Crack-free rendering of dynamically tesselated B-Rep models

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

EUROGRAPHICS 2014 / B. Lévy and J. Kautz
(Guest Editors)

Volume 33 (2014), Number 2

Crack-free rendering of dynamically tesselated B-Rep models

Frédéric Claux, Loïc Barthe, David Vanderhaeghe, Jean-Pierre Jessel, Mathias Paulin

IRIT - Université de Toulouse

Abstract

We propose a versatile pipeline to render B-Rep models interactively, precisely and without rendering-related artifacts such as cracks. Our rendering method is based on dynamic surface evaluation using both tesselation and ray-casting, and direct GPU surface trimming. An initial rendering of the scene is performed using dynamic tesselation. The algorithm we propose reliably detects then fills up cracks in the rendered image. Crack detection works in image space, using depth information, while crack-filling is either achieved in image space using a simple classification process, or performed in object space through selective ray-casting. The crack filling method can be dynamically changed at runtime. Our image space crack filling approach has a limited runtime cost and enables high quality, real-time navigation. Our higher quality, object space approach results in a rendering of similar quality than full-scene ray-casting, but is 2 to 6 times faster, can be used during navigation and provides accurate, reliable rendering. Integration of our work with existing tesselation-based rendering engines is straightforward.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Boundary representations

1. Introduction

Computer-Aided Design (CAD) software applications allow mechanical engineers to reliably design machined parts to be used in large-scale projects such as building an aircraft or a car. Software programs usually define parts with an internal, proprietary model definition but also use a common representation called B-Rep (for *Boundary Representation*). B-Rep has been popular since the 1980s, is the de-facto export format for manufacturing, and is widely used as an internal representation of the model in CAD applications for rendering purposes.

B-Rep models consist of a set of faces, each defined with a basis surface and a set of trimming curves defined in the parametric space of each basis surface (Figure 2). This representation is obtained by successively applying modeling operations on the exterior faces of geometric primitives. Because small gaps may exist between adjacent faces for the sake of representing shared edges in a relatively simple fashion [SAG84], manufacturing relies on explicit topological information enforcing geometrical continuity to produce machined parts of good quality. Although these gaps can be apparent during model visualization, it is only the case at extremely high zoom levels and, up to a certain extent, CAD operators usually cope with their presence while designing their parts. For this reason, the complex and costly methods that exist for eliminating them in a post-process on the B-Rep model [SSZ*04, SFL*08] are yet to be widely adopted. Rendering B-Rep models as they are strictly geometrically defined, accurately and at interactive performance is, however, very largely desired, and is a challenging task that we address in this paper.

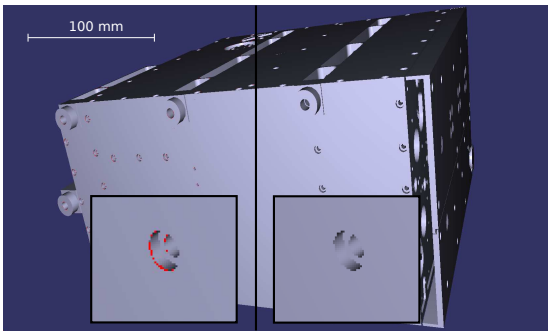

Figure 1: *Rendering using dynamic tesselation and direct GPU trimming produces cracks (in red). Our method fills them appropriately.*

Several approaches exist to fulfill this need (Section 2). All methods based on full-model micropolygonization or ray-casting usually offer a great image quality but fail to provide interactive performance, making tesselation the best candidate for high-performance rendering. Rendering based on static, model-wide tesselation suffers from inaccuracies in the display because of the approximate nature of the static tesselation and its fixed, predetermined resolution. Rendering based on dynamic surface tesselation and direct GPU trimming leaves visual artifacts, denoted as *cracks* between adjacent faces, lowering the overall image quality (Figure 3(a)). While a very fine tesselation can reduce the amount and size of cracks, it cannot remove them completely. Existing methods to remove these cracks may themselves leave visual artifacts and are complex to implement.

We propose an algorithm to get a rendering of high visual quality, with no crack, and interactive performance. Our algorithm is broken down into successive steps (Figure 4). A first step performs an initial model rendering through dynamic surface tesselation and direct GPU trimming, using a carefully chosen error tolerance keeping the generation of cracks under control (Section 3.1). Following the initial rendering step, a second step (Section 3.2) detects crack-induced pixels and flags them for later crack filling. Subsequent steps are where the crack filling and final image rendering are performed. During model navigation, we propose a simple single-step image space crack-filling method based on neighboring surface and color information around crack pixels (Section 3.3). For still images, or if higher quality is desired during navigation, we also propose a 3-step object-space crack filling method. With this method, a third step (Section 3.4.1) builds a depth buffer to limit GPU work done in a fourth step (Section 3.4.2) that performs the actual crack filling by ray-casting the model at the flagged pixel locations only. Our key idea is to leverage the efficient GPU model tesselation and rasterization in this step to limit the quantity of ray-casting, reducing the impact on rendering performance, while also guaranteeing that cracks will be effectively filled. A final step (Section 3.4.3) eventually writes color and depth data obtained from both the first and fourth steps out to the render buffer, and produces the final image.

Our contributions reside in the image-space crack detection, image-space crack filling for interactive rendering and object-space crack filling for either interactive or static, still image rendering. Our tests (Section 4) show that our image-space crack-filling method has acceptable quality and limited runtime cost. Our object-space crack-filling method significantly reduces the gap in image quality between dynamic tesselation and ray-casting, does not suffer from the rendering imperfections that can be observed with full-scene ray-casting (Figure 18), and is comparatively 2 to 6 times faster. Our crack filling algorithms can be plugged into existing rendering engines based on dynamic tesselation and do not require dedicated crack-filling geometry to be sent alongside model data, making them convenient to implement.

