Eurographics Symposium on Rendering (2022)
A. Ghosh and L.-Y. Wei (Editors)

GPU-Driven Real-Time Mesh Contour Vectorization

Wangziwei Jiang, Guiqing Li*, Yongwei Nie, Chuhua Xian

South China University of Technology, Institute of Computer Science and Engineering, China

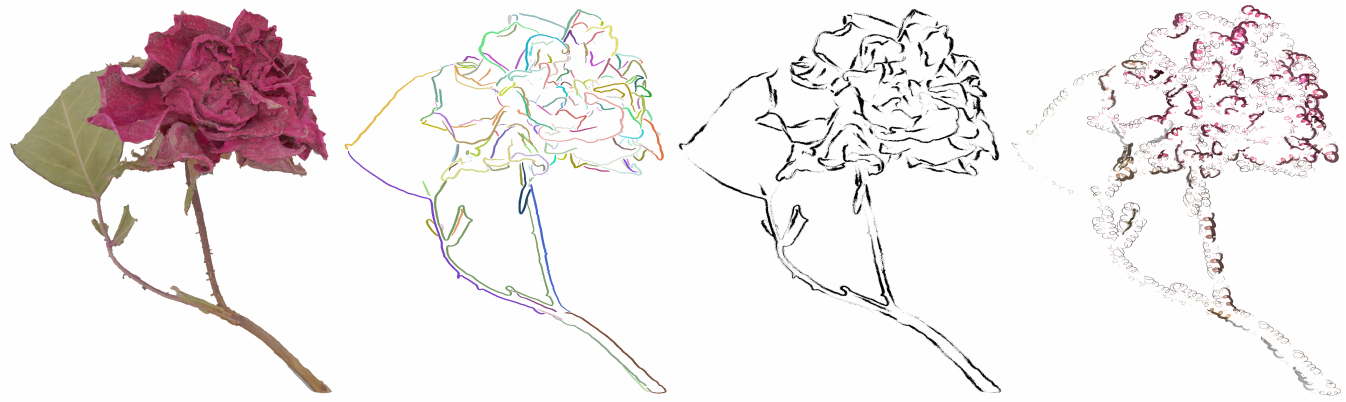


Figure 1: *Real-time vectorization and stylization of a rose (158k triangles) under* 2560×1440 *resolution: From left to right are respectively input 3D mesh, vectorized stroke curves rendered with different colors, and two different stylizations based on extracted stroke curves*

Abstract

Rendering contours of 3D meshes has a wide range of applications. Previous CPU-based contour rendering algorithms support advanced stylized effects but cannot achieve realtime performance. On the other hand, real-time algorithms based on GPU have to sacrifice some advanced stylization effects due to the difficulty of linking contour elements into stroke curves. This paper proposes a GPU-based mesh contour rendering method which includes the following steps: (1) before rendering, a preprocessing step analyzes the adjacency and geometric information from the 3d mesh model; (2) at runtime, an extraction stage firstly selects contour edges from the 3D mesh model, then the parallelized Bresenham algorithm rasterizes the contour edges into a set of oriented contour pixels; (3) next, Potrace is parallelized to extract (pixel) edge loops from the contour pixels; (4) subsequently, a novel segmentation procedure is designed to partition the edge loops into strokes; (5) finally, these strokes are then converted into 2D strip meshes in order to support rendering with controllable styles. Except the preprocessing step, all other procedures are implemented in parallel on a GPU. This enables our framework to achieve real-time performance for high-resolution rendering of dense mesh models.

CCS Concepts

• ***Computing methodologies*** → ***Non-photorealistic rendering;*** *Image processing;*

1. Introduction

Contour of 3D meshes, which reveals the essential shape of objects, plays a crucial role in painting and other arts. In particular, stylized contours are essential for artistic expression. That is why contour rendering which extracts and stylizes contours of 3D meshes has long been a fundamental topic in non-photorealistic rendering (NPR).

CPU-based contour rendering methods usually first convert the contour of a given 3D model into 2D or 3D stroke curves [GTDS10], then render those curves in different styles, e.g., changing contour width [GVH07], line-drawing density simplification [GDS04], stroke texturing [BCGF10], and stroke abstraction [BJC*12]. Though CPU-based approaches are able to produce stroke curves from the 3d contour, it is difficult for them to achieve real-time performance.

Most contour rendering methods for real-time applications are

DOI: 10.2312/sr.20221159

fully implemented on GPU in order to avoid frequent communication between CPU and GPU. GPU-based methods can be roughly divided into two categories: contour-edge-based rendering [MH04, CF09], and image-filtering-based rendering [ST90, ND04]. The former directly extract contour edges from the mesh and then renders each edge as a line segment or a rectangle while the latter first renders the geometry (for example, depth and normals) into textures, then finds feature pixels via image processing filters. Unfortunately, to our knowledge, existing full GPU-based methods cannot link contour pixels / edges together to form stroke curves, which however is the key for contour stylization. Recently, we have also witnessed deep neural networks being utilized to produce stylized line drawings [LNHK20, LFHK21], which mainly focus on learning styles rather than curve generation.

Chaining contour elements (image pixels or mesh edges) into a curve usually starts from a contour element and then continuously links the current element to its adjacent neighbor, until arriving at a singular point where the chain's visibility changes [BH19]. When linking 2D pixels, this process can be considered as a particular genre of image vectorization. It is difficult to parallelize the linking procedure due to its sequential nature and the irregular topology of contour edges or pixels.

To address the issue, we propose a GPU-based system to generate stroke curves from 3D mesh models. It first works on the CPU to prepare the adjacency information between vertices, edges and faces, and also geometric attributes including vertex positions and face normals. A GPU scheme is then designed to quickly locate contour edges between front and back faces of the mesh. After that, the parallelized Bresenham algorithm [Wri90] is adopted to rasterize these contour edges. To trace the boundaries of these rasterized contours efficiently, we parallelize the Potrace algorithm [Sel03] on the GPU, where the pixel-edge chaining step is parallelized by the technique of parallel list ranking [Wyl79]. Finally, based on the orientation of mesh contours and the traced boundaries, we devise a simple heuristic that is also parallelized to extract stroke polylines from image boundaries.

