# Correct resolution rendering of trimmed spline surfaces

Ruijin Wu[a], Jorg Peters[a]

[a] *University of Florida*

**Abstract**

Current strategies for real-time rendering of trimmed spline surfaces re-approximate the data, pre-process extensively or introduce visual artifacts. This paper presents a new approach to rendering trimmed spline surfaces that guarantees visual accuracy efficiently, even under interactive adjustment of trim curves and spline surfaces. The technique achieves robustness and speed by discretizing at a near-minimal correct resolution based on a tight, low-cost estimate of adaptive domain griding. The algorithm is highly parallel, with each trim curve writing itself into a slim lookup table. Each surface fragment then makes its trim decision robustly by comparing its parameters against the sorted table entries. Adding the table-and-test to the rendering pass of a modern graphics pipeline achieves anti-aliased sub-pixel accuracy at high render-speed, while using little additional memory and fragment shader effort, even during interactive trim manipulation.

*Keywords:* trimming, spline, accurate, real-time, scan density

## 1. Introduction

A standard approach to designing geometry in computer aided design, is to "overfit", i.e. create spline surfaces that are larger than needed and subsequently trim the surfaces back to match functional constraints or join to other surfaces (see Fig. 1). This approach persists both for historical reasons and for design simplicity: for historical reasons in that overfitting and trimming pre-dates alternative approaches such as subdivision surfaces [1, 2] and finite geometrically-smooth patch complexes (see e.g. [3, 4]); for practical reasons, in that it is often more convenient to control the shape of a signature piece in isolation than when constraints have to be taken into consideration. For example, a car's dashboard can be prepared in one piece without consideration of cut-outs for instrumentation and the steering column.

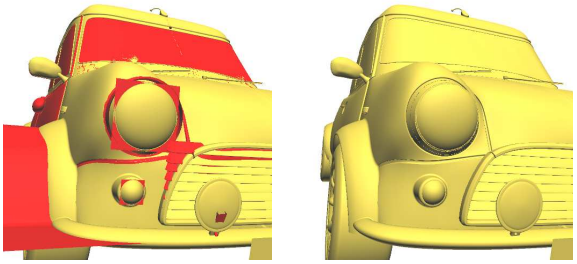


Figure 1: Geometric design with trimmed surfaces. The red spline pieces need to be trimmed away. See also Fig. 2.

The prevailing practice in computer aided design environments is to generate and display a fixed-resolution triangulation on the CPU and transfer it to the GPU. This process interrupts the design process and can yield unsatisfactory results as close-ups reveal a jagged or otherwise incorrect approximation. Conversely, an overly fine triangulation wastes resources: there is no need to highly resolve a complex trim curve when the corresponding surface measures only a few pixels on the screen.

The computer graphics community has developed a number of clever techniques, reviewed in Section 2, to deliver real-time display of trimmed spline surfaces. The present paper advances the state-of-the-art by carefully *predicting how fine an evaluation of the trim curves results in correct trim decisions at screen resolution*. This tight prediction makes it possible to construct, as a prelude to each modified view or model rendering pass, a slim and adaptive trim-query acceleration table that supports a light-weight per-fragment trim test. This simple add-on to any rendering pass is efficient enough to allow interactive trim-curve editing.

*Overview.* Section 2 reviews existing techniques for fast rendering of trimmed spline surfaces. Section 3 reviews basic concepts and establishes notation. Section 4 explains how correct resolution can be determined. Section 5 explains how to build and use the trim-query acceleration table. Section 6 measures the performance of a full implementation.

## 2. Real-time rendering of trimmed surfaces and related techniques

The *trim decision* is to determine to which side, of a set of trim curves, lies the $uv$-pre-image of a pixel. The underlying challenge is the same as when determining the fill region of a planar decal [5] – except that in planar filling, the accuracy of rendering is measured in the $uv$-plane, while for trimmed

*Email addresses:* ruijin@cise.ufl.edu (Ruijin Wu), jorg@cise.ufl.edu (Jorg Peters)

surfaces, the accuracy is measured in screen space, i.e. after applying the non-linear surface map followed by projection onto the screen.

A straightforward approach, used in 2D vector graphics [6, Sec 8.7], is to *ray-test*: each pixel's $uv$-pre-image in the domain sends a ray to the domain boundary to determine the number of intersections (and possibly the intersection curve orientation). The intersections decide whether the fragment is to be discarded. For example, Pabst et al. [7] test scan-line curve intersection directly in the Fragment Shader. Direct testing without an acceleration structure is impractical since the number and complexity of intersections can be unpredictably high so that robustness and accuracy are difficult to assure within a fixed time. It pays to pre-process the trim curves and map them into a hierarchical search structure in order to localize testing to a single trim curve segment. For example, Schollmeyer et al. [8] break the segments into monotone pieces and test scan-line curve intersection in the Fragment Shader by robust binary search. This pre-processing is view-point independent but becomes expensive for interactive trim-curve manipulation.

An alternative is to generate a *trim texture*: the $uv$-pre-image of each fragment indexes into a texture that returns whether the point is to be trimmed or not. Such a trim texture can be generated in a separate rendering pass, using the stencil buffer [9]. The trim-test is highly efficient, requiring only a single texture look-up to classify a domain point. However trim textures need to be recomputed for every viewpoint change and the separate pass can noticeably lower overall performance as each segment of every trim curve generates and atomically inserts a triangle into the stencil buffer. Moreover, the trim-texture represents a uniform, limited resolution sampling of the $uv$-domain. Projective fore-shortening must be accounted for separately: when rendering a curved surface in 3-space, the non-uniform distortion of the domain caused by the non-linear map of the surface followed by perspective projection can result in low render quality even where the texture resolution is high.

