.2. *Patch mapping*

Once a patch is selected for fitting, it must be mapped from the original template space to the space in which the object is being constructed (referred to as *construction space*). For the initial patch being fit, we simply use the $t = 0$ position of the defined parameterization; further patches are then fit relative to previous ones. To fit the patch in construction space, a transformation must be applied to all of its elements (i.e., its sample and sub-samples). Algorithm 2 provides an overview of this process.

Algorithm 2

```
1. $param = GetParameterization();$
2. for $x \in samples \bigcup subsamples$
3.     $t = FindRelativeTValue(x, param);$
4.     $TransformPositionAlongParameterization(x, t, param);$
```

For each element in the patch, a parameter value t is found within model space and used to determine the correct position along the parameterization in construction space. Each element is then translated to the position such that it maintains the features of the item in template space, but is fit relative to the defined parameterization in construction space. Finally, the element is

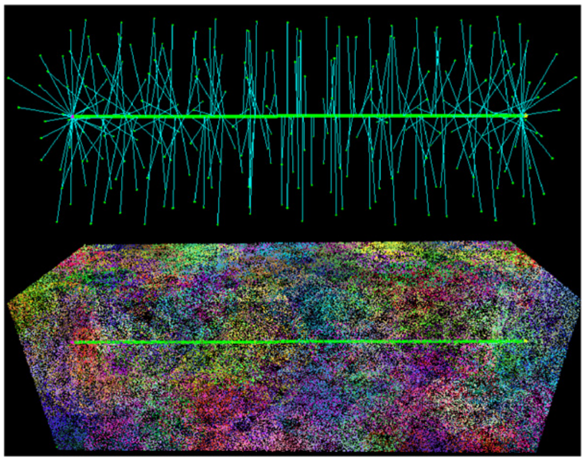

Fig. 7. Mapping of samples and sub-samples along a defined parameterization. Each fitted patch is assigned a unique color.

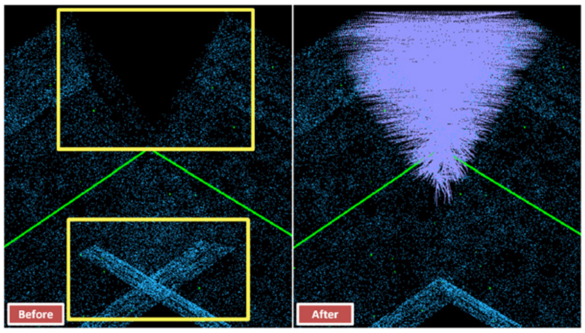

Fig. 8. The before/after states for cleanly mapping patches across articulated joints.

rotated to align the three principal axes from the model to the defined curve. The result of this fitting process is a sample and set of sub-samples that encompass the features of the template object, but are fit around the parameterization. Fig. 6 provides a two-dimensional illustration of this mapping process.

Additional patches must be fit relative to the initial item to ensure a consistent surface definition. For each, we again find a t value for the item taken from template space, but this value must be relative to the initial fitted patch in construction space. Thus, this process fits the patches exactly as they exist in template space, but simply transformed around the new parameterization. Fig. 7 shows an example of the results from this mapping process. The blue lines illustrate the t mapping for each sample to the underlying parameterization, and each fitted patch is illustrated with a unique color.

3.4.3. *Handling articulation*

While the template model is parameterized along a straight axis, the given parameterization can include bends (similar to an articulated joint). This can result in gaps and excess points in the regions around a bend in the curve. The left side of Fig. 8 illustrates an example of such a case. In order to ensure a well-defined solid model is constructed during the later stages, these gaps must be adequately filled with samples and the excess points inside the surface removed. Algorithm 3 provides an overview of this process.

Algorithm 3
```
1.  for $x \in patchesAffectedByArticulation$
2.      $crease = FindCreaseDetails(x)$;
3.      $FindRegionsAlongCrease(x, crease, leftReg, rightReg)$;
4.      $leftSamples = FindSamplesInsideRegion(leftReg)$;
5.      $rightSamples = FindSamplesInsideRegion(rightReg)$;
6.      for $l \in leftSamples$
7.          $adjNeigh = FindAdjNeighbor(l, rightSamples)$;
8.          $lineSamples = ConnectAndSubsample(l, adjNeigh)$;
9.          $TransformAndStoreSubsamples(lineSamples)$;
10.
11.     for $r \in rightSamples$
12.         $adjNeigh = FindAdjNeighbor(r, leftSamples)$;
13.         $lineSamples = ConnectAndSubsample(r, adjNeigh)$;
14.         $TransformAndStoreSubsamples(lineSamples)$;
15.
16.     $plane = FindBisectionPlane(crease)$;
17.     for $s \in samples \bigcup subsamples$
18.         if $ResidesOnOppositeSideOfBisectionPlane(s, plane)$
19.           then $RemovePoint(s)$;
20.           else  $KeepPoint(s)$;
```

To fill a gap, we begin with the original patch as defined in template space, and identify the crease in which the patch is being broken. A region of a fixed width on either side, and along the full length of this crease, is found. The samples residing within the region on either side are then identified. The nearest neighbor for each sample residing in the opposite region is found, and a line between the two samples is created, sub-sampled, and transformed into construction space. The collection of these sub-sampled lines "stretches" the patch across the gap and provides the necessary filler to ensure proper reconstruction. Fig. 9 illustrates this process and Fig. 8 shows an example of its results.