In summary, our contributions include:

- We parallelize the Potrace algorithm [Sel03] that is previously designed for CPU-based image vectorization, overcoming the sequential nature of boundary tracing by using the technique of parallel list ranking [Wyl79].
- We also propose a heuristic-rule-based parallel algorithm to extract stroke curves from the traced boundaries.
- Our method is fully in parallel. By exploiting the sparsity of contour edges and pixels, we further improve the performance of our method, achieving real-time performance.

2. Related work

An large amount of literature has been contributed to contour extraction and stylization, which can be roughly classified into three categories: image-based contour rendering, mesh-edge-based contour rendering, and the hybrid methods. This work focuses on real-time approaches that can be implemented on a GPU. We refer the readers to the survey by Bénard and Hertzmann [BH19] for more details.

2.1. Image-based contour rendering

Image-based approaches directly apply image filters to extract feature pixels from a rendered image. Some CPU-based algorithms further exploit image vectorization algorithms to convert feature pixels into continuous planar curves. For example, CPU-based image vectorization algorithms [Sel03] can be used for tracing feature curves. Xiong et al. [XFZ16] used GPU to accelerate the vectorization process, however their method relies on CPU to finish the sequential contouring.

GPU-based approaches usually make use of a fragment shader to apply edge detection filter on G-buffers. A G-buffer generally consists of three components: scene color, depth and normal images. A pixel in the G-buffer is considered as a line feature if its gradient is higher than a specified threshold [CS16]. Mesh contour can be approximately extracted as a subset of these line features. As only a set of scattered pixels are generated, this kind of methods can only support limited control over the stylization [ND04, Har07]. For example, it is challenging to achieve thick lines since the detected pixels are usually highly noisy under a low gradient threshold. It is often inaccurate too. For example, contours may be missed when the depth varies slowly around the contour area.

"Inverted hull", a special GPU-based method by Raskar and Cohen [RC99], is very popular in industries due to its simplicity and efficiency. The given mesh model is rendered twice to reveal its outline. The first pass renders front faces into a depth buffer while the second pass renders slightly enlarged back faces in black color, so that contour appears as black borders.

Bénard et al. [BJC*12] proposed a method to track feature curves in the image space with temporal coherence. Their algorithm is mainly based on CPU, except for the stages of line pixel filtering and final rendering that are done on a GPU. Each curve is represented as a polyline initialized using a CPU-based image vectorization algorithm. In each frame, they avoided the cost of vectorization (essentially reconstruction) by tracking and deforming a set of curves. Although being able to achieve excellent temporal coherence for meshes of moderate complexity, the approach suffers from a performance bottleneck due to multiple readbacks from GPU to CPU. As having little knowledge about the underlying 3D scene, its curve topology sometimes deviates from scene occlusions and details.

2.2. Mesh-edge-based contour rendering

Instead of extracting contour strokes from rendered images, some methods directly compute and render contour edges from the 3D model. An edge is considered on the contour when one of its two adjacent faces is forward and the other one backward with respect to the current viewpoint.

The earliest GPU-based methods [CM02, Goo03] treat each mesh edge as a degenerated quad and select contour edges in a vertex shader. Each quad contains four vertices (two are the endpoints of the mesh edge and the other two are its opposite vertices on its two adjacent faces) in order to determine whether the corresponding edge is a contour one. A fragment shader is then devised to scan-convert the contour edges. Noticing that it may lead to gaps

between adjacent edges when vertex normals fail to reflect the contour curvature well, McGuire and Hughes [MH04] drew caps at the ends of each contour edge. There are also efforts using GPU to extract mesh edges for other purposes. For example, Peciva et al. [PSM*13] and wachter et al. [WKS07] used GPU to efficiently compute shadow volumes.

Cole and Finkelstein [CF10] noted that early GPU-based methods suffer from visibility issues. They utilized geometry shader and advanced fragment shader techniques to achieve accurate visibility determination for contour edges. Each edge is projected onto the screen and sliced into small 2D segments, then the visibility of each segment is estimated via comparing its depth against the scene depth buffer, and finally each contour edge is individually rendered as textured quads.

2.3. Hybrid approaches

Hybrid method combines both the geometric information of contour edges and texture information of the rasterized pixels to generate contour stroke curves during the whole process. Both contour edges and pixels have their own advantages and disadvantages for rendering. The former may lead to small and frequent zig-zag artifacts when rendered as strokes [NM00]. On the contrary, the latter has simpler topology and natural appearance but usually loses accurate 3D information.

A typical hybrid approach by Isenberg et al. [IHS02] extracts 3D curves from contour edges with the help of a image-precision line visibility algorithm adapted for contours. The algorithm is essentially a software depth test in which contour edges are scan-converted into pixel-sized fragments and each fragment compares its depth against its 3×3 neighbors in the z-buffer.

Our approach analyzes the strokes in the image space. We also record geometric information such as vertex positions, face normals, primitive adjacency, and projected direction of contour edges for use. Therefore, our method can also be viewed as a GPU-based hybrid algorithm.

3. Overview

Our method takes a triangular mesh as input and generates vectorized contour curves. Specifically, it consists of five stages as shown in Figure 2. From left to right: (1) Preprocessing is conducted on CPU to collect the adjacency information and geometric attributes from the given mesh models; (2) Rasterization is responsible for recognizing the contour edges by checking the orientation of faces sharing the edge and then rasterizing the edges into pixels via a parallelized Bresenham algorithm; (3) The vectorization state parallelizes Potrace algorithm to trace the loops of the pixel boundaries; (4) The stroke generation stage employs a simple heuristic to extract the strokes from the pixel edge loops; (5) Finally, the stylization stage yields the rendering result of contour edges with a specific style.

4. Contour rasterization

Conventional hardware rasterization only yields a whole image instead of generating the desired contour pixels. Hence, we develop a specialized rasterization scheme to collect contour pixels only. Our scheme includes three sequential stages: recognition of contour edges, rasterization of the recognized edges and visibility decision on the rasterized fragments (pixels).