Another pre-processing choice is to convert the piecewise rational trim curves into an implicit representation, via resultants (see e.g. [10, 11, 12]). Evaluating the resultant will generate a signed number and the sign can be used to determine whether a pixel is to be trimmed. In principle, this yields unlimited accuracy. However, there are several caveats to this approach. First, the use of resultants increases the degree and number of variables. The coefficients of the implicit representation are typically complicated expressions in terms of the coefficients of the trim curve segments. Therefore the evaluation in the Fragment Shader can be expensive even if the derivation of the implicit expression is done off-line prior to rendering. There are more efficient approaches than full implicitization, e.g. [13]. While useful for ray-tracing, these expressions do not presently yield a signed test as required for trimming. Second, implicitization converts the entire rational curve, not just the required rational *piece*. Use of resultants therefore requires a careful restriction of the test region, for example by isolating bounding triangles in the domain that contain a single indicator function whose zero level set represents the trim. Determining such restrictions is in general tricky since the implicit can have extraneous branches. For conics, the conversion expressions are sufficiently simple and for fixed shapes, such as fonts, determining bounding triangles can be done offline once and for all [14]. For less static scenarios, also pre-processing of conics and triangulation of the domain is not easily parallelized. Stencil buffers can avoid careful triangulation [5] but the fixed resolution inherits the challenges of texture-based trimming.

Computing the trim curves from CSG operations addresses a related but somewhat different problem than rendering. Here the trim curves (exact intersection pre-images) are not given. For simple CSG primitives such as quadrics the render decision can be based on an implicit in/out test. For more complex B-reps, faceted models are compared and proper resolution of the B-rep into facets remains a challenge. Practical implementations use stencil operations, depth and occlusion testing [15, 16].

Figure 2: Correct resolution real-time rendering of trimmed spline surfaces.

## 3. Definitions and Concepts

*Coordinates and Projection.* In the OpenGL graphics pipeline [17, Sec 13.6], the non-orthogonal projection $P$

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \rightarrow \begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} := \begin{pmatrix} P_{11} & 0 & 0 & 0 \\ 0 & P_{22} & 0 & 0 \\ 0 & 0 & P_{33} & P_{34} \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

maps camera coordinates $(x, y, z, 1)^T$ with the camera is at the origin pointing in the negative $z$-direction, to clip coordinates $(x_c, y_c, z_c, w_c)^T$. The entries $P_{33} := (\overline{z}-\underline{z})/(\overline{z}+\underline{z})$ and $P_{34} := 2\overline{z}\underline{z}/(\underline{z}-\overline{z})$ define two planes at depth $\underline{z}$ (near) and $\overline{z}$ (far) such

that any geometry with depth outside the range $[\underline{z}, \overline{z}]$ is clipped. Perspective division converts the clip coordinates into normalized device coordinates $(x_d, y_d, z_d) := (x_c, y_c, z_c)/w_c$, and the viewport transformation converts normalized device coordinates to screen coordinates: $(\tilde{\mathtt{x}}, \tilde{\mathtt{y}}) := (Wx_d/2 + O_x, Hy_d/2 + O_y)$. Here $W$ and $H$ are the width, respectively height of the viewport in pixels and $O_x$ and $O_y$ are the screen space coordinates of the viewport center, typically $O_x = W/2$, $O_y = H/2$. Together, the projection from $\mathbb{R}^3$ to the rasterized screen is

$$\begin{aligned} &\mathtt{P} : \mathbb{R}^3 \to \mathbb{R}^2 \qquad w_x := \frac{P_{11}W}{2}, w_y := \frac{P_{22}H}{2}, \\ &(x, y, z) \to (\tilde{\mathtt{x}}, \tilde{\mathtt{y}}) := \left( w_x \frac{x}{z} + c_x, w_y \frac{y}{z} + c_y \right). \end{aligned} \tag{1}$$

*Surfaces and trim curves.* Models consist of rational parametric *surfaces*

$$\mathbf{x} : U \subsetneq \mathbb{R}^2 \to \mathbb{R}^3 \qquad (u, v) \to \mathbf{x}(u, v) := \begin{pmatrix} x(u, v) \\ y(u, v) \\ z(u, v) \end{pmatrix} \tag{2}$$

defined on a domain $U$. Often $U := [0..1]^2$, the unit square, and the surface (patch) is in tensor-product form with basis functions $b_k$,

$$\mathbf{x}(u, v) := \sum_i \sum_j \mathbf{c}_{ij} b_i(u) b_j(v).$$

To *trim* a patch means to restrict its domain to one side of a piecewise rational *trim curve*

$$\gamma : T \subsetneq \mathbb{R} \to U \qquad t \to \gamma(t) := (u(t), v(t)). \tag{3}$$

There can be multiple trim curves per patch. When trim curves are nested, their orientation can be used to determine which side is trimmed away. A simplified convention applies an alternating count from a domain boundary.

Trim curves are often rational approximations $\gamma_1(t)$, $\gamma_2(t)$ to the solutions of the surface-surface intersection (SSI) problem, $\mathbf{x}_1(u_1, v_1) = \mathbf{x}_2(u_2, v_2) \in \mathbb{R}^3$. We are not concerned with solving such potentially hard SSI systems of 3 equations in 4 unknowns, whose solution is generically a pair of non-rational algebraic curves. Rather we assume the trim curves are given as rational curves and address the challenge of efficient accurate display of the resulting trimmed surface.

Trim curves can also be artist-defined or they can reference planar shapes. Such shapes, for example fonts, are mapped to the surface reshaped by the curvature of the map $\mathbf{x}$.

*Faithful surface tessellation.* Ray-casting guarantees that each pixel represents a correctly-placed surface piece and finds each pixel's $uv$-pre-image. Remarkably, such pixel-accurate rendering can also be achieved using the highly efficient standard graphics pipeline. In [18], a compute shader determines a near-optimal (minimal) tessellation factor $\tau$ for each patch $\mathbf{x}$, so that evaluating $\mathbf{x}$ on a $\tau \times \tau$ partition of its domain $U$ yields a proxy triangulation $\hat{\mathbf{x}} \in \mathbb{R}^3$ of $\mathbf{x}$ such that its image under the screen projection $\mathtt{P}$ of (1) agrees with that of the correct image. That is the difference $\mathtt{P}\mathbf{x}(\mathtt{u}) - \mathtt{P}\hat{\mathbf{x}}(\mathtt{u})$ is below the visible pixel threshold. We call $\hat{\mathbf{x}}$ a faithful triangulation of $\mathbf{x}$ and will use $\hat{\mathbf{x}}$ to render the trimmed surfaces.


