An Efficient Trim Structure for Rendering Large B-Rep Models

Frédéric Claux, David Vanderhaeghe, Loic Barthe, Mathias Paulin, Jean Pierre Jessel, David Croenne

Vision, Modeling, and Visualization (2012)
M. Goesele, T. Grosch, B. Preim, H. Theisel, and K. Toennies (Eds.)

An Efficient Trim Structure for Rendering Large B-Rep Models

Frédéric Claux[1,2], David Vanderhaeghe[1], Loïc Barthe[1], Mathias Paulin[1], Jean-Pierre Jessel[1], David Croenne[2]

[1]IRIT - Université de Toulouse [2]Global Vision Systems

Abstract

We present a multiresolution trim structure for fast and accurate B-Rep model visualization. To get a good tradeoff between performance and visual accuracy, we propose to use a vectorial but approximated representation of the model that allows efficient, real-time GPU exploitation. Our structure, based on a quadtree, enables us to do shallow lookups for distant fragments. For closeups, we leverage hardware tessellation. We get interactive frame rates for models that consists of hundreds of thousands of B-Rep faces, regardless of the zoom level.

1. Introduction

Boundary Representation (B-Rep) models are the de-facto representation for the geometry produced by Computer Aided Design (CAD) software applications. B-Rep models contain a set of *faces*, where each face is described by a *basis surface* such as a plane, a cone or a NURBS, and *trimming curves* defined in the *parametric space* of each basis surface. Each face has one outer trimming curve and zero or more inner trimming curves defining holes, possibly with islands. The representation of CAD models often requires the use of faces with a very large number of holes, as the ones appearing in Figure 1. Trimming curves are defined with parametric representations such as lines, circles, ellipses, polynomial or rational curves. The accurate visualization of such models in real time is of high interest for the industry but remains an open scientific challenge, especially for very large models.

CAD model visualization, which we are dealing with in this paper, ideally requires unlimited precision (Figure 2). As users zoom into the model, they expect to see a very accurate rendering with smooth curves and sharp, crisp features, just like when they observe actual machined parts.

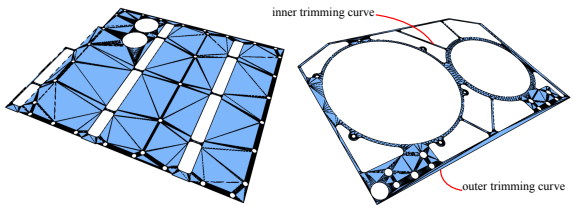


Figure 1: *Trimmed surface tessellation produces a lot of ill-configured triangles.*

Prior to being given to a 3D rendering engine for display, B-Rep models are usually tessellated into triangles, readily renderable by Graphics Processing Units (GPUs). One major drawback of this process is that tessellation is carried out at a fixed resolution. The higher the precision, the more accurate the generated mesh and the display but the lower the rendering performance due to the higher number of tessellated polygons. Because B-Rep faces sometimes contain hundreds of trimming curves, a large number of vertices and triangles are produced to match the shape of these holes and contours, leading to an increased memory usage and decreased rendering performance. Tessellation examples are presented in Figure 1.

As addressed in previous work, it is possible to use GPUs to render faces on the fly during rendering using geometrical representations. A face is represented by a basis surface and a binary texture, called a trim texture. The trim texture defines points that are trimmed away from the basis surface and points that remain on the trimmed basis surface. This process is called *point classification*. We call trimmed points *off-face* points and remaining points *on-face* points. The basis surface is rendered using either dynamic tessellation [SS09] or ray casting [TL08], and during rasterization, off-face and on-face fragments are identified by a query in the trim texture. Finally, only the on-face fragments are displayed. Even though very elegant, such state of the art techniques (Section 2) provide insufficient rendering interactivity or require prohibitive memory to be effective when dealing with very large models.

Contributions: Based on trim textures, we present a method allowing us to render large B-Rep models with high visual quality, at interactive to real-time frame rates. Our

Authors version
The definitive version is available at diglib.eg.org
doi: 10.2312/PE/VMV/VMV12/031-038

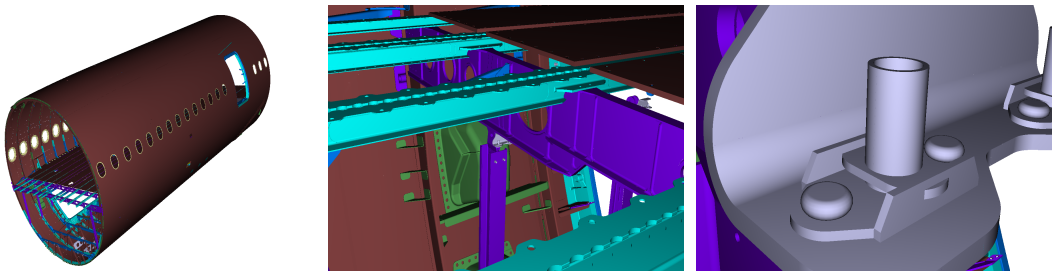


Figure 2: *Our point classification method for trimmed surface rendering scales well with the zoom level. When surfaces are zoomed out, a multiresolution mechanism reduces the cost of point classification. When surfaces are zoomed in, we leverage hardware tessellation when possible and perform triangle classification to accelerate overall surface rendering.*

