CAD Shape Grammar: Procedural generation for Massive CAD Model

Wallas H. S. dos Santos
Pontifícia Universidade Católica
do Rio de Janeiro
Departamento de Informática
Rio de Janeiro - RJ, Brasil
Email: wallashss@tecgraf.puc-rio.br

Paulo Ivson
Pontifícia Universidade Católica
do Rio de Janeiro
Instituto Tecgraf
Rio de Janeiro - RJ, Brasil
Email: psantos@tecgraf.puc-rio.br

Alberto Barbosa Raposo
Pontifícia Universidade Católica
do Rio de Janeiro
Departamento de Informática
Rio de Janeiro - RJ, Brasil
Email: abraposo@inf.puc-rio.br

***Abstract*—This work presents a procedural modeling technique based on shape grammars for representing and rendering massive 3D CAD models in real time. Procedural modeling is an appealing approach to quickly generate massive scenes while maintaining compact representation. Until now, procedural modeling has not been explored in the domain of large industrial projects. Traditional procedural modeling techniques generate parameterized random scenes. In order to achieve an accurate representation for pre-existing 3D CAD scenes, we propose a specialized Shape grammar. We used common geometric primitives found in real 3D CAD models to create a compact model representation. In addition, we describe an efficient rendering algorithm to draw CAD Shape Grammars in real time. We evaluated both performance and memory consumption of our proposed technique using real-world CAD models. Results indicate not only high rendering performance, but a significant reduction in the memory required to represent massive 3D CAD models.**

I. Introduction

3D CAD models are important tools for planning, constructing, maintaining and operating large engineering projects. A high-precision and high-fidelity CAD model can be used as a source of engineering information and as an analysis and simulation tool. Examples include evaluating the viability of plans, minimizing costs through optimization procedures and checking for safety prior to any real operations. This work focuses on massive 3D CAD models of industrial plants, driven by the increasing demands of the Oil & Gas industry.

By definition, models are simplified representation of a real entity. Nevertheless, complex engineering projects require massive CAD models that become challenging to handle even by modern computers. Modern industrial plants contain in the order of tens of millions of geometric objects. Moreover, modeling an industrial plant like an entire oil refinery, is a time-consuming and error-prone task. Models with defects can negatively impact planning of real activities, leading to money expending and even safety risks. These challenges and demands motivate the research for new solutions for 3D CAD modeling and algorithms to handle data scalability.

Procedural modeling has been used to create 3D scenes where specific patterns are well defined. Repetitive patterns can be employed to represent buildings [1] and forests [2], while recursive growing patterns can be found in the topology of trees and fractals [3]. The benefits of this technique include generating convincing scenes while at the same time reducing human work to simply adjusting a few parameters. Another advantage is the reduction of memory consumption, since often the computational rule that express a given pattern is more compact than its explicit geometric representation.

Within this context, shape grammars are a popular technique due to its simplicity and power of expression. Even so, there is still a challenge to use it for other domains like general 3D CAD models. Typical shape grammars generate parameterized random scenes, while CAD models must represent real-world constructs as faithfully as possible. Nevertheless, we have observed that 3D CAD models of industrial plants exhibit distinct geometric patterns and repetitions. These have motivated our research on applying shape grammars to this new domain. In this case, the rules for model generation should guarantee correct geometric representation of several different CAD structures like piping, buildings and equipment.

This work proposes CAD shape grammar, a new approach to use procedural modeling for representing 3D CAD models. The use of a Shape Grammar on CAD is not straightforward. We address this problem by specializing a new grammar based on typical CAD geometries. In addition, we developed a rendering pipeline to efficiently draw 3D scenes represented by CAD shape grammars.

This works is organized as follows: Section II shows related work that motivated our research. Section III presents the original definition of shape grammar that we used as basis. Section IV presents our proposed shape grammar for massive CAD models. Section V describes the experiments used to evaluate our grammar. Section VI discusses the results of our work. Finally on Section VII presents conclusions and further work.

A. Contributions

This work demonstrates a procedural modeling technique that uses shape grammars to represent 3D CAD models of industrial plants. Our contributions can be summarized as follows:

Procedural modeling for massive 3D CAD models: we were not able to find any previous work that uses procedural modeling techniques for massive CAD models. This work

2377-5416/17 $31.00 © 2017 IEEE
DOI 10.1109/SIBGRAPI.2017.11

demonstrates its feasibility and evaluates the benefits of applying shape grammars to this new domain.

CAD Shape Grammar: we created a purpose-built shape grammar for representing 3D CAD models. Common geometric primitives found in real models can be used to procedurally model a vast array of scenes. This representation has memory requirements equal or less than a typical mesh representation.

Optimized rendering: we developed a rendering pipeline optimized for processing our shape grammar data structure. This procedure can reach high rendering rates even for massive CAD scenes.