(a) insufficient resolution (b) correct resolution

Figure 3: Choosing a *correct* domain partition into $uv$-cells. i.e. sufficiently fine so that no two pixels' $uv$-pre-images share one $uv$-cell.

## 4. Predicting correct resolution

Since any screen has a fixed discrete resolution, trimmed surfaces can and need only be resolved up to pixel width (subpixel width in the case of subsampling for anti-aliasing). Our approach is to pull back the pixel-grid to the domain $U$ and partition $U$ into $uv$-cells – in such a way that no two pixels' $uv$-pre-images share one $uv$-cell (cf. Fig. 3). This guarantees each pixel, in particular of a group straddling the projection of a surface trim, its individual trim test. Next we evaluate the trim curve in the domain $U$ so that the resulting broken line differs from the exact trim curve by less than one $uv$-cell. The two discretizations, of the domain $U$ and of the curve $\gamma$, provide a consistent, **correct resolution** in the sense that testing the $uv$-pre-image of a pixel $(\tilde{\mathtt{x}}_i, \tilde{\mathtt{y}}_i)$ against the broken line yields a correct trim decision up to pixel resolution.

For a non-linear surface $\mathbf{x}$ followed by a perspective projection $\mathtt{P}$, pulling the pixel-grid back exactly is not practical since the Jacobian of $\mathtt{P}\mathbf{x}$ can vary strongly. Multi-resolution can accommodate adaptively scaled $uv$-cells, but a complex hierarchical lookup slows down each fragment's trim test. We therefore opt for a simple two-level hierarchy. At the first level, we partition the domain $U$ into $n_V$ $v$-strips $V_i$:

$$V_i := \{(u, v) \in U, \quad \nu_i \le v < \nu_{i+1}\}. \tag{4}$$

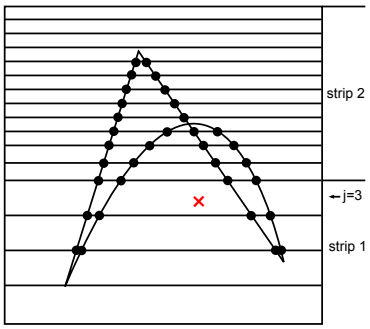
Each $v$-strip is a 'fat scan-line' where the $u$-coordinate is free while the $v$-coordinate is sandwiched between an upper and a lower bound. The default choice is to space the $v_i$ uniformly and set $n_V = \tau$, where $\tau$ is the surface tessellation factor already computed according to [18]. At the second level, each $v$-strip $V_i$ is uniformly partitioned once more in the $v$-direction into a minimal number $\mu_i$ of $v$-scan lines that guarantees correct trim resolution (see Fig. 4). To determine the critical number $\mu_i$ that guarantees that the distance between the screen images of two $v$-scan lines is less than one pixel, we proceed as follows. By the mean value theorem, for some $v^* \in [v, v + h]$,

$$|\tilde{\mathtt{x}}(u, v) - \tilde{\mathtt{x}}(u, v + h)| = h \, |\tilde{\mathtt{x}}_v(u, v^*)|, \qquad \tilde{\mathtt{x}}_v := \frac{\partial \tilde{\mathtt{x}}}{\partial v}. \tag{5}$$

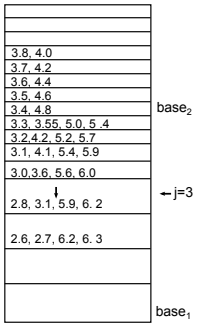
If, for all $v \in V_i$ and the $v$-scan line-spacing $h > 0$,

$$\begin{aligned} &h \, \rho_i(v) < 1, \qquad v \in [\nu_i..\nu_{i+1}], \\ &\rho_i(v) := \max\{\sup_u |\tilde{\mathtt{x}}_v(u, v)|, \sup_u |\tilde{\mathtt{y}}_v(u, v)|\}, \end{aligned} \tag{6}$$

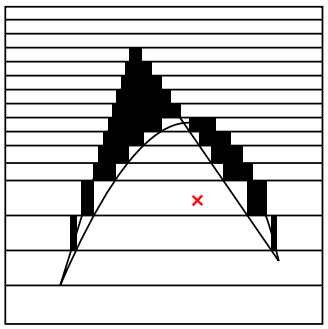
then the $\tilde{\mathtt{x}}$-distance between the screen images of the two $v$-scan lines $\tilde{\mathtt{x}}(u, v_j)$ and $\tilde{\mathtt{x}}(u, v_j + h)$ is less than a pixel and so is the



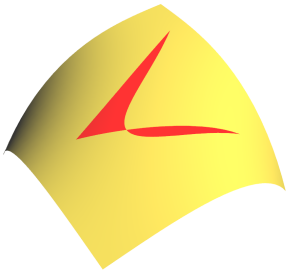

(a) Trim curve in $uv$ space and $v$-scan line intersections for two $v$-strips




(b) $u$-intercept table



(c) Adaptive resolution of trimmed region



(d) Trimmed region (red) on patch at correct resolution.

Figure 4: **Construction and testing of the $u$-intercept table.** (b) Two $v$-strips $V_1$ and $V_2$ partitioned into $\mu_1 = 4$ and $\mu_2 = 12$ $v$-scan lines respectively. (c) The $\leftarrow$ and $\downarrow$ indicate the $v$-scan line and the relative $u$ coordinate of x in the $u$-intercept table. (d) In the absence of orientation information, the parity of the table entry just smaller than $u$ decides the trim-query.

$\tilde{\mathrm{y}}$-distance. Therefore setting

$$\mu_i := \lceil (\nu_{i+1} - \nu_i) \sup_{v \in [\nu_i .. \nu_{i+1}]} \rho_i(v) \rceil \tag{7}$$

guarantees correct resolution. Fig. 5a shows an artifact when $\mu_i$ is chosen too small. Fig. 5c shows the correct and near-minimal choice. If the projection is orthographic and the surface is a faithful triangulation then $\mu_i$ is simply the maximum size of the $\tilde{\mathrm{x}}$ and the $\tilde{\mathrm{y}}$ projection of the strip in pixels.