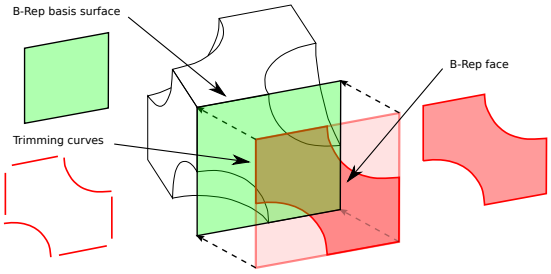

Figure 2: *Anatomy of a B-Rep model face.*

2. Related work

The techniques used in interactive B-Rep model rendering can be divided into two categories, using either model-wide tesselation or direct trimming approaches. We quickly go over model-wide tesselation and concentrate this section on methods based on direct trimming, which our algorithm is based on.

Model-wide tesselation rendering strategies rely on the tesselation of individual B-Rep faces [PR95]. A set of triangles approximating as closely as possible a B-Rep face is generated based on the geometrical definition of the face (eg. the B-Rep basis surface and associated trimming curves, illustrated in Figure 2). Special care must be taken to make sure that cracks do not exist between independently tesselated faces, in the vicinity of trimming curves [SK03]. It is generally done by using the topological information that links adjacent faces together. Model-wide, watertight, face-based tesselation is heavy on resources and cannot be done on the fly during rendering. As a consequence, models are pretesselated using a fixed error tolerance and the resulting mesh is used as input in the rendering system. Because the tesselation of the model can create a significant amount of geometry and hamper rendering performance, many methods also propose the definition of Level-Of-Details (LODs) schemes [Hop96, HSH10]. The highest achievable visual precision of the rendering is therefore bound to the original world-space error tolerance the highest-level LOD mesh has been built with. This encourages LODs of very high precision to be defined, occupying a sometimes prohibitive amount of memory.

More recently, with the advances in GPU hardware, methods based on dynamic surface tesselation with **direct trimming** have gained attention. Rendering based on direct trimming follows the trim definition of the B-Rep faces for rendering purposes in that every basis support surface is rendered then dynamically trimmed, independently. Rendering the basis surface can be done using either tesselation or ray-casting, while the trimming is usually done on a fragment basis, using a representation of the trimming space in a trim structure. This structure can either be a discretization [GBK05] or a vectorial representation [SF09,CVB*12].

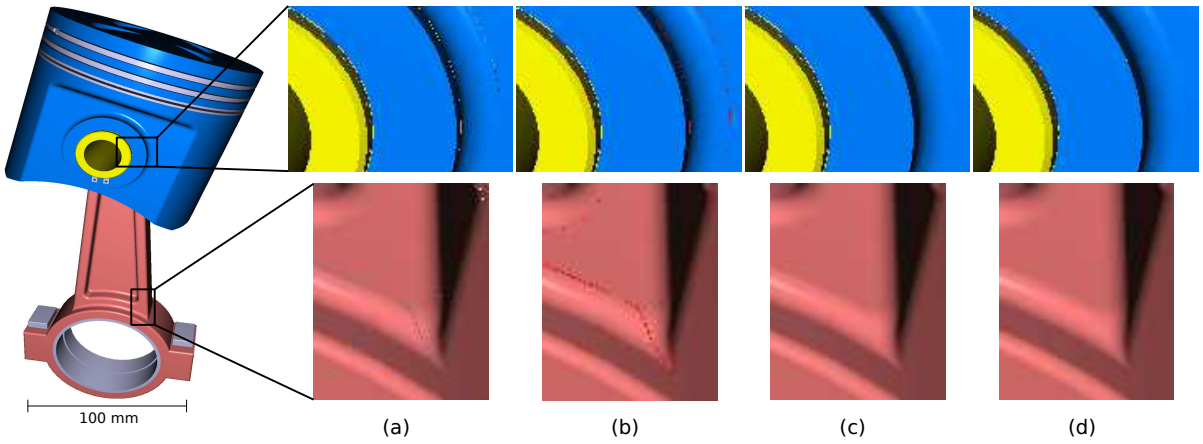

Figure 3: *(a) result of step 1 after dynamic model tesselation, with numerous cracks in the rendering. (b) step 2 crack identification, with crack pixels in red. (c) step 4 crack filling achieved using our selective ray-casting. (d) reference rendering using full-scene ray-casting.*

Schollmeyer and Fröhlich [SF09] propose a method to do the trimming of parametric surfaces efficiently and accurately. A pixel-precise B-Rep rendering can be achieved when their method is combined with ray-casting, though occasional, parasite pixels may show up around face boundaries (Figure 18, bottom-left). These pixels can be observed because of the limitations of the iterative Newton method used for root finding [FP79], which they implement to calculate ray-patch intersections. Rendering performance is mainly limited by the systematic ray-casting that needs to be done for each fragment of every input, untrimmed B-Rep basis surface, and by the trimming that follows. When tesselation is used to render basis surfaces, cracks between adjacently rendered trimmed B-Rep faces appear. These cracks are the manifestation of the gaps artificially created at the junction of trimming curves between two faces by the tesselation process.

Hanniel and Haller [HH11] ray-cast trimmed CAD models in a watertight fashion. Fragments that are in the vicinity of trimming curves are projected onto the surfaces adjacent to these curves. A test based on which side of the surfaces the fragment lies on is then done to determine whether to keep it for display. Their rendering yields no tesselation-related cracks and automatically removes topological gaps between faces. Their trimming algorithm suffers from weaknesses where trimming curve endpoints are tangential to neighboring curves, or when these trimming curves define a tangential transition across two adjacent surfaces, which create visual artifacts. Based on full-scene ray-casting, their method has limitations in terms of performance when dealing with larger models.

Yeo et al. [YBP12] propose pixel-accurate rendering of untrimmed NURBS models. Their method is based on the dynamic tesselation of rational Bézier surfaces. They use SLEFEs [Pet03] to calculate the optimal tesselation factors to be used when rendering a surface. Cracks between adjacent patches are avoided by defining the tesselation factors at the shared edges level. When used with trimmed models, cracks do appear however as nothing is done to fill up the space left unrasterized between trimming curves of adjacent faces – only the transition between untrimmed *surfaces* is taken care of.