To remove the excess points, the bisection plane for the joint is found and a simple test performed. All points that are determined to belong to one side of the split in template space, yet reside on the opposite side of the bisection plane in construction space, are removed. Fig. 10 illustrates this process and Fig. 8 shows an example of its results.

3.4.4. *Handling repetition of parts*

The defined parameterization may be much longer than the one originally defined for the template object. The example shown in Fig. 11 illustrates such behavior. To handle these cases, repetition of the previously described user annotated regions are used. A set of these regions are selected such that when combined with the established anchor regions, they fully encompass the defined parameterization. Fitting repeated parts is then easily handled through storing additional values within the tokens in the Petri net, and using post-firing manipulation of these tokens. Algorithm 4 provides an overview of this process.

As tokens are moved across edges into a repetition region, the values stored in the token are adjusted to represent the current iteration of repetition. If a token is entering into a region for the first time, it is marked with a repetition iteration value of zero. This value is a simple counter that tracks which chosen repetition the token belongs to, and helps determine the necessary offset transformation for correct fitting. Note that a given place may contain tokens from several different iterations. Thus, this value must be checked during transition firing and for a place to fire, all tokens must be of the same iteration. Finally, if this is the first repetition place to receive a token, it is recorded as the initial starting place for the repetition region.

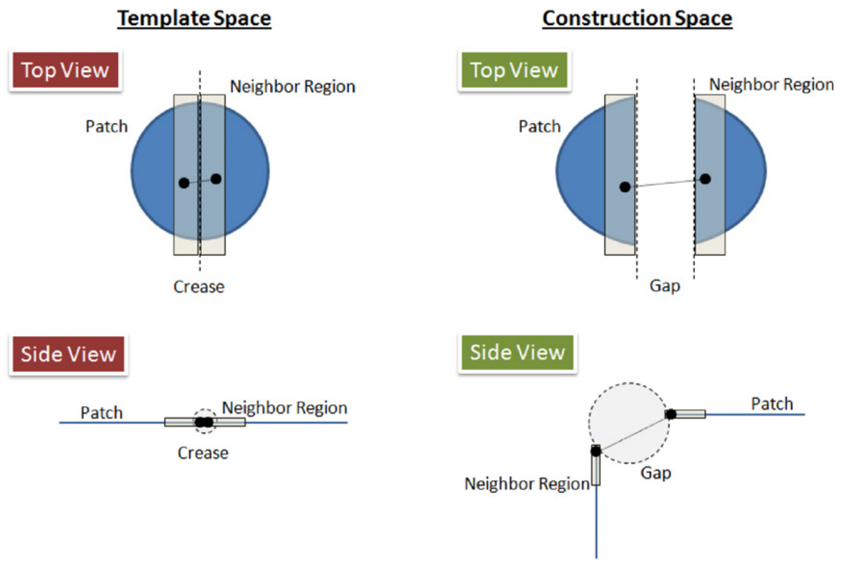

Fig. 9. Gaps in the surface definition caused by articulation can be filled by "stretching" a patch across the void. This is performed by identifying a fixed width region on either side of the crease in template space, locating the samples in each region and their corresponding nearest neighbor in the opposite region, and then sub-sampling the line connecting each pair of samples.

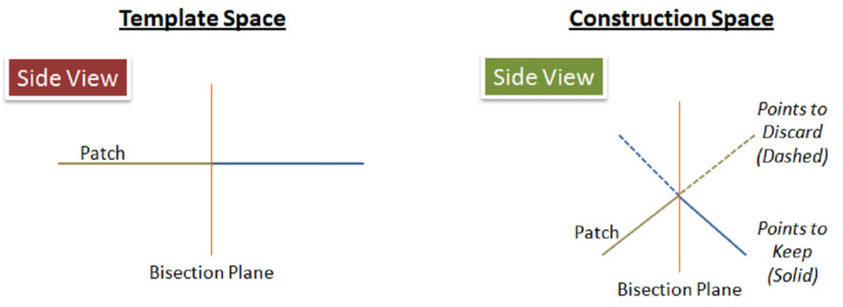

Fig. 10. Points from an inter-surface intersection caused by articulation can be removed by finding the bisection plane at the joint, and performing a simple spatial test to identify the points that lie on one side of this plane in template space and the opposite side in construction space.