(a) insufficient scan resolution $\mu_i$ (b) insufficient curve tessellation (c) correct resolution

Figure 5: Render quality affected by scan density (jaggies in (a)) and curve tessellation level (corners (b)). Figures are enlarged.

*Estimating $\rho_i$.* Since for any reasonable view the near-plane of the scene is at some minimal distance to the viewer, i.e. $z \geq \underline{z} > 0$, the expansion

$$\tilde{\mathrm{x}}_v = \tilde{\mathrm{x}}_x x_v + \tilde{\mathrm{x}}_y y_v + \tilde{\mathrm{x}}_z z_v = \frac{w_x}{z}(x_v - \frac{x}{z} z_v) \tag{8}$$

is well-defined, although potentially large for small $z$. For our (faithfully) triangulated surface, $x$ and $z$ are piecewise linear maps and $x_v$ and $z_v$ are piecewise constant. Determining an upper bound on $\rho_i(v)$ over the three vertices $(u_k, v_k)$, $k = 1, 2, 3$ of each triangle $\triangle$,

$$\rho_\triangle := \max_k \left|\frac{1}{z}\right| \max_k \left| w_x (x_v - \frac{x}{z} z_v) \right|,$$

and setting $\mu_i$ to the per-strip maximum,

$$(\nu_{i+1} - \nu_i) \max_\triangle \rho_\triangle, \qquad \triangle \cap V_i \neq \emptyset,$$

typically yields a tight estimate. However, the estimate $\rho_\triangle$ can include (parts of) triangles outside the viewing frustum so that, if a user zooms in and a part of a triangle approaches the camera, $z$ can be arbitrarily small. Instead, using the pixel's $uv$-preimage sent down through the graphics pipeline, we compute the analog of (8) for each pixel $\alpha$ as

$$\rho_\alpha := \max\{|\tilde{\mathrm{x}}_v(u_\alpha, v_\alpha)|, |\tilde{\mathrm{y}}_v(u_\alpha, v_\alpha)|\} \tag{9}$$

and set

$$\mu_i := (\nu_{i+1} - \nu_i) \max_\alpha \rho_\alpha, \qquad \alpha \in V_i. \tag{10}$$

Now parts of triangles outside the viewing frustum do not contribute; and if a $v$-strip $V_i$ lies outside the viewing frustum, it receives no votes $\rho_\alpha$ in (10) and $\mu_i = 0$. That is, $\mu_i$ provides a correct upper bound on $\rho_i(v)$ for all *visible samples*; and that is exactly what is needed for correct resolution rendering.

## 5. Filling and using the acceleration table

The overall algorithm is summarized as follows. Each trim curve writes itself, at the *correct resolution* (determined by the $v$-scan line density $\mu_i$ defined in the previous section) into a slim $u$-intercept table. Each fragment then makes its trim decision by testing against the table entries.

*Initializing the $u$-intercept table.* The key data structure is the $u$-intercept table. There is one $u$-intercept table per sub-surface, e.g. a tensor-product spline patch complex with $\ell_u \times \ell_v$ pieces. The $u$-intercept table is an array of size $\mu \times n$ where $\mu := \sum_i \mu_i$ is the total number of $v$-scan lines and $n$ is an upper bound on the number of trim curves that may cross a $v$-scan line of the surface piece. The row of the table with index

$$\mathrm{base}_i + j, \quad \mathrm{base}_i := \sum_{k=1}^{i-1} \mu_k, \quad j := \frac{v - \nu_i}{\nu_{i+1} - \nu_i} \mu_i, \tag{11}$$

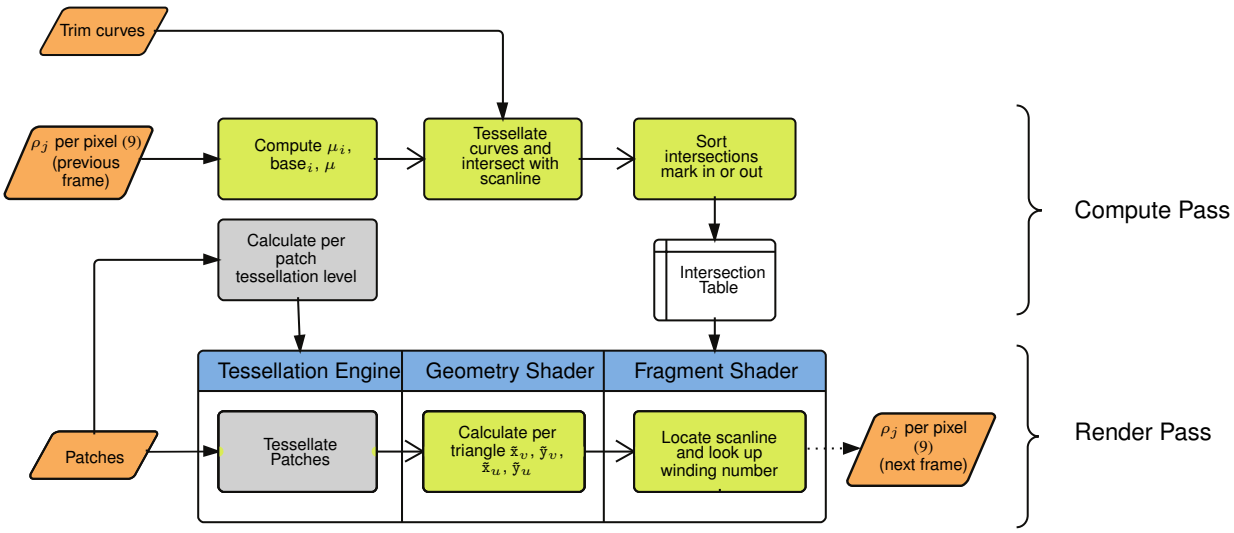



Figure 6: Flowchart of the surface trimming add-on (light green) to pixel-accurate GPU rendering according to [18].

will contain a sorted list of $u$-intercepts: the intersections of the (linear segments of the correctly tessellated) trim curves $\gamma$ with the $v$-scan line $(u, v) : v = \nu_i + \frac{j}{\mu_i}(\nu_{i+1} - \nu_i)$. Each $v$-strip $V_i$ has its own number of rows $\mu_i$ computed from the per-pixel densities $\rho_\alpha$ by a prefix sum and (10). The per-pixel densities $\rho_\alpha$ are computed via (8) at the end of the rendering pass before the next compute pass: The Fragment Shader has access to $x$ and $z$ and, the Geometry Shader provides the Fragment Shader with the Jacobian of each fragment's triangle.

