EUROGRAPHICS 2011 / M. Chen and O. Deussen
(Guest Editors)

Volume 30 (2011), Number 2

Procedural Modeling of Interconnected Structures

Lars Krecklau and Leif Kobbelt

RWTH Aachen University, Germany

Abstract

The complexity and detail of geometric scenes that are used in today's computer animated films and interactive games have reached a level where the manual creation by traditional 3D modeling tools has become infeasible. This is why procedural modeling concepts have been developed which generate highly complex 3D models by automatically executing a set of formal construction rules. Well-known examples are variants of L-systems which describe the bottom-up growth process of plants and shape grammars which define architectural buildings by decomposing blocks in a top-down fashion. However, none of these approaches allows for the easy generation of interconnected structures such as bridges or roller coasters where a functional interaction between rigid and deformable parts of an object is needed. Our approach mainly relies on the top-down decomposition principle of shape grammars to create an arbitrarily complex but well structured layout. During this process, potential attaching points *are collected in* containers *which represent the set of candidates to establish interconnections. Our grammar then uses either* abstract connection patterns *or* geometric queries *to determine elements in those containers that are to be connected. The two different types of connections that our system supports are* rigid object chains *and* deformable beams*. The former type is constructed by inverse kinematics, the latter by spline interpolation. We demonstrate the descriptive power of our grammar by example models of bridges, roller coasters, and wall-mounted catenaries.*

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Languages—I.3.5 [Computer Graphics]: Geometric algorithms, languages, and systems—I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—

1. Introduction

Modeling highly complex 3D scenes with rich detail is one of the most important topics for movie production and nowadays also for the game industry [WMV*08]. Procedural modeling has become a well established approach when realistically looking landscapes [EMP*02], architecture [MWH*06], or plants [PHL*09] have to be generated.

In general, procedural models describe a scene by a textual grammar consisting of several rules which recursively replace *non-terminal* input symbols by a sequence of new *terminal* or *non-terminal* output symbols. There exist several different strategies of evaluating a grammar with regard to 3D content creation. String replacement, e.g., is commonly used for plant modeling. In this case, the evaluation of the grammar is based on a given input text. After some iteration steps of modifying the text the resulting string is visually interpreted. This convention is best suited to model the growth

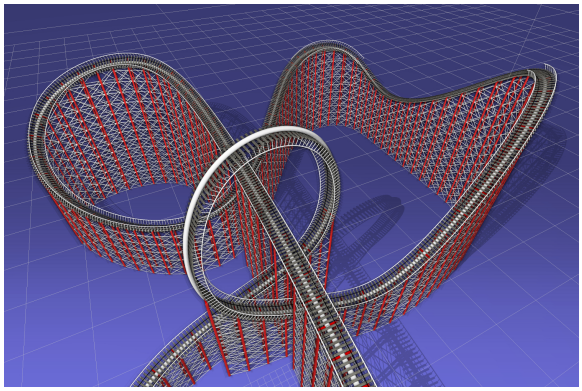

Figure 1: *Our system presents a novel method to easily describe complex interconnected structures with a high level of detail like this roller coaster.*

process of plants. Another approach for 3D content creation based on rules are shape grammars. Here rules are applied to replace a certain shape in the current scene by one or several others. Usually, the new shapes represent a higher level of detail, e.g. a box is replaced by four smaller ones. Since this approach corresponds to the idea of modeling in a coarse-to-fine fashion, it is preferably used in architectural modeling.

A drawback of both strategies is, however, that they do not support the definition of connections between existing objects in the scene. Such connections are very important for the creation of complex structures that are often seen in the real world, e.g. bridge constructions, architectural framings or roller coasters to mention only a few. Another important observation regarding complex interconnected structures is, that we have to deal with two different classes of objects, namely rigid objects, such as metal joints, that are contained within a bounding box and deformable objects, like bended steel girders or beams, which can be modeled by a sequence of trilinear freeform deformation (FFD) cages. While high level mechanisms are used to layout the coarse scene of rigid and deformable objects, splitting operations on the boxes and the FFDs will produce the fine details.

Our modeling strategy is based on a simple idea. First, we collect meaningful locations in *containers*, like possible connection points on a steel girder. Then, we use the elements of the containers to create mappings between two locations which satisfy a number of *relations* between a source and a target location. Alternatively, geometric queries can be used to define those relations. Finally, an inverse kinematics (IK) algorithm calculates a chain of rigid objects between the source and the target location or a spline is used to generate a deformable beam between the end points.

For a fast and easy scene assembly, our procedural modeling setup is based on the following key points:

Containers — During the scene generation, meaningful connection points are collected that can be utilized later on to establish relations between source and target locations to construct geometric connections. Furthermore, our system also uses containers to define freeform spline curves or to define avoidance areas that may prevent two end locations of a relation from being connected.

Rigid & Deformable — Our paper presents an easy scene description that allows us to apply the well-known splitting operators from previous approaches to rigid as well as to deformable objects. Therefore, no new concepts have to be learned by the user in order to generate the fine details of the geometry in our scenes.

Inverse Kinematics — Our grammar provides an intuitive way to define how two end locations will be connected by a chain of rigid objects. The user just has to set the degrees of freedoms in terms of translational and rotational joints and our system will automatically generate a plausible layout of boxes by using an inverse kinematics algorithm.

Curves — Our system uses spline curves as a tool to generate a continuous sequence of trilinear freeform deformation cages. In our approach, the control points will be extended to form rectangular cross sections along the curve that gives a high level control over the roll of the FFDs along the curve which would not be the case, if only control points would be interpolated.

1.1. Related Work

Procedural modeling approaches have become well established for plant modeling after Lindenmayer and Prusinkiewicz have introduced L-Systems into graphics [PL96]. A LOGO-style turtle is used for the visual interpretation of the resulting text string. Furthermore, several extensions have been developed to enhance the expressive power of the textual grammar like parameters, stochastic rule application or string-based context-sensitivity, which is used to simulate information flow within a plant [PHHM97]. The concept of self-sensitivity was introduced for the automatic creation of street networks [PM01]. The plant modeling language *cpfg* [PHM00, PKMH00] provides mechanisms called open L-System (OLS) for a more sophisticated environmental-sensitivity [PJM94, MP96]. The communication between an OLS and external modules goes via a fixed set of parameters. Since our approach needs to define IK systems of a dynamic size, an OLS would need to create a separate external module for every possible number of elements in an IK chain of joints. Furthermore, L-systems are designed for growth processes where each step is determined in a parallel and greedy fashion with no planning ahead and no backtracking optimizations. In order to produce interconnected structures, the OLS formalism would have to be extended at least by the capability of passing parameter lists to a module that define the degrees of freedom of a rigid chain. Additionally, the replacement strategy would have to be changed to guarantee that certain objects are created before others in order to apply geometrically queries on fully generated geometry.