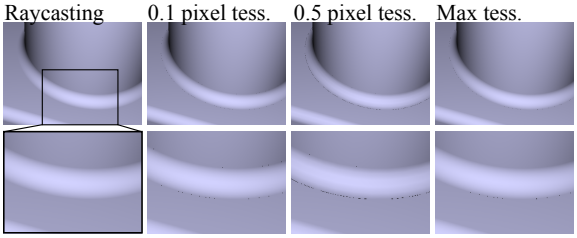


Figure 3: *Cracks between two faces caused by tessellation. The circular trimming of the plane, around the tore, is performed with the method of Schollmeyer and Fröhlich that has no precision problem. Left: Error-free but slow rendering using raycasting. Middle: Tore tessellation using a chordal error of 0.5 or 0.1 screen pixel, leaving cracks. Right: pushing the hardware tessellation factor to its maximum value is not always sufficient to completely remove cracks either.*

main contribution lies in the way we represent and exploit a vector trim structure for efficient on-face and off-face point classification. We have observed (Section 2) that raycasting only can lead to an error-free rendering of B-Rep models. Nevertheless, it is too slow. Using tessellation almost inevitably leads to cracks between faces (Figure 3). Based on these observations, we choose to deliberately approximate trimming curves, which does create a few more cracks between faces, at very high zoom level, but enables extra performance gains. The maximum allowed size of these cracks is defined during a preprocess. The multiresolution nature of our trim structure also enables us to reduce computation costs for distant fragments. Finally, we leverage hardware tessellation to perform early triangle classification, our structure being well suited to tessellation-based rendering. Our trim structure is unbalanced which reduces memory usage, especially compared to methods based on mipmaps.

Overview: Ideally, point classification should be done on the original trimming curves but this requires a high computational cost at render time. Instead, we approximate all types of input trimming curves, within a controllable error, with sets of connected 2D quadratic Bézier curves represented implicitly (Section 3) and stored in a quadtree built over the parametric space of each basis surface (Section 4). Each node of the quadtree contains a binary coverage value telling us if the quadtree cell is mainly on or off the face. Leaf nodes that are not completely on or off the face also contain references to the quadratic approximation curves crossing the node. This structure, efficiently handled by the GPU (Section 7), allows us to save on memory compared to previous works and to enable a multiresolution lookup reducing screen flicker and accelerate point classification on distant (Section 5, Multiresolution access) and close (Section 6) objects.

Our data structure is only bound to the *uv* coordinates in the parametric domain of the basis surfaces, that will be referred to as the *parametric space* onwards. Although it is independent of the rendering method being used, it shows most of its benefits when used in conjunction with hardware surface tessellation. For all rendered fragments, we calculate their *uv* coordinates in parametric space and access our quadtree trim structure to do the classification. During the traversal of this quadtree, we stop the traversal as soon as we reach a node that covers less than a pixel, in which case we use the node coverage value to do the classification. If we reach a leaf node, if this leaf node covers more than one fragment and has references to quadratic curves, we resort to a more expensive quadratic evaluation (Section 5). For object close-ups, we classify triangles issued from the tessellation unit (Section 6).

With our method, model features are displayed in a smoother fashion than when using static model tessellation, at any zoom level. We demonstrate real-world model rendering at interactive frame rates with models containing hundreds of thousands of B-Rep faces (Section 8).

2. Previous Work

Each rendered point (or fragment) of a basis surface needs to be classified as either on or off the face. The idea of exploiting trim textures in order to classify points on NURBS-based models has been introduced by Guthe et al. [GBK05]. Guthe et al. convert NURBS and T-Splines into a hierarchy of Bézier patches and binary trim textures where each node matches a given world space approximation error. Data is then streamed to the GPU on demand during rendering. Their work heavily relies on continuous streaming and has

problems dealing with close-ups, where the texture resolution is exploding.

In an effort to overcome this resolution problem, Schollmeyer et al. [SF09] define an approximation-free trimming model preserving geometrical properties. They first break down input NURBS curves into piecewise, bimonotonic rational Bézier curves. Dealing with such curves allows them to determine iteratively if a point shall be classified as on or off the face. Curve segments are stored in an acceleration structure based on a dual binary tree. Their method is very accurate as they do not need to approximate input trimming curves. When used in conjunction with raycasting, rendering is guaranteed to be pixel-precise. On the downside, performance starts to drop when too many fragments need to be classified, using either raycasting or tessellation.