Algorithm 4

```
1.  DetermineRepetitionsRequired(param);
2.  for tok ∈ tokensCrossingRepetitionBoundary
3.      selReg = GetRegionCrossed(tok);
4.      if DidTokenEnterRegion(tok, selReg)
5.        then
6.          enterStatus = IsFirstTimeRegionEntered(tok, selReg);
7.          if enterStatus == true
8.            then
9.              IncreaseRepetitionCounter(tok.repCounter);
10.             markedEnterNode = LookupNode(tok);
11.           else
12.             DiscardToken(tok); // part already started
13.     if DidTokenExitRegion(tok, selReg)
14.       then
15.         if IsAnotherRepetitionNeeded(selReg)
16.           then
17.             // move token to first marked node of repetition
18.             removeNode = LookupNode(tok);
19.             RemoveTokenFromNode(tok, removeNode);
20.             IncreaseRepetitionCounter(tok.repCounter);
21.             UpdateOffset(tok.offset);
22.             AddTokenToNode(tok, markedEnterNode);
23.           else
24.             // leave the token where it is, but invalidate it
25.             InvalidateRepetitionCounter(tok.repCounter);
26.             UpdateOffset(tok.offset);
```

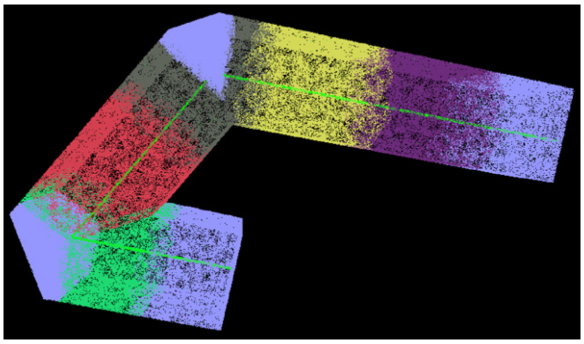

Fig. 11. Results of the patch fitting process using articulation and repetition of parts. The different colored regions correspond to the repeated parts used to fully model the defined parameterization.

When a token leaves a repetition region, going back into an anchor region, we must determine whether an additional repetition is allowed or desirable. This is determined based on the remaining length of the given parameterization, and the amount of space taken up by the repetition and anchor regions fitted thus far. If another iteration is to be started, then the token is removed from the outgoing place it currently resides in, and it is reinserted at the previously captured starting place for the repetition with an incremented iteration value. Finally, when it is determined that the repetition is done and tokens move out of the repetition region and into an anchor region, the token is marked as invalid and the offset value stored in the token (recall that the offset determines

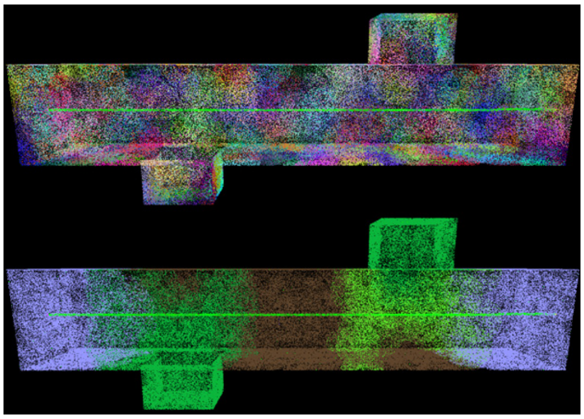

Fig. 12. Results of the patch fitting process using interchangeable parts. The top image shows the fitted patches and the bottom image shows the randomly selected parts.

where patches are located in construction space) must be updated.

Fig. 11 shows the result of the patch fitting process when using repetition of parts. In this figure, the different colored regions correspond to the repeated parts (i.e., each repetition is assigned a unique color, and repetition is performed until the parameterization has been modeled).

3.4.5. Handling interchanging of parts

Finally, we must handle any interchangeable parts obtained from the input template. Fig. 12 shows an example of this behavior using the annotations defined in Fig. 4. In this example, the user marked two different parts as being interchangeable (i.e., the two blue spheres). One part corresponds to a peg, and the other to a flat region. Interchanging these two parts randomly, along repeated regions, allows the creation of many different configurations of the object (one such example is shown in Fig. 12). Similar to the repetition of parts, through analyzing and manipulating the tokens moving through the Petri net, this behavior can be easily handled. Algorithm 5 provides an overview of this process.

Algorithm 5

```
1.  for $tok \in tokensCrossingInterchangeableBoundary$
2.        $selReg = GetRegionCrossed(tok)$;
3.        if $DidTokenEnterRegion(tok, selReg)$
4.          then
5.              if $IsFirstTimeRegionEntered(tok, selReg)$
6.                then
7.                    $annList = GetInterchangeAnnotations(selReg)$;
8.                    $randReg = ChooseRandomRegion(annList)$;
9.                    $fitParams = GetFitParams(selReg, randReg)$;
10.                   $off = GetRelativeOffset(tok)$;
11.                   $node = GetClosestNodeInRegion(randReg, off)$;
12.                   $tokenCreated = InstantiateToken(node)$;
13.                   $AddTokenToNode(tokenCreated, node)$;
14.             else
15.                   $DiscardToken(tok)$; // part already started
16.       if $DidTokenExitRegion(tok, selReg)$
17.         then $DiscardToken(tok)$; // discard the token on exiting
```

As a token moves through the network from a non-interchangeable region into a region defining an interchangeable part, a check must be performed to see if this is the first time the region and repetition instance has been entered. If it is the first time, then a new part must be selected for fitting. If it is not the first time, then the token is discarded as the region has already been handled and a fitting is underway.