Instead on working on text strings, decomposition of shapes into other shapes better fits for the generation of man-made structures like architecture [WWSR03]. Stiny pioneered the concept of rule-based shape replacement by introducing shape grammars [Sti75, Sti80]. This basic idea was then transferred to the creation of procedural buildings with help of the *CGA shape* modeling language [MWH*06]. In their work, Müller et. al. present a modeling strategy of placing mass models in a first step. Those are refined by splitting rules leading to detailed building geometry. Occlusion queries are taken into account to avoid the creation of objects like windows when they would be partly hidden by an intersecting wall. We basically rely on the concepts of *CGA shape* as this formalism turned out to work well for the creation of man-made structures like buildings. By slightly modifying the modeling strategy, we are then able to provide a mechanism for the generation of interconnected structures.

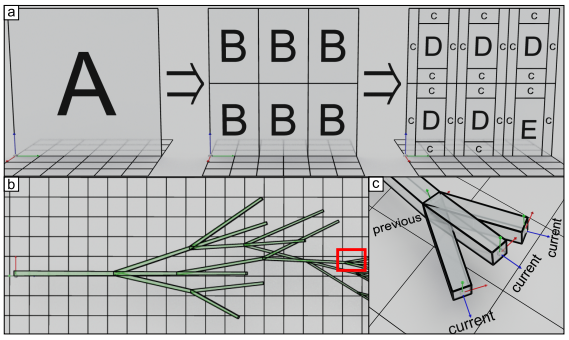

Figure 2: *(a) Decomposition rules provide an easy mechanism to create man-made structures like facades. (b) Controlling a rectangular LOGO-style turtle generating complex tree structures by successively spanning trilinear FFDs. (c) Close-up of (b).*

There are several other procedural modeling approaches for specific problems like noise functions for procedural terrains [EMP*02] or wall grammars for simple 2.5D facades [LG06]. The creation of building interiors, e.g., can be achieved by recursive splitting [HBW06]. Galin et al. presented a system for the generation of more complex roads by approximating the anisotropic shortest path problem [GPMG10]. During the generation, a cost function allows for an automatic placement of tunnels and bridges. Applying a physical simulation of the elements that are placed by a grammar like *CGA shape*, Whiting et al. are able to optimize a certain set of parameters in order to create stable configurations of masonry buildings [WOD09]. Ganster et al. combine different modeling strategies in a visual framework [GK07] whereas Krecklau et al. present the unified procedural modeling language G^2 which handles multiple non-terminal classes [KPK10] such as boxes or freeform deformations (FFDs) [SP86]. Our system is also capable of creating *FFD based deformable objects*, but in contrast to G^2, we use high level modeling primitives such as spline curves [PT97] to define sequences of FFDs between two given end locations.

Furthermore, our approach was inspired by the practical application of inverse kinematics, which is usually applied in character animation or robot control [MWS05]. Based on the assumption, that a set of rigid objects is given and that each of these objects is at least connected to one other object of the set by a rotational or a translational joint, IK algorithms numerically solve the problem of finding a valid joint configuration such that a subset of the rigid objects can be placed at specific positions and orientations in the scene. Our approach uses an IK algorithm [Bus04] for an easy definition of *box based rigid chains* between two end locations as an alternative to the deformable interconnections. Both concepts are unified in our modeling grammar for an intuitive and fast creation of interconnected structures that would be hard or even impossible to achieve with existing grammar systems.

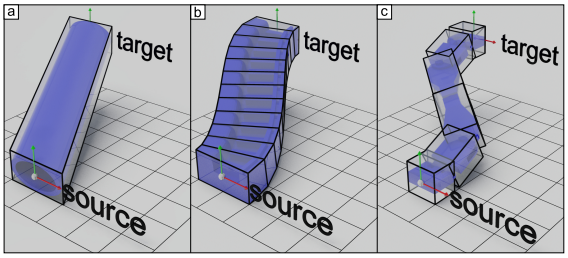

Figure 3: *Our approach presents operations for an easy handling of interconnected structures. When precise connections have to be established, the user can choose to create a simple FFD (a), a deformable beam (b) or a rigid chain (c).*

2. Procedural Modeling

In general, a context-free formal grammar is a 4-tuple (N,T,P,S), where N is a set of non-terminal symbols, T is a set of terminal symbols, P is a set of production rules of the form $N \rightarrow (N \cup T)^*$ and $S \in N$ is the start symbol. Whenever a non-terminal symbol matches the left-hand side of a production rule, that rule can be applied and the symbol will be replaced by the right-hand side of the rule.

Basically, our procedural modeling language builds upon the idea of *CGA shape* by Müller et al. [MWH*06] which has been successfully used for the easy creation of man-made structures like buildings. In their work, they simplify the concept of general shape grammars by associating a textual non-terminal symbol with a *scope*, which can be seen as a bounding box in 3D space that may contain an arbitrary geometry. The right-hand side of a rule in *CGA shape* contains a sequence of operators that are applied to the scope. The most important operators to mention here are local transformations, like a translation or a rotation, to simulate LOGO-style turtle movements or splitting operators that subdivide the scope into smaller scopes, like dividing the scope along the local x-axis into parts that have approximately the same width (cf. Figure 2.a). Operators that generate new scopes in 3D space also have to set the attached non-terminal symbol in order to further process the scope by a certain rule.

The principle of using scopes only as boxes is not sufficient, if interconnected structures have to be generated that contain deformed pieces such as steel girders or beams. Therefore, we adopt the idea of increasing the degrees of freedom of the scope from a box to a trilinear freeform deformation (FFD) as it was presented in the *generalized grammar (G^2)* by Krecklau et al. [KPK10]. In addition to the previously explained concept of splitting boxes in order to create buildings, they also show how to generate complex tree structures by controlling a rectangularly shaped turtle through space and spanning FFDs between these rectangles (cf. Figure 2.b). We rely on the syntax of G^2, because it provides useful features such as implicit class and operator attributes or a method to encapsulate several rules to build modules that can be easily reused throughout the grammar.