II. Related Work

1) Massive Model: A typical approach to describe 3D scenes is a *scene graph* [4] [5]. The concept of a scene graph is widely used on 3D game engines, simulators and other modeling programs. In general, the data structure is a tree of objects, where algebraic transformations and state (color, texture) are shared from parent to descendant nodes. The hierarchical structure also supports branch culling if we know a priori if a parent is outside of view frustum, thus efficiently discarding the processing of entire subtrees. Therefore, many typical 3D scenes can be suitably represented by a scene graph.

On the other hand, scene graphs are not a scalable solution for massive 3D CAD models. Industrial CAD models can contain thousands objects, and processing the whole graph for rendering can be a bottleneck to a naive renderer implementation. Another disadvantage of scene graphs is the lack of reuse of common geometric layouts. Equipment, pipings and buildings are a combination of small objects that share common templates. During modeling phase, if the template is changed, a badly designed system would oblige user to search and change manually every derived instance.

For these reasons, many precious research have explored alternative means to efficiently represent and render massive 3D CAD models in real time. Peng et. al. developed an out-of-core approach to render massive models using the GPU [6]. The method consists of a mesh simplification algorithm implemented inside the GPU and mesh data transfers from CPU to GPU according to the camera viewpoint. The mesh data exchange exploits frame-to-frame coherence since objects from a given frame have high probability to be present on the next one. The meshes are simplified on the GPU with information to reformulate the triangles and recover the original geometry. The reformulation is defined by the LOD of the object in a any given instance. The main drawback of this method is that fast camera movements can lead to frames with missing objects, since the system may not have enough time to transfer newly visible meshes to the GPU.

Santos et. al. presented a method to render massive models based on instancing of repetitive objects [7]. First the model is preprocessed in order to find redundant meshes using *shape matching*, which tries to find an optimal affine transformation between vertices of different objects. The algorithm significantly reduces memory footprint and improves the model rendering speed, once multiple objects share the same mesh only differing by an algebraic transformation. Besides these positive results, the authors admitted lack of visibility culling and LOD support to improve performance and scalability.

Xue et al proposed [8] a framework to render massive models using voxel representations and out-of-core algorithms. The voxelization is also used to generate shadows in order to increase scene realism. To handle large scenes, the authors proposed a method to compress the data to be transferred from CPU to GPU. Vertex and triangle data are quantized and compressed, while normals are deleted. Once the data of a given frame is available inside the GPU, triangle data are decompressed by a vertex shader and normals are recalculated generated using a geometry shader. The authors reported good performance of the developed prototype, however the voxelization process needed to load the scene was very time consuming. Moreover, the voxelization and data compression decreased the objects' visual quality.

2) Procedural modeling: G. Stiny introduced the shape grammar formalism in 1980 [9]. Later, Müller et. al. introduced CGA shape grammars to procedurally model buildings [1], which contain basic operators that modify a mass model and generates various types of buildings architecture. Section III discusses more details of this approach.

Procedural modeling has been used on various domains in order to reduce human work and render urban environment scenes [1], [10], [11], [12], [13]. By using shape grammars and an L-system as data structure, other works explored the scene generation directly on the GPU [2], [14], [15], [16]. These approaches were capable of achieving real-time rendering of massive 3D scenes.

These works mostly used shape grammars or L-systems to explore repetition patterns adjusted by user-provided parameters. However, most solutions are domain-specific and can not be used as is for 3D CAD models. Random 3D CAD models are not suitable to represent industrial projects since every object must accurately correspond a real physical entity. The main intention of our shape grammar is to ease the modeling process of 3D CAD models, making this task more efficient and storing the results in a data structure with high scalability. The generation rules used to control the procedural modeling define project constraints to obtain a well-formed design.

Similarly, Krecklau et. al. proposed the language G^2 (*Generalized Grammar*) as a tool to procedurally model scenes of different types of domain [17]. The language can express buildings, normally done by shape grammars, and plants by L-systems. The expression was increased by applying free-form deformation on non terminal symbols. The authors report the method can be used on real-time applications, although it is not adequate for massive models 3D CAD model.

III. Shape grammar

A shape grammar is made of a set of symbols (terminal and non terminal) and a set of production rules. The production rule is in the form of $L \rightarrow R$, where L is a single symbol on the left side called predecessor, a non terminal symbol, and R, on the right side, is called successor, which is composed

of one or more terminal or non terminal symbols. An initial symbol is required to start the derivation, and derivation must end when it generates only terminal symbols.

The production rules can be parametric. The parameters can be used in arithmetic expressions to derive successors. Also, the production rules can be conditional, *i.e.* multiple production rules with the same predecessor symbol. In this case, the condition that fits to the current context determines which rule to generate.

There are two special symbols that are useful to save the current scope and make derivation branches. These symbols are the *push* and *pop* denoted by "[" and "]" respectively. When a push occurs, the current scope is saved and can be freely changed by any operation, until a pop operation restores the previous scope.

