Visualization of Industrial Structures with Implicit GPU Primitives

Rodrigo de Toledo, Bruno Lévy

Visualization of Industrial Structures with Implicit GPU Primitives

Rodrigo de Toledo[1] and Bruno Levy[2]

[1] Petrobras – CENPES/PDP/GR, Brazil
[2] INRIA – ALICE, France

Abstract. We present a method to interactively visualize large industrial models by replacing most triangles with implicit GPU primitives: cylinders, cone and torus slices. After a reverse-engineering process that recovers these primitives from triangle meshes, we encode their implicit parameters in a texture that is sent to the GPU. In rendering time, the implicit primitives are visualized seamlessly with other triangles in the scene. The method was tested on two massive industrial models, achieving better performance and image quality while reducing memory use.

1 Introduction

Large industrial models are mainly composed of multiple sequences of pipes, tubes and other technical networks. Most of these objects are combinations of simple primitives, e.g., cylinders, cones, tori and planes. Each CAD application has its own internal data format and different ways to manipulate the objects.

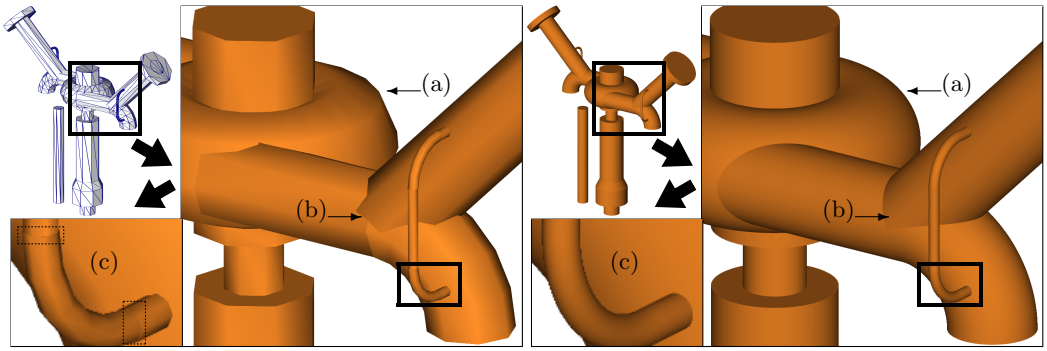

Fig. 1. An industrial piece in PowerPlant (section 13). *Left:* Original tessellated data. *Right:* Rendering with implicit GPU primitives. *Image quality enhancement:* **(a)** silhouette roundness; **(b)** precise intersection; **(c)** continuity between consecutive primitives.

However, the most common way to export data is through a triangular mesh (triangles are currently the *lingua franca* in all 3D software). Representing simple primitives by tessellated approximations results in an excessive number of triangles for these massive models. Interactive visualization is only possible when applying advanced and complex algorithms such as *Far Voxels* [1].

In this paper we propose a **hybrid approach**, using GPU ray-casting primitives [2], in combination with rasterized triangles, for visualizing industrial models. By substituting GPU primitives (cones, cylinders and torus slices) for most of the original triangles, we can achieve several improvements: **quality enhancement** (smooth silhouettes, per-pixel depth and shading, and continuity between pipe primitives, see Figure 1); **faster rendering speed** (CPU/GPU transference reduction and balancing between vertex and fragment pipelines). **Memory reduction** (we only keep implicit data such as radius, height and position, instead of storing vertices, normals and topology of tessellated models).

2 Related work

The idea of replacing meshes with other representations for the purpose of interactive visualization has been widely use. Both billboard and nailboard, which is a billboard that includes per-pixel depth information, have been explored in recent GPUs (in the CG tutorial [3] they are called *depth sprites*). More complex solutions have been proposed, such as billboard clouds [4] and relief texture mapping [5]. In massive model visualization the concept of texture depth meshes was used to obtain interactive rendering [6, 7]. Despite being fast, this method presents some visible regions where the mesh stretches into *skins* to cover missing geometric information. Another issue is storage space (original 482MB PowerPlant uses 10GB of memory).