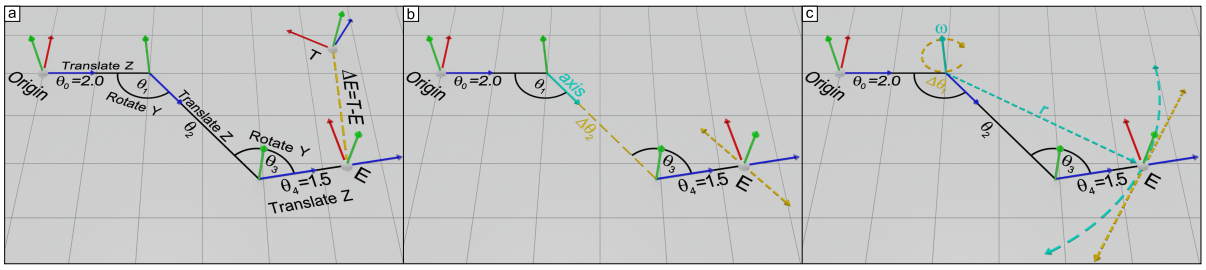

Figure 4: *(a) A rigid object chain is defined by translational* (θ_0,θ_2,θ_4) *and rotational* (θ_1,θ_3) *joints with one degree of freedom where some are fixed* (θ_0,θ_4) *and some are free* (θ_1,θ_2,θ_3)*. The free parameters are optimized to yield the smallest distance between an end effector E and a target pose T. (b-c) The Jacobian of the system is used to estimate the effect at the end effector, if the corresponding value* θ *changes. (b) Derivation principle for a translational joint. (c) Derivation principle for a rotational joint.*

While it turned out that boxes are easy to handle for the creation of precise structures in a strict coarse-to-fine fashion and FFDs can handle growth processes that start at a certain location and expand in random but guided directions, our approach aims for a whole new class of objects. We present a system that provides easy operations to establish connections between semantically meaningful locations in the scene. Our paper describes two completely different ways of generating geometry between two arbitrary locations in space. First, a chain of rigid objects can be created automatically between the locations by applying an inverse kinematics algorithm (cf. Figure 3.c). Details of the rigid objects are then easily generated in the manner of *CGA shape*. Alternatively, deformable connections are represented by a sequence of trilinear FFDs which are defined by a spline curve that interpolates the locations (cf. Figure 3.b). In order to generate details along the deformed connections we adapt the splitting concept of the boxes and apply them in a similar way to the FFD cages (cf. Figure 11.c).

3. Inverse Kinematics

A *rigid chain system* is a sequence of *revolute (rotational)* and *prismatic (translational)* joints. The joints in our kinematic system only have one degree of freedom (DoF) which is a scalar value θ representing an angle of a revolute joint for a rotation *around* a fixed axis or representing a length of a prismatic joint for a translation *along* a certain vector. More complex behaviors can be simulated by using a combination of these joints (cf. Figure 4.a).

Each translational joint can be expressed by a rigid transformation $T(\theta) \in \mathbb{R}^{4\times4}$ consisting of an identity matrix $I \in \mathbb{R}^{3\times3}$ as rotational part and a translation vector $\vec{t} \in \mathbb{R}^{3\times1}$ that is scaled by the length θ:

$$T(\theta) = \begin{bmatrix} I & \vec{t}*\theta \\ 0 & 1 \end{bmatrix}$$

Each rotational joint can be expressed by a rigid transformation $T(\theta) \in \mathbb{R}^{4\times4}$ consisting of a zero vector $\vec{z}$ as translational part and a rotation matrix $R \in \mathbb{R}^{3\times3}$ which is defined by a rotation axis $\vec{a}$ and the angle θ:

$$T(\theta) = \begin{bmatrix} R(\vec{a},\theta) & \vec{z} \\ 0 & 1 \end{bmatrix}$$

If a set of parameters $\Theta = \{\theta_0,\cdots,\theta_{n-1}\}$ is applied to a sequence of n rigid transformations, this will result in an end effector $E \in \mathbb{R}^{4\times4}$:

$$f(\Theta) = T_{n-1}(\theta_{n-1}) * \cdots * T_0(\theta_0) = E$$

Changing the parameters θ_i corresponds to the idea of *forward kinematics* [MWS05] which perfectly matches the concept of the LOGO-style turtle movement that is used for procedural modeling, because both systems rely on the idea of defining a path by just using relative transformations.

In terms of interconnected structures, we have to solve the problem, that we have only given the wanted target pose T of the end effector E and the sequence of rigid transformations where only a subset $\Phi \subseteq \Theta$ are free parameters that have to be optimized by the solver while $\Theta \setminus \Phi$ are fixed parameters that are necessary to fully describe the shape of the rigid chain. Finding the values for the free parameters $\theta_i \in \Phi$ is a hard problem, because f is highly non-linear due to the rotations. Inverse kinematics algorithms are numerical methods that solve $f^{-1}(E) = \Theta$ by iteratively updating the parameters Φ by $\Delta\Phi$ (cf. Figure 4.b,c). The algorithms usually terminate, when the distance ΔE between the current end effector E and the target pose T falls below a certain threshold. Further details on inverse kinematics can be taken from the book of Spong et. al. [MWS05].

4. Freeform Curves

Since the primitive scopes of our system can be trilinear FFDs which have many degree of freedom, we need higher order operations for their creation. Therefore, our system is capable of sweeping rectangles along a spline curve thereby creating a sequence of FFDs between a rectangle and its successor (cf. Figure 5). A curve that is parameterized in the in-