In this work we built a procedural modeling based on CGA shape grammar proposed by Müller et. al. [1]. The terminal symbols generated by the derivation of the shape grammar are interpreted similarly to a turtle interpreter based on LOGO, as in L-systems [3]. Every symbol is processed and generates an object or changes the state of the interpreter. The state of the interpreter is called *scope*. It represents an oriented box and can be transformed by operations like scale ($S(x, y, z)$), translation ($T(x, y, z)$), and rotation ($R(x, y, z)$). CGA shape grammars also introduced special operators to help generate buildings: *Split* and *Repeat*, these operators will subdivide the scope in smaller scopes and are useful to express subdivisions and repetition patterns. Finally, the operation *instance* ($I("instance\_name")$) generates an object inside the scope by applying the scope transformation.

Next, we describe a typical shape grammar example with the basic operations. The production rule $axiom$ is the first production rule to be evaluated, its successors are other production rules. The production rules A, B, C, and D illustrate scope operations, E and F split are repeat operations and $cube$, $cylinder$ and $sphere$ represent geometries and colors to instantiate objects during scene generation. Figure 1 shows the output generated by the shape grammar example.

```
axiom -> A B C D E F
A -> cube
B -> T(2,0,0) cylinder
C -> T(2,0,0) R(45,45,0) cylinder
D -> T(2,0,0) S(1.5,1.5,1.5) cylinder
E -> T(2.5,0,0) Split("Z",0.25,0.25,0.25,0.25)
     {cube cylinder cube cylinder}
F -> T(2.5,0,0) Repeat("XYZ", 27){ sphere }
cube -> C(1, 0, 0)I("cube")
cylinder -> C(0, 1, 0)I("cylinder")
sphere -> C(0, 0, 1)I("sphere")
```

IV. CAD Shape Grammar

This Section describes our proposed CAD Shape Grammar and an optimized renderer for it.

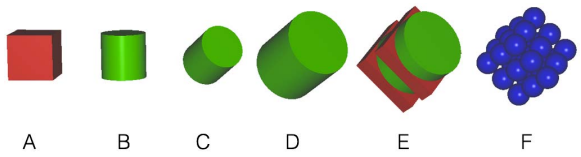

Fig. 1. Render output of an example shape grammar.

A. Grammar definition

Our shape grammar overcomes the problem of geometry scalability by specializing object instances. We call these object instances as *parametric objects*. A large portion of a 3D CAD scene can be expressed solely by these parametric objects, as shown on Figure 2. We extended the shape grammar described on the previous section to fit the CAD domain. The operations Repeat and Split remain as the original definition.

1) Scope: Scope operations generate a transformation for a 3D object. We define scope operation as two types: relative and absolute. Absolute operations receive absolute parameters: current scope will not influence the output of the transformation. For example, we can set the translation to a specific position independently of the current scope while the scale and rotation remain the same. Relative operation will transform the current scope by appending transformations. The summary of the Scope operation is as follows:

- $T(x, y, z)$ *Relative translation*: translate the current scope by (x, y, z).
- $M(x, y, z)$ *Absolute translation*: translate the scope to coordinates (x, y, z).
- $R(x, y, z)$ *Relative rotation*: rotates the current scope by (x, y, z) in Euler angles in *yaw pitch roll* convention.
- $G(x, y, z)$ *Absolute rotation*: set the rotation to (x, y, z) in Euler angles in *yaw pitch roll* convention.
- $S(x, y, z)$ *Relative scale*: scales the current scope by dimensions (x, y, z).
- $E(x, y, z)$ *Absolute scale*: set the scale by dimensions (x, y, z).

Absolute scope operations are useful to transform a single object without the influence of any previous generation. In certain cases it is not possible to take advantage of the procedural modeling expression. Hence, the absolute operation will just instantiate the 3D object at its particular algebraic transformation on the scene.

2) Instance: The operator instance ($I(\text{“}instance\_name\text{”})$) is used to generate an object of label “$instance\_name$”. Typically, the instance name leads to a mesh to be rendered in the current scope. In order to reduce the memory footprint for massive models it is interesting to define built-in primitives to be reused. For CAD models there are several types of common primitives useful to represent the 3D scene. Therefore, our grammar incorporates these specialized primitives as built-in objects. The specialized primitive types of our grammar are as follow:

- *Cube*
- *Cylinder*

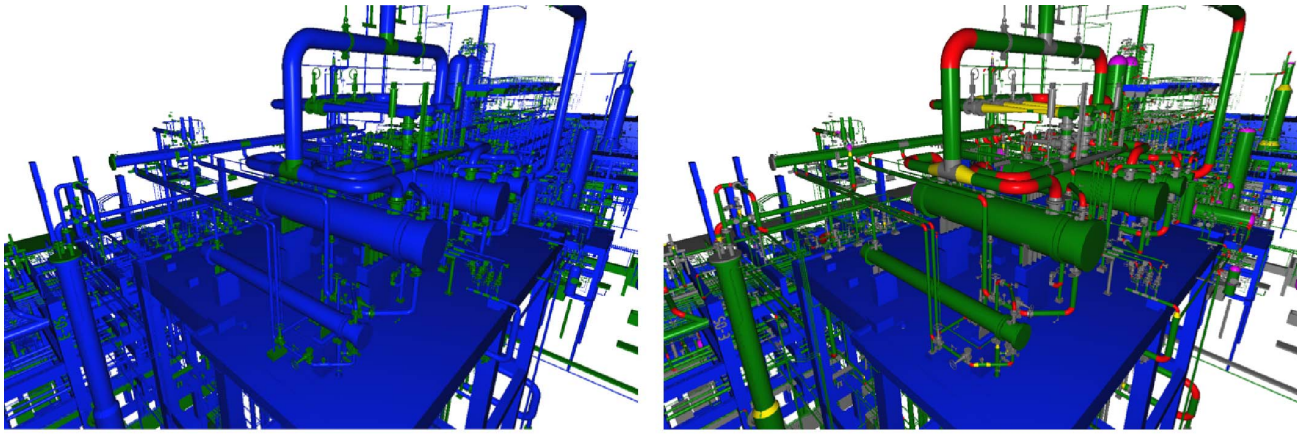

Fig. 2. Left image shows the distribution between parametric objects (blue) and meshes (green). Right image shows primitives of the same type with same color: blue for box, green for cylinders, yellow for cones, red for torus, magenta for dish, and gray for meshes.

- *Cone*
- *Sphere*
- *Dish*
- *Torus*

The primitives cube, cylinder, sphere and dish are what we call *normalized primitives*. These primitives do not have explicit parameters, because we can obtain any variation by applying only scope transformations. For example, we can generate a box with dimensions (10, 1, 1) by applying a scale $S(10, 1, 1)$ and invoke $I(\text{“}cube\text{”})$. By the other hand, cone and torus must have explicit parameters. The cone has bottom radius, top radius, x offset and y offset. The torus has sweep angle, inner radius and outer radius. To maintain these primitives normalized, the parameters must be relative to the scope. These objects can be generated by invoking instance operator as follows:

$$I(\text{”}torus\ sweep\_angle\ inner\_radius\ outer\_radius\text{”}) \quad (1)$$

$$I(\text{”}cone\ bottom\_radius\ top\_radius\ offset\_x\ offset\_y\text{”}) \quad (2)$$

3) Color: In typical 3D CAD models, the colors contain semantic information that identify specific types of components. We can add the color property to the scope as $C(r, g, b)$ or $C(c)$, where r (red), g (green), b (blue) are the color components or c is a 32-bit integer color ID.

B. Optimized Renderer

This section describes the developed rendering algorithm to efficiently draw a 3D scene represented by a CAD shape grammar. To handle large scenes, we optimized the rendering of our specialized primitives. In this pipeline we were able to send minimal data to generate a 3D geometry and use traditional optimization techniques like view-frustum culling and level-of-detail.

Interpretation. First, the shape grammar is interpreted on the CPU side as described in Section III. From the axiom, production rules generate symbols until only terminal symbols remain. Afterwards, the terminal symbols are processed to generate primitive objects or generic meshes by invoking instancing operations. There is a buffer for each type of object that contains the instance data, and it is drawn by binding the respective shader programs for the specific type. Repeated generic meshes can be rendered using geometry instancing, while unique meshes are appended to the buffer and are drawn as usual. The instance data are: scope (a float matrix 4x3), solid color (3 floats) and, to the torus and cone, their respective additional parameters (a float for each parameter). Figure 5 illustrates the buffer layout to be sent to GPU.

Parametric surface. The specialized primitives of our grammar can be rendered as parametric surfaces. We use tessellation shader programs available on modern GPUs to generate a regular grid and deform it as a parametric surface, see Figure 4. For every instance we can use only the basic information to generate the triangles by the shader, reducing memory and processing. After the tessellation performed by the GPU hardware, the tessellation evaluation shader receives the grid and deform it applying parametric equations of the respective surface. Figure 3 shows the overview of the rendering pipeline for these primitives.

LOD and Frustum culling. In our CAD shape grammar, the scope is analogous to a model matrix from a traditional rendering pipeline. However, unlike a typical model matrix, the scope must represent the bounding box of the current instance. For example, the mesh may already be transformed to world space and its corresponding model matrix would be an identity matrix. Therefore, if we ensure the scope is the oriented bounding box of the instance, we can estimate where the object will be placed to cull it or choose the proper level of detail for rendering.

The shader performs view-frustum culling by checking the scope against the current viewpoint and setting an empty or

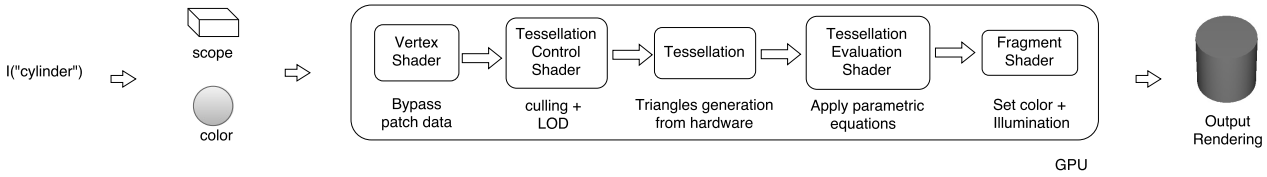

Fig. 3. Rendering pipeline of CAD shape grammars. It starts by the invocation of an instance and by getting the current scope. The data is passed to the GPU where each specialized shader will process the parameters and draw the instantiated primitive.