First, a random selection is made from the list of available parts as defined by the user annotations. We must then fit this selected part to the correct location along the given parameterization. This process begins by finding the parameters needed to transform the randomly selected part to the part it is replacing in the template model (i.e., a transformation between each of these items as they are defined in template space is calculated).

Next, the relative offset of the selected interchangeable sample to the center of the interchangeable region is found. The offset and transformation is then used to find the closest corresponding sample within the randomly selected part. This sample will serve as the starting point for fitting the remainder of this selected part.

Now that the starting sample has been found, a token can be instantiated and initialized with values (e.g., interchange index and repetition in the standard template, index of the randomly selected region that will replace the template, and location). Finally, the token is added to the corresponding place for this starting element to begin the fitting process of the part. On subsequent iterations of fitting, the patches making up the selected region will be fit until all have been handled and the part is fully defined.

Note that the interchange case must be handled in conjunction with the repetition case, as repeated regions can encompass different interchangeable parts. The image in Fig. 12 shows an example of this behavior. With each repetition, a different set of randomly selected parts is chosen. Using a combination of these parts allows easy generation of a wide variety of different model configurations.

3.5. Solid model reconstruction and integration

Once the inference-based patch fitting process has completed, we have a set of sub-sampled surface patches fit around the parameterization that need to be integrated into a solid model. The set of these samples will serve as the basis of a reconstruction process where the overall goal is to join them together to form a watertight and smooth surface definition.

The reconstruction process begins by taking the point samples from all of the fitted patches, and those that were added to fill the gaps, and combines them to form a single point cloud representation of the object (i.e., a set of samples that fully defines the boundary of the object). A projection-based Moving Least Squares (MLS) approach [37] is then used to obtain a smooth and continuous surface definition. A Marching Cubes contouring algorithm [47] is then used to generate a polygonal representation from the implicit surface. Marching Cubes is a simple and efficient technique, but many other alternative contouring algorithms could also be used (e.g., [48–50]). An example constructed object is shown in Fig. 13 and the underlying template object used is shown in the top left corner.

The final stage of our process takes the constructed object and integrates it into an overall virtual environment. Integration is an important step, as the object must be inserted into the environment in a believable position and pose such that it looks natural to the user. Since our work is focused on creating very cluttered environments such as the real-world example shown in Fig. 1, we use a simple method for insertion. Each object is dropped from a predefined height and a physically based simulation determines its final resting place. The result provides a re-creation of such environments adequate for further applications. Fig. 1 provides an example of a generated environment.

Table 1
Details of the examples.

Dataset	Num. objects	Total sampling (s)	Ave. fitting (s)	Ave. solid (s)	Total runtime (s)
Example 1	25	3.06	0.95	14.97	648.79
Example 2	20	5.81	2.02	27.97	1198.79
Example 3	25	25.15	4.19	43.52	1899.00
Example 4	30	19.31	2.43	16.51	1034.40

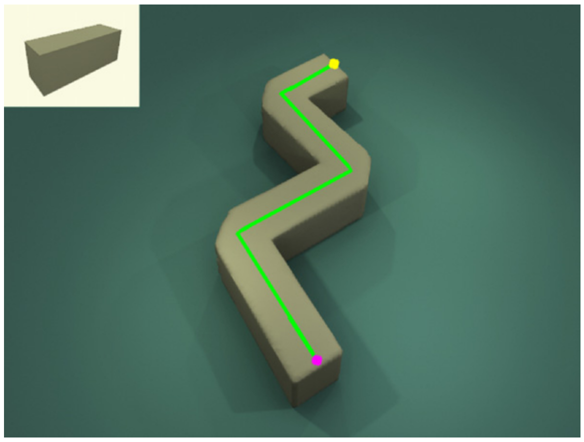
Fig. 13. An example constructed object (center) from a block template (top left).