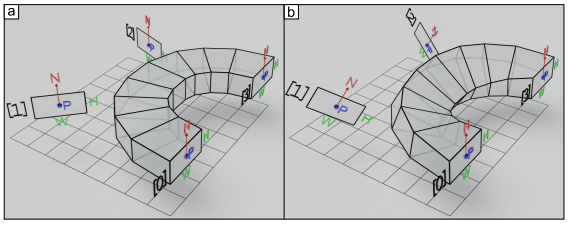

Figure 5: *(a) Deformable beams are created by sweeping rectangles along a spline curve. (b) The interpolation of normals yields an intuitive control over the roll of the rectangle.*

terval $t \in [0,1]$ with n control points P_i and basis functions N_i is described by the following formula:

$$C(t) = \sum_{i=0}^{n-1} N_i(t) P_i$$

For the precise details on this function we want to refer the reader to *The NURBS Book* [PT97]. Creating rectangles at the sample points to span trilinear FFDs can be done in several ways. For a natural behavior, the generated rectangles have their center in $C(t)$ and are spanned orthogonal to the tangent, which is the first derivative $C'(t)$ of the curve. The rotation of the rectangle around the tangent has still to be determined. In order to get full control over the roll of the rectangle, e.g. producing arbitrary twists of the rectangle along the curve, we extend the definition of the control points P_i:

$$P_i = [\underbrace{P_x, P_y, P_z}_{position}, \underbrace{N_x, N_y, N_z}_{normal}, \underbrace{W, H}_{size}]^T$$

We address $C_P(t)$ as the positional part, $C_N(t)$ as the normal part and $C_S(t)$ as the size part of the curve at parameter t. Note that the *normal* vector $C_N(t)$ is normalized after the interpolation in order to guarantee unit length. The local rectangle will have its center at $C_P(t)$. The local right direction R and the local up direction U are then calculated as follows:

$$R = \frac{C_N(t) \times C'_P(t)}{|C_N(t) \times C'_P(t)|}, \; U = \frac{R \times C'_P(t)}{|R \times C'_P(t)|}$$

Note that $U \neq C_N(t)$ in most cases, because the interpolated normal $C_N(t)$ is just used to determine the roll around the tangent direction.

5. Containers

By recursively replacing non-terminal scopes in the scene, a lot of detail can be created in a controlled way. Parameters are used to vary the behavior of the rules thereby changing the look of the generated scene. This is sufficient as long as the relation between objects or parts is unidirectional in the sense of a recursive replacement (parent → child). If interconnected structures have to be modeled, it is very intuitive to create independent architectural objects first and later establish connections between meaningful locations that are

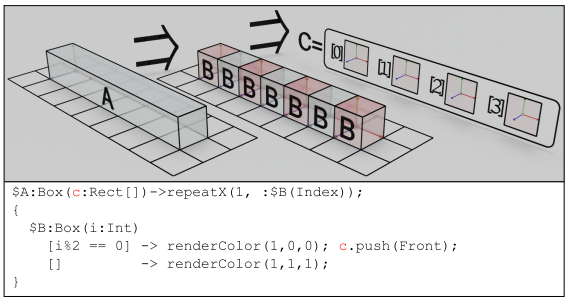

Figure 6: *Containers are used to store rectangles which are the end locations for creating interconnected structures. In the grammar,* index *is an implicit operator attribute and* front *is an implicit class attribute [KPK10].*

defined by the individual objects. Please note, that this would not be possible with a simple coarse-to-fine modeling strategy, since the connections are generated *between* existing objects in the scene and do not result from a decomposition or growth process as presented in previous approaches.

5.1. Collecting Meaningful Objects

In order to collect geometrically meaningful objects during the evaluation process, we introduce the concept of *containers*. In general, a container could be any data structure to hold geometric objects like a tree or a list. In our implementation, containers are multidimensional lists that are easy to handle, when fixed connection patterns have to be defined later on. Formally, containers are handled just as any other parameter in the grammar and can be passed to any rule or operator. Their definition can be understood analogously to the definition of a multidimensional array in C++: `name:type[]···[]`. Since our lists are dynamic, new elements can be pushed into a container by using `name.push(element)` or `name.push()` where the latter one automatically appends a new element with its default value. Furthermore, a convenient syntax to directly create a list with n entries is to use a sequence of elements in curly brackets: $\{e_1, \cdots, e_n\}$ (cf. Figure 6). Accessing an element of a container is also defined analogously to C++: $name[index_1]\cdots[index_n]$. Note that a default value will be returned, if any index is out of range.

5.2. Utilizing Stored Objects

Since we want to generate interconnections that are either rigid boxes or deformable freeform deformations, the most natural way to specify the end locations is done via the common 2D counterpart of the scopes which are rectangles. In this chapter, we will therefore address rectangle containers as a very important special case of general containers.

In order to make our system very flexible, we uncouple the actual operator that generates an interconnection from

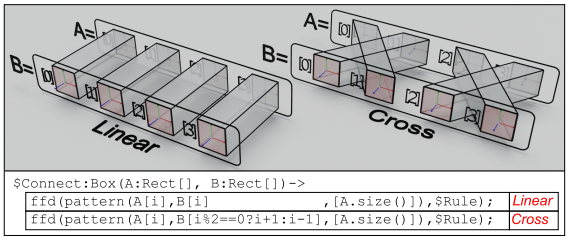

Figure 7: *Mappings are defined for the creation of fixed patterns which could end up in regular parallel or regular cross connections. In the grammar,* i *is an implicit operator attribute that returns the current value of the counter [KPK10]. Note, that* i *is the short version of* i[0]*, if only one counter is provided.*

the function that defines which rectangle instances have to be connected. In this chapter, we first explain two possible options that define the correspondences, i.e. a fixed pattern or geometrical queries. Formally, the functions return a pair containing the start and end location in form of a rectangle:

```
Pair := (source:Rect, target:Rect)
```

A simple visualization is achieved by spanning a FFD for each pair in a given pair container. The newly created non terminal FFDs will then be associated with the rule *next*:

```
ffd(correspondences:Pair[], next:Rule)
```

The second part of this chapter will then focus on two operators that create the complex interconnections by producing a deformable beam or a rigid chain.