Recently, a method that has achieved good results for massive models visualization was Far Voxels [1]. It is based on the idea of scene partitioning, grouped into a tree by volumetric clusters. Leaf nodes keep the original triangles and inner nodes have a volumetric grid representation. In rendering time, the grid's voxels are splatted, respecting a maximal screen area. The negative points are: very expensive preprocessing; significant memory use (70MB per million vertices); and an overly complex visualization system (including LOD, occlusion culling and out-of-core data management). In an attempt to accelerate the rendering process, most aforementioned algorithms have as side effects reducing quality and/or increasing memory space. As opposed to them, our solution acheives better performance while enhancing quality and reducing memory.

Ray tracing on GPU Since the beginning of programmable GPUs, ray tracing is a target application that explores their parallelism [8–10]. The z-buffer algorithm is not used to determine visible surfaces because ray tracing already discovers, for each pixel, the closest intersected object given the viewing ray. Purcell et al. [9] broke the ray-tracing algorithm into kernels that access the entire

scene data, which is mapped on textures. The idea of encoding the scene in texture was followed by other methods [11–13]. All these methods suffer from GPU memory limitation. They require space to represent geometry and hierarchical structures, such as kD-trees or bounding volumes. In a recent work, Gunther et al. [13] succeeded in loading the PowerPlant and rendering it at 3 fps.

We also have encoded geometry in texture. However, in contrast with explicitly representing triangles, we record the parameters of implicit surfaces in texture. Thus, we do not have memory space problems (see Section 6.3).

Extended GPU primitives The concept of extended GPU primitives was first introduced by Toledo and Levy [2]. They have created a framework to render quadrics on GPU without tessellation. GPU primitives are visualized through a ray-casting algorithm implemented on fragment shaders. The rasterization of a simple proxy triggers the fragment algorithm. To keep GPU primitives compatible with other surfaces, the visibility issue between objects is solved by the z-buffer. Some recent work have concentrated in ray casting cubics and quartics [14, 15].

3 Topological Reverse Engineering

There are several reverse engineering algorithms for scanned data, composed by a dense set of points laid down on objects surfaces. However, the data we are dealing with present a different situation, since CAD models are generated without any scanning or capturing step. The main differences are: CAD data have sparser samples; the vertices are regularly positioned (spatially and topologically); and it is partially segmented.

Toledo et al. [16] has shown that numerical reverse engineering is not the best choice for CAD triangular meshes. Treating sparse data is a weakness for optimization algorithms [17–19]. On the other hand, it is possible to explore the topological information to reconstruct original implicit data. The topological algorithm is exclusively designed to segment tubes composed of cylinders, truncated cones and torus slices ("elbow junctions"). After applying this reverse-engineering algorithm on triangular meshes, the result is a set of higher-order primitives (see Figure 2) plus a set of triangles for the unrecovered objects. Two data sets, PowerPlant and P40 Oil Platform, were used for tests (see Table 1).

Data set	triangles (Δ)	cylinders	cones	tori	unrecovered Δ	effectiveness
PowerPlant	12,742,978	117,863	2,150	82,359	1,251,019	90.18%
Oil Platform	27,320,034	215,705	40,001	85,707	2,932,177	89.26%

Table 1. Reverse engineering results with PowerPlant and P40 Oil Platform.

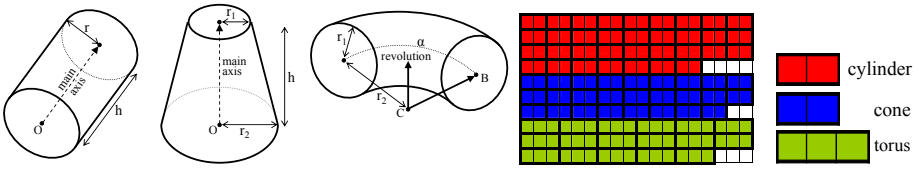

Fig. 2. Recovered primitives: cylinder, cone section and torus slice. Grouping all implicit information in a floating-point texture (each texel has four floats).