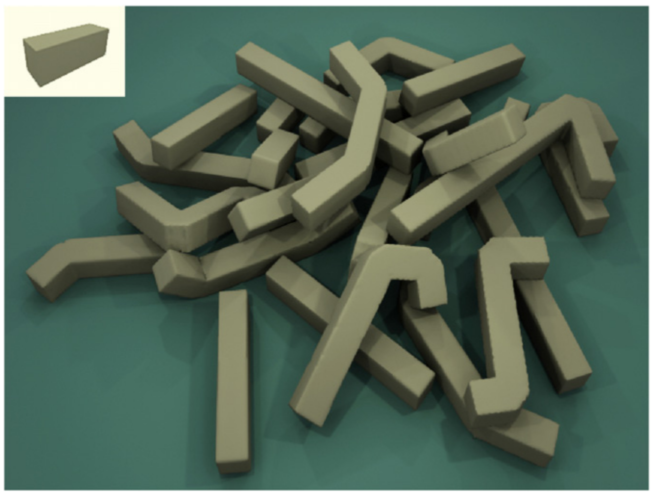
Fig. 14. Constructed blocks using articulation and part repetition (Example 1).

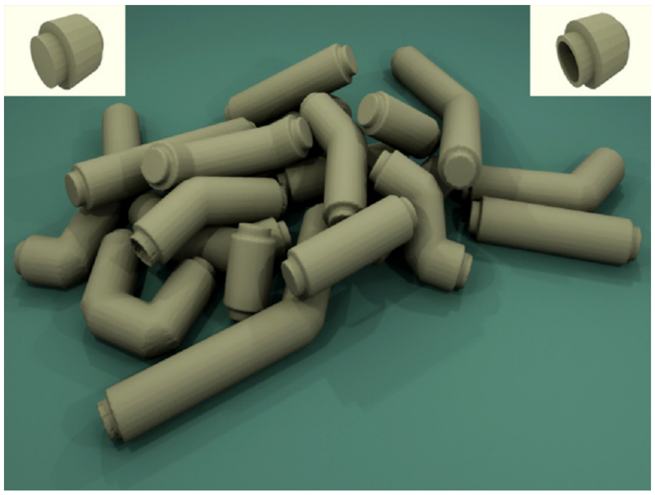
Fig. 15. Constructed pipes using articulation and part repetition (Example 2).

4. Results

In order to demonstrate the application of our approach, we provide several examples along with the details of their execution. All results were captured using an Intel Xeon 2.67 GHz CPU with 4 GB of memory. Currently, our algorithm implementation performs all computation on the CPU and uses the GPU only for rendering. The dataset details and respective execution times for each stage (i.e., structure sampling, patch fitting, and solid model construction) of the provided examples are shown in Table 1. The total runtime summarizes the complete construction time, combined with the amount of time for integrating the objects into a simulation-based environment.

Figs. 14–17 show some results generated by our approach. In these examples, we show how different types of objects can be easily constructed. Through the use of articulation, repetition, and interchanging of parts, many different object configurations can be quickly developed. The template object used for each example is shown in the outside corners of the figure. Finally, Fig. 18 shows the results of a constructed environment integrated into a physically based simulation.

5. Discussion

The closest methods to our proposed approach are the model synthesis techniques described by Merrell et al. [4] and Bokeloh et al. [21], the point cloud augmentation approach by Gal et al. [29], and the many feature-based modeling methods [39,40].

The model synthesis approaches [4,21] have a common theme to ours, but each takes a different underlying approach. Similar to our method, Merrell et al. [4] allow arbitrary generation of random models by extracting regions from a provided template. However, their approach centers around the satisfaction of different categories of constraints (e.g., adjacency, algebraic, incidence, connectivity, etc.). These constraints govern a dynamic growing/creation of a model using model synthesis. The method by Bokeloh et al. [21] also uses an example-based method for procedurally constructing models, but their approach cuts along symmetric regions and then builds shape operations that use these pieces while maintaining model similarity within the final construction. Their approach develops a shape grammar to guide the random construction process. Our approach is different and is focused on fitting the extracted patches around an already defined parameterization using the implicitly defined rules captured in the Petri net.

The augmentation approach [29] is focused on obtaining higher quality reconstructions by fitting extracted patches from prior models with similar surface characteristics. Their approach is able to smooth noisy data, and fill small voids in the final reconstruction. Our approach loosely extends their idea and uses extracted patches from a template model, but we fit them to fully define the boundary of an object around a characterizing parameterization.

Many different feature-based methods have been proposed [39, 40]. These methods typically rely on explicitly defined rules for how primitives (typically solids) can be used to construct larger and more complex objects. Our approach uses a similar notion,

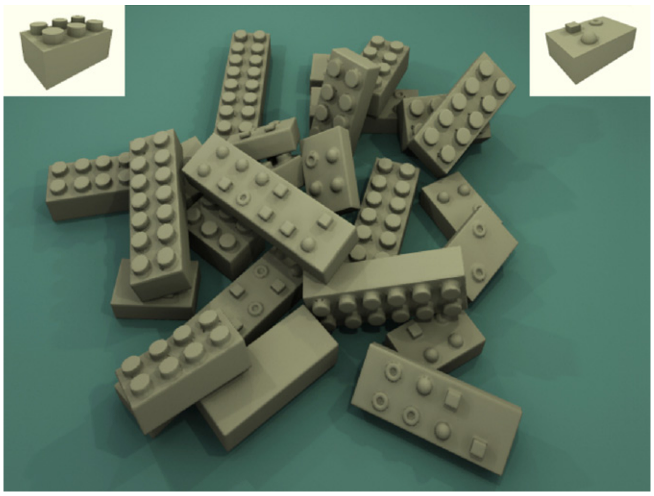

Fig. 16. Constructed bricks using repetition and interchangeable parts (Example 3). The different style pegs can be interchanged and then repeated iteratively across the model.