Some methods propose to address the crack problem introduced by the individual, independent rendering of B-Rep faces by creating additional geometry to be rendered along the trimming curves. The geometries are either lines or triangle strips that overlap neighboring faces, and that are rendered to fill up the space artificially created by the tesselation. Balázs et al. [BGK04] concentrate on tesselation-induced cracks, and render two-pixel thick primitives along trimming curves. Pavanaskar and McMains [PM13] dynamically adjust the thickness of triangle strip geometries with respects to the curvature of matching trimming curves. Their method can fill topological gaps between faces in addition to rendering-related cracks. Both methods can suffer from minor artifacts caused by the additional geometry to be rendered, and are limited by the discretization of crack-filling primitives.

Image-based methods such as morphological and bilateral filtering [Soi03, KTD09] are unlikely to provide the accuracy we are looking for and which can be offered by geometrical approaches. Nevertheless, we introduce in Section 3.3 a dedicated image-space crack-filling method that has good

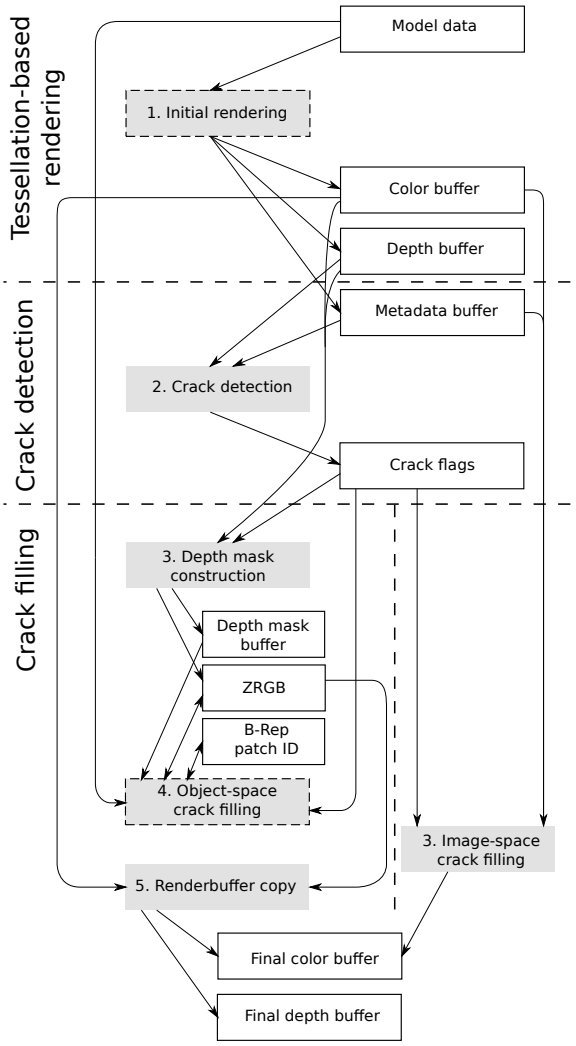

Figure 4: *Overview of our rendering pipeline. Boxes with a gray background represent rendering steps. Note that final crack-filling steps are commonly numbered for the object space (steps 3,4 and 5, left) or image space (step 3, right) methods. Dashed boxes perform full-model rendering, other boxes operate on a screen-sized quad. Boxes with white background represent storage data buffers. Arrows indicate whether step boxes read and/or write the data.*

performance but shows the limitations of this family of algorithms.

3. Our algorithm

Prior to being given to our renderer, the input model needs to be converted. This conversion, done once in a CPU preprocess, is a standard lossless transformation of the input model. Each B-Rep basis surface is transformed into a NURBS surface, which is then decomposed into multiple rational Bézier patches [CLR80]. Trimming curves are also similarly transformed into NURBS curves and split up into multiple rational Bézier curves, which are finally gathered in a trim structure [SF09]. The trim structure is used to do the trimming of individual fragments every time this is needed.

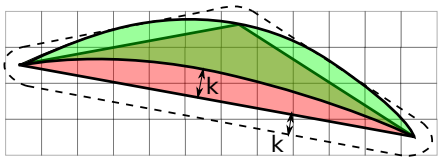

Figure 5: *The red triangle is a k-pixel approximation of the B-Rep surface (green) section it represents. The k-pixel covering accuracy defined by Yeo et al. guarantees that the B-Rep surface footprint on screen is located within the k-pixel sized ring (dashed line) around the triangle.*

An overview of our pipeline, which entirely runs on the GPU, and the different buffers used is given in Figure 4. Figure 3 illustrates a result of the crack detection and filling operations. Individual rendering steps are detailed onwards.

3.1. Step 1: initial rendering

In the first step, the whole model is rendered using the programmable graphics pipeline. For basis surface rendering, dynamic tesselation is performed using the method described by Yeo et al. [YBP12]. Following their work, rational Bézier surface patches are tesselated and rendered in such a way that the tesselated primitives never deviate more than k pixels from their corresponding projected footprint as defined by their analytical representation. This concept is defined by Yeo et al. as a k pixel *covering* accuracy. A *parametric* accuracy also guarantees that fragments of a given u,v coordinate never get rendered further than k pixels away from the corresponding location on the footprint of the analytically defined surface (Figure 5). We set k to 0.5 pixel to make sure all projected fragments end up being within a 1 screen pixel neighborhood ring of their real pixel location. k can be set to a lower value, which results in the appearance of less cracks at the end of step 1 – changing the value of k is discussed in Section 4.

Each rendered fragment goes through the trimming process [SF09], and off-face fragments are discarded. For on-face fragments, alongside the fragment color, we store in a framebuffer-sized *metadata buffer* the Bézier patch ID, the parent B-Rep face ID, as well as the u,v coordinate of the fragment in the parametric space of the Bézier patch.