*Filling the $u$-intercept tables.* (see Fig. 4) Intersecting $v$-scan lines with a piecewise rational trim curve can be tricky. However, the required, correct $v$ resolution of the trim curve, $\min_i(\nu_{i+1} - \nu_i)/\mu_i$, is known. Applying formula (9) with $\tilde{\mathrm{x}}_v, \tilde{\mathrm{y}}_v$ replaced by $\tilde{\mathrm{x}}_u, \tilde{\mathrm{y}}_u$ yields a $u$-evaluation density that guarantees correct resolution. (Fig. 5b illustrates insufficient trim curve tessellation.) The bounds of [19] could be leveraged, but since the Compute Shader already uses slefe-based bounds to generate the faithful triangulation $\hat{\mathbf{x}}$ according to [18], we use the same estimators to determine the curve tessellation factor for correct resolution (as defined in Section 4). For the correct curve tessellation number, the Compute Shader threads calculate, for each trim curve segment $k$ in parallel, the end points $\gamma(t_{k-1})$ and $\gamma(t_k)$. Each thread then inserts the $u$-coordinates of all $v$-scan line-intersections with the trim curve segment into the $u$-intercept table. A second generation of threads subsequently sorts all trim curve intersections of a $v$-scan line by their $u$-coordinates.

*Testing against the $u$-intercept table.* To determine whether a fragment with pre-image $(u, v)$ should be discarded, the Fragment Shader reads the row of the table determined from $v$ by (11) and locates $u$, by binary search in the table (see Fig. 4b). The complexity of the binary search is $\log_2 n$, i.e. 4 tests if the number of intercepts is $n = 16$. The Fragment Shader then makes the trim decision based on the parity of the entry just smaller than $u$ (and possibly orientation hints).

### 5.1. Implementation detail and optimization

Fig. 6 shows our mapping of accurate trim display onto the graphics pipeline. The main add-on is the Compute Shader pass, shown as the top row of the flowchart. The trim test is performed in the Fragment Shader. The Geometry Shader can be removed by adding its work to the Fragment Shader.

*Consistent intersections.* Care has to be taken to avoid duplicate or missing intersections where the $v$-scan line intersects two consecutive trim curve segments at the common end point. If the $v$-scan line does not cross but just touches the two segments, either two or zero intersections should be recorded, not one. We achieve consistency by treating every segment as a bottom-open top-closed interval (see Fig. 7).

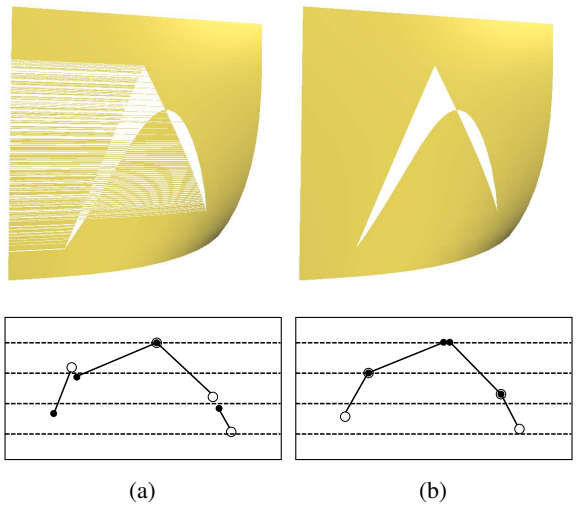
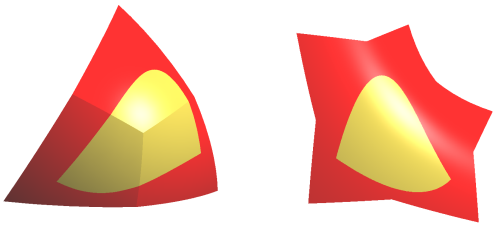



Figure 7: (a) Artifacts due to duplicated/missing intersections of curve segments that end at $v$-scan lines. (b) Choosing segments bottom-open top-closed prevents artifacts.

*Merging $u$-intercept tables.* To minimize overhead in launching a compute shader and memory allocation we can and do

group together the $u$-intercept tables of several sub-surfaces, such as several tensor-product NURBS patches (see next).

*Jointly Trimming multiple patches.* The grouping in the previous paragraph was aimed at improving performance. It also makes sense to have collections of patches in irregular layout share one domain. For example, when patches $\mathbf{x}_k : U_k \subsetneq \mathbb{R}^2 \to \mathbb{R}^3$ are joined in a geometrically-continuous fashion their charts are related by a reparameterization $(\tilde{u}, \tilde{v}) : U_k \to \mathcal{U} \subsetneq \mathbb{R}^2$ that maps the individual domains to a joint region $\mathcal{U}$ in the plane. Our approach applies to this scenario with $(u, v)$ replaced by $(\tilde{u}, \tilde{v})$ and the joint trim curve specified in $\mathcal{U}$. Fig. 8 illustrates the use of such a joint domain for two multi-spline caps.