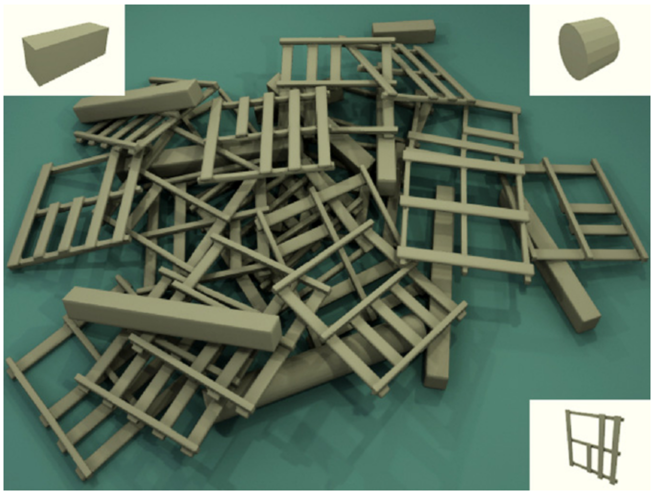

Fig. 17. Constructed environment similar to the provided real-world example that inspired our approach (Example 4). The different configurations of pallets were generated using repetition and random interchanging of board configurations.

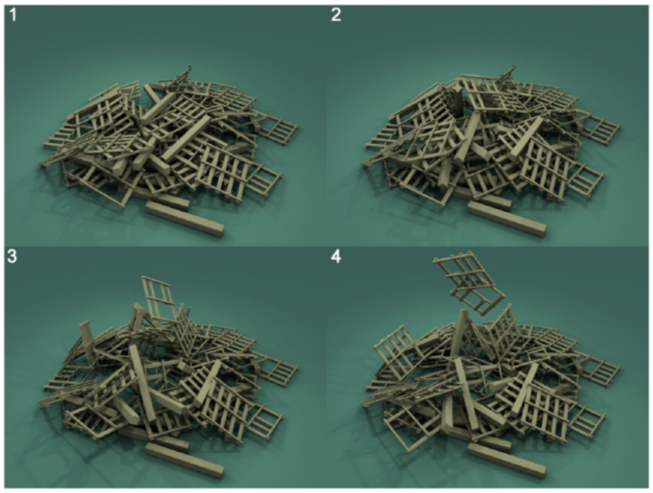

Fig. 18. Example environment incorporated into a physically based simulation. Several stages are shown after applying forces to the pile.

except rather than using explicitly defined rules, our approach incorporates a Petri net structure automatically developed from the provided template model. We use this Petri net structure to guide the dynamic fitting of surface patches around a parameterization, which when incorporating behavior such as repetition and interchanging of parts, allows for the construction of a wide variety of randomly defined objects.

We have introduced the first procedural solid generation approach (as far as we are aware) based on Petri nets. Petri nets provide a very general structure for describing the relationships between various parts of a model, and they allow a nice means for tracking the distributed fitting process of these parts and ensuring patches are fit in a consistent fashion. While our implementation is somewhat limited in scope, it demonstrates that Petri nets can provide a good foundation for a procedural generation algorithm.

Our procedural approach allows the definition of many different object configurations, but currently has several limitations that hinder its creativeness. First, it only allows fitting of patches from within a single provided model. It also requires two adjacent patches being fit to have smooth and similar transitions in overlapping areas (i.e., to correctly join the patches together). MLS can smooth small discontinuities, but large gaps between patches will result in poorly constructed models. Finally, the one-dimensional parameterizations proposed allow for only limited control over the object being constructed.

There are several ways in which our work might be extended. First, the parameterization-based approach can be extended beyond the simple definition we use, to a more expressive system that allows the user to draw symbols and better defined illustrations for annotating desired behaviors. Second, by alleviating the user from annotating the input models (for marking specialized regions), and replacing this step with a more automated approach, the overall construction process becomes much simpler. Finally, mixing and matching patches from different models would allow for a wider variety of models to be defined. However, this would increase the complexity of our algorithm as potentially very different patches would have to be consistently fit together to define a solid object. The incorporation of these different ideas, combined with addressing our approach's current limitations, would allow a more flexible approach that could generate a wider variety of models.

6. Conclusion

We have introduced an inference-based procedural solid model generation algorithm based on Petri nets. Our approach provides an efficient means for creating variations of a user provided template. This process can function in both a fully automated and semi-automated fashion, and can robustly construct solid models that include articulation, repetition, and interchanging of parts obtained from a template model.