5.2.1. Fixed Patterns

The first function simplifies the creation of non-trivial interconnected structures by defining *how* the elements of a container have to be connected:

```
Pair[] pattern(source:Rect, target:Rect, counters:Int[])
```

Just as with a one dimensional array, one can think of a loop that iterates over a container and for each source rectangle with index i, it determines a target rectangle based on the index i. Figure 7 shows two possible mappings between two containers A and B. The linear connection pattern is defined as a mapping $i \rightarrow i$ whereas the cross connection pattern is defined as a mapping $i \rightarrow (i\%2 == 0\ ?\ i+1 : i-1)$. We use the conditional operator from C++ that turned out to be very convenient for our purposes:

```
condition ? value if true : value if false
```

If the container is structured into more than one dimension and the connection pattern becomes more complex, it is useful to provide cascaded loops. Therefore, the third parameter of the operator is a list that defines the number of elements n_i for each of the m cascaded loops:

$$\texttt{for(i}_0\texttt{=0;i}_0\texttt{<n}_0\texttt{;i}_0\texttt{++)}\cdots\texttt{for(i}_{m-1}\texttt{=0;i}_{m-1}\texttt{<n}_{m-1}\texttt{;i}_{m-1}\texttt{++)\{\}}$$

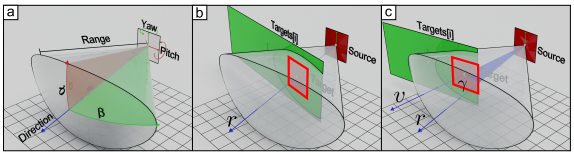

Figure 8: *Shooting a ray to find rectangles to connect with. (a) Defining an elliptical conical volume along the shooting direction to reduce the number of potential target rectangles. (b) The ray intersects a rectangle in the volume, so that the target rectangle (red) lies inside the hit rectangle (green) with its center at the intersection point. (c) If no rectangle is hit directly, among all other rectangles in the volume, the one closest to the ray is chosen with respect to the angle* γ*.*

In the sense of implicit operator attributes as they were presented by Krecklau et al. [KPK10], the first two parameters of the operator can now use the indices to define the actual mapping.

5.2.2. Geometrical Queries

The second function assumes, that a number of possible target rectangles have been defined by another procedural model. Conceptually, the operator takes a source rectangle B as a basis and shoots a ray $\vec{r}$ into some target direction to define a connection pair:

```
Pair shoot(source:Rect, targets:Rect[],
           yaw:Float, pitch:Float,
           alpha:Float, beta:Float, range:Float)
```

Figure 8 depicts the volume that is spanned to determine a target rectangle from the container. Basically, the shooting direction $\vec{r}$ is defined by a yaw and a pitch angle. This is an intuitive choice, because seen from the local coordinate system of B it describes the direction by turning left or right (yaw) and afterwards turning up or down (pitch). For each rectangle T in the target container, we determine the closest sub rectangle S that has the same size as B, the same orientation as T, and completely lies in T. Letting $\vec{v}$ be the vector pointing from the center of B to the center of S, we define γ to be the angle between $\vec{v}$ and $\vec{r}$ (cf. Figure 8.c). If the vector $\vec{v}$ is longer than a given maximum distance or it lies outside the elliptical cone that is spanned by α and β (cf. Figure 8.a), we skip the rectangle S. The connection will then be established from B to the sub rectangle S that has the smallest angular deviation γ from the shooting direction $\vec{r}$. If there are more than one possible S with the same deviation γ, we take the one with the shortest vector $\vec{v}$. Note, that our system could easily adopt other geometrical queries as a function returning the pair construct.

5.2.3. Deformable Beams

Spanning trilinear FFDs between two end rectangles is most of the time not sufficient for the creation of many real world examples, e.g. if a rope or cable has to be spanned between two end points. Therefore, our system extends the previous

definitions by a stiffness value of the material and a gravity value that creates a plausible sag of a deformable beam:

```
beam(correspondences:Pair[], next:Rule,
     stiffness:Float, gravity:Float,
     step:Float, threshold:Float)
```

The operator will automatically create the following five rectangles whose position, normal and size will be interpolated in the sense of Chapter 4. Based on the two end rectangles, two additional rectangles will be inserted by creating a copy of the end rectangles and moving the copy along their normal direction with respect to the defined stiffness value. The resulting tangents guarantee, that the orientation of the first and last rectangle of the interpolating spline are oriented in the same way as the given end rectangles. Additionally, a fifth rectangle will be inserted in the middle of the parameter domain. It will be moved down along the world y-axis with respect to the gravity value in order to simulate plausible sags that usually occur when cables or ropes are spanned between two points. Typical examples for such a scenario are power lines, cabin lift cables or other deformable supporting structures.

In order to let this operator behave analogously to the repeat operator of *CGA shape*, the curve has to be arc length parameterized. We approximate the arc length by recursively inserting sampling points. Whenever the distance between two successive samples lies above a certain threshold, new sample points will be inserted. Based on this piecewise linear approximation, the curve is then resampled uniformly with a step width given by the second last parameter, to create the cross sectional rectangles (cf. Figure 5).

5.2.4. Rigid Chains

In contrast to deformable beams, there is another class of real world examples that is better modeled with rigid interconnections, e.g. bridges, cranes and other complex steel constructions. We provide an operator that automatically creates a chain of rigid segments between the two end rectangles:

```
chain(correspondences:Pair[], segments:Segment[])
Segment := (type:Enum, value:Float, next:Rule, ) |
           (type:Enum, start:Float, min:Float,
            max:Float, next:Rule)
```

Every created segment can get another non-terminal symbol attached to it and could thereby be processed in another way. For the calculation of the rigid chain, we first transform the local coordinate system of the second rectangle into the local coordinate system of the first rectangle, so that we can build up the rigid chain from the origin. The segments are then defined by a sequence of transformations analogously to Chapter 3, where a fixed segment is defined by a 3-Tuple and a free segment is defined by a 5-Tuple. In both cases, one of the six types of transformation has to be chosen, which can be either a translation or a rotation along the local x-, y- or z-axis. The second parameter of a fixed segment is the scalar value that is used for the chosen transformation type. In contrast, for a free segment, the user has to specify a start value (commonly set to 0), a minimum value and a maximum value that is allowed for the optimization. By setting the minimum or the maximum to infinity, the user can disable the lower or upper bound of the joint, respectively. Note, that if it is not possible to generate a rigid chain between the end rectangles, the operator will skip the current pair.

5.2.5. Modeling Curves

The freeform modeling process of arbitrary spline curves is achieved by controlling the box scope in the usual fashion, i.e. applying transformations like a rotation or a translation to it, and at certain positions storing a face of the box in a rectangle container. Afterwards, the container can be passed to the curve operator which will automatically generate a sequence of FFD scopes:

```
curve(control:Rect[], closed:Bool, next:Rule,
      step:Float, threshold:Float)
```

The parameters for the step size and for the arc length threshold can be understood analogously to the creation of the deformable beams. In addition, a flag can be set to interprete the rectangle container as a closed curve and a non-terminal symbol is defined to further process the resulting freeform deformation scopes.