4 Storing implicit parameters on GPU memory

In our application, we charge GPU memory with the parameters of the implicit shapes in pre-processing. The goal is to avoid sending this information at each frame to increase frame rate (see results in Section 6). The drawback is that primitive parameters are difficult to modify on the fly. Before the first rendering frame, we include information from all primitives in a floating-point texture, loading it into video memory. Each texel contains four floating-point scalars. We store the following information for each primitive:

cone (8 scalars in 2 texels): origin; main axis scaled by height; two radii.
cylinder (8 scalars in 2 texels): origin; main axis; radius and height.
torus (12 scalars in 3 texels): center; revolution vector; center-to-begin vector; slice angle; two radii.

In texture, the implicit information is grouped by primitive type (Figure 2). We have tested both random and sequential accesses and they do not produce any difference in performance (in NVidia 7900 graphics card)[3]. In rendering time, we draw the primitives by type to avoid too much shader switching.

Recent graphics cards enable the use of floating-point textures that can be read in vertex shaders[4]. In our case, this strategy is better than using vertex buffers because several vertices share the same parameter values. Using a vertex texture makes it possible to represent this level of indirection. This procedure resulted in doubling the speed of our primitives (see Section 6.2).

5 Rendering implicit primitives on GPU

We have implemented some high-performance GPU primitives aiming at visualization of industrial models. We have developed two specific quadrics extending Toledo's et al. work [2]: cones (Subsection 5.1) and cylinders (Subsection 5.2). Based on [15], we have created a novel *torus slice* primitive (Subsection 5.3).

Each surface uses a different fragment shader that executes its ray casting. To trigger fragment execution we rasterize the faces of a specific proxy for each

[3] Horn et al. [11] indicate that, in their case, coherent access was better than random.
[4] We use the RECT texture target without filtering.

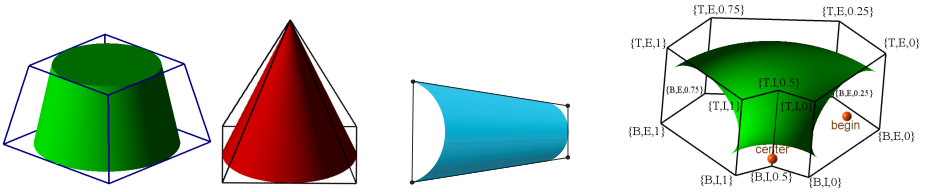

Fig. 3. The front faces of the cone's bounding-box are used to trigger the fragment shader responsible for rendering the cone. A billboard is the best solution for a cylinder without caps. For torus slices, we use an adapted bounding polyhedron with 14 vertices. Each vertex has a parameterized coordinate representing base/top, external/internal and partial angle (1 is the complete slice angle, in this case, 90 degrees).

primitive type: hexahedron for cones; quadrilateral for cylinders; and an adapted polyhedron for tori. To accelerate ray casting, implicit surfaces are locally described in canonical positions. Fragments that do not intersect the surface are discarded, otherwise the fragment shader computes the z-value (or depth).

The coordinates of proxy vertices encode (u, v) texture coordinate rather than a 3D positional info. For example, we use `glRect(-u,-v,u,v)` to call the quadrilateral rasterization for one cylinder. The vertex shader reads implicit information from the texture starting in the position (u, v).

5.1 Cone sections

The ray casting executed inside the hexahedron uses a local coordinate system. In this system, the base of the cone is zero-centered and it has unit radius and unit *total height* (distance between the base and the apex). A very simple fragment-shader computes the intersection between a ray and this canonical cone. The fragment shader is also responsible for drawing the cone caps. Note that *complete* cones (those that include an apex point) are more efficintly rendered by using a pyramid as their bounding-box (see Figure 3).