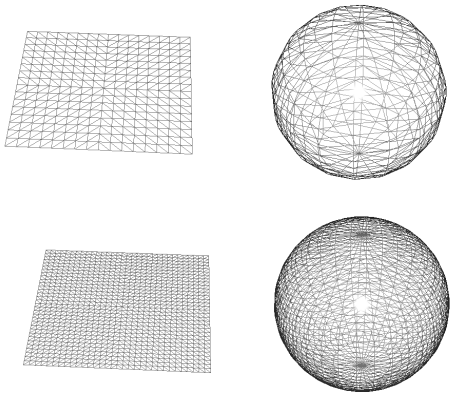

Fig. 4. Base grids and generated spheres by deforming it using parametric equations. On the top a coarser representation of a sphere and on the bottom a smooth version.

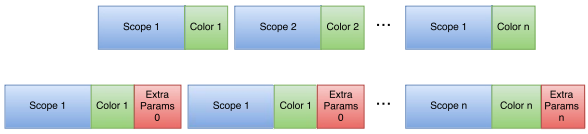

Fig. 5. Layout of the buffer to send to GPU to render parametric surfaces. Above the buffer for normalized primitives and below for primitives that need additional parameters.

null tessellation. If the object is not culled, then the shader sets the tessellation level based on the object's size on screen. Figure 4 shows a parametric surface with different levels of detail and their respective base grids generated by the tessellation shader.

V. Results

We tested our CAD shape grammar by converting pre-existing 3D CAD models and re-modeling common structures present on them. All tests were executed in a desktop PC with an Intel Core i7 2.93 GHz Quad Core processor, 6GB of RAM and an Geforce GTX 770 graphics card running Windows 10 as operating system.

A. Modeling

To evaluate our proposed shape grammar, we explored the features that it offers to generate some types of recurring structures in large industrial projects.

Tank reservoir. The first shape grammar example models a tank reservoir. We used the split operation to divide the scope based on the specified length parameter of the rule. In this example, the parameter of split operation with suffix "r" will be relative to the amount of scope in axis Z left from the other absolute parameters. The front and the back will always have the scope size fixed. The length can be estimated by the volume of liquid desired to storage in this model representation, then the parameter can have a semantic relation to the project specifications. Figure 6 shows the output generated by this shape grammar.

```
tank(length) ->
E(2.3,2.3,length+1)
Split("Z",0.5,1r,0.5){front body back}

body -> I("cylinder")

front ->
R(0,180,0)I("dish")
front_connector
[support][bottom_connector]

back ->
I("dish")
[support] bottom_connector

front_connector ->
T(0,0,0.25)E(0.6,0.6,0.25)I("cylinder")
T(0,0,0.125)E(1,1,0.1)I("cylinder")

bottom_connector ->
T(0,1.3,-1.3)R(90,0,0)E(1,1,0.1)I("cylinder")
T(0,0,0.17)E(0.7,0.7,0.25)I("cylinder")

support ->
T(0,0.95,-2.1)E(2.1,0.1,0.8)R(90, 0, 0)
I("support_mesh")
```

Stairs. The next shape grammar generates a staircase with lifeline. In this case we explore the repeat rule that generates the desired quantity of equally spaced steps. Based on the step count the rules also resize the cylinders to be placed as the stairs' lifeline. Figure 7 shows the output generated with several different parameters.

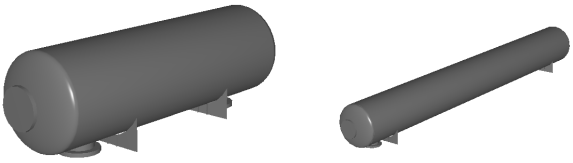

Fig. 6. Output generated by production rule $tank$. On the left with $length = 8$ and on the right $length = 20$. Note that both objects have the same cylinder radius, the size difference is due to different viewing perspectives.