6. Use Cases

We present three different use cases including a bridge, a catenary in a street and a roller coaster (also shown in our accompanying video). The grammar snippets in this section focus on the structural part of the model design.

Sydney Harbour Bridge (cf. Figure 9) — In this example we reconstructed the structure of the Sydney Harbour Bridge (Fig. 9.e,9.f). A small grammar snippet is shown in the top right of Figure 9. First, the main steel girders are created by applying several splitting rules and transforming the parts to fit a quadratic curve (Fig. 9.a). During the evaluation, meaningful junctions are stored in three different containers for the side, the bottom and the top. Afterwards, a fixed pattern, colored purple in the grammar, is applied to create the interconnections of one layer (Fig. 9.c). In this case, the rigid chain is very simple, because the end locations are well aligned. Consequently, only two free rotations around the local y-axis are needed in combination with one free translational joint which is associated to a rule that will create the geometry of the interconnection. Note, that we use a two dimensional container here to store the side junctions of the two girders, i.e. the left side has index 0 whereas the right side has index 1. Several layers can then be connected in the same way (Fig. 9.d). Finally, the high details of the interconnections are modeled and inserted.

Catenary in a Street (cf. Figure 10) — For this example we first modeled different facades by the decomposition of boxes (Fig. 10.d). We extended the rules by providing a container of mounting areas that can be used by other objects. In our case, we modeled a catenary that shoots four rays for

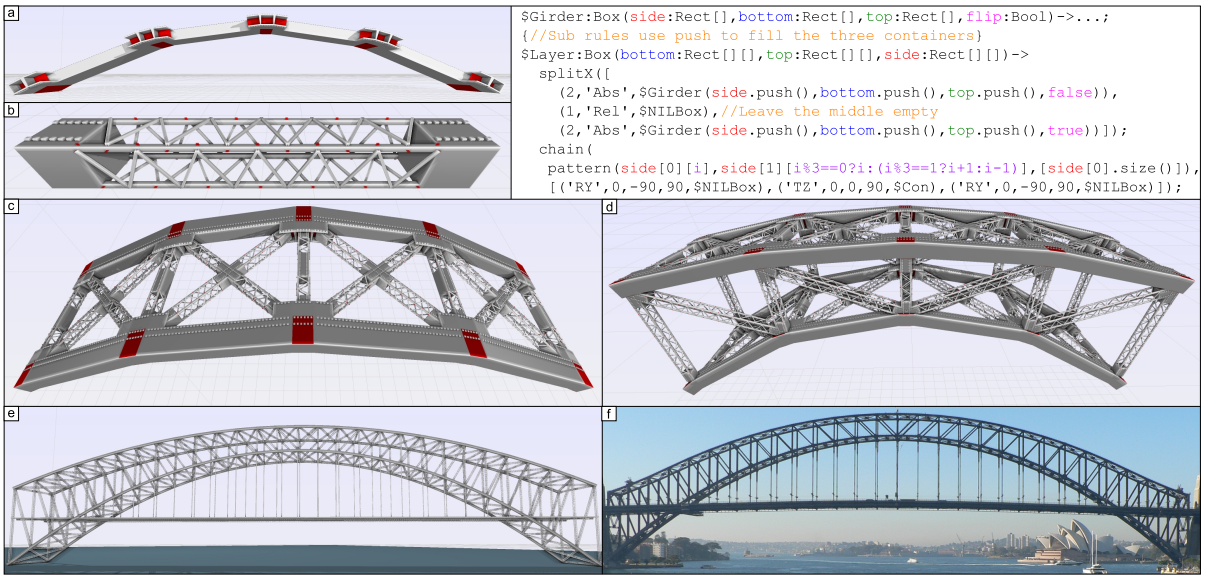

Figure 9: *Reconstruction of the Sydney Harbour Bridge (f). First, independent steel girders define connection points (a). Then, fixed connection patterns are applied to generate complex structures (c,d). Finally, the details are generated (b) which results in a virtual model of the bridge (e).*

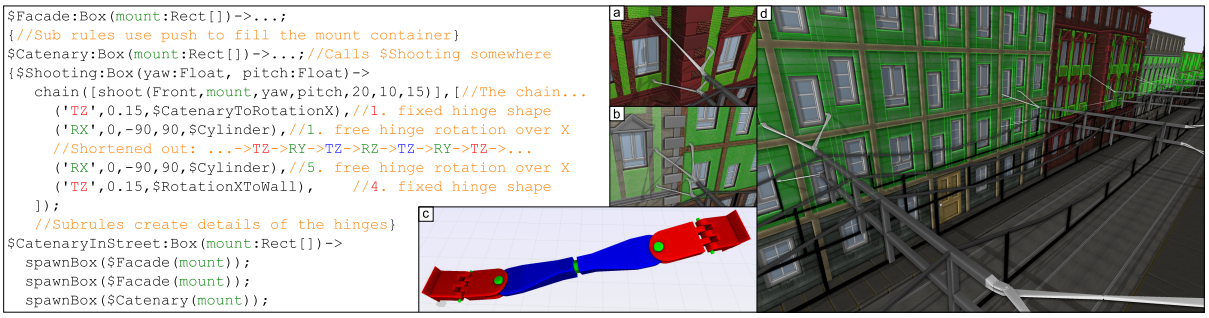

Figure 10: *Creation of a catenary that automatically connects with mounting areas on independently defined facades (a,b,d). The connections are based on rigid chains that only have to define the degrees of freedom (c).*

each side in direction of the facades (Fig. 10.a-10.b). In detail, every connection is created by a rigid chain of 11 elements. There are 4 fixed translational joints, colored red in the grammar, which are not optimized but needed to create the hinge geometry around the end locations (Fig 10.c, red). Furthermore, the 5 free rotational joints, colored green in the grammar, correspond to cylinder objects in the rigid chain geometry (Fig 10.c, green). Finally, there are 2 free translational joints, colored blue in the grammar, to bridge the distance (Fig 10.c, blue). The catenary itself uses the deformable beam principle to generate the bend of the hanging cables. The scene can now be easily composed by first creating several facades that fill the container of mounting areas and afterwards the caternary is created. Note, that the catenary is able to establish a connection to two neighboring facades which have been generated by two different rule sets (Fig. 10.b).