5.2 Cylinders

For cylinders a quadrilateral billboard is the best choice to trigger their fragment shader. It uses only four vertices and it has a very tight projection enclosing the cylinder, which reduces the discarded fragments. The vertex shader computes the vertex position based on the cylinder's dimensions, in such a way that the 4 vertices always form a perfect convex hull for the cylinder's visible body (Figure 4b). The local coordinate system can be deduced directly from the cylinder's main direction and the computed convex hull axes directions. This is a kind of *view-dependent* coordinate system, where one of the axes is fixed relatively to the world (the cylinder's main axis) and the other axes depend on the viewing

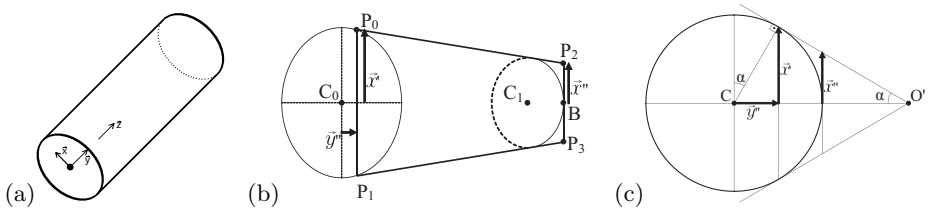

Fig. 4. (a) View-dependent Coordinate System. (b) Billboard in perspective covering the cylinder's body. (c) We reduce the geometric problem to $\mathbb{R}^2$ (O' is the observer projected onto the plane defined by the cylinder's main axis). We can compute the vertices position based on cylinder radius r, distance $|O'C|$ and unit vectors $\boldsymbol{x}$ and $\boldsymbol{y}$.

direction. Finally, the unique per-vertex information required is the relative position in the convex hull (front/back and left/right). This information is implicitly given by negative/positive values combined with (u, v) texture coordinates.

View-dependent Coordinate System. Generally, in a cylinder's local coordinate system, $\boldsymbol{z}$ coincides with the main axis while $\boldsymbol{x}$ and $\boldsymbol{y}$ are in a plane perpendicular to $\boldsymbol{z}$, observing the right-hand rule. In our *View-dependent Coordinate System*, we impose one more restriction: $\boldsymbol{x}$ must be perpendicular to the viewing direction $\boldsymbol{v}$.

$$\boldsymbol{z} = \frac{(C_1 - C_0)}{|\,(C_1 - C_0)\,|}\ , \quad \boldsymbol{x} = \boldsymbol{v} \times \boldsymbol{z}\ , \quad \boldsymbol{y} = \boldsymbol{z} \times \boldsymbol{x}$$

The vector $\boldsymbol{v}$ is a normalized vector pointing to the observer (as if it was perpendicular to the sheet of paper for the reader in Figure 4a). Note that $\boldsymbol{y}$ and $\boldsymbol{z}$ are aligned when projected to the screen, although perpendicular in $\mathbb{R}^3$.

To compute the final position of the vertices P_0, P_1, P_2, P_3, we must consider the viewer position and perspective distortion. As indicated on Figure 4, we can compute these positions as follows:

$$\begin{aligned} P_0 = C_0 + \boldsymbol{y''} + \boldsymbol{x'},\ P_1 = C_0 + \boldsymbol{y''} - \boldsymbol{x'} \\ P_2 = C_1 + \boldsymbol{y'} + \boldsymbol{x''},\ P_3 = C_1 + \boldsymbol{y'} - \boldsymbol{x''}\ \text{, where} \\ \boldsymbol{x'} = \boldsymbol{x} \cdot r \cdot \cos\alpha,\ \boldsymbol{x''} = \boldsymbol{x} \cdot (|O'C| - r) \cdot \tan\alpha, \\ \boldsymbol{y'} = \boldsymbol{y} \cdot r, \boldsymbol{y''} = \boldsymbol{y} \cdot r \cdot \sin\alpha, \text{ and } \alpha = \arcsin\left(\frac{r}{|O'C|}\right). \end{aligned}$$

Thickness control is a special feature developed for the GPU cylinders. In the vertex shader we guarantee that vectors $\boldsymbol{x'}$ and $\boldsymbol{x''}$ have a minimum size of 1 pixel on the screen. This avoids the dashed line aspect of thin and high cylinder rendering, which may occur when the viewer is far away.