3.2. Step 2: crack pixel detection

Step 2 operates in screen space. For each pixel of the framebuffer area, we decide whether to flag it as a crack or not

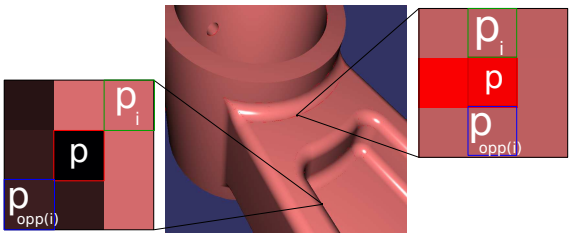

Figure 6: *The four $(p_i, p_{opp(i)})$ pixel pairs around p are examined. Top-right inset: p is a crack, as the world-space distance between the B-Rep surface points defined by p and p_i is greater than d_{min}, the same is true between p and $p_{opp(i)}$, and p does not have the same B-Rep face ID as either p_i or $p_{opp(i)}$. This latter condition is not met in the bottom-left inset, causing p to not be classified as a crack.*

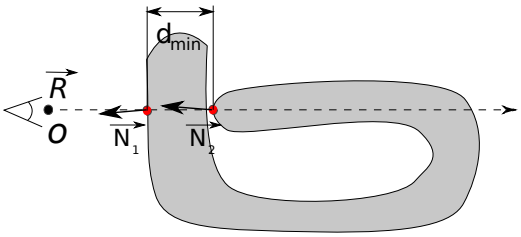

Figure 7: *Calculation of d_{min}, which holds the minimum distance between surface points bound to two adjacent pixels to be used for crack pixel detection. B-Rep models identify solid models and only their exterior, front-facing faces are visible and therefore rendered. d_{min} can be set to the minimum possible distance travelled by a single ray (dotted line) intersecting the B-Rep model at front-facing locations (red) with respects to the observer (left).*

depending on the information of the 1-ring, 8-pixel neighborhood. Crack detection results in the creation of boolean values in a framebuffer-sized *crack flag* buffer (see Figure 4), where a value set to true means the corresponding pixel has been identified as a crack.

The crack detection routine scans through the neighboring pixels. The central pixel p is flagged as a potential crack when both a neighboring pixel p_i and its opposite counterpart $p_{opp(i)}$ on the neighborhood ring are closer to the observer by an offset of at least d_{min}, expressed in world units (Figure 6, top-right inset). The value of d_{min} should be set to the minimum distance between two points on the surface of the model over any possible intersecting ray $(o, \overrightarrow{R})$, and where the dot product between the ray and the two corresponding model normals $\overrightarrow{N_1}$ and $\overrightarrow{N_2}$ is negative (see Figure 7). This precalculated value is difficult to come by in practice. For the models we tested, we set it to 1.0 millimeter in world-space allowing a safe, conservative crack detection to take place, regardless of the camera position and view angle. The pixel is not flagged as a crack if p and a neighboring pixel (p_i or $p_{opp(i)}$) have the same associated B-Rep face ID, suggesting we are in a high-perspective rendered area of the same B-Rep face. Testing p_i and $p_{opp(i)}$ simultaneously against p is needed so that only crack-induced pixels are really identified as cracks, and that pixels in contour edge areas are left out undetected (Figure 8 and bottom-left inset of Figure 6). Note that our crack detection being conservative, flagged pixels identify cracks but may also include non-crack pixels.

In the two next sections we describe two different methods to fill cracks. The method described in the next section exclusively works in image space and targets frame rendering during navigation. The method that follows works in object space and benefits from the geometrical accuracy of ray-casting. Even though slower, it performs fast enough to be used during navigation and provides accurate, reliable rendering.

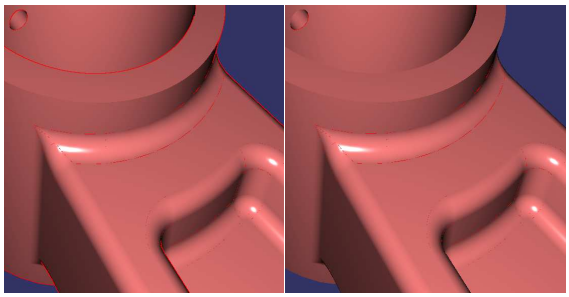

Figure 8: *Our algorithm isolates crack-induced pixels (in red, right) by comparing the depth value of neighboring screen pixels. The "opposite fragment" test makes sure pixels nearby contour edges are not taken into account. The left screenshot has the opposite fragment test disabled, causing all silhouette pixels to be flagged as cracks.*

3.3. Image-space crack filling

For each crack pixel p of the framebuffer area, we update its color depending on the information of the 8 neighboring pixels. We first determine the Bézier patch B_p that covers the highest number of neighboring pixels around p. The pixel p then receives the average color of the pixels that relate to B_p.

Image-space crack filling does not rely on local geometric information beyond the patch IDs and may fill topological gaps and small features such as tiny holes viewed at very low zoom level. We discuss image quality in Section 4.

3.4. Object-space crack filling

In this section we detail an algorithm to fill cracks in a robust and reliable way. It works by precisely detecting which B-Rep model face is visible at the crack pixel locations.

3.4.1. Step 3: depth mask construction

Aiming at minimizing the number of fragments that we process in the crack filling step, we first need to construct an adequate depth buffer that allows early Z tests to be performed by the GPU. The objective is to early cull all fragments that are known to have no possible impact on subsequent crack filling.

Step 3 works in screen space. It outputs a depth buffer value for every screen pixel computed as the maximum of the depth of the pixel created in step 1 and the depth of neighboring crack pixels. For pixels that neither define nor are in the vicinity of cracks, this value is the original depth buffer value created in step 1. For crack pixels or those in 1-ring vicinities, this depth buffer value holds the maximum depth beyond which step 4 fragments should be ignored. Fragments having higher depths are occluded by the crack pixel and therefore have no impact on the crack filling. The depth buffer created in step 3 is referred to as the *depth mask* in Figure 4.