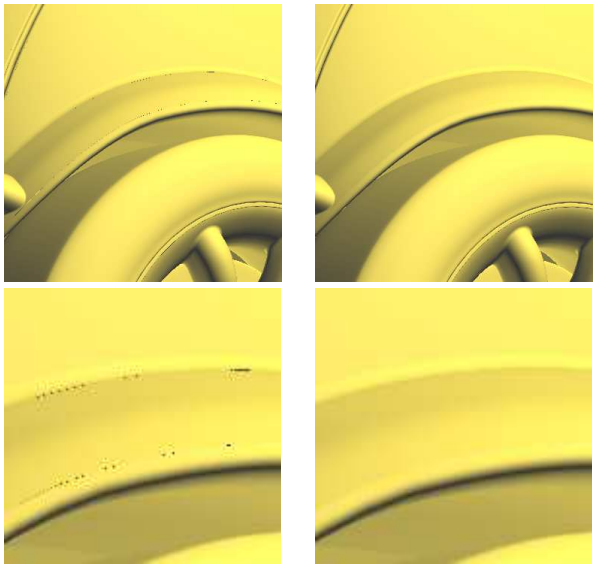


(a) 3-patch $C^0$ cap (b) 5-patch $G^1$ cap

Figure 8: Trimmed Bézier patches joining at a point. The part to be trimmed away is colored in red.

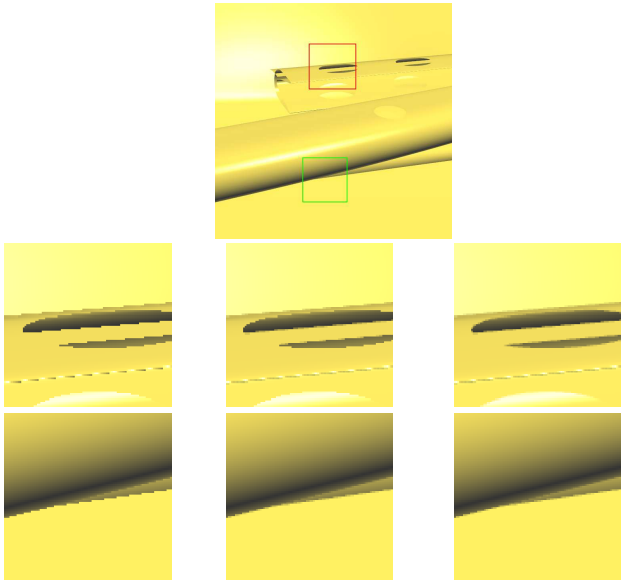
*Treating trim-data mismatches.* When a CAD modeling-kernel exports a trim derived from an intersection, the algebraic trim curves are approximated by rational trim curves whose images only match up to system-default or user-set tolerances. It is not the job of the rendering engine to fill such gaps and adjust SSI tolerances: correct resolution should display such gaps and let the designer know. However, *pixel dropout* can also result from more subtle sub-pixel resolution mismatches. When multiple surfaces partly cover a pixel, the pixel's sample location(s) may not be covered at all. Fig. 9a illustrates the concern. While this type of pixel dropout is tricky to deal with in the general setting [20], the faithful triangulation $\hat{\mathbf{x}}$ of our setup guarantees that these mismatches are of size less than one pixel. To cover these pixel-sized gaps, our algorithm additionally draws the patch trim boundaries $\hat{\mathbf{x}} \circ \gamma$ (as images of the already correctly tessellated trim curves) and so achieves correct coverage Fig. 9b.

*Multi-sample anti-aliasing.* ($m$-MSAA) improves silhouettes and (trim) boundaries by testing object coverage at $m$ locations per pixel (compare Fig. 10a to Fig. 10b where $m = 4$). We add sub-pixel trim testing by decreasing the screen spacing of $v$-scan lines in (6) so that the trim has sub-pixel accuracy. The Fragment Shader then corrects the coverage mask according to the per sub-pixel trim test result (see Fig. 10c). The main cost of $m$-MSAA is in creating finer $u$-intercept tables, not in the Fragment Shader test.



(a) Pixel-gaps due to resolution mismatch (b) After adding the images of the trim curves

Figure 9: Pixel-gap filling by drawing the patch trim boundary



66fps 55fps 43fps

(a) Without MSAA (b) 4xMSAA + per-pixel trim test (c) 4xMSAA + **sub**-pixel trim test

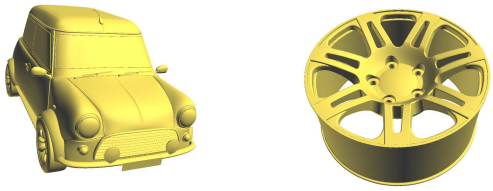
Figure 10: Anti-aliasing for silhouettes and trim boundaries. The screen occupation is 100%. The images are cropped and scaled. In (b, bottom), standard 4xMSAA correctly anti-aliases the silhouette (corresponding to the green square in the overview image on top), but not the boundary of the circular hole (middle, red square). In (c), our method correctly anti-aliases both silhouettes and trims.

*Opportunistic optimization.* Whenever the view is unchanged, neither the surface tessellation nor the $u$-intercept table need to be recomputed. Whenever the geometry is unchanged, neither

the surface slefe-boxes of the pixel-accurate patch rendering nor the trim curve partition need to be recomputed.

## 6. Results

We implemented the trim-render add-on in the OpenGL 4.3 API and benchmarked the implementation on the CAD models in Fig. 11. To make the benchmark easily replicable, we fixed the number of $v$-strips per spline surface to 128 and packed their $v$-scan lines into $u$-intercept tables of 16384 rows. We allocated space so that each row can hold up to $n = 32$ intersections (but never encountered more than 8 intersections; see Fig. 12).