The vectorial representation of trim texture shares common ideas with vector textures and path rendering. Nehab and Hoppe [NH08] perform path rendering in realtime. Their approach is to build a regular grid over a vector drawing, and then restrict the path rendering work to each cell. Their approach has no level of details mechanism. Cell size is typically fixed manually with a tradeoff between performance and memory consumption.

Ray et al. [RNCL05] introduce Vector Texture Maps. A Vector Texture Map is a multiresolution vector function representation, using monotonic representations of vector curves restricted to the cell of a 4x4 subdivision tree. Point classification is done by evaluating up to two curves bound to each cell. Their multiresolution structure is based on a mipmap. The trimming curves found in CAD models would sometimes push the quadtree depth down to 12 or more, triggering the creation of a 2^{12} by 2^{12} texture. This memory-hungry representation is unsuitable in our context.

Hanniel and Haller [HH11] propose an approach for pixel-accurate rendering using a point classification algorithm based on a 2-pass ray casting. Their method does not scale up well for large B-Rep models and their point classification method only works with raycasting. Finally, they need adjacency information between faces, which is not always available in B-Rep models.

3. Trimming Curves Approximation

The first step of our approach, done in a pre-process, is to generate a representation of trimming curves suitable for efficient GPU access and evaluation.

3.1. Approximation with connected Bézier curves

Trimming curves are first approximated by piecewise quadratic Bézier curves following a world space approximation error ε, as shown in Figure 4. We first use a curve fitting algorithm interpolating discretized points of the original curve and producing a fixed-degree (in our case, two) uniform Bspline. This Bspline is then simplified using a knot reduction algorithm guided by a parametric space approximation error calculated from ε, and broken down into a sequence of connected quadratic Bézier curves [LM87,LM88]. C^1 continuity is maintained across curve endpoints when the input curves are themselves C^1 continuous. Flat quadratic Bézier curves are converted to their line segment definition.

3.2. Local Implicit Reconstruction

Following the convention used for the input trimming curves, our quadratic curves are always represented with globally consistent orientation that ensures that on-face points are on the left side of the curve. Point classification just consists in evaluating on which side of a curve a point lies. Inside the convex hull of the control polygon of quadratic Bézier curves, classifying a point is achieved by using the method introduced by Loop and Blinn [LB05].

Loop and Blinn use a field function $f : \mathbb{R}^2 \to \mathbb{R}$, defined as $f(x,y) = y - x^2$ if the face is locally convex and $f(x,y) = x^2 - y$ if the face is locally concave, to represent a quadratic Bézier curve, with (x,y) expressed in a *quadratic space* defined by a *quadratic frame* (Figure 5). In this quadratic frame the quadratic Bézier curve is the 0-set of f. The quadratic frame is also defined such that, in this frame, the successive control points p_i, $i = 0,1,2$ of the Bézier curve have their coordinates (x_i, y_i) successively equal to $(0,0)$, $(0.5,0)$ and $(1,1)$ (Figure 5). Within the convex hull of the Bézier curve control polygon, (x,y) point coordinates evaluating to $f(x,y) \geq 0$ are on-face.

When classifying a point of coordinates (u,v) in the parametric space, its coordinates are transformed into (x,y) in the quadratic space as follows:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 2(u_{p_1} - u_{p_0}) & u_{p_0} - 2u_{p_1} + u_{p_2} & -u_{p_0} \\ 2(v_{p_1} - v_{p_0}) & v_{p_0} - 2v_{p_1} + v_{p_2} & -v_{p_0} \\ 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

where (u_{p_i}, v_{p_i}), $i = 0,1,2$ are the coordinates of the Bézier curve control points in the parametric space. Once this is done, the evaluation of $f(x,y)$ tells us if the point is on-face ($f(x,y) \geq 0$) or off-face ($f(x,y) < 0$).

4. Building The Multiresolution Structure

The multiresolution representation of the face is generated off of the parametric space as a recursive power of two subdivision of uniform cells.

Leaf cells can identify an area that is fully on or off the face, or that intersects one or two quadratic curves. Cell subdivision continues until each leaf cell respects these constraints. Leaves store a reference to the one or two quadratic curves they intersect. This is shown in Figure 4(b).

Having two quadratic curves referenced in a cell is inevitable because junction points between adjacent piecewise quadratic segments will not always fall onto cell boundaries. Also, two distinct trimming curves might be so close to each other that subdividing cells until these trimming curves end up in different cells would force the subdivision process to

Authors versionThe definitive version is available at diglib.eg.orgdoi: 10.2312/PE/VMV/VMV12/031-038