4.1. Computation of contour edges

An edge of a mesh is considered contour if and only if one of its two adjacent triangles is a front face and the other one is a back face with respect to the viewpoint. Given local information of all edges collected on the CPU, our GPU-based procedure first computes the orientation of all faces, and then collects the contour edges while getting rid of non-contour ones.

Pre-processing. Recognizing an contour edge needs to know the local geometry near the edge, therefore we collect all related information in CPU. This includes the following five buffers:

- edge-vertex buffer B_{ev}: store the index of 2 vertices for each edge;
- edge-face buffer B_{ef}: store the index of 2 adjacent faces of each edge;
- vertex buffer B_{vc}: record vertex coordinates;
- face-vertex buffer B_{fv}: save vertex index for each face;
- face-normal buffer B_{fn}: record face normals.

Considering that concave edges, whose internal dihedral angles are greater than π, cannot be a part of a visible contour [BH19], we discard all this kind of edges in B_{ev} and B_{ef} to save resources. To our experience, about 40% of the total mesh edges can be removed (see Figure 12).

Orientation of triangles with respect to viewpoint. We dispatch a GPU kernel to calculate the orientation of each face with respect to the viewpoint based on buffers B_{fv} and B_{fn} and store it in the face orientation buffer B_{fo} in which a back face is labelled with '1' while a front face is labelled with '0'.

Detection of contour edges. With B_{fo} as input, a GPU kernel is created to recognize contour edges from B_{ef}. An edge is a contour edge if its two adjacent faces have different orientations, namely one with label '1' and the other with label '0'. Next, we use parallel stream-compaction [BOA09] to select contour edges while discarding the rest. This yields a new buffer, the contour edge buffer B_{ce}. The subsequent GPU threads will only process edges in B_{ce} instead of those in B_{ev}. According to McGuire [McG04, MH04], the number of contour edges is close to $N_f^{0.8}$ where N_f is the number of mesh faces.

4.2. Fragment generation

A parallelized Bresenham algorithm [Wri90] is designed to scan-convert the contour edges into fragments. Each fragment is a pixel-sized primitive with geometric attributes and a pointer to its contour edge. The algorithm consists of two passes: a counting pass and an allocation pass.

Fragment counting pass. With B_{ce} and B_{vc} as input, this pass counts how many fragments are covered by each contour edge. If the absolute slope of the projection of the contour edge is less than 1, the number of pixels equals to the length of its projection along

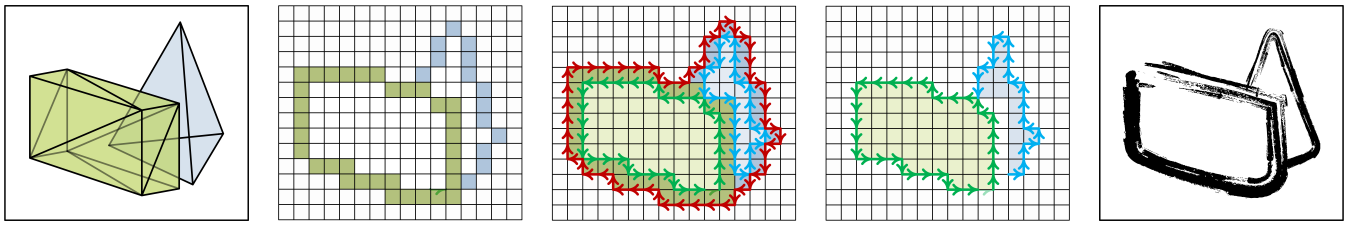


Figure 2: *Our approach consists of five stages. From left to right are respectively preprocessing, rasterization, vectorization, stroke generation, and stylization rendering. In the middle three pictures, white pixels stand for background regions.*

x-axis. Otherwise the number is the length of its projection along y-axis. Fragment count is stored as a sub-buffer B_{cf} within B_{ce}. We call these fragments *contour fragments*.

Fragment generation pass. This pass allocates a fragment attribute buffer B_{fa} for the fragments according to the total fragment number. We record pixel coordinates, projection of its associated edge vector (edge vertices oriented by its adjacent front face), depth and normal for each fragment in B_{fa}. For each contour edge, its fragments are sequentially stored in B_{fa}. To achieve such an allocation scheme, we need the mapping between contour edges in B_{ce} and contour fragments in B_{fa}. We apply an exclusive add-scan upon the fragment count buffer B_{cf} to build the mapping B_{ce_f} for each edge to its starting fragment index. The fragment-to-edge mapping B_{f_ce} is initialized with negative ones. We use B_{ce_f} to build the mapping at starting fragments in B_{f_ce}. Then we broadcast the mapping to other fragments via a segmented max-scan [Ble90] upon B_{f_ce}, with each starting fragment seen as the segment head. B_{f_ce} and B_{ce_f} enables each fragment (resp. contour) to access attributes from the corresponding contour (resp. fragments). Finally, we apply the parallel Bresenham algorithm [Wri90] to compute the coordinate for each fragment. Note that the depth and normal should be interpolated from vertex attributes in a perspective-correct manner.

4.3. Contour pixel generation

We need to extract visible fragments from B_{fa}. Accurate contour visibility has long been a challenging problem [CF10, BHK14]. We address the issue by a two-pass procedure on GPU. A soft depth test picks up pixels covered by visible contour fragments, referred to as *contour pixels*. A hardware z-test pass then selects the front-most fragment for each contour pixel.

Soft depth-test pass. A scene depth texture is rendered in advance. For each contour fragment, we compare its depth from B_{fa} against depth samples from its 3×3 neighborhood in the depth texture. A fragment passes the test if it is in front of no fewer than two neighbors and is called a pseudo-visible-fragment (abbrev. as pv-frag). This relaxed depth test allows multiple pv-frags to cluster in the same contour pixel as shown in Figure 3 in which 'e', 'd' and 'f' among 6 fragments pass the test to be a pv-frag within the same screen pixel.