While each of these methods provides a means for modeling objects under different conditions, the broader objective of our work is to develop high-fidelity virtual environments for anything from games and entertainment, to various simulation-based applications. In each of these application areas, having a solid representation of each object for simulation purposes is critical. Figs. 1 and 18 provide a demonstration of such an approach. Overall, our techniques help to address the requirements of many different application areas by supporting the automated construction of more complex environments in a quicker and easier fashion.

Acknowledgments

This work was supported in part by NSF Grant IIS-0917286 and by Award No. KUS-C1-016-04 from King Abdullah University of Science and Technology (KAUST).

References

[1] Ebert DS, Musgrave FK, Peachey D, Perlin K, Worley S. Texturing and modeling: a procedural approach. 3rd ed. San Francisco (CA): Morgan Kaufmann Publishers; 2002.
[2] Kelly G, McCabe H. A survey of procedural techniques for city generation. ITB J 2006;14:87–130.
[3] Watson B, Müller P, Wonka P, Sexton C, Veryovka O, Fuller A. Procedural urban modeling in practice. IEEE Comput Graph Appl 2008;28(3):18–26.
[4] Merrell P, Manocha D. Constraint-based model synthesis. In: Proc. of the 2009 SIAM/ACM joint conf. on geometric and physical modeling. ACM; 2009. p. 101–11.
[5] Merrell P, Schkufza E, Koltun V. Computer-generated residential building layouts. ACM Trans Graph 2010;29(6):1–12.
[6] Procedural I. Cityengine: 3D modeling software for urban environments. February 2011. http://www.procedural.com/.
[7] Prusinkiewicz P, Lindenmayer A. The algorithmic beauty of plants. New York (NY): Springer-Verlag, Inc.; 1990.
[8] Weber J, Penn J. Creation and rendering of realistic trees. In: Proc. of the 22nd annual conf. on computer graphics and interactive techniques. ACM; 1995. p. 119–28.
[9] Měch R, Prusinkiewicz P. Visual models of plants interacting with their environment. In: Proc. of the 23rd annual conf. on computer graphics and interactive techniques. ACM; 1996. p. 397–410.
[10] Prusinkiewicz P, Mündermann L, Karwowski R, Lane B. The use of positional information in the modeling of plants. In: Proc. of the 28th annual conf. on computer graphics and interactive techniques. ACM; 2001. p. 289–300.
[11] IDV Inc. Speedtree. February 2011. http://www.speedtree.com/.
[12] Geiss R. GPU gems 3. Generating complex procedural terrains using the GPU, New York (NY): Addison-Wesley Publishing Co.; 2007. [Chapter 1].
[13] Smelik RM, deKraker KJ, Groenewegen SA, Tutenel T, Bidarra R. A survey of procedural methods for terrain modeling. In: A. Egges, W. Hürst, and R.C. Veltkamp (Eds.), Proc. of the CASA workshop on 3D advanced media in gaming and simulation. 2009. p. 25–34.
[14] Software P. Planetside. February 2011. http://www.planetside.co.uk/.
[15] P. Inc. Mojoworld. February 2011. http://www.mojoworld.org/.
[16] Barnsley M. Fractals everywhere. 2nd ed. San Diego (CA): Academic Press Professional, Inc.; 1993.
[17] Perlin K. An image synthesizer. SIGGRAPH Comput Graph 1985;19(3):287–96.
[18] Lindenmayer A. Mathematical models for cellular interactions in development I and II. J Theoret Biol 1968;18(3):280–315.
[19] Lefebvre S, Neyret F. Pattern based procedural textures. In: Proc. of the 2003 symp. on interactive 3D graphics. ACM; 2003. p. 203–12.
[20] Stiny GN. Pictorial and formal aspects of shape and shape grammars and aesthetic systems. Ph.D. thesis. Los Angeles (CA): University of California; 1975.
[21] Bokeloh M, Wand M, Seidel H-P. A connection between partial symmetry and inverse procedural modeling. ACM Trans Graph 2010;29(4):1–10.
[22] Schnabel R, Wahl R, Klein R. Efficient RANSAC for point-cloud shape detection. Comput Graph Forum Proc Eurograph 2007;26(2):214–26.
[23] Jenke P, Krückeberg B, Straßer W. Surface reconstruction from fitted shape primitives. In: Proc. of vision, modeling and visualization 2008. IOS Press; 2008. p. 31–40.
[24] Schnabel R, Wessel R, Wahl R, Klein R. Shape recognition in 3D point-clouds. In: Proc. of the 16th int. conf. in central Europe on computer graphics, visualization and computer vision. UNION Agency-Science Press; 2008. p. 1–8.