Finally, this step also constructs a ZRGB value for each pixel on screen. This value is an unsigned integer value, obtained by combining the depth and primary color buffer values of the associated pixel produced in step 1, and storing them as a single value that can be atomically accessed by the GPU. The depth value is converted into an unsigned integer and stored in the upper bits, while the R,G and B color component values are stored in the lower bits. Thanks to this layout, a *minimum* or *maximum* operation carried out using two ZRGB values will act as if depth values were compared and updated, with the RGB value being updated *alongside* its corresponding depth value.

3.4.2. Step 4: object-space crack filling

The rasterization of the k pixel tolerant tesselation done in step 1 may have missed the rendering of fragments located within a k pixel ring (so essentially a ring of 1 pixel) around each triangle primitive (Figure 5), potentially leaving cracks.

Step 4 carries out the crack filling properly speaking. What we want to do now is ray-cast surfaces that can potentially fill the cracks. We define these surfaces as those being in a 1-ring neighborhood of a crack and located in front of it. Our algorithm cannot just work out of the B-Rep face ID and u,v coordinates created during step 1. This data is only available for the frontmost fragment in the Z order and therefore only enables to do the ray-casting on the corresponding surface, which is insufficient to fill many cracks. For each pixel around a crack that is either fully or partly covered by a tesselation primitive, we want the GPU to generate a corresponding fragment. In order to do this, we render the model twice, successively using filled triangle and wireframe modes. The wireframe mode makes sure that fragments are generated for thin triangles (Figures 10 and 13). Thanks to early Z rejection offered by the depth mask and adequate GPU pipeline setup, step 4 only processes fragments that are in the 1-ring neighborhood of a crack (Figure 9), and have a lower depth.

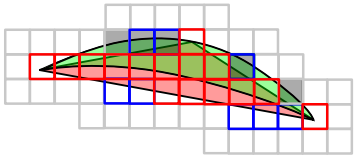

Figure 9: *Green: actual B-Rep surface footprint. Red triangle: footprint on screen representing a $k = 0.5$ pixel approximation of a B-Rep surface section. Red squares: filled-mode triangle fragments. Blue squares: wireframe-mode triangle fragments. Gray squares: pixels located on a 1-pixel ring around red and blue fragments are candidate for crack filling. Gray background squares: pixel locations flagged as cracks for which crack filling is done.*

3.4.2.a. Ray-casting B-Rep faces at crack locations

For each incoming fragment passing the early Z test, we proceed with the ray-casting of the related surface at the location of the neighboring flagged pixel. Ray-casting only takes place if the current fragment has a lower Z value in the ZRGB buffer than the neighboring pixel identified as a crack, otherwise the fragment is ignored. Using the surface and the u,v coordinate of the incoming fragment as the starting point for successive Newton root finding iterations [FP79], we determine if a ray actually intersects the surface at the crack pixel location. If it does, we look up the u,v coordinate resulting from the Newton iteration in the trim structure to determine if the intersection actually lies on the B-Rep face. This follows the fragment classification method of Schollmeyer and Fröhlich [SF09]. If the Newton iterations do not converge, or if the trim structure lookup reveals the fragment is off the face, execution continues on to the next neighboring fragment. Otherwise, the depth and color values of the ray-casted surface fragment are updated in the ZRGB buffer, but only if the Z value is lower than the one already in the buffer. An integer *atomicMin* operation is done for this purpose against the ZRGB value considered as an integer and the ZRGB buffer value, keeping Z (and associated color data) of always lower values.

Fragment classification is reported to be on-face only if the location of the u,v coordinates associated with the *center* of each fragment is classified as such. Some fragments may be classified as off-face, not be rendered in step 1, and not be processed in the current step, even though they do partially cover an on-face area of the surface. For thin triangles in silhouette areas, a single off-face classification might lead to missed, but nonetheless critical opportunities to fill nearby crack pixels. Having trimming disabled for wireframe mode

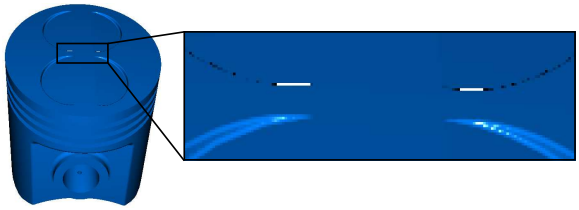

Figure 10: *Failure to render the model using both filled and wireframe modes results in cracks. The horizontal lines in the inset are background pixels that do not get filled when wireframe rendering is disabled.*

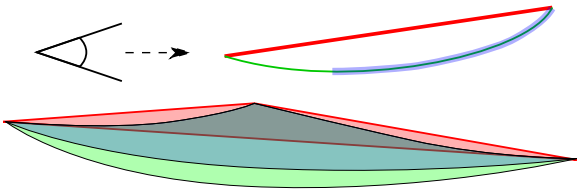

Figure 11: *With back-face culling enabled, this red, back-facing triangle is discarded. The corresponding B-Rep surface section (green and turquoise) it approximates is not entirely back-facing though. The green surface part is front-facing. This situation happens mostly with very thin triangles.*

in step 4 ensures that all triangle edge fragments are always processed by our fragment shader (Figure 12).

We do want to generate fragments for **back-facing triangles** that have their associated analytic B-Rep surface section partially front-facing over the definition domain of the supporting primitive (Figure 11). These triangles may be generated around silhouette edges, and if back-face culling is enabled, they are ignored. We therefore disable back-face culling when rendering the model in wireframe mode.

At the end of step 4, all pixels flagged in step 2 have their ZRGB buffer value updated so that this value reflects the closest ray/surface intersection for the model at the associated screen pixel location, effectively filling cracks.