To generate the contour pixels, we use a texture with all pixels assigned to 0. Each pv-frag atomically reads its pixel value from the texture, and then mark the pixel value as 1. We record the coordinate of a pixel in the pv-frag first visiting the pixel and then employ the parallel stream compaction algorithm [SHG*] to obtain a contour pixel buffer B_{cp} with pixel coordinates.

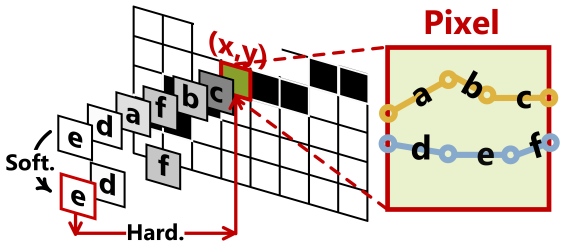


Figure 3: *Creation of contour pixels: A soft test generates a buffer of contour pixels such as (x,y) and a set of pv-frags for each pixel, 'e', 'd' and 'f' on (x,y); A hard test selects the front-most one for each contour pixel, e. g. 'e' among 'e', 'd', and 'f' on (x,y).*

Hardware depth test pass. This pass picks the front-most pv-frag for each contour pixel and copies the fragment attributes into the corresponding pixel in B_{cp}. We treat each pv-frag as a 1-pixel-size point whose depth is the fragment depth and whose color is computed by packing bits of the fragment attributes from B_{fa}. The pv-frag points are then rendered into a texture with hardware z-test. At last, each contour pixel samples the texture at its coordinate and decodes the sampled color to the corresponding fragment attributes. In Figure 3, pv-frag 'e' is finally selected in this test.

Figure 4 presents an example of the visibility test: visible (resp. hidden) fragments are marked as green (resp. red) on the left column; the right column illustrates contour pixels colored with encoded geometrical attributes. Rasterized contour-pixels only occupy a tiny portion of the screen, making it possible to achieve realtime image vectorization.

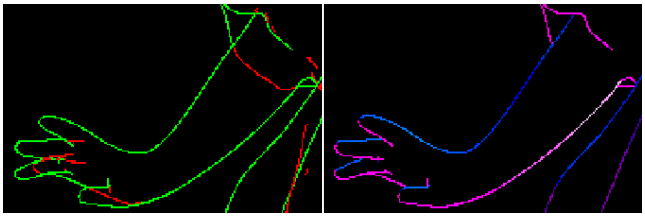


Figure 4: *An example showing results before (left: fragments) and after (right: the contour pixels) generating contour pixels.*

5. Contour chaining

So far, we have obtained B_{cp}, the buffer of contour pixels with geometric attributes, in which contour pixels generally form long and thin strands in the corresponding image. A chaining process should be conducted to link the contour pixels into a set of long curves [GTDS10].

Our chaining process is inspired by Potrace [Sel03], which is designed for vectorizing the boundary of a binary image, where the boundary consists of a sequence of boundary pixel edges. A pixel has four pixel edges by viewing it as a square and a boundary pixel edge is one shared by a foreground pixel and a background pixel as shown in Figure 5. Each boundary is an oriented pixel-edge loop and encloses a connected region. These loops act as a superset of our final stroke curves.

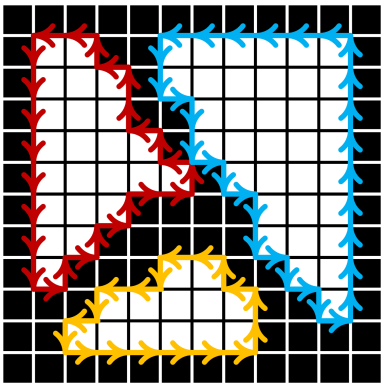


Figure 5: *Edge-loops: black and white squares are foreground (contour) and background pixels, respectively; edges shared by white and black squares are boundary pixel edges (red, blue and yellow one with arrow indicating their direction); three colored polygons are pixel-edge loops.*

5.1. Generation of pixel edges and creation of their linkage

We follow the 'path decomposition' scheme of Potrace to generate oriented pixel-edges for each contour pixel and build their linkage according to different contour pixel configurations.

Each pixel-edge is clockwise oriented around its contour pixel, therefore for the left pixel-edge of a contour pixel, we need to consider the 2×2 block where the contour pixel is at the bottom-right corner as shown in Figure 6. In this case, there are four possible configurations for the next pixel-edge. Other three cases, namely top, right and bottom pixel-edges of a contour pixel can be dealt with in a similar manner. Furthermore, it also requires to find the previous pixel-edge of the current one for each of the above four cases, which is needed in loop breaking process.

The whole task only involves 3×3 neighborhood of a contour pixel in the bitmap and it is trivial to parallelize. GPU threads only work on B_{cp}, i.e., the buffer of contour-pixels (foreground pixels), in order to improve performance. A binary bitmap with contour pixels as the foreground is required to support neighboring pixel queries. It finally outputs a pixel-edge loop buffer denoted by B_{pe_l} in which each element knows the indices of its previous and next pixel-edge neighbors.

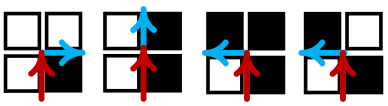


Figure 6: *Four pixel configurations. Given the left pixel-edge (red arrow) of a contour pixel (black one), its next pixel-edge should be the blue one.*

5.2. Edge loop flattening

We propose a parallel solution to replace the highly sequential process of Potrace to extract all pixel-edge loops from B_{pe_l} and flatten them onto a linear array as shown in Figure 7. Our solution consists of two passes: loop breaking and list ranking. The first pass selects a head element to break edge-loops while the later pass ranks pixel-edges in each edge-loop with respect to the head element.

In our setting, each edge-loop is a circular linked list and each pixel-edge is a list node randomly scattered in B_{pe_l}. It is quite suitable for Wyllie's parallel list ranking algorithm [Wyl79] to determine the rank of each pixel-edge in the pixel-edge loop. With ranks calculated, organizing the pixel-edges into linear arrays becomes trivial.