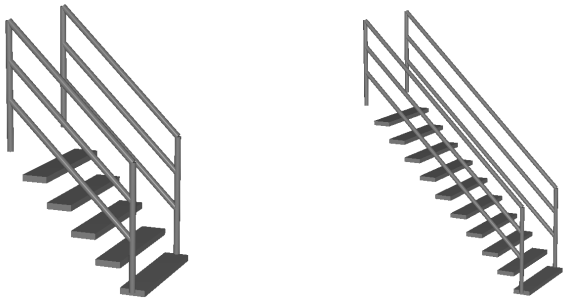

Fig. 7. Output generated by production rule $stairs$. On the top with $step\_count = 5$; bottom left with $step\_count = 10$; bottom right with $step\_count = 100$.

```
stairs(step_count) ->
lifelines(step_count)
E(20, 4, 1) Repeat("", step_count){step}

step -> I("cube")T(0, 4, 4)

lifelines(step_count)->
[T( 10, 0, 0) lifeline(step_count)]
[T(-10, 0, 0) lifeline(step_count)]

lifeline(step_count) ->
E(1, 1, 20)
base_lifeline(step_count)
lateral_lifeline(step_count)

base_lifeline(step_count) ->
[T(0, 0, 10)I("cylinder")]
[T(0, step_count*4, step_count*4+10)
 I("cylinder")]

lateral_lifeline(step_count) ->
S(1, 1, 0.2828*step_count)
T(0, step_count*4*0.5, step_count*4*0.5+20)
[R(-45, 0,0)I("cylinder")]
[T(0, 0, -6)R(-45, 0,0)I("cylinder")]
[T(0, 0, -12)R(-45, 0,0)I("cylinder")]
```

B. Memory footprint and Rendering

To evaluate the performance of the renderer and memory footprint for the generated scene of our grammar we converted three existing 3D CAD models. The models are tagged as *small*, *medium* and *large*. Figure 8 shows the evaluated models.

Table I shows object counts by type. The *small* model has more generic meshes than the primitive ones. In contrast, the *medium* and *large* models have more primitive types than meshes, around 69% and 79%. These specialized primitives allowed for a reduced the memory footprint: in Table II we can see that the *large* model originally have 4.85GB of data, and represented as parametric primitives the memory footprint dropped to 499MB, a reduction to 10%.

In Table II the interpretation time is the time to generate the scene after parsing of grammar. The results indicate that all models take less than a second to be interpreted. The FPS was measured rendering the whole scene at once. The *large* model is rendered at 22 FPS, which is equivalent to drawing a scene with over 900K objects and 145M triangles.

Object type	Small	Medium	Large
Box	22,475	176,578	248,276
Cylinders	14,924	221,377	389,589
Dishes	984	3,986	6,858
Cones	4,704	23,825	38,897
Spheres	0	600	2,706
Torus	3,109	22,395	53,931
Meshes	115,206	193,914	192,753
Total Parametric	46,196	448,761	740,257
Total	158,293	642,675	933,010

TABLE I
COUNT OF OBJECTS PER TYPE.

VI. DISCUSSION

modeling. Procedural modeling techniques are commonly used to generate well defined patterns. Typical 3D CAD models contain variety of patterns that can be exploited by procedural modeling to generate the 3D scene. However, sometimes it is not possible or worthwhile to create rules to generate specific objects. To overcome this problem, our solution uses absolute scope operations. With this approach, the user can place an object anywhere without extra memory consumption. We present the following example of an absolute operation:

```
A -> M(10,0,0)E(2,5,1)G(0,0,90)I("cube")
```

An advantage in using procedural modeling for industrial CAD models that the procedural formalism can be used to represent industrial project specifications. The user can define a set of rules to be a template of common structure types, so the grammar promotes reuse in order to gain productivity on modeling. Moreover, the rules can add semantic and constraints to the modeling. For example, we created rules to generates stairs, and we defined the distance between each step. These constraints can be defined by engineering specifications such as safety or cost minimization. By using shape grammars, the user can express these specifications intrinsically during the modeling workflow. In addition, changing structure templates is both easy and fast: since the rules are reused, a change in project specification would be automatically propagated for every instance. For example, if we have a systematic repetition pattern of a box, and there is a need to change it to a cylinder, it can be done by just replacing the box instance by cylinder instance in the production rule that is being repeated.