3.4.2.b. Patch mailboxing

The GPU rasterizer tends to naturally group fragments in primitive order, and process fragments of the same primitive concurrently. To improve performance, we propose a mailboxing mechanism to prevent fragments of the same triangle primitive from exceedingly trying to ray-cast the same patch, at the same pixel location, through the same ray. For each flagged pixel we maintain in a buffer the last Bézier patch ID for which a ray-casting and trimming operation has been

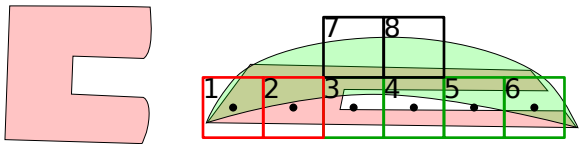

Figure 12: *Left: trimming structure. Right: footprint of the corresponding B-Rep surface on the screen, as approximated by a single tesselated quad (1 quad = 2 triangles in red). When trimming is enabled, the red triangles get rasterized as only 2 fragments (1 and 2). Fragments 3, 4, 5 and 6 do not get rasterized as the result of their classification is off-face, their center being off the B-Rep face. Fragments 7 and 8 are likely to be identified as cracks. While fragment 7 is within the neighborhood ring of fragment 2 and thus can potentially be crack-filled, fragment 8 cannot, as it is the neighbor of neither 1 nor 2.*

Figure 13: *Footprint of a B-Rep surface tesselated into a set of triangles, rendered by the GPU. Left: filled polygon mode. The fragments affected by the rasterization leave a discontinuous footprint on screen, as only 4 fragments have their centerpoint actually lying within the boundaries of the tesselated primitives. The rendering obtained is illustrated in Figure 10. Middle: 4x multisampling. The footprint left out by the surface on the screen still is non-continuous. Right: wireframe mode provides a suitable result.*

successful. Prior to initiating the ray-casting at a given pixel location, we check the patch ID and discard the processing if the value is the same as in the patch ID buffer. Because adjacent fragments of the current primitive being processed by the GPU affect the filling of common nearby flagged pixels, this process ensures the first fragment shader invocation to fill a specific crack pixel can relieve other invocations relating to the same patch from doing expensive ray-casting and trimming work at this location.

Note that due to inaccuracies of the Newton method for root finding, convergence may occur when the ray-casting is done using the u, v coordinate of some fragments, but may not occur when being initiated from the u, v coordinate of other nearby fragments of the same patch. Similarly, when a convergence occurs, the resulting u, v coordinate may lead the fragment to being classified as on or off the face if the u, v coordinate is very close to a trimming curve. Therefore, we only store the patch ID if the Newton iterations actually converge and trimming yields a classification as on-face. If these conditions are not met, other primitive fragments can still contribute to the crack filling for a specific pixel.

3.4.3. Step 5: output to renderbuffer

Step 5 works in screen space. Flagged pixels receive color and depth information stored in the ZRGB buffer value, while other pixels receive the original rendering color and depth information. This is to make sure the ZRGB value, which may only be 32 bit wide on some hardware and hence offers lower precision than the original separate color and depth buffers, is only used when ray-casting has actually taken place.

4. Results

We compare rendering performance and quality of our two crack filling methods. We also compare our object-space crack filling method with full-scene ray-casting. Full-scene ray-casting is implemented using the convex hull of the control polygon of Bézier patches, used to construct bounding volumes in which the ray-casting is to take place. Ray-patch intersections are calculated using the Newton iterative root finding method.

4.1. Rendering quality

4.1.1. Comparaison of the two crack filling methods

Comparing the image quality of the two methods is done by evaluating our crack detection routine, and by analyzing how cracks are effectively filled. To do this we compare crack-filled pixels with corresponding pixels in a reference image rendered by full-scene ray-casting.

The crack detection routine is mainly affected by the value of the d_{min} parameter. An implementation should estimate d_{min} accurately (see Section 3.2) or set it to a lower bound value, in order for all crack pixels to be detected. Either way, pixels that are not identifiable as cracks are occasionally flagged, i.e. crack detection is conservative. Thus, we need to assess how our two crack filling methods actually behave with extraneously detected crack pixels. For our object-space method, more crack-filling attempts linked to false positives means reduced performance and a marginal impact on image quality. We can observe here the limitations associated with our image-space crack filling strategy that fills all flagged pixels with neighboring color information, which it is not supposed to do for false positives. This leads to noticeable artifacts as shown in Figure 14. Increasing d_{min} beyond its accurate estimate as defined by our calculation method (Section 3.2) would result in the missed detection of an increasing number of cracks (Figure 15).

When performing an analysis of the image quality, with the image-space method, we can point out artifacts that can be seen across surface junctions where many aligned crack pixels show up. This method does not reliably choose the best surface to fill crack pixels with, as both the surfaces above and below the cracks comprise 3 pixels neighboring the crack (Figure 16). Other artifacts can be seen in some situations where there is a large discrepancy in the depth of the viewing space (Figure 17).

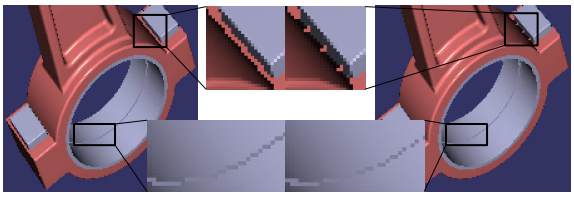

Figure 14: *Object-space versus image-space crack filling with safe, conservative crack detection ($d_{min} = 1.0$). Crack pixels are filled based on neighboring color data with the image space crack filling algorithm, which may create visual artifacts (upper-right inset). Our object-based crack filling method produces a rendering of good quality as crack pixels are ray-casted against nearby model surfaces (left).*

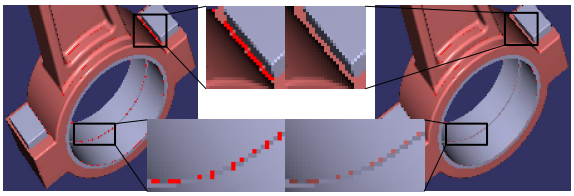

Figure 15: *Influence of d_{min} on crack detection. d_{min} set to 1.0 (left) and 5.0 (right). With $d_{min} = 5.0$, some cracks have not been detected and will not be filled (bottom-right inset). When $d_{min} = 1.0$, all cracks are detected, but some pixels are wrongly reported as cracks (upper-left inset).*

4.1.2. Comparaison with full-scene ray-casting

The visual quality of the image obtained with our rendering pipeline is on par with ray-casting, offers smooth display and has no cracks. There does exist a discrepancy between the rendering of tesselated fragments and neighboring ray-casted fragments. However, considered the deviation between the analytical surface and its tesselation only is $k = 0.5$ or less pixels, normals interpolated through tesselation are similar to those obtained from ray-casting the surface. The same shading model is used for both ray-casted and rasterized fragments.