Roller Coaster (cf. Figure 11) — In our last example, we created a complex support structure of a roller coaster. First, we transformed the scope of a box several times in order to store a rectangle as the ground and to place control rectangles which result in the coarse layout of the basic roller coaster track by generating FFDs along a spline (Fig. 11.a). We use FFDs instead of generalized cylinders [PL96] in order to apply adaptive splitting for the generation of the details similar to *CGA Shape* (Fig. 11.c). During this process, the segments will also generate some avoidance areas that are queried in a condition [KPK10] when the red columns are created in the next step (Fig. 11.b, blue). The defined spline is sampled again with larger distances to apply the shooting operator which builds the basic poles from the track to the ground (Fig. 11.b). The newly created red interconnections provide several junctions in a cascaded container that is structured in 4 dimensions as depicted in Figures 11.d, 11.e.

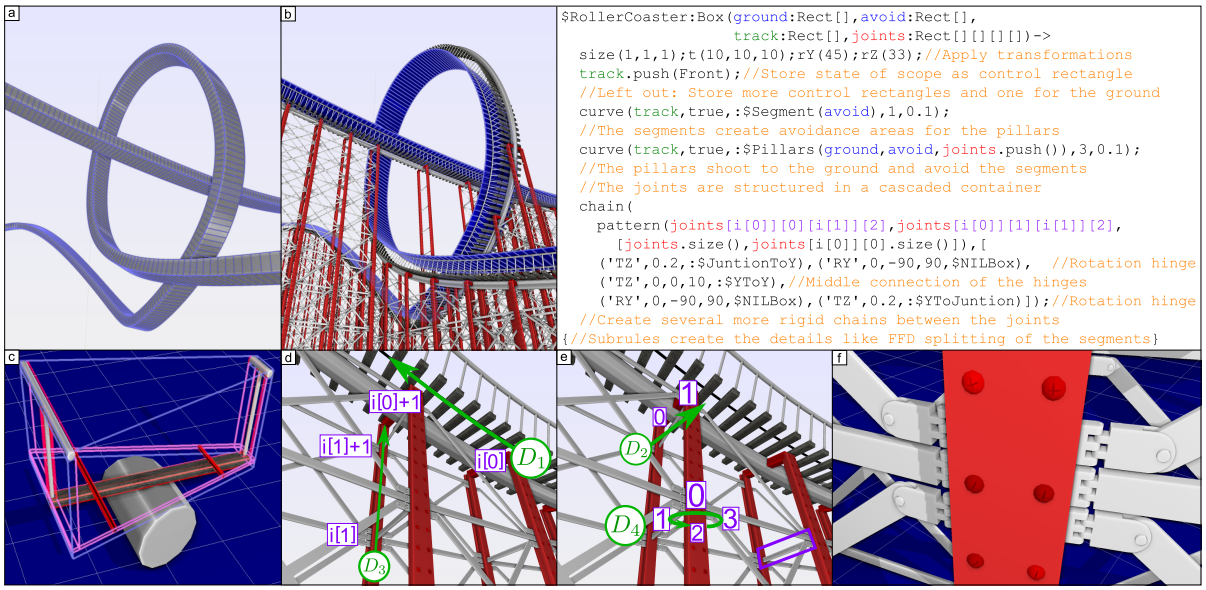

Figure 11: *Complex example of a roller coaster that is defined by a spline curve (a,b). The details are produced by splitting FFD cages (c). Complex connection patterns are defined from junctions that are stored in cascaded containers (d,e). A high amount of detail is reached by using many small IK systems for the creation of the supporting structure (f).*

D_1 runs along the track. D_2 has two elements expressing the right (0) and left (1) interconnections. D_3 runs along each interconnection. D_4 has four elements expressing the sides of the interconnection (front:0, right:1, back:2, left:3). One pattern, colored purple in the grammar, is highlighted in the bottom right of Figure 11.e. It is a two dimensional mapping that establishes a connection for all elements $i[0]$ along the track and all elements $i[1]$ along the current interconnection from the right (0), back (2) junction to the left (1), back (2) junction. With this well-structured container of the junctions a lot of different patterns can be established (Fig. 11.f) which finally results in a complex roller coaster model (Fig. 1).

7. Discussion

Implementation — Our application is implemented in C++ using OpenGL for the renderings as seen throughout this paper. The grammar is parsed into an internal data structure using boost spirit [Spi10]. We run our application on an Intel Core i7 with 2.67GHz with 6 GB ram and a GeForce GTX 470 with 1 GB. Since our application is not parallelized so far, we only utilize one of the cores. The complex grammar of Figure 11, which uses a lot of queries and IK systems, takes about 6 seconds for the evaluation resulting in 1.4M elements of the scenegraph with 6.8M triangles in total. The rendering of the scenegraph is done in 1.7 seconds since a lot of draw calls are needed. If we load the geometry to the graphics card once, which takes also about 1.7 seconds, the rendering can be done in real time with about 11 ms per frame. Shadowmapping and screen space ambient occlusion is used to enhance the visual appearance of the scene.

Commercial Software — Based on the procedural framework by Krecklau et al. [KPK10] our method can be easily integrated into professional modeling applications such as Houdini from Side Effects Software Inc. which heavily relies on a procedural methodology. The GenerativeComponents software from Bentley, which focuses on complex parameterized man-made structures, is another alternative that could adopt our presented approach. In a similar way to our modeling system, the geometry is saved by storing the algorithms that were applied after each other resulting in a very compact scene description. The system basically builds upon the idea of establishing relations between certain parameters in the object description. Rigid chains between two locations could be integrated as a script which is automatically invoked whenever one of the locations in the relation is changed. We further believe that our approach is a very useful extension for the CityEngine from Procedural Inc., since our approach already inherits the basic idea of producing the details by adaptive splitting of boxes or, more general, of freeform deformation cages.

Usability — Assuming that writing a shape grammar for previous approaches is already known [MWH*06, KPK10], containers are an easy to learn extension since they are similarly utilized as usual parameters. The structure of the grammar remains clear and human readable. This is especially important, if other designers have to understand a given grammar. Our presented operators give an intuitive option to define interconnected structures by just collecting the end locations, which would be hard to achieve with grammars that rely on the principle of decomposition or growth.

Fail Cases — If the defined rigid chain does not provide enough degrees of freedom to reach the target location, the IK system will fail and the resulting rigid chain will be ending in free space (but as close as possible to the target location). We regard such situations as design failures, which the user can easily resolve by providing further degrees of freedom in the definition of the rigid chain. Our accompanying video also shows a fail case during the creation process.