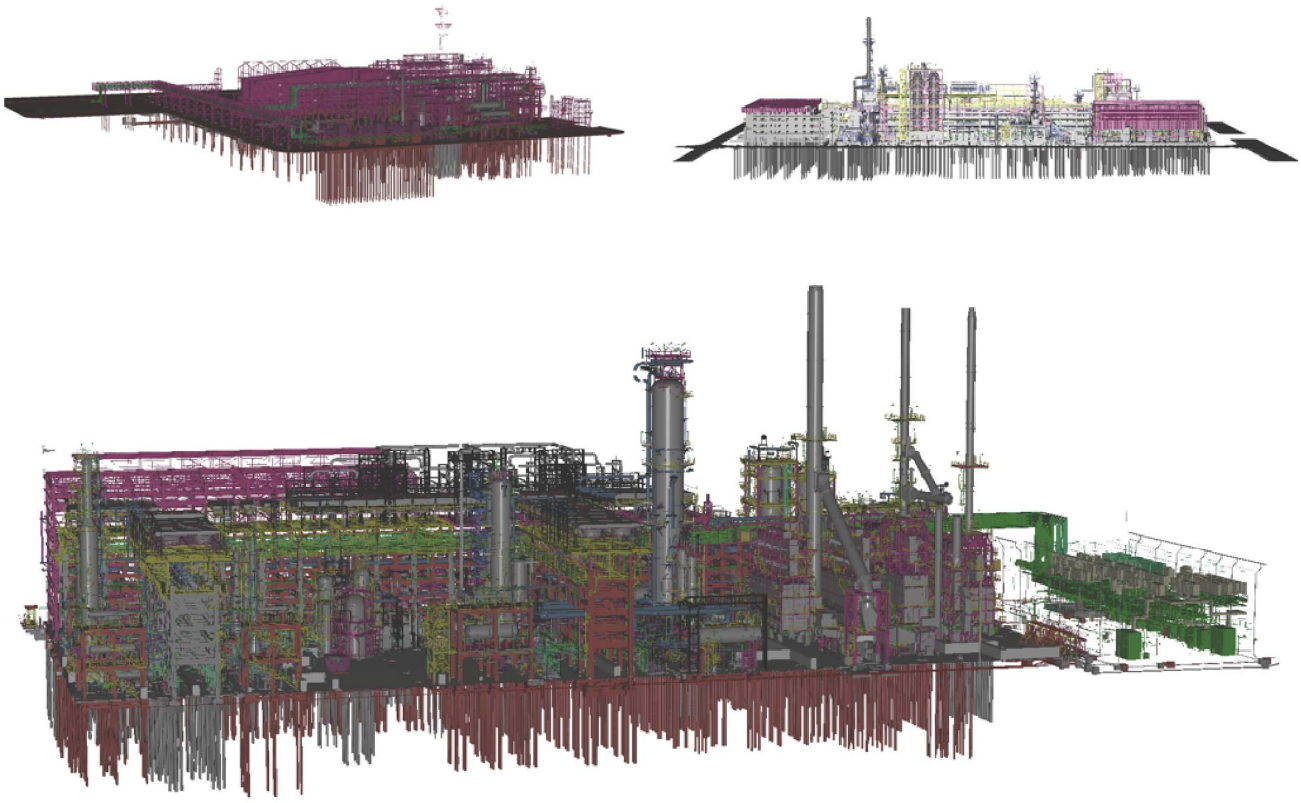

Fig. 8. 3D CAD scenes generated by the proposed CAD shape grammar. On the top left the *small* model with 159K objects and 11M triangles, on the top right the *medium* model with 641K objects and 74M triangles. On bottom the *large* model, it contains over 900K objects and is rendered at 22 FPS

Model	Triangles	Objects	Parametric	Memory Full	Reduced Memory	Interpretation	Render
Small	11M	158K	43K	455MB	196MB	127ms	154 FPS
Medium	74M	641K	447K	2.56GB	293MB	422ms	35 FPS
Large	145M	933K	736K	4.85GB	499MB	673ms	22 FPS

TABLE II
RESULTS OF OUR PROPOSED TECHNIQUE: REDUCED MEMORY FOOTPRINT FOR EACH MODEL, INTERPRETATION TIMES IN MS AND RENDERING PERFORMANCE MEASURED IN FPS.

In summary, the CAD shape grammar can help to specify industrial CAD models following accurate design guidelines.

Memory footprint. In Figure 2 we show how the basic primitives are very often present in industrial CAD models. These primitives can represent CAD components in a very compact form: essentially the scope and in some cases a few additional parameters. Additionally, for generic meshes we can also reduce memory by grouping multiple object instances that share the same triangle mesh in a similar way how is done in [7]. Since our grammar promotes reuse of rules, generic meshes are drawn instanced, where each instance differ only by the scope.

Rendering performance. By taking the benefit of specialized primitives we can efficiently generate triangle meshes in the GPU using tessellation shaders. We evaluated the performance of the renderer using real massive models and found it suitable for interactive 3D applications. Also it is possible to extend the renderer to support another types of parametric surfaces. LOD and culling would be easily implemented by following the ideas in Section III.

Visual quality. Most CAD systems draw surface objects with a "good enough" discretization, and sometimes contours do not accurately match the corresponding physical entities. Since we use parametric objects and control their LOD, we can also setup the renderer to tessellate the primitive objects to obtain very smooth surfaces.

Limitations. The first obvious limitation is writing shape grammar rules to procedurally generate 3D models. A first look on a shape grammar text file can be not very engaging to a user. Ideally, the user that will create the rules should have a basic programming background. A good practice is to write production rules naming successors with intuitive description

to facility changes and "debugging". However, once the rules are well formed, the modeling workflow could be very efficient by just reusing existing parametric rules. Nevertheless, a user-friendly abstraction layer such as well-designed modeling graphics interface could facilitate the creation and edition of the CAD shape grammar.

To improve rendering performance the model should have as many parametric objects as possible. In an ideal scenario, the modeling should start using our proposed grammar. However, in some cases, to maintain interoperability with other system, a conversion must be performed. Many design systems already contain similar basic objects but their usage depends on the project modeler. Possible solutions to this problem are trying to convert the internal format of a similar primitive to our grammar or reverse engineering triangle meshes to our primitive objects.

Another limitation is how we handle generic meshes. We did not implemented LOD or frustum culling in order to improve their rendering performance. However, since most massive data of the models can be represented by the primitive objects, the absence of these feature did not harm substantially the interactivity of the scene.