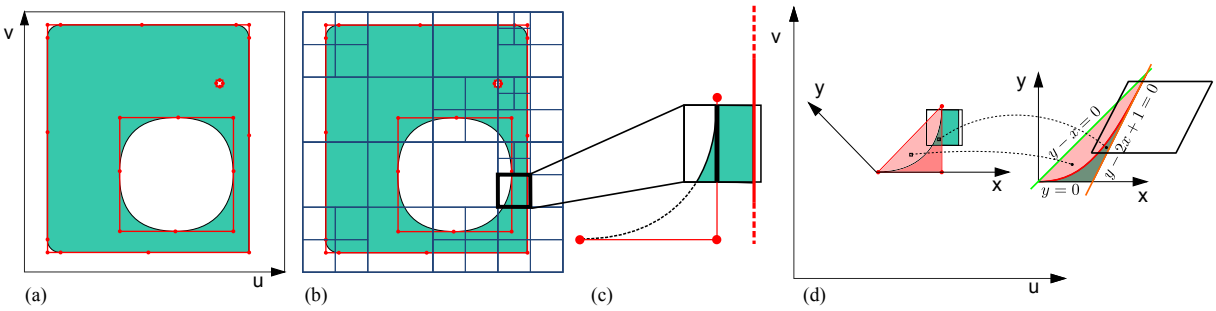


Figure 4: *(a) Piecewise quadratic approximation of trimming curves. The control points and the control polygons of Bézier quadratic curves are shown in red. (b) Quadtree structure: each leaf is either fully inside or outside the trimmed surface, or intersecting at most two quadratic curves. (c) When a leaf intersects two quadratic curves, the curves are separated by a line that is bound to the cell. (d) To evaluate a quadratic Bézier curve implicitly, we transform point coordinates from the uv parametric space to the xy quadratic frame.*

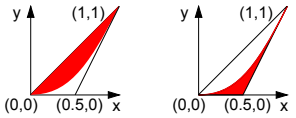


Figure 5: *Illustration of the on-face area (in red) defined by the implicit representation of the Bézier curve in the quadratic frame. Left: on-face convex area verifying* $y-x^2 \geq 0$*. Right: concave area verifying* $x^2-y \geq 0$*.*

be extremely deep in order to respect the constraints. Figure 6 shows a quadtree built over the parametric domain for two complex B-Rep faces.

The top-down recursive cell subdivision process is followed by a bottom-up on-face cell coverage calculation. The coverage value is used at runtime as part of the multiresolution lookup (Section 5, Multiresolution access).

5. Point Classification Within Cells

Point classification for completely on or off face cells is trivial to handle. We now describe the classification process for cells with intersecting curves.

Cells with one quadratic curve: In a cell intersected by a single convex curve, the point coordinates (u,v) in parametric space are transposed to (x,y) quadratic frame coordinates

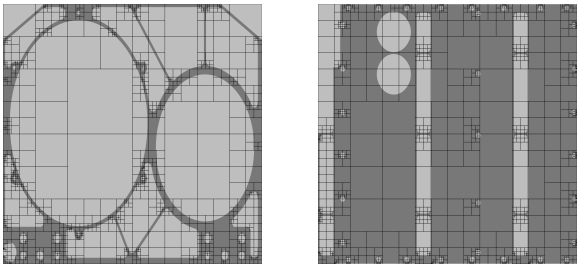


Figure 6: *Quadtree structure generated for the faces shown in Figure 1*

as explained in Section 3.2. A point is first tested against the control polygon of the curve and classified as off-face if $y<0$ or $y-2x+1<0$, as on-face if $y-x \geq 0$; outside the control polygon, we classify the point with the evaluation of the field function f presented in Section 3.2 (Figure 4(d)). In the concave case, on-face becomes off-face and the opposite way around.

Cells with two quadratic curves: When two quadratic curves intersect a cell, we precompute a separating line. Considering only the two curve parts located within a cell, a separating line is a line that intersects none of the two quadratic curves, with one curve on each of its sides (see Figure 4(c)), and can be calculated with the separating axis theorem [Ebe06, p. 393]: given two non-intersecting convex polygons, there always exists a line with one polygon on one side of the line, and the other polygon on the other side. In addition, it exists a segment of one of the polygons that is the support of a separating line (Figure 4(c)). Such a segment is determined by subdividing these polygons until they do not intersect. This always happens since faces are 2D-Manifold and thus, Bézier curves do not cross. Then we test lines supported by the refined control polygons segments and we take as separating line the first line that does not intersect the other polygon (Figure 14 right). At runtime, we first evaluate on which side of the line a point is, then proceed with the point classification method described in the previous paragraph.

Multiresolution access: The binary coverage value determines if a cell mainly ($>50\%$) represents an on-face or off-face surface area in the cell. Leaf cells that are completely on or off the face are respectively identified in the quadtree by a coverage value of 1 or 0, with no reference to quadratic curves.

At runtime, the quadtree traversal is stopped when a node covering an area of less than a screen pixel is reached. When reaching a leaf covering several pixels, point classification is performed following the procedure presented in Section 5. This multiresolution access enables us to both increase rendering performance by limiting the depth of the quadtree

traversal (Figure 7 and 8(a)) and to avoid visually inelegant moiré patterns (Figure 8(b)) at low zoom levels.