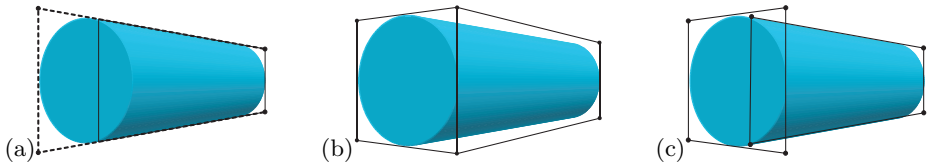

Fig. 5. Three different ways to render the GPU cylinder with caps: (a) extending the billboard used for the body; (b) using two faces and six vertices of a bounding box; or (c) rendering two separated billboards, one for the body and another one for the caps.

Cylinders' caps. Similarly to cones, it is straightforward to compute the intersection between a ray and the canonical cylinder's body. However, to render the cylinder's caps, there are three possible strategies to attach them:

- Extending the billboard in directions $\boldsymbol{P_2P_0}$ and $\boldsymbol{P_3P_1}$ (Figure 5a).
- Using a partial 3D bounding box, with 2 faces and 6 vertices (Figure 5b).
- Using a second billboard showing the cap turned to the viewer (Figure 5d).

In the first option, note that $\boldsymbol{P_2P_0}$ and $\boldsymbol{P_3P_1}$ directions are divergent and extending them may include too many fragments that will be discarded at the end. On the other hand, the second and third options have the drawback of increasing the total number of vertices.

In our implementation we have chosen the last option, because it is the one that better reduces the number of discarded fragments. The vertex shader of this *extra* GPU primitive uses the same parameters stored on the floating-point texture. Notice that only one extra primitive is used to render both caps, because they are never visible together in the same frame. Finally, in our industrial model visualization, most cylinders do not have caps because they are part of a pipe sequence. Thus, the extra billboard is used only for few ones.

5.3 Torus slices

In industrial structures, torus shapes appear in junctions, chains and CAD patterns. Actually, in most cases, the objects contain only a *slice* of a torus. We use a bounding box well adapted to its form rather than a simple hexahedron. This polyhedron was chosen to reduce pixel wasting by better fitting the slice. At the same time, it is possible to surround torus slices from small angles up to 180 degrees without a significant vertex cost.

We have implemented a special vertex shader to automatically locate vertices around the torus slice based on its parameterization. Once more, (u, v) texture coordinates are associated with positive/negative values, determining internal/external and top/base position. The third coordinate represents the relative angle in the interval $[0, 1]$. The vertex shader fetches torus parameters and computes vertex world and local coordinates.

We use an iterative algorithm for ray casting tori, the Newton-Raphson root finder presented by Toledo et al. [15]. They have compared several solutions, considering this one the fastest solution for ray casting torus on GPU.

6 Results

6.1 Enhancing Image Quality

There are five image quality improvements obtained with our method.

Smooth silhouette. The tessellation of curved surfaces impedes the rendering of continuously smooth silhouette, since it is restricted by mesh discretization. The GPU primitives are computed by pixel, therefore the silhouettes are always smooth, even after a huge zoom. See Figure 1(a).

Intersections (Per-pixel z-computation). In conventional triangle rasterization, the z-buffer depth of each pixel is the result of interpolation of per-vertex depth. On the other hand, the GPU primitives compute per-pixel depth, which is much more accurate. When two or more GPU primitives intersect, the boundary has a correct shape due to this precise visibility decision (Figure 1(b)).

Continuity between pipe primitives. In industrial plant models, the pipes are a long sequence of primitives. With the tessellated solution, there is a risk of cracks appearing between primitives. To avoid them, the tessellation should keep the same resolution and the same alignment for consecutive primitives. In Figure 1(c), a misalignment has caused cracks. This kind of undesirable situation is also a problem in applications using LOD in pipes (see [20]), whereas with GPU primitives continuity is natural since there is no discretization.

Per-pixel shading. Our primitives compute shading by pixel (Phong shading), enhancing image quality if compared to default Gouraud shading.

Cylinder thickness control. The thickness control adopted in the vertex shader of GPU cylinders avoids *dashed* rendering when zooming out of a thin and high cylinder. Moreover, when this control is associated with color computation based on the normal average, it significantly reduces the aliasing pattern effect for a group of parallel cylinders (see Figure 6b).