(a) Mini (b) rim

Figure 11: Two models built from trimmed surfaces.

| | Mini | rim | Mini x 3 |
|---|---|---|---|
| Bézier Patches | 32,714 | 5,303 | 98,142 |
| Trim Curves | 8,390 | 7,227 | 25,170 |
| ms with trim | 6.57 | 3.20 | 17.22 |
| calculate patch tessellation level | 1.09 | 0.25 | 3.21 |
| *percentage* | *17%* | *8%* | *20%* |
| build $u$-intercept table | 1.33 | 0.79 | 3.15 |
| *percentage* | *20%* | *25%* | *18%* |
| render surfaces | 4.01 | 2.06 | 10.08 |
| *percentage* | *61%* | *52%* | *58%* |
| ms w/o trim | 4.72 | 2.15 | 12.81 |
| calculate patch tessellation level | 1.09 | 0.25 | 3.27 |
| render surfaces | 3.39 | 1.36 | 8.85 |

Table 1: Break down of the rendering time for the models in Fig. 11. The timings were collected for an i5-2500 3.3GHz processor and a GTX 780 graphics card, for a screen resolution of $1000 \times 1000$.

Table 1 breaks down the overall work and the work for maintaining the $u$-intercept table. To measure the maximally interactive case, we did not use opportunistic optimization. The full algorithm displayed by the flowchart of Fig. 6 was executed for each render pass. As observed in [8], a direct performance comparison to earlier implementations is challenging due to their implementation complexity and choice of scenes and textures. Moreover, recent advances in hardware acceleration, notably the introduction of the tessellation engine, have changed the bottlenecks when rendering spline surfaces. To nevertheless give an idea of the magnitude of acceleration, we note that none of the earlier trim-surface rendering algorithms promised interactive trim adjustment. We show the increased flexibility and speed of the trim-rendering add-on in the accompanying video, by demonstrating interactive trim modification.

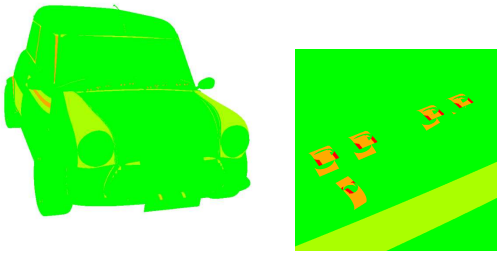


Figure 12: Number of $u$-intercept table entries: green=1 to red=6.

The *GPU memory* used by the add-on is that of the slim $u$-intercept table, set to $2^{14}$ rows and $2^5$ columns plus, for each surface piece consisting of many patches, $2^7$ $v$-strip indices and the numbers $\mu_i$. The graph in Fig. 13 compares GPU memory usage for different zoom levels and the Mini data set of the $u$-intercept table vs. texture-based trimming. (Identical trimming precision was enforced by setting the texture resolution to $n_V \cdot \max_i \mu_i$.)

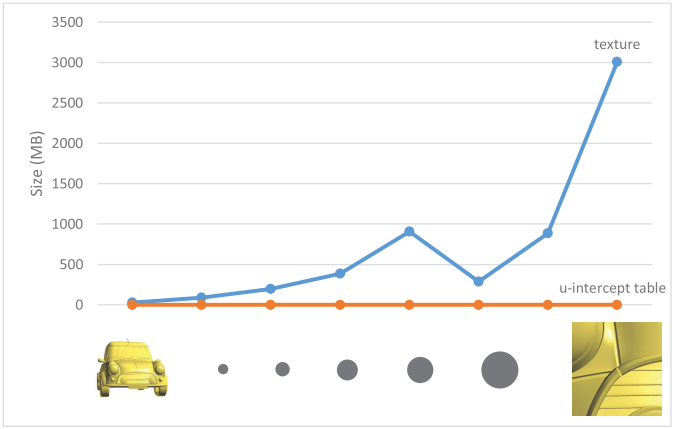



Figure 13: GPU memory usage for different zoom levels.

*Performance under zoom and different resolution.* Fig. 14 illustrates work distribution when zooming in. As the surface(s) fill more and more of the screen, the surface rendering time (red) dominates, while work for calculating the patch tessellation levels decreases since many patches are recognized to fall outside the viewing frustum by the algorithm of [18] and are discarded. Fig. 15 shows the increase in run time when the render window increases from $400 \times 400$ to $1200 \times 1200$. Since the patch-based work, for determining the tessellation factor that guarantees pixel-accuracy [18], stays constant, the proportional increase of work for drawing (pixel fill) and the trim test determine the overall cost.

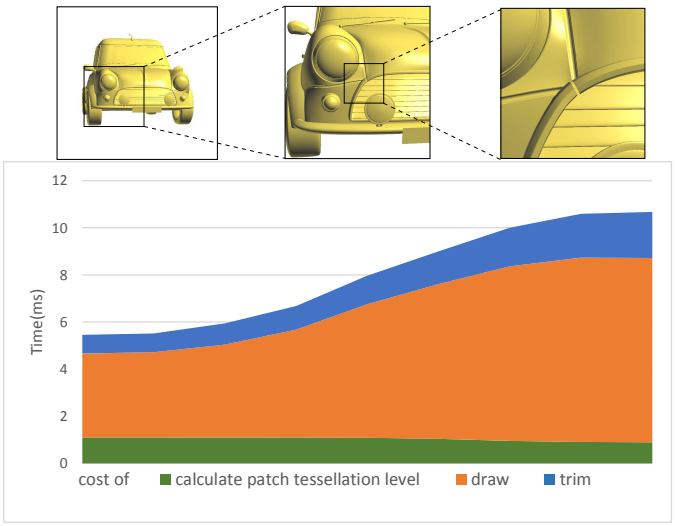



Figure 14: Performance at different zoom levels.

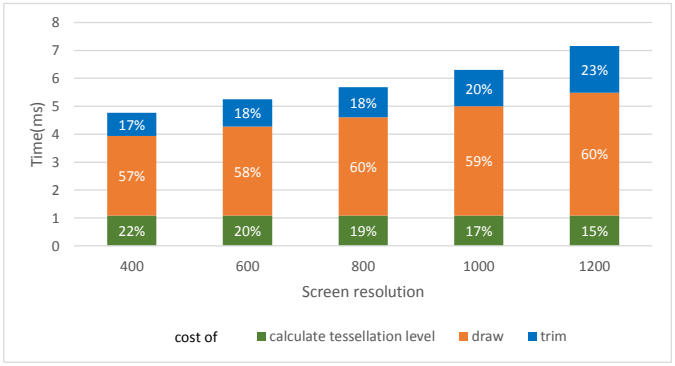



Figure 15: Performance at different screen resolution for model Fig. 11a.