Comparatively with our method, full-model ray-casting exhibits a few artifacts (Figure 18). The main reason for this is that the ray-casting of a surface may be done multiple times with our method for a single screen pixel identified as a crack, using different starting parameters. The ray-casting process only stops for a specific surface if the Newton root finding convergence is effective, and if the u, v coordinate resulting from the iteration is effectively classified as on-face. Our tests show that having multiple ray-casting attempts does have a positive impact on the rendering quality.

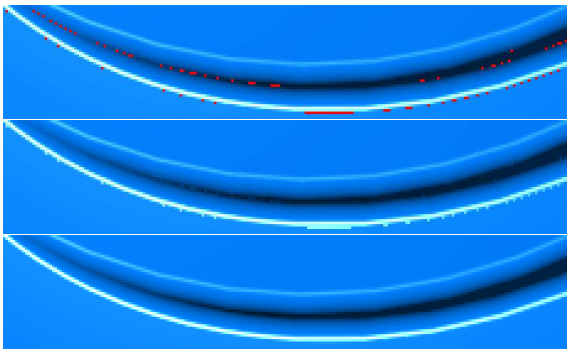

Figure 16: *Top: a large number of crack pixels can sometimes be aligned (bottom crack line). Middle: when these cracks are filled, the wrong surface may be chosen by our image-space algorithm, leading to visually disturbing appearance. Bottom: our object-space algorithm leaves no artifact.*

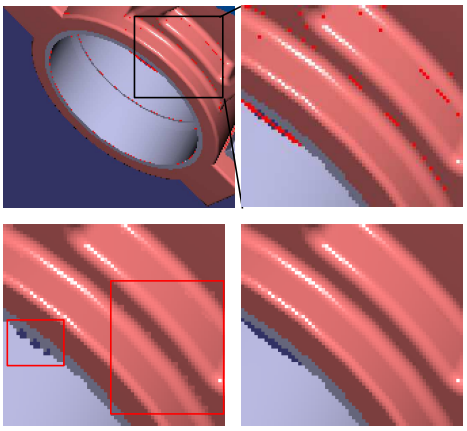

Figure 17: *Top: cracks detected in the rendered image, in red. Bottom-left: image-space crack filling wrongly attempts to fill the space between two silhouette edges not related to each other (left red inset). Averaging color values along surface junctions leaves minor but frequent imperfections (right red inset). Bottom-right: object-space method provides the expected result.*

See Figure 18 for a quality comparison between our method and full-scene ray-casting.

4.2. Performance

We evaluate the performance of our method on a Intel Core i7-860 processor at 2.8 Ghz and a GeForce GTX 780 graphics card. The resolution used for the tests is 1655x988 in all cases.

Rendering performance is evaluated for each step in Table 1. Because cracks are detected in image space using

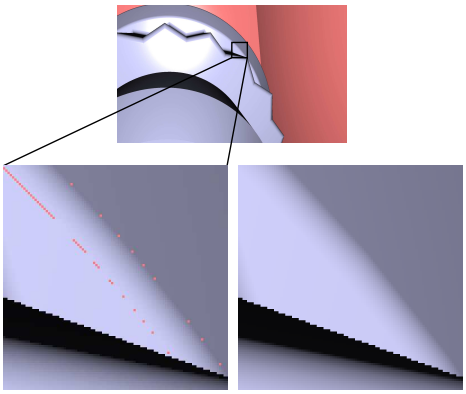

Figure 18: *Image quality comparaison between full-model ray-casting and dynamic tesselation with object-space crack filling. Bottom-left: full-model ray-casting leaves imperfections. Bottom-right: our object-space crack filling method does not. In both cases, the Newton algorithm is set to the same error tolerance (0.001 millimeter) and a maximum allowable number of iterations high enough for convergence to always take place when appropriate.*

depth information, the number of cracks detected does not linearly increase with the number of primitives, where there typically is a lot of overdraw and therefore occluded cracks pixels, which we do not need to fill. Our performance is limited by the number of actual cracks that are visible on screen. Our algorithm efficiently minimizes the amount of ray-casting to be done through early Z cull in step 4. Also, the Newton iterative method used for root finding starts with a u_i, v_i coordinate that is relatively close to the final u_f, v_f coordinate resulting from convergence, which helps reduce the cost of step 4.

According to our tests, increasing the tesselation factors so that a parametric accuracy of less than k with $k < 0.5$ pixel is honored does not generally affect performance, though it can occasionally have a slight, negative impact. k has an impact on wireframe rendering with regards to the number of generated fragments potentially falling into the 1-ring neighborhood of crack pixels, negatively impacting the performance.

5. Conclusion and future work

We have presented an algorithm that renders accurately B-Rep models typically 2 to 6 times faster than the speed of ray-casting the same scene. Our method detects cracks artificially created by the tesselation process, and fills them out using two different methods. Our image-space approach has a low impact on performance and acceptable image quality. Our object-space approach uses color and depth data obtained through ray-casting and trimming to fill cracks, providing similar image quality than full-scene ray-casting

and faster rendering speed. Our algorithm is implemented as add-on stages that can be plugged into rendering systems based on dynamic tesselation. Our crack filling techniques can be switched at runtime depending on the need for speed or image quality during navigation.