[25] Funkhouser T, Kazhdan M, Shilane P, Min P, Kiefer W, Tal A, et al. Modeling by example. ACM Trans Graph 2004;23(3):652–63.
[26] Kreavoy V, Julius D, Sheffer A. Model composition from interchangeable components. In: Proc. of the 15th Pacific conf. on computer graphics and applications. IEEE Computer Society; 2007. p. 129–38.
[27] Gal R, Cohen-Or D. Salient geometric features for partial shape matching and similarity. ACM Trans Graph 2006;25(1):130–50.
[28] Shalom S, Shapira L, Shamir A, Cohen-Or D. Part analogies in sets of objects. In: Proc. of eurographics symp. on 3D object retrieval. Eurographics Assoc.; 2008. p. 33–40.
[29] Gal R, Shamir A, Hassner T, Pauly M, Cohen-Or D. Surface reconstruction using local shape priors. In: Proc. of the fifth eurographics symp. on geometry processing. Eurographics Assoc.; 2007. p. 253–62.
[30] Pauly M, Mitra NJ, Wallner J, Pottmann H, Guibas LJ. Discovering structural regularity in 3D geometry. ACM Trans Graph 2008;27(3):1–11.
[31] Berner A, Bokeloh M, Wand M, Schilling A, Seidel H-P. A graph-based approach to symmetry detection. In: Proc. of the IEEE/EG int. symp. on volume and point-based graphics. Eurographics Assoc.; 2008. p. 1–8.
[32] Bokeloh M, Berner A, Wand M, Seidel H-P, Schilling A. Symmetry detection using feature lines. Comput Graph Forum Proc Eurograph 2009;28(2): 697–706.
[33] Amenta N, Bern M, Kamvysselis M. A new voronoi-based surface reconstruction algorith. In: Proc. of the 25th annual conf. on computer graphics and interactive techniques. ACM; 1998. p. 415–21.
[34] Dey TK, Giesen J, Hudson J. Delaunay based shape reconstruction from large data. In: Proc. of the IEEE 2001 symp. on parallel and large-data visualization and graphics. IEEE Computer Society; 2001. p. 19–27.
[35] Hoppe H, DeRose T, Duchamp T, McDonald J, Stuetzle W. Surface reconstruction from unorganized points. In: Proc. of the 19th annual conf. on computer graphics and interactive techniques. ACM; 1992. p. 71–8.
[36] Alexa M, Behr J, Cohen-Or D, Fleishman S, Levin D, Silva CT. Computing and rendering point set surfaces. IEEE Trans Vis Comput Graphics 2003;9(1): 3–15.
[37] Amenta N, Kil YJ. Defining point-set surfaces. ACM Trans Graph 2004;23(3): 264–70.
[38] Kazhdan M, Bolitho M, Hoppe H. Poisson surface reconstruction. In: Proc. of the fourth eurographics symp. on geometry processing. Eurographics Assoc.; 2006. p. 61–70.
[39] Han J, Pratt M, Regli WC. Manufacturing feature recognition from solid models: a status report. IEEE Trans Robot Autom 2000;16(6):782–96.
[40] Shah JJ, Anderson D, Kim YSe, Joshi S. A discourse on geometric feature recognition from CAD models. J Comput Inf Sci Eng 2001;1(1):41–51.
[41] Alberti MA, Evi P, Marini D. Modeling constrained geometric objects with OBJSA nets. In: Concurrent object-oriented programming and Petri nets. Secaucus (NJ): Springer-Verlag New York, Inc.; 2001. p. 319–37. [Chapter].
[42] Turk G. Generating textures on arbitrary surfaces using reaction–diffusion. Comput Graph 1991;25(4):289–98.
[43] Elad M, Tal A, Ar S. Content based retrieval of VRML objects: an iterative and interactive approach. In: Proc. of the sixth eurographics workshop on multimedia. Springer-Verlag, Inc.; 2002. p. 107–18.
[44] Peterson JL. Petri nets. ACM Comput Surv 1977;9(3):223–52.
[45] Abelson H, diSessa A. Turtle geometry: the computer as a medium for exploring mathematics. Cambridge (MA): MIT Press; 1986.
[46] Sagan H. Space-filling curves. New York (NY): Springer-Verlag, Inc.; 1994.
[47] Lorensen WE, Cline HE. Marching cubes: a high resolution 3D surface construction algorith. In: Proc. of the 14th annual conf. on computer graphics and interactive techniques. ACM; 1987. p. 163–9.
[48] Ju T, Losasso F, Schaefer S, Warren J. Dual contouring of hermite data. ACM Trans Graph 2002;21(3):339–46.
[49] Schaefer S, Warren J. Dual marching cubes: primal contouring of dual grids. In: Proc. of the 12th Pacific conf. on computer graphics and applications. IEEE Computer Society; 2004. p. 70–6.
[50] Kazhdan M, Klein A, Dalal K, Hoppe H. Unconstrained isosurface extraction on arbitrary octrees. In: Proc. of the fifth eurographics symp. on geometry processing. Eurographics Assoc.; 2007. p. 125–33.