## 7. Discussion

As surveyed in Section 2, existing approaches to rendering trimmed spline surfaces require extensive pre-processing and re-approximation of the data or expensive off-line ray casting that interfere with the designer's work flow. Correct resolution rendering can give immediate and visually accurate feedback by discretizing at just the right level. A direct timing comparison with the implementations of [9], [7] and [8] is not possible since the GPU hardware has evolved; and meaningful comparison requires that all rendering be (sub-)pixel accurate. However timing comparisons are not crucial since a qualitative comparison already brings out the relative strengths. On one hand, the correct resolution approach avoids generating large trim textures that are a bottleneck in [9]. On the other hand, the trim test based on the $u$-intercept table is faster and more robust than retrieving and solving equations in [7]. And while approaches such as [8] leverage extensive pre-processing, the $u$-intercept table is built on the fly, enabling interactive trim adjustment. Accuracy via the simple $u$-intercept table data structure is made possible by tight estimates and the fact that the screen resolution is known at run time.

In terms of GPU usage, an important characteristic of correct resolution rendering is that the Fragment Shader does little extra work. This is important since rendering is typically Fragment Shader bound due to heavy use by expensive pixel shading operations.

*Worst case complications.* Computing the per-pixel densities $\rho_\alpha$ at the end of the previous rendering pass creates an information lag. This lag is imperceptible at the targeted frame rates above ten.

The $v$-partition number $\mu_i$ bounds the arclength of the projection of a (piece of a) $u$-scan line onto the screen-space. For typical surfaces the arclength, and hence the size of the $u$-intercept table, is a (small) fraction of the screen size $w_x$, at ca. 1000 a relatively small number. If, however, a surface piece is highly oscillating, $\mu_i$ can in principle exceed $w_x$ and increase work and memory requirements.

*In summary.* The light-weight trim add-on, based on tight conservative estimates of correct resolution, enables interactive anti-aliased display during the design of trimmed spline surfaces. By adapting the curve tessellation level and the $v$-scan line density, correct resolution rendering can guarantee accuracy while keeping the memory footprint small.

Figure 16: Correct resolution real-time rendering of trimmed spline surfaces.

*Acknowledgements.* We thank Michael Guthe, Bernd Froehlich and Andre Schollmeyer for pointers to trimmed NURBS models and helpful comments concerning their algorithms. We thank GrabCad.com for providing models. This work was supported in part by NSF grant CCF-1117695.

[1] E. Catmull, J. Clark, Recursively generated B-spline surfaces on arbitrary topological meshes, Computer-Aided Design 10 (1978) 350–355.

[2] J. Peters, U. Reif, Subdivision Surfaces, Vol. 3 of Geometry and Computing, Springer-Verlag, New York, 2008.
[3] C. Loop, Second order smoothness over extraordinary vertices, in: R. Scopigno, D. Zorin (Eds.), Eurographics Symposium on Geometry Processing, Eurographics Association, Nice, France, 2004, pp. 169–178.
[4] K. Karčiauskas, J. Peters, Rational bi-cubic $G^2$ splines for design with basic shapes, Computer Graphics Forum (SGP2011) 30 (5) (2011) 1389–1395.
[5] M. J. Kilgard, J. Bolz, GPU-accelerated path rendering, ACM Transactions on Graphics 31 (6) (2012) 1.
[6] D. Rice, R. J. Simpson, OpenVG specification version 1.1 (Dec. 2008).
[7] H.-F. Pabst, J. Springer, A. Schollmeyer, R. Lenhardt, C. Lessig, B. Froehlich, Ray casting of trimmed NURBS surfaces on the GPU, in: Interactive Ray Tracing 2006, IEEE Symposium on, 2006, pp. 151–160.
[8] A. Schollmeyer, B. Fröhlich, Direct trimming of NURBS surfaces on the GPU, ACM Transactions on Graphics 28 (3) (2009) 1.
[9] M. Guthe, A. Balázs, R. Klein, GPU-based trimming and tessellation of NURBS and T-spline surfaces, ACM Transactions on Graphics 24 (3) (2005) 1016.
[10] J. T. Kajiya, Ray tracing parametric patches, in: Computer Graphics (SIGGRAPH '82 Proceedings), Vol. 16 (3), 1982, pp. 245–54, ray tracing bivariate polynomial patches.
[11] D. Manocha, J. F. Canny, MultiPolynomial resultant algorithms, Journal of Symbolic Computation 15 (2) (1993) 99–122.
[12] T. W. Sederberg, F. Chen, Implicitization using moving curves and surfaces, in: SIGGRAPH, 1995, pp. 301–308.
[13] B. Laurent, Implicit matrix representations of rational Bézier curves and surfaces, Computer-Aided Design 46 (2014) 14–24.
[14] C. Loop, J. Blinn, Resolution independent curve rendering using programmable graphics hardware, ACM Transactions on Graphics (TOG) 24 (3) (2005) 1000.
[15] T. E. F. Wiegand, Interactive Rendering of CSG Models, Computer Graphics Forum 15 (4) (1996) 249–261.
[16] J. Rossignac, A. Megahed, B. Schneider, Interactive inspection of solids: cross-sections and interferences, ACM SIGGRAPH Computer Graphics 26 (2) (1992) 353–360.
[17] M. Segal, K. Akeley, The OpenGL graphics system: A specification (Oct. 2013).
[18] Y. I. Yeo, L. Bin, J. Peters, Efficient pixel-accurate rendering of curved surfaces, in: Proc ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games - I3D '12, ACM Press, New York, New York, USA, 2012, p. 165.
[19] D. Filip, R. Magedson, R. Markot, Surface algorithms using bounds on derivatives, Computer Aided Geometric Design 3 (4) (1986) 295–311.
[20] A. Balázs, M. Guthe, R. Klein, Efficient trimmed NURBS tessellation, Journal of WSCG 12 (2004) 27–33.3.