6. Triangle classification

The input B-Rep structure and topology of a model usually reflects the way the model has been designed by the CAD operator, before the B-Rep export took place. Very large B-Rep surfaces containing only very sparse on or off-face areas are common. They are prone to hamper performance as they generate a very large number of fragments during rasterization that could ideally be trivially classified. See the airplane model section in Figure 9.

When B-Rep faces are close to the observer, performance starts to drop dramatically as point classification needs to be carried out for a very large number of fragments: even small surfaces generate many fragments when they are zoomed in.

The cost of point classification in our method mostly comes from the traversal of the quadtree, which is proportional to the node depth. The classification against the curves in the reached node, when needed, is relatively fast, as we use implicit evaluations. To reduce the number of individual fragment classifications, we propose to classify the triangles issued from dynamic tessellation. Additionally, for triangles that we cannot classify, we speed up the classification of underlying fragments.

Basis surfaces are tessellated over the u,v parametric domain at runtime respecting a screen-space error (see Section 7). For each surface, we precompute a quadtree level k_i at which we know we will be able to quickly classify at runtime at least 40% of the parametric domain when covered by triangles issued from tessellation using a pair of tessellation factors $t_u = 2^{k_i}, t_v = 2^{k_i}$. At runtime, we want the tessellated triangles to both respect the screen space error and also never cross node boundaries up to the k_i^{th} level in the quadtree, guarantying that at least 40% of them are classified during rendering. We thus compute tessellation factors to respect the screen space error, and adjust them so that their value match the nearest superior power of two value greater than or equal to 2^{k_i}, giving us the final tessellation factors $2^{k_{uf}}, 2^{k_{vf}}$. Figure 10 shows how we adjust the tessellation factors so that enough triangles are classified.

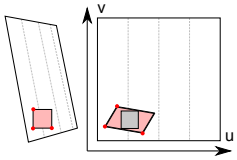
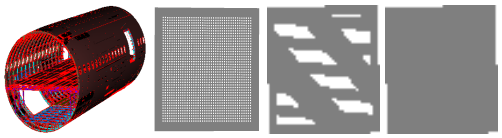


Figure 7: *Left: fragment footprint in screen space. The parametric domain is distorted to reflect the perspective projection of the current view. Right: fragment footprint in parametric space. We calculate the size of the largest square fitting in the footprint to limit quadtree traversal depth.*



(a) (b)

Figure 8: ***(a)*** *Red fragments making use of our multiresolution access. They identify fragments for which we stopped the traversal of the quadtree at an intermediate node, or at a leaf node but without resorting to actual curve evaluation.* ***(b)*** *Left: example B-Rep face with many tiny holes. Center: moiré pattern visible at low zoom levels. Right: when multiresolution access is used, holes disappear altogether, and there is no moiré pattern.*

For every tessellated triangle we classify it before rasterization takes place as shown in Figure 11. A single node lookup is performed by traversing the quadtree up to a maximum level $k_f = \min(k_{uf}, k_{vf})$, once per triangle, using its barycentric center. At this point, a triangle can be classified as on-face or off-face if the node is a quadtree leaf that

- identifies a completely on-face or off-face area,
- holds one quadratic curve, and the triangle resides in the convex side of the curve,
- holds two quadratic curves, the triangle points fit on the same side of the separating line, and reside in the convex side of the corresponding quadratic curve.

Otherwise, when the triangle cannot be classified, the reference to the quadtree node is passed to the fragment shader, which quadtree traversal for the contained fragments will resume from.

The performance increase achieved through triangle classification must outweight the penalty incurred by doing the extra tessellation of the surface that would otherwise typi-

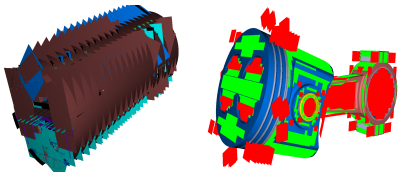


Figure 9: *Left: untrimmed airplane model revealing many planar faces cutting the fuselage across. The fuselage has many circular sections that have been designed as one single part in the CAD program. Right: green and red zones in the piston model match triangles for which triangle classification was successful. Red triangles are discarded.*

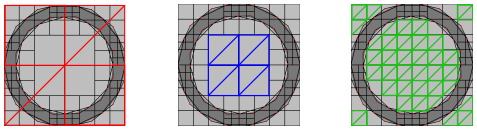


Figure 10: *Level 1-aligned triangles (red) cannot be classified, while level 2 and 3 = k_i triangles shown in blue and green can, and cover 25% and 50% of the parametric space.*

Authors versionThe definitive version is available at diglib.eg.orgdoi: 10.2312/PE/VMV/VMV12/031-038