Loop breaking. In this step, we determine the head pixel-edge for each pixel-edge-loop. We specify the pixel-edge with the largest Morton code [Mor66] as the head of the loop, where the Morton code, unique for each pixel-edge, encodes its direction and related pixel coordinates. After obtaining the Morton codes of all pixel-edges, we employ Wyllie's algorithm with its operator set as integer maximum to pick up the head pixel-edge with maximal Morton code. The tail node of an edge-loop will be chosen as the predecessor of the starting one. Two traced edge-loops are shown in the top of Figure 7 in which the red arrows stand for the head.

List ranking. This pass ranks the above linked lists with head and tail nodes via Wyllie algorithm [Wyl79]. After ranking, we use the rank of each node (pixel-edge) as its array index and serialize all pixel-edge loops into an array, i.e., B_{pe_l}, such that the pixel-edges belonging to the same loop occupy a continuous segment as shown in the bottom of Figure 7.

5.3. Operations on edge-loop pool

The pixel-edge loop buffer B_{pe_l} forms the basis of our following screen-space algorithms. We call it an edge-loop pool. We develop two special operations: spatial filtering and segmentation. Classical parallel computing primitives like the segmented scan [SHG*] can be applied to the edge-loop pool by treating each edge-loop as a segment.

Spatial filtering. Spatial filtering can be considered as a 1D convolution on each edge-loop: each edge navigates around its edge-loop and collects data from the neighboring pixel-edges. In our implementation, GPU threads linearly map to all pixel edges. Each thread caches data into the thread group shared memory. In most

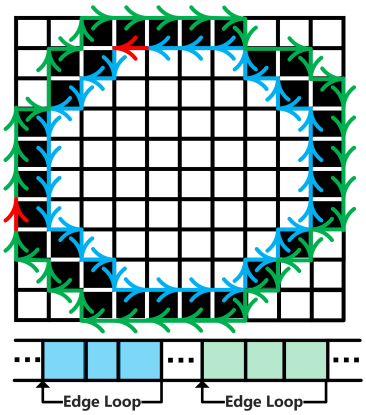


Figure 7: *Pixel-edge loops: two pixel-edge loops with a red arrow as the head node (top) and their pixel-edge loop array (bottom).*

cases, neighboring data can be found and fetched efficiently from this cache. However, there are a few cache misses: (1) Pixel edge is mapped to the start or end of a thread group; (2) Pixel edge is at the start (or end) of an edge-loop, and its predecessor (or successor) is not mapped to the same thread group. This can only happen to the first or last edge-loop mapped to the thread group. We detect both scenarios and load missed data to the group shared memory. Since the topology of edge-loop is fixed each frame, we can prepare missed data for each thread group and reuse it the whole frame.

Segmentation. Given a key for each edge, segmentation splits each edge-loop into segments; pixel-edges inside a segment share the same key, and two adjacent segments have different keys. Segmentation requires each pixel-edge to evaluate where its segment starts and ends, which can be implemented via two segmented scans, one for the starting index and another one for the ending index.

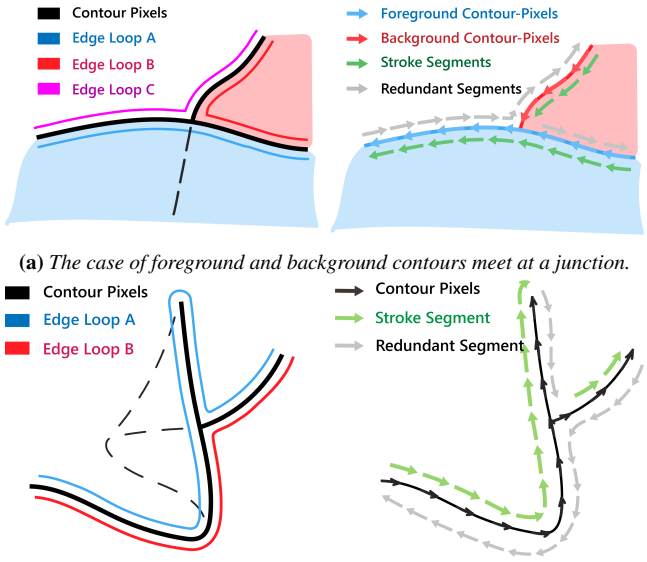
6. Stroke extraction

Edge-loops excessively cover contour features and neither start nor end at visibility changes. To resolve this issue, we select desired pixel-edges from loops, which we call *stroke segments*. Each stroke segment starts or ends as its underlying mesh contour became visible or hidden, and each contour feature is covered by exactly one stroke segment (Figure 8). An "inner" edge-loop without visibility change will be extracted as a stroke (see the inner loop in Figure 7).

We match the orientation of each pixel-edge with its surrounding contour-pixels along the edge-loop; pixel-edges with coherent orientation will be selected as stroke segments. According to Bénard and Hertzmann [BH19], there are two kinds of visibility change among contour-pixels: cusp and junction (see Figure 8b). Our heuristic can resolve both cases owing to the oriented nature of edge-loops and mesh contours.

6.1. Orientation of contour pixels

Seen from the viewpoint, vertices at a front face (resp. back face) have a counter-clockwise (resp. clockwise) winding order. Let each



(a) *The case of foreground and background contours meet at a junction.*

(b) *The case of a self-occlusion model generates a cusp and a junction.*

Figure 8: *Orientation-based stroke extraction.*

contour edge share the same vertex order as its adjacent front face. The contour of a smooth mesh will form counter-clockwise curves on the screen. After rasterization, the hidden contour is discarded while the visible contour becomes thin and long strands of contour-pixels. As the camera projection preserves the orientation of a triangle face, the strands of contour-pixels share a counter-clockwise orientation (see Figure 8).

During the rasterization stage (Section 4), we numerate vertices v_{c0} and v_{c1} of each contour edge according to its winding order in the adjacent front face, project them to the screen positions v_{s0} and v_{s1} respectively, and finally copy the edge direction $v_{s1} - v_{s0}$ to its rasterized contour fragments.