VII. Conclusion

This paper proposes a new shape grammar capable of representing and rendering 3D CAD models of industrial projects. Until now, procedural generation has not be applied on the modeling of massive industrial engineering projects. Using procedural generation brings many advantages to improve the productivity of designing massive CAD projects. We found a strong correlation between procedural rules and engineering specifications, which can be intrinsically represented by our shape grammar. We also identified common objects to this domain and incorporated them in our grammar for reduced memory consumption and efficient rendering. The optimized rendering results demonstrate that is possible to efficiently navigate massive 3D CAD scenes generated by our proposed grammar.

As future work we suggest implementing grammar generation fully inside GPU as in [2], [14], [15], [16]. A compact shape grammar could be transferred from the CPU to GPU to reduce overhead and allow editing on the fly. We also suggest investigating alternative methods to modeling some types of structures to improve memory consumption and/or human readability. The L-system approach could be used to generate piping or similar structures, which are very common on industrial CAD models.

Another interesting contribution to the grammar is incorporating new primitives. We showed that the built-in objects help reducing memory footprint and improving visual quality. NURBS is a strong candidate to be the next primitive to be added to the grammar vocabulary. Its incorporation would fit well within our rendering pipeline and would take naturally take advantage of LOD and frustum culling.

VIII. Acknowledgments

We would like to thank CNPq, Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brasil for the financial supporting of this work. We also wish to thank Petrobras for providing the 3D CAD models and for supporting scientific research and development in its partnership with Tecgraf Institute at PUC- Rio. We thank the reviewers for their helpful comments and suggestions.

References

[1] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool, "Procedural modeling of buildings," in *Acm Transactions On Graphics (Tog)*, vol. 25, no. 3. ACM, 2006, pp. 614–623.

[2] M. Lipp, P. Wonka, and M. Wimmer, "Parallel generation of multiple l-systems," *Computers & Graphics*, vol. 34, no. 5, pp. 585–593, 2010.

[3] P. Prusinkiewicz and A. Lindenmayer, *The algorithmic beauty of plants*. Springer Science & Business Media, 2012.

[4] R. Osfield, D. Burns *et al.*, "Open scene graph," *Library–OSG. http://www. openscenegraph. org*, 2004.

[5] D. Brutzman and L. Daly, *X3D: extensible 3D graphics for Web authors*. Morgan Kaufmann, 2010.

[6] C. Peng and Y. Cao, "A gpu-based approach for massive model rendering with frame-to-frame coherence," in *Computer Graphics Forum*, vol. 31, no. 2pt2. Wiley Online Library, 2012, pp. 393–402.

[7] P. I. N. Santos and W. Celes Filho, "Instanced rendering of massive cad models using shape matching," in *Graphics, Patterns and Images (SIBGRAPI), 2014 27th SIBGRAPI Conference on*. IEEE, 2014, pp. 335–342.

[8] J. Xue, G. Zhao, and W. Xiao, "Efficient gpu out-of-core visualization of large-scale cad models with voxel representations," *Advances in Engineering Software*, vol. 99, pp. 73–80, 2016.

[9] G. Stiny, "Introduction to shape and shape grammars," *Environment and planning B*, vol. 7, no. 3, pp. 343–351, 1980.

[10] Y. I. Parish and P. Müller, "Procedural modeling of cities," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. ACM, 2001, pp. 301–308.

[11] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky, *Instant architecture*. ACM, 2003, vol. 22, no. 3.

[12] P. Merrell, E. Schkufza, and V. Koltun, "Computer-generated residential building layouts," in *ACM Transactions on Graphics (TOG)*, vol. 29, no. 6. ACM, 2010, p. 181.

[13] F. Bao, M. Schwarz, and P. Wonka, "Procedural facade variations from a single layout," *ACM Transactions on Graphics (TOG)*, vol. 32, no. 1, p. 8, 2013.

[14] J.-E. Marvie, C. Buron, P. Gautron, P. Hirtzlin, and G. Sourimant, "Gpu shape grammars," in *Computer Graphics Forum*, vol. 31, no. 7. Wiley Online Library, 2012, pp. 2087–2095.

[15] M. Steinberger, M. Kenzel, B. Kainz, P. Wonka, and D. Schmalstieg, "On-the-fly generation and rendering of infinite cities on the gpu," in *Computer graphics forum*, vol. 33, no. 2. Wiley Online Library, 2014, pp. 105–114.

[16] M. Steinberger, M. Kenzel, B. Kainz, J. Müller, W. Peter, and D. Schmalstieg, "Parallel generation of architecture on the gpu," in *Computer graphics forum*, vol. 33, no. 2. Wiley Online Library, 2014, pp. 73–82.

[17] L. Krecklau, D. Pavic, and L. Kobbelt, "Generalized use of non-terminal symbols for procedural modeling," in *Computer Graphics Forum*, vol. 29, no. 8. Wiley Online Library, 2010, pp. 2291–2303.