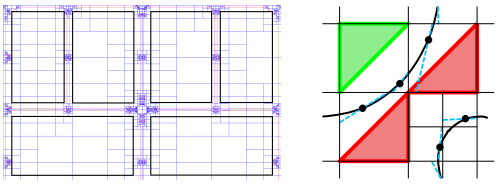


Figure 11: *Left: planar surface with large off-face areas highlighted in black. Right: triangle classification. The red triangles classification fails as their vertices are either on both sides of the node's separating line (right), or on the concave side of the curve (bottom). Green triangles that have passed classification will either be quickly classified as on-face areas, or discarded by the geometry shader altogether.*

cally be tessellated into the minimum amount of triangles with respects to the screen-space tessellation error. Triangle classification is increasingly interesting as the average number of fragments per triangle increases. On average, the cost of classifying n fragments lying in a screen-space triangle T must be lower when tessellation and triangle setup has been done for T, than when it has not, i.e.

$$C^{T(k_{uf},k_{vf})} + \sum_{i=1}^{n} C_f^{T(k_{uf},k_{vf})}(i) < \sum_{i=1}^{n} C_f(i)$$

Where $C^{T(k_{uf},k_{vf})}$ is the cost of triangle classification, C_f the cost of fragment classification and $C_f^{T(k_{uf},k_{vf})}$ the cost of fragment classification taking into account triangle-wide calculations reused for the fragment.

We have tested triangle classification with a uniform $t_u = t_v$ tessellation factor for plane surfaces, due to their large quantity in CAD models (see Table 1), since the tessellation factor has no impact on the screen space error, and because we can reliably calculate the number of fragments early in the rendering pipeline. We enable extra tessellation only when triangles occupy on average at least 200 fragments on the screen. This is enough to considerably enhance the performance. For instance, View 1 of the Piston model runs twice as fast with triangle classification for planar surfaces than without.

7. Implementation

CAD models consist of a large variety of basis surfaces. As demonstrated by Toledo and Levy [TL08] raycasting is well suited to CAD visualization. It produces high quality and accurate rendering, with an increase of performance for simple basis surfaces, comparable to (and sometimes challenging) tessellation. Nevertheless, raycasting is computationally heavy for complex basis surfaces. Our data structure being orthogonal to the rendering algorithm, our rendering engine can either tessellate all basis surfaces, or raycast simple surfaces and tessellate on the fly more complex ones. We implement raycasting for spheres, cylinders and cones ; torus, fillets and cubic Bézier patches have dedicated tessellation shaders. Other basis surface types such as revolution and extrusion surfaces are first converted into cubic Bézier patches [LM87, LM88], respecting a predefined world-space approximation error.

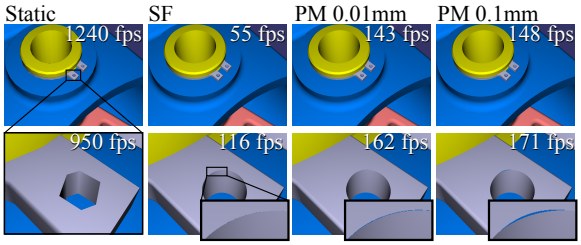


Figure 12: *Rendering quality for the piston model. Static: 0.1mm static tess., SF: Schollmeyer and Fröhlich, PM: proposed method with $\varepsilon = 0.01mm$ and $0.1mm$. Dynamic tessellation rendering using $\varepsilon_s = 0.1$ screen pixel error.*

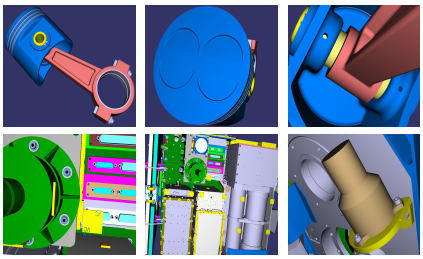


Figure 13: *Views used for performance evaluation for the Piston (top) and Satellite (bottom) models.*

When tessellation is used, each basis surface is delivered to the tessellation shaders as a single patch, with surface data needed for tessellation read off of GPU texture memory. We use two tessellation factors by basis surface, one along each dimension of the parametric domain. The tessellation factor supported by current hardware is limited to 64, creating at most 4096 vertices for a single basis surface patch. Our tessellation strategy is governed by a screen-space error ε_s defining the maximum deviation between the actual basis surface and its discretization and by the tessellation factors t_u and t_v (Section 6).

8. Results

We use three CATIA V5 models for our tests, a piston, a satellite and an airplane section. Table 1 gives details about surface types and trimming curve types for these models.

Visual quality: Our rendering method provides very high visual accuracy with close-ups. While static tessellation cannot handle both large scale, global object visualization as well as small scale feature-level rendering, with a limited amount of memory, our approach can display smooth trimming curves, see Figure 12.