6.2 Performance

We have done two sets of performance measures targeting different comparisons. The first set uses the PowerPlant and compares GPU primitives with tessellated data of the original model. The second set uses P40 model to compare the visualization of GPU primitives with different levels of mesh tessellation.

All tests were done on an AMD Athlon 2.41GHz, 2GB memory, with GeForce 7900 GTX512MB. We render the models entirely in frustum view, fulfilling a 1024×768 screen. In this scenario, the bottleneck is not in the fragment shader because primitives are not so large on the screen.

Important Note. All the rendering tests were executed without any culling technique, which would surely speed up all the frame rates. We have intentionally presented the results in this way to avoid covering up speed results when comparing GPU primitives and exclusively-rasterized polygons.

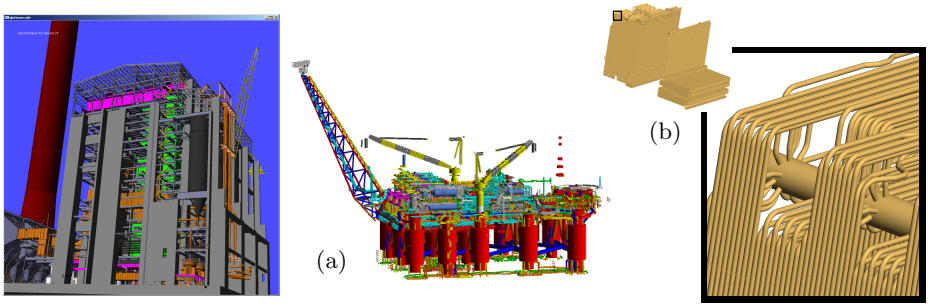

Fig. 6. (a) PowerPlant and P40 Oil-Platform models used on performance tests. (b) PowerPlant Section 1, about 60,000 primitives recovered from 3,500,000 triangles. The thickness control prevents undesired visible patterns when cylinders are seen from far.

Sec.	FPS				Other information		
	GPU prim.		VBO				
	isolate	group	multiple	single	#triangles	#UT	#prim.
1	48.5	**93.0**	**51.0**	2.9	3,429,528	0	57,938
15	138.5	**268.5**	**177.5**	165.5	1,141,240	0	21,839
19	62.5	**128.0**	**73.5**	3.8	2,650,680	0	42,046
20	66.5	**130.5**	**66.9**	4.1	2,415,976	0	41,872
	group	+UT	all	only UT	#triangles	#UT	#prim.
12	911	**719**	**524**	2935	360,872	38,067	6,205
all	27.8	**21**	**12.9**	81	12,742,978	1,251,019	202,372

Table 2. PowerPlant performance test. In first block, Sections 1, 15, 19 and 20 do not have unrecovered triangle (UT), in other words, triangles were 100% converted into primitives by the reverse engineering. Frames per second (FPS) measured without v-sync. In second block, Section 12 and the entire PowerPlant have some UT.

PowerPlant. PowerPlant sections 1, 15, 19 and 20, used in Table 2, were intentionally chosen because they were 100% recovered by the reverse engineering algorithm. This way, we can directly compare rasterization and GPU primitive techniques. We have created two situations for GPU primitives: isolated (without using the floating-point texture, but passing their implicit information at each frame) and grouped. The latter is twice as fast as the isolated solution. We also have compared two rasterization strategies based on VBO (Vertex Buffer Object): decomposing the model in multiple VBOs and using a single VBO. Decomposing gives better results for large models.

Comparing the fastest solutions for GPU primitives and for triangle rasterization (second and third columns in Table 2), grouped GPU primitives are clearly faster, almost doubling the speed with multiple VBO implementation.