6.2. Orientation of pixel-edges

All pixel-edges are originally clockwise oriented around their contour-pixel square. To estimate an accurate orientation of the pixel-edge, we fit a curve to the local shape on its edge-loop. Our fitting algorithm takes the framework by Lewiner et al. [LGJLC05].

We dispatch two kernels to realize the local curve fitting. The first kernel samples the midpoint of each pixel edge, and then applies Laplacian operator to smooth the midpoints by using the spatial filtering discussed in Subsection 5.3. The second kernel applies the spatial filtering again to collect for each pixel edge e_0 the midpoints $W = \{m_{-n}, \dots\ m_0, \dots m_n\}$ of its neighborhood along the corresponding edge loop ($n = 8$ in our experiments). In addition, we compute the arc-length parametrization of W as follows

$$s_k = \begin{cases} 0 & k = 0; \\ s_{k+1} + \|m_k - m_{k+1}\| & k = -1, -2 \cdots, -n; \\ s_{k-1} + \|m_k - m_{k-1}\| & k = 1, 2, \cdots, n; \end{cases} \quad (1)$$

A quadratic parametric curve us then use to fit W_e

$$\mathbf{r}(s) = \mathbf{a}s + \mathbf{b}s^2, \quad (2)$$

where $\mathbf{r}(s) = (x(s), y(s))$, $\mathbf{a} = (a_x, a_y)$ and $\mathbf{b} = (b_x, b_y)$. This leads to the following optimization

$$argmin_{\{\mathbf{a},\mathbf{b}\}} \sum_{k=-n}^{n} ||w_k(m_k - \mathbf{r}(s_k))||^2, \quad (3)$$

where $w_k = e^{-\frac{(p_{k+e}-p_e)^2}{\sigma^2}}$ are Gaussian weights. The orientation of pixel edge e is then computed as $t_e = \frac{(a)}{\|(a)\|}$.

6.3. Stroke generation based on inside-outside test

Combining orientations of contour-pixels estimated in Subsection 4.2 and orientations of pixel-edges obtained in Subsection 6.2, we can extract strokes from edge-loops. Concretely, for each pixel edge e_0, we again collect its neighbor $e_n, \cdots, e_{-1}, e_0, e_1, \cdots, e_n$ on the same edge loop like having done in Subsection 6.2 and then find their corresponding contour pixels $p_n, \cdots, p_{-1}, p_0, p_1, \cdots, p_n$. If more than half of the inner products between the orientations of e_0 and p_k is greater than τ (an adjustable threshold set to 0.6 in default), then e_0 is labeled as inside the surface contour. Otherwise, it is assigned an outside label.

A Visibility change event happens if adjacent pixel-edges on an edge-loop switching between inside and outside (visible and occluded). The segmentation operator described in Section 5.3 is then used to trace (inside) stroke segments and discard (outside) redundant segments as shown in Figure 9.

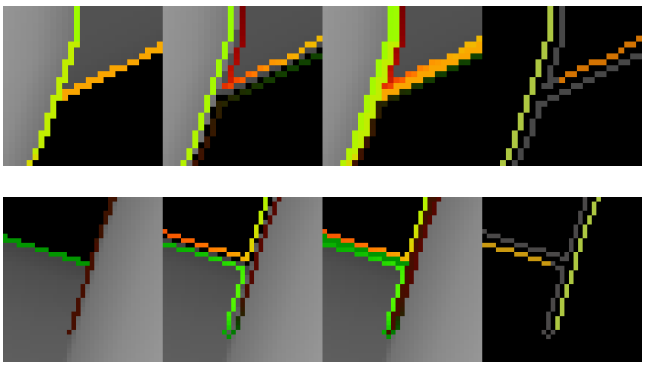


Figure 9: *Inside-outside test: the case of junction (top row) and the case of a cusp and a junction (bottom row). Each row shows contour-pixels, pixel-edges, orientation match, extracted strokes (colored) and redundant segments (gray) rrom left to right.*

The above heuristic may lead to noisy results due to bad mesh quality or image sampling. We leverage the spatial filtering (Section 5.3) to smooth the inside-outside values to make our algorithm more robust. Very short visible strokes are also given up to improve the visual appearance.

Note that each stroke is actually a sequence of pixel edges which should be converted into polylines for rendering. Let $\{e_1, e_2 \ldots e_n\}$ be such a stroke without loss of generality. We simply compute the vertices of its polyline $\{p_0, p_1 \ldots p_n\}$ by setting p_i to the midpoint of e_i. We can smooth the stroke polylines if necessary. In addition, we can compute tangent $\mathbf{t}_i$ of p_i as the orientation of e_i and normal $\mathbf{n}_i$ orthogonal to f $\mathbf{t}_i$ for latter use.

Conventional stroke rendering algorithms [DiV13] can be easily applied to these stroke polylines to achieve stylized results. In order to collaborate with texture mapping, we extend each vertex of a polyline along its the nomral direction [HLW93] to obtain a strip mesh as illustrated in Figure 10, and then create the texture coordinates for each of the mesh vertex on the given texture.

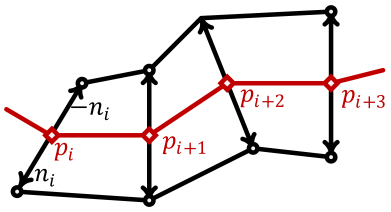


Figure 10: *Stroke parameterization for texturing. Extending points p_i on the path along its normal direction n_i to two sides yields a strip planar mesh which is then mapped to the texture space.*

7. Experimental results

This section first describes our implementation details and then shows the advantages of the proposed framework via a variety of experiments. We will elaborately evaluate the time performance of our approach and discuss its main influencing factors, and then compare our approach with three popular contour rendering systems (Freestyle, Line Art, Pencil+4) and Active Strokes [BJC*12] both in time cost and rendering quality. Our system is developed as a render pipeline in Unity Engine. All runtime procedures are implemented on the GPU by using HLSL shaders.