Performance: Table 2 shows performance results obtained with our method and the one proposed by Schollmeyer et al. Tessellation is performed with the Datakit CATIA file reading API (`www.datakit.com`). In this test, we set the tol-

Basis surface type	Satellite	Airplane	Piston
Planes	36518	276788	210
Cylinders	41832	295262	390
Cones	9722	20488	26
Spheres	667	1595	24
Tores	6517	34266	66
Fillets	166	22125	74
Nurbs	853	20871	0
Other type	173	30669	0
Total basis surfaces	96448	702064	790
Cubic Bézier	1271	71894	0
Total patches	96693	722418	790

	Trimming curves		
Curve type	Satellite	Airplane	Piston
Line	354197	2767813	2680
Elliptical arc	58797	438295	294
Total Nurbs	58236	496808	606
Nurbs degree 1	12479	2251	0
Nurbs degree 2,3,4	98	407	8
Nurbs degree 5	45571	493723	598
Nurbs degree 6,7,8,11	88	427	0
Total curves	471230	3702916	4186
Quadratics (0.1mm)	690140	5975794	5695
Quadratics (0.01mm)	941476	8661440	7854

Table 1: *Model information broken down by basis surface type and trimming curve type. The Cubic Bézier line counts the number of cubic Bézier surface patches that are created as a result of a 0.1mm approximation of the original basis surfaces. Any surface which type is not one of the six primitive types (plane, cone, cylinder, tore, sphere or fillet) has to be approximated into one or more patches. Otherwise, each basis surface gives birth to one patch.*

erance to 0.1mm for Bézier surfaces and fillet curves approximation. We set the trimming curve quadratic approximation so that it matches either a 0.1 or 0.01mm world-space error. The results are obtained with a GeForce GTS 450 GPU with 1GB of video RAM, on an Intel Core i7-860 CPU with 8GB of RAM. For our method, they show a performance gain varying between 10% and 240% depending on the view or model used.

Memory usage: Statically tessellated models require memory space that is proportional to the discretization factors. For surface data, the memory requirements of the proposed approach is affected by the error tolerance used to approximate complex surfaces with cubic Bézier patches, and fillet curves with cubic Bézier curves. The memory usage of trim structures is affected by the quadratic approximation error used for trimming curves. Table 3 presents memory usage for both our method and Schollmeyer's, as well as for statically tessellated models. With an acceptable trimming curve approximation error of ε = 0.1mm, we consume 30-45% less memory than Schollmeyer et al. With ε = 0.01mm, we roughly equal their memory consumption.

Limitations: On some rare occasions, the cell subdivision process might degenerate when three or more quadratic Bézier curves are very close, which mostly happens when a quadratic curve extremity of a trimming loop is very close to another trimming loop (Figure 14 left). In this case, cell subdivision goes very deep as the quadratic curve extremity creates a multi-curve situation which we cannot handle

	Rendering time (ms)					
	(a)	(b)	(a)	(b)	(a)	(b)
Airplane	View 1		View 2		View 3	
SF	145	130	115	101	50	56
P.M. 0.1mm	108	98	103	92	38	37
P.M. 0.01mm	109	99	93	92	40	37
Satellite	View 1		View 2		View 3	
SF	44	28	45	31	87	22
P.M. 0.1mm	34	21	30	21	65	14
P.M. 0.01mm	33	20	30	19	65	15
Piston	View 1		View 2		View 3	
SF	15	8.3	31	15	53	25
P.M. 0.1mm	8.8	3.1	19	4.5	31	8
P.M. 0.01mm	8.9	3.1	19	4.6	33	8

Table 2: *Frame rendering time in ms for our proposed method (P.M.) and Schollmeyer and Fröhlich's (SF). Model views used are from Figures 2 and 13. Surface approximation ε = 0.1mm; Screen-space tessellation error ε_s = 0.5 pixel. Results are presented using either raytracing and tessellation **(a)** or tessellation only **(b)***

Airplane	B.surface	Trimming	Total
SF	64	296	360
P.M. 0.1mm	64	197	261
P.M. 0.01mm	64	294	358
Discretization 0.1mm			246
Discretization 0.01mm			1010
Satellite	B.surface	Trimming	Total
SF	10.5	48.1	58.6
P.M. 0.1mm	10.5	31	41.5
P.M. 0.01mm	10.5	43.7	54.2
Discretization 0.1mm			42.5
Discretization 0.01mm			139.3
Piston	B.surface	Trimming	Total
SF	0.119	0.656	0.775
P.M. 0.1mm	0.119	0.309	0.428
P.M. 0.01mm	0.119	0.440	0.559
Discretization 0.1mm			0.629
Discretization 0.01mm			2.683

Table 3: *Memory usage, in megabytes, for Schollmeyer and Fröhlich (SF), our proposed method (P.M.), and statically tessellated models (for which each vertex takes 36 bytes, accounting for the space used by vertex coordinates, normal and color, each defined by three floating point values and each triangle uses three 32-bit vertex indices). Surface approximation is in all cases done using a tolerance of 0.1mm.*

Authors versionThe definitive version is available at diglib.eg.orgdoi: 10.2312/PE/VMV/VMV12/031-038