A different situation is presented on the two-last lines of Table 2, where some of the original triangles (10%) were not converted to implicit primitives (e.g.,

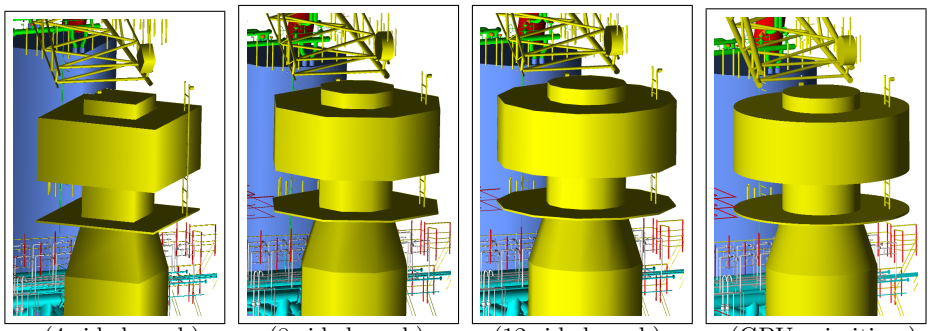

Fig. 7. Comparing GPU primitives and tessellations for the P40 model.

Method	FPS	Memory	Triangles
4-sided mesh	40fps	283MB	4,475,852
8-sided mesh	21fps	464MB	10,559,474
12-sided mesh	13fps	740MB	18,394,158
GPU primitives	22fps	89MB	(*)

Table 3. Performance results for oil-platform P40. (*) 341,413 primitives: 215,705 cylinders, 40,001 cones and 85,707 tori.

walls). We compare hybrid rendering (GPU primitives + UT, second column) with the best VBO rasterization rendering (third column). The hybrid solution is between 40% and 60% faster than rasterization.

Oil platform - P40. In this set of measures we have compared the visualization of GPU primitives with mesh rasterization with different levels of tessellation. The tests were done with 4, 8 and 12-sided meshes (see Figure 7). We were not able to load a 16-sided mesh version because of memory restriction.

As can be seen in Figure 7, the GPU primitives achieve a much better image quality than any tessellated solution. The performance of GPU primitives is comparable to the 8-sided solution (see Table 3).

6.3 Memory use

In Table 4 we summarize memory-space information before and after recovering implicit surfaces. The topological recovery procedure converts 90% of the original data of industrial models (PowerPlant section 1 is an exception because it is only composed by tubes and pipes resulting in 100% of conversion rate). The memory-space of recovered data is reduced to at most 2% (98% of reduction). This is a consequence of compact implicit representation used for recovered primitives. If

Model	Triangles	Initial space	Conversion	Primitives space	Final space
PP Sec. 1	3,429,528	119MB	100%	1.9MB	1.9MB
PPlant	12,742,978	482MB	90.18%	6.5MB	79.3MB
P40	27,320,034	1033MB	89.26%	10.25MB	121.19MB

Table 4. Memory space comparison.

we consider the remaining 10% of unrecovered triangles, industrial models such as oil platforms and power plants can be stored in about 15% of their original data size. The size of floating-point textures on recent graphics cards is limited to 4096×4096 (16M texels with 4 floating-points), using 256MB. Our biggest example, P40 model, only uses 4% of this space. Note that it is also possible to use multiple textures, which would push further the scene-size limits.

7 Conclusion

The use of GPU primitives for CAD and industrial models is very promising. The benefits are classified in three categories: image quality (e.g., perfect silhouette and per-pixel depth), memory and rendering efficiency. Grouping primitive information in a GPU texture has proven to be a good strategy for performance purposes without any memory space problem.

As future work, we suggest the application of well-known acceleration techniques that are usually applied on conventional visualization of massive models: frustum culling, occlusion culling, coherent memory cache, and so on. Since GPU primitives update the Z-buffer, special techniques, which are usually applied to triangles, can also be combined with our primitives. Two interesting future work are shadow maps application and occlusion culling based on queries.

Most implementations of GPU ray tracing have adopted triangles as their only primitive. We suggest the use of implicit primitives, which would probably reduce memory use, which is a critical issue for them.

We also recommend the implementation of other surfaces found in industrial plant that were not covered in our work (e.g., sheared cylinders and half spheres).