Our approach and the first three systems run on a PC with Intel i7-7700HQ 2.8 GHz of 8 GB RAM and NVIDIA GTX 1070 while Active Strokes [BJC*12] works on a PC with Intel Core i7-4790K 4GHz of 32 GB RAM and NVIDIA GTX 980Ti due technique reasons. Nonetheless, both hardware configurations are fairly close. All results are generated under 1920×1080 resolution.

7.1. Implementation details

Mesh data is preprocessed and stored in GPU buffers persistently while runtime data such as information for contour edges and pixel-edges is generated from scratch in each frame. For example, the edge-loop pool (Section 5.3) contains many sub-arrays sequentially storing pixel-edges of edge-loops. Each pixel-edge records its index in the sub-array and the length of the sub-array. The order between sub-arrays is determined by the allocation process [Har10].

In the chaining stage, Wyllie algorithm requires $log(n)$ iterations to resolve linked lists with a list length n. In practice, we found that 18 iterations are enough for screen resolution of 2048×2048. One or more stream compactions can be inserted inbetween these iterations to discard short lists that have already finished ranking.

7.2. Performance evaluation

We observe the performance by separating our system into two stages: one is the rasterization process (Section 4) and the other

is the vectorization process consisting of pixel-edge chaining (Section 5) and stroke extraction (Section 6). The model is placed to cover the screen as much as possible in all experiments.

Time cost distribution. Figure 11 depicts the performance under a 1920×1080 resolution, commonly used in real-time rendering applications. Generally, The performance of the first stage is mainly governed by shape complexity and mesh size while performance of the latter is primarily determined by screen resolution. We also known from the figure that the time cost mainly comes from the vectorization stage and is merely affected by the mesh complexity. Figure 11 shows that our system achieves highly real-time performance considering the conventional budget of realtime rendering applications (16ms per frame).

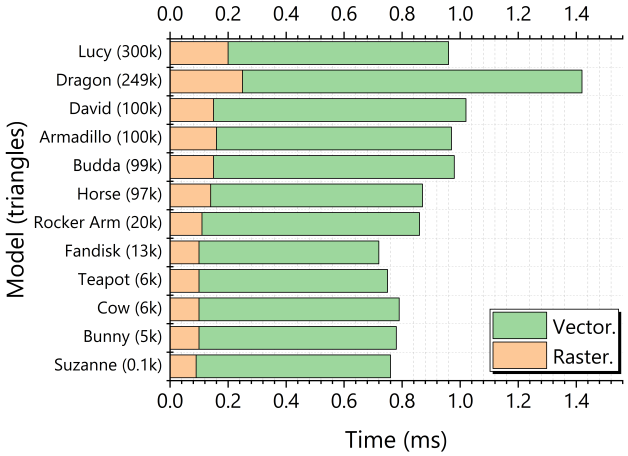


Figure 11: *GPU runtime performance per stage, under* 1920×1080 *resolution*

Sparsity analysis of contour primitives. Our approach greatly benefits in performance from the sparsity of contour edges and contour pixels which determines the GPU workload. To verify this, we rotate the mesh model and record the average of three ratios - ratio between the amount of convex edges and the total edges, contour edges and the total edges, contour-pixels and the pixels covered by the mesh's screen bounding box. Figure 12) (a) illustrates that about 60% convex edges remain after preprocessing while only 4% of total edges are contour edges. Similarly, near 4% of the rendering pixels are contour ones.

7.3. Comparison with previous work

As no existing GPU-based approaches support contour vectorization, all approaches to be compared are CPU-based: Freestyle, Line Art, Pencil+4 and Active Strokes are CPU-based. Both Freestyle and Line Art are line drawing modules of Blender, among which Freestyle is based on a series of work by Grabli and Turquin et al. [GTDS10, GTDS04]. Pencil+4 is a closed-source line drawing renderer, with implementation across multiple softwares. Considering that our system is developed in Unity Engine, we choose the Unity version of Pencil+4 for comparison.

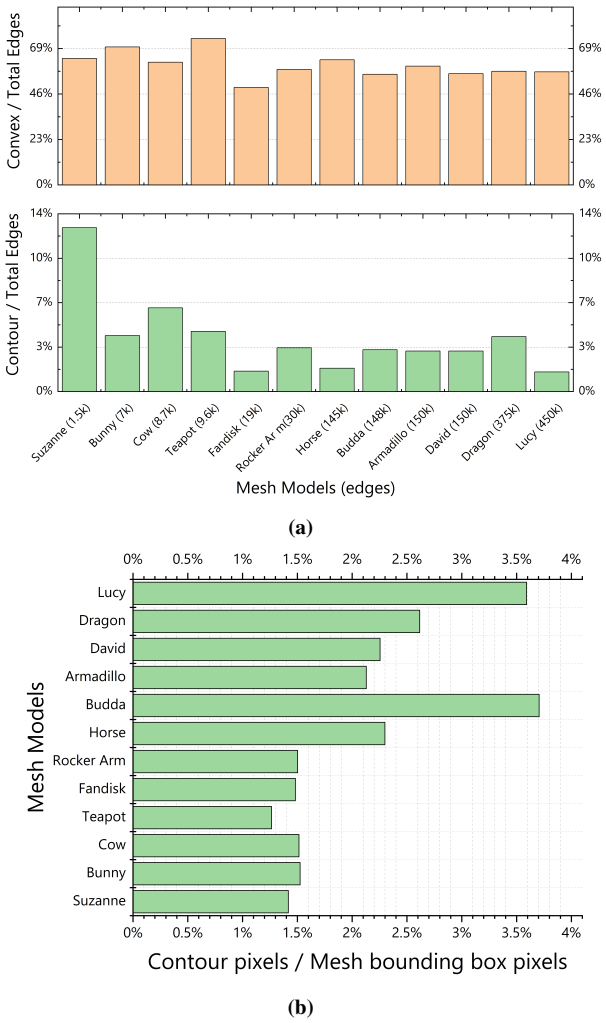


Figure 12: *Sparsity of primitives: (a) Ratios between convex edges and the total edges, and the contour edges and the total edges ; (b) Ratio between contour pixels and screen pixels covered by the mesh bounding box.*

Active Stroke is a prototype based on an image-space line rendering method [BJC*12], which is different from the other three tools in two critical points: (1) it only generates curves in the first frame and then maintains a set of curves throughout the subsequent frames while the other three methods generate stroke curves for each frame; (2) it generates curves from feature samples in the depth buffer with image-space filters (not from actual 3D mesh contours) while the other three methods generate curves from the 3D contour of the mesh model.