Limitations — Currently the containers are implemented as simple cascaded lists. Although this structure simplifies the creation of the structure patterns, applying queries on this data structure has a linear time complexity. Hence, it would be a great improvement to use data structures like octrees or kd-trees which would reduce the query complexity to logarithmic time. Grammars can already be created interactively for architecture [LWW08], but designing the interconnections with intuitive metaphors is not easy to achieve, since abstract structures like connection patterns have to be handled. For establishing the logical association between scene objects, a graphical node system could be used.

8. Conclusion

This paper proposes a novel method to describe 3D objects with complex interconnected structures. Procedural objects provide containers storing meaningful rectangles as possible end locations. Interconnections are then generated by applying a geometric query on the stored rectangles or by defining a pattern between elements of any containers. With the automatic generation of rigid chains or deformable beams between two end locations, a variety of different real world examples can be easily modeled such as bridges, catenaries or even complex roller coasters.

Acknowledgement

This work was supported in part by NRW State within the B-IT Research School.

References

[Bus04] BUSS S. R.: *Introduction to Inverse Kinematics with Jacobian Transpose, Pseudoinverse and Damped Least Squares methods*. Tech. rep., IEEE Journal of Robotics and Automation, 2004. 3

[EMP*02] EBERT D. S., MUSGRAVE K. F., PEACHEY D., PERLIN K., WORLEY S.: *Texturing & Modeling: A Procedural Approach, Third Edition*. Morgan Kaufmann, 2002. 1, 3

[GK07] GANSTER B., KLEIN R.: An integrated framework for procedural modeling. In *SCCG 2007* (Apr. 2007), Sbert M., (Ed.), Comenius University, Bratislava, pp. 150–157. 3

[GPMG10] GALIN E., PEYTAVIE A., MARÉCHAL N., GUÉRIN E.: Procedural generation of roads. *Computer Graphics Forum 29*, 2 (2010), 429–438. 3

[HBW06] HAHN E., BOSE P., WHITEHEAD A.: Persistent real-time building interior generation. In *Sandbox '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames* (NY, USA, 2006), ACM, pp. 179–186. 3

[KPK10] KRECKLAU L., PAVIC D., KOBBELT L.: Generalized use of non-terminal symbols for procedural modeling. *Computer Graphics Forum 29* (2010), 2291–2303. 3, 5, 6, 8, 9

[LG06] LARIVE M., GAILDRAT V.: Wall grammar for building generation. In *GRAPHITE '06* (NY, USA, 2006), ACM, pp. 429–437. 3

[LWW08] LIPP M., WONKA P., WIMMER M.: Interactive visual editing of grammars for procedural architecture. In *SIGGRAPH '08* (NY, USA, 2008), ACM, pp. 1–10. 10

[MP96] MĚCH R., PRUSINKIEWICZ P.: Visual models of plants interacting with their environment. In *SIGGRAPH '96* (NY, USA, 1996), ACM, pp. 397–410. 2

[MWH*06] MÜLLER P., WONKA P., HAEGLER S., ULMER A., GOOL L. V.: Procedural modeling of buildings. *ACM TOG 25*, 3 (2006), 614–623. 1, 2, 3, 9

[MWS05] MARK W. SPONG SETH HUTCHINSON M. V.: *Robot Modeling and Control*. Wiley, 2005. 3, 4

[PHHM97] PRUSINKIEWICZ P., HAMMEL M., HANAN J., MĚCH R.: Visual models of plant development. 535–597. 2

[PHL*09] PALUBICKI W., HOREL K., LONGAY S., RUNIONS A., LANE B., MĚCH R., PRUSINKIEWICZ P.: Self-organizing tree models for image synthesis. *ACM TOG 28*, 3 (2009). 1

[PHM00] PRUSINKIEWICZ P., HANAN J., MĚCH R.: An l-system-based plant modeling language. In *Applications of Graph Transformations with Industrial Relevance*, Lecture Notes in Computer Science. 2000, pp. 258–261. 2

[PJM94] PRUSINKIEWICZ P., JAMES M., MĚCH R.: Synthetic topiary. In *SIGGRAPH '94* (NY, USA, 1994), ACM, pp. 351–358. 2

[PKMH00] PRUSINKIEWICZ P., KARWOWSKI R., MECH R., HANAN J.: L-studio/cpfg: A software system for modeling plants. In *AGTIVE '99: Proceedings of the International Workshop on Applications of Graph Transformations with Industrial Relevance* (London, UK, 2000), Springer-Verlag, pp. 457–464. 2

[PL96] PRUSINKIEWICZ P., LINDENMAYER A.: *The algorithmic beauty of plants*. Springer-Verlag New York, Inc., NY, USA, 1996. 2, 8

[PM01] PARISH Y. I. H., MÜLLER P.: Procedural modeling of cities. In *SIGGRAPH '01* (NY, USA, 2001), ACM Press, pp. 301–308. 2

[PT97] PIEGL L., TILLER W.: *The NURBS book (2nd ed.)*. Springer-Verlag New York, Inc., NY, USA, 1997. 3, 5

[SP86] SEDERBERG T. W., PARRY S. R.: Free-form deformation of solid geometric models. *SIGGRAPH '86 20*, 4 (1986), 151–160. 3

[Spi10] Boost spirit. http://spirit.sourceforge.net/, Jan., 2010. 9

[Sti75] STINY G.: *Pictorial and Formal Aspects of Shape and Shape Grammars*. Birkhauser Verlag, Basel, 1975. 2

[Sti80] STINY G.: Introduction to shape and shape grammars. *Environment and Planning B 7* (1980), 343–361. 2

[WMV*08] WATSON B., MÜLLER P., VERYOVKA O., FULLER A., WONKA P., SEXTON C.: Procedural urban modeling in practice. *IEEE Computer Graphics and Applications 28*, 3 (2008), 18–26. 1

[WOD09] WHITING E., OCHSENDORF J., DURAND F.: Procedural modeling of structurally-sound masonry buildings. *ACM TOG 28*, 5 (2009), 112. 3

[WWSR03] WONKA P., WIMMER M., SILLION F., RIBARSKY W.: Instant architecture. *ACM TOG 22*, 3 (2003), 669–677. 2