Even though our implementation works with tensor-product Bézier patches and uses Bézier patches-specific SLEFEs [Pet03] to calculate tesselation factors, our algorithm can work with any kind of parametric patches, as long as the tesselation factors can be calculated so that a 0.5 pixel *covering* and *parametric* accuracy, as defined by Yeo et al. [YBP12], is guaranteed for the projection of the patches on screen.

Our method guarantees watertight appearance only as far as the B-Rep geometrical definition permits. Junctions between some faces may not be topologically watertight, even though this is only visible at very high zoom levels. In particular, this may be the case for NURBS faces that are adjacent to one another. One way to modify our algorithm in order to remove these topological gaps could be to carry out the ray-casting and let some fragments be rendered out of the trimming area, slightly beyond the trimming curves, or even out of the surface domain for parametric patches mathematically defined beyond it.

Thanks

We would like to thank Datakit for having given us a developer license for their CATIA V5 file reading library.

References

[BGK04] BALÁZS Á., GUTHE M., KLEIN R.: Fat borders: gap filling for efficient view-dependent lod nurbs rendering. *Computers & Graphics 28*, 1 (2004), 79–85. 3

[CLR80] COHEN E., LYCHE T., RIESENFELD R.: Discrete b-splines and subdivision techniques in computer-aided geometric design and computer graphics. *Computer Graphics and Image Processing 14*, 2 (1980), 87 – 111. 4

	Fig. 3	Fig. 18	Fig. 1
Step 1 initial rendering	2.78	3.2	4.38
Step 2 crack detection	2.32	2.35	2.33
Image-space crack filling			
Step 3 crack filling	0.4	0.26	0.43
Total image-space method	5.5	5.81	7.14
Object-space crack filling			
Step 3 depth mask prep.	0.36	0.4	0.33
Step 4 crack filling	7.52	2.18	6.82
Step 5 renderbuffer output	0.22	0.2	0.18
Total object-space method	13.2	8.33	14.04
Full-scene ray-casting	37.24	119	48

Table 1: *Rendering time in milliseconds, with* $k = 0.5$ *pixel,* $d_{min} = 1.0$ *world units* $= 1$ *millimeter*

[CVB*12] CLAUX F., VANDERHAEGHE D., BARTHE L., PAULIN M., JESSEL J.-P., CROENNE D.: An efficient trim structure for rendering large b-rep models. In *Proceedings of the Vision, Modeling, & Visualization Workshop* (2012), pp. 31–38. 2

[FP79] FAUX I. D., PRATT M. J.: *Computational Geometry for Design and Manufacture*. Halsted Press, New York, NY, USA, 1979. 3, 6

[GBK05] GUTHE M., BALÁZS A., KLEIN R.: Gpu-based trimming and tessellation of nurbs and t-spline surfaces. *ACM Trans. Graph. 24*, 3 (July 2005), 1016–1023. 2

[HH11] HANNIEL I., HALLER K.: Direct rendering of solid cad models on the gpu. In *Proceedings of the 2011 12th International Conference on Computer-Aided Design and Computer Graphics* (Washington, DC, USA, 2011), CADGRAPHICS '11, IEEE Computer Society, pp. 25–32. 3

[Hop96] HOPPE H.: Progressive meshes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1996), SIGGRAPH '96, ACM, pp. 99–108. 2

[HSH10] HU L., SANDER P. V., HOPPE H.: Parallel view-dependent level-of-detail control. *IEEE Trans. on Visualization & Computer Graphics 16* (September 2010), 718–728. 2

[KTD09] KORNPROBST P., TUMBLIN J., DURAND F.: Bilateral filtering: Theory and applications. *Foundations and Trends in Computer Graphics and Vision 4*, 1 (2009), 1–74. 3

[Pet03] PETERS J.: Efficient one-sided linearization of spline geometry. In *IMA Conference on the Mathematics of Surfaces* (2003), Wilson M. J., Martin R. R., (Eds.), vol. 2768 of *Lecture Notes in Computer Science*, Springer, pp. 297–319. 3, 10

[PM13] PAVANASKAR S., MCMAINS S.: Filling trim cracks on gpu-rendered solid models. *Computer-Aided Design 45*, 2 (2013), 535–539. 3

[PR95] PIEGL L. A., RICHARD A. M.: Tessellating trimmed nurbs surfaces. *Computer-Aided Design 27*, 1 (1995), 16–26. 2

[SAG84] SEDERBERG T., ANDERSON D., GOLDMAN R.: Implicit representation of parametric curves and surfaces. *Computer Vision, Graphics, and Image Processing 28*, 1 (1984), 72 – 84. 1

[SF09] SCHOLLMEYER A., FRÖHLICH B.: Direct trimming of nurbs surfaces on the gpu. *ACM Trans. Graph. 28*, 3 (July 2009), 47:1–47:9. 2, 3, 4, 6

[SFL*08] SEDERBERG T. W., FINNIGAN G. T., LI X., LIN H., IPSON H.: Watertight trimmed nurbs. *ACM Trans. Graph. 27*, 3 (Aug. 2008), 79:1–79:8. 1

[SK03] STÖGER W. A. G., KURKA G.: Watertight tessellation of b-rep nurbs cad-models using connectivity information. In *Proceedings of the International Conference on Imaging Science Systems and Technology* (2003), Arabnia H. R., Mun Y., (Eds.), CSREA Press, pp. 602–606. 2

[Soi03] SOILLE P.: *Morphological Image Analysis: Principles and Applications*, 2 ed. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003. 3

[SSZ*04] SONG X., SEDERBERG T. W., ZHENG J., FAROUKI R. T., HASS J.: Linear perturbation methods for topologically consistent representations of free-form surface intersections. *Comp. Aided Geom. Design 21*, 3 (2004), 303–319. 1

[YBP12] YEO Y. I., BIN L., PETERS J.: Efficient pixel-accurate rendering of curved surfaces. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2012), I3D '12, ACM, pp. 165–174. 3, 4, 10