References

1. Gobbetti, E., Marton, F.: Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. ACM Trans. Graph. **24** (2005) 878–885
2. Toledo, R., Levy, B.: Extending the graphic pipeline with new gpu-accelerated primitives. In: 24th International gOcad Meeting, Nancy, France. (2004) also presented in Visgraf Seminar 2004, IMPA, Rio de Janeiro, Brazil.
3. Fernando, R., Kilgard, M.J.: The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2003)

4. Dcoret, X., Durand, F., Sillion, F.X., Dorsey, J.: Billboard clouds for extreme model simplification. ACM Trans. Graph. **22** (2003) 689–696
5. Oliveira, M.M., Bishop, G., McAllister, D.: Relief texture mapping. In: Proceedings of ACM SIGGRAPH 2000. Computer Graphics Proceedings, Annual Conference Series (2000) 359–368
6. Aliaga, D., Cohen, J., Wilson, A., Baker, E., Zhang, H., Erikson, C., Hoff, K., Hudson, T., Stuerzlinger, W., Bastos, R., Whitton, M., Brooks, F., Manocha, D.: Mmr: an interactive massive model rendering system using geometric and image-based acceleration. In: SI3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics, ACM Press (1999) 199–206
7. Wilson, A., Manocha, D.: Simplifying complex environments using incremental textured depth meshes. ACM Trans. Graph. **22** (2003) 678–688
8. Carr, N.A., Hall, J.D., Hart, J.C.: The ray engine. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, Eurographics Association (2002) 37–46
9. Purcell, T.J., Buck, I., Mark, W.R., Hanrahan, P.: Ray tracing on programmable graphics hardware. ACM Transactions on Graphics **21** (2002) 703–712 ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).
10. Wald, I., Purcell, T.J., Schmittler, J., Benthin, C., Slusallek, P.: Realtime Ray Tracing and its use for Interactive Global Illumination. In: Eurographics State of the Art Reports. (2003)
11. Horn, D.R., Sugerman, J., Houston, M., Hanrahan, P.: Interactive k-d tree gpu raytracing. In: I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games, New York, NY, USA, ACM Press (2007) 167–174
12. Popov, S., Günther, J., Seidel, H.P., Slusallek, P.: Stackless kd-tree traversal for high performance GPU ray tracing. Computer Graphics Forum **26** (2007) 415–424 (Proceedings of Eurographics).
13. Günther, J., Popov, S., Seidel, H.P., Slusallek, P.: Realtime ray tracing on gpu with bvh-based packet traversal. In Keller, A., Christensen, P., eds.: IEEE/Eurographics Symposium on Interactive Ray Tracing, Ulm, Germany, IEEE, Eurographics (2007)
14. Loop, C., Blinn, J.: Real-time gpu rendering of piecewise algebraic surfaces. In: SIGGRAPH '06: ACM SIGGRAPH 2006 Papers, New York, NY, USA, ACM Press (2006) 664–670
15. de Toledo, R., Levy, B., Paul, J.C.: Iterative methods for visualization of implicit surfaces on gpu. In: ISVC, International Symposium on Visual Computing. Lecture Notes in Computer Science, Lake Tahoe, Nevada, Springer (2007) 598–609
16. de Toledo, R., Levy, B., Paul, J.C.: Reverse engineering for industrial-plant cad models. In: TMCE, Tools and Methods for Competitive Engineering, 2008, Izmir, Turkey (2008) 1021–1034
17. Thompson, W., Owen, J., de St. Germain, H., Stark, S., Henderson, T.: Feature-based reverse engineering of mechanical parts. IEEE Transactions on Robotics and Automation (1999) 57–66
18. Fitzgibbon, A.W., Eggert, D.W., Fisher, R.B.: High-level CAD model acquisition from range images. Computer-aided Design **29** (1997) 321–330
19. Petitjean, S.: A survey of methods for recovering quadrics in triangle meshes. ACM Comput. Surv. **34** (2002) 211–262
20. Krus, M., Bourdot, P., Osorio, A., Guisnel, F., Thibault, G.: Adaptive tessellation of connected primitives for interactive walkthroughs in complex industrial virtual environments. In: Virtual Environments '99. Proceedings of the Eurographics Workshop in Vienna, Austria. (1999) 23–32