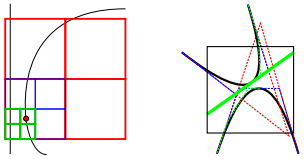


Figure 14: *Left: the red circle identifies a junction point between two quadratic curves. Quadtree subdivision sometimes has to go a long way before converging to a 2-or-less curve situation. Right: quadtree node with 2 overlapping curves. A suitable separating line is found after 3 control polygon subdivisions.*

Airplane	0.1mm	0.01mm
Number of patches	365	668
% of total patches	0.12	0.22
Avg. nodes per patch	2.5	4.4
Satellite		
Number of patches	147	333
% of total patches	0.23	0.5
Avg. nodes per patch	1.2	3
Piston		
Number of patches	0	0
% of total patches	0	0
Avg. nodes per patch	0	0

Table 4: *Number of patches for which the quadtree node level reached 13 and we gave up subdivision (See Figure 14 left). The table also shows the average number of problematic nodes per problematic trimming structure, eg. 1.2 means that, on average, a problematic trimming structure has 1.2 problematic node.*

in a single cell. This situation occurs mainly because we do not control the location of the junction points between adjacent quadratic segments during the approximation process. Table 4 shows this situation is quite rare though, even for the very large airplane model.

The method we use for calculating the separating line (Section 5) may take some time to converge to a solution. When the curve segments are very close to each other, control polygon subdivision has to go a long way before a suitable separating line is found (Figure 14 right).

9. Conclusion and future work

We have presented a rendering method based on the direct trimming of either raytraced or dynamically tessellated B-Rep surfaces. The multiresolution nature of our quadtree trimming structure and the availability of the coverage value allow us to perform shallow lookups when possible, when trimming is done on surfaces distant from the observer, increasing performance in situations where a large number of distant fragments are to be drawn. When tessellation is used for rendering, we propose a method to leverage dynamic tessellation to speed up the classification of large sections of screen fragments for object close-ups.

We have tested the latter technique for plane faces. Further work needs to be done to find a reliable generalization for any type of basis surface. In particular, cylinders, cones and other surface types typically tessellated into $[1,n]$ subdivisions can have a rendering performance that suffers from the extra tessellation in some situations. Our tests show that the tessellation factor being aligned on a power of two scale that follows the quadtree structure is not always beneficial to performance if we cannot reliably estimate at runtime the number of fragments being rendered for a particular primitive. A heuristic could be worked out to find the optimal tessellation factors for a surface in a view-dependent context so that the overall surface classification is the most efficient, taking into account the number of fragments and the complexity of the quadtree structure. Our method may also be applied to subdivision-based tessellation – a process our quadtree should be friendly with. It might also be possible to adapt our algorithm or structure so that triangles can be classified efficiently at runtime in a wider range of situations, not just when they are known to fit in a single quadtree node.

References

[Ebe06] EBERLY D. H.: *3D Game Engine Design, Second Edition: A Practical Approach to Real-Time Computer Graphics (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. 4

[GBK05] GUTHE M., BALÁZS A., KLEIN R.: Gpu-based trimming and tessellation of nurbs and t-spline surfaces. *ACM Trans. Graph. 24* (July 2005), 1016–1023. 2

[HH11] HANNIEL I., HALLER K.: Direct rendering of solid cad models on the gpu. In *Computer-Aided Design and Computer Graphics (CAD/Graphics), 2011 12th International Conference on* (Sept. 2011), pp. 25 –32. 3

[LB05] LOOP C., BLINN J.: Resolution independent curve rendering using programmable graphics hardware. *ACM Trans. Graph. 24* (July 2005), 1000–1009. 3

[LM87] LYCHE T., MØRKEN K.: Knot removal for parametric b-spline curves and surfaces. *Computer Aided Geometric Design 4*, 3 (1987), 217 – 230. 3, 6

[LM88] LYCHE T., MØRKEN K.: A data-reduction strategy for splines with applications to the approximation of functions and data. *IMA Journal of Numerical Analysis 8*, 2 (1988), 185–208. 3, 6

[NH08] NEHAB D., HOPPE H.: Random-access rendering of general vector graphics. *ACM Trans. Graph. 27* (December 2008), 135:1–135:10. 3

[RNCL05] RAY N., NEIGER T., CAVIN X., LÉVY B.: *Vector Texture Maps*. Tech. rep., INRIA - ALICE, 2005. 3

[SF09] SCHOLLMEYER A., FRÖHLICH B.: Direct trimming of nurbs surfaces on the gpu. *ACM Trans. Graph. 28* (July 2009), 47:1–47:9. 3

[SS09] SCHWARZ M., STAMMINGER M.: Fast gpu-based adaptive tessellation with cuda. *Computer Graphics Forum 28*, 2 (2009), 365–374. 1

[TL08] TOLEDO R., LEVY B.: Visualization of industrial structures with implicit gpu primitives. In *Proceedings of the 4th International Symposium on Advances in Visual Computing* (Berlin, Heidelberg, 2008), ISVC '08, Springer-Verlag, pp. 139–150. 1, 6