Runtime Performance. For methods generating curves in each frame (all except Active Strokes), we profile the time of contour extraction and stroke vectorization, which are the main focus of our method. For Active Strokes, we record the total cost of feature pixel extraction (extract samples and image readback) and curve processing (advection, relaxation, and topology adjustments). Table 1

Table 1: *Comparison of timings among five approaches: ours, Freestyle, Line Art, Pencil+4 and Active Strokes [BJC*12].*

Mesh Model (tris)	Suzanne (0.1k)	Bunny (5k)	Cow (6k)	Teapot (6k)	Fandisk (13k)	Rocker. (20k)
Ours	0.76	0.78	0.7	0.75	0.63	0.86
Pencil+4	7.93	8.43	9.32	8.21	9.22	14.84
Freestyle	27.10	43.32	43.75	44.79	53.34	67.13
Line Art	11.30	18.80	19.65	20.29	28.71	44.69
Active Strokes	54.40	54.68	53.79	46.70	52.26	54.40
Mesh Model (tris)	Horse (97k)	Buddha (99k)	Arm. (100k)	David (100k)	Dragon (249k)	Lucy (300k)
Ours	0.87	0.98	0.97	1.02	1.42	0.96
Pencil+4	68.10	97.67	111.61	67.00	208.00	183.60
Freestyle	242.37	331.10	370.56	383.69	828.47	845.61
Line Art	165.90	189.49	233.44	166.54	445.14	551.52
Active Strokes	51.45	97.09	124.56	91.49	107.43	73.73

demonstrates that our approach achieves tens to hundreds folds of acceleration over other methods. This is because the serial nature of CPU makes it challenging to process massive geometric data. In addition, the iterative processes such as contour vectorization also consume considerable time on CPU (linear time complexity). In contrast, the time complexity of our vectorization algorithm is only $O(logn)$.

Stroke chaining quality. Generally, our method produces stroke curves with quality comparable to or even better than the CPU-based approaches regardless of the complexity of mesh shapes. In our experiments, hybrid methods (all except Active Strokes) were tuned to ensure a coherent configuration: only mesh contour is extracted, and contour elements are chained to stroke curves starting and ending at visibility events. Since the stroke topology of Active Strokes is mainly determined by the curve tracking process, instead of rendering a static scene, we animate the scene to bring motion to the curves and take a screen capture.

All experimental results are presented in Figure 14 in which strokes are drawn with different colors. As shown in areas C and D in the figures, contour by other methods is inappropriately broken into fragmented curves, and the curve topology fails to reflect the occlusion relationship. In contrast, continuous curves by our method match the occlusion relationship better. Nevertheless, area A depicts that our image-based line extraction process can not recognize those endpoints with subtle visibility change on the screen, while other hybrid methods (all except Active Strokes) produce more accurate line distribution. This defect is even more visible for Active Strokes, which links all pixels as a single curve. For dense meshes shown in the third ("David") and fourth column ("Lucy"), our method yields a line topology similar to that by Pencil+4 and Active Strokes and much better than those by Line Art and Freestyle which are highly fragmented as shown in area G to K.

Our algorithm achieves a balance between Pencil+4 and Active Strokes: (1) It leads to more coherent and continuous curves compared with Pencil+4 which links curves directly on meshes because contour-pixels on image often have cleaner topology and smoother geometric attributes than the contour edges on the mesh; (2) It catches more details and better reflects the occlusion relationship than Active Strokes.

7.4. Stylized rendering of complex models

At the end, we present some stylized results of complex models. Figure 15 depicts three types of stroke patterns for Lucy model while Figure 16) illustrates the final rendering results and stylized strokes for two complex monstrous models.

8. Limitations, conclusions and future work

Our method suffers from some disadvantages in some special situations. First, it may ignore subtle contour visibility changes due to lack of accurate 3D contour information to guide the stroke extraction and therefore wrongly connect different strokes together (see Region A in Figure 14). Second, a stroke may be falsely broken if being occluded by other primitive or objects as shown in the top left of Figure 13a. This case worsens for contour features highly clustering on the screen such as thin objects as shown in the bottom left of Figure 13a. It usually does not happen for geometry-based algorithms such as Pencil+4, e.g. seeing the second column in Figure 13b. Third, a minor limitation is that the extracted strokes have a pixel offset from the actual screen-space contour. This artefact will not be perceived generally and can be amended by moving the stroke pixels towards their associated contour-pixels.

Regardless of the aforementioned drawbacks, our method achieves acceleration of hundreds of times against CPU-based methods and is enough to make up for these disadvantages in real-time applications. As future work is to extend our framework to generate temporally coherent stylized contour animations. It is also interesting to integrate the proposed framework into a more complete and powerful GPU contour stylization pipeline [BH19]. A reference implementation of the proposed method is available at `https://github.com/JiangWZW/Realtime-GPU-Contour-Curves-from-3D-Mesh`.

9. Acknowledgements

We thank anonymous reviewers, especially the primary reviewer, for the valuable and careful comments. We thank Pierre Bénard for kindly providing the experiment data of Active Strokes [BJC*12]. We also thank Wengrui Ma and Yiming Wu for helpful discussions on Freestyle and Line Art.

This research is sponsored in part by the National Natural Science Foundation of China (61972160, 62072191), in part by the Natural Science Foundation of Guangdong Province (2019A1515012301 , 2019A1515010860). Guiqing Li is the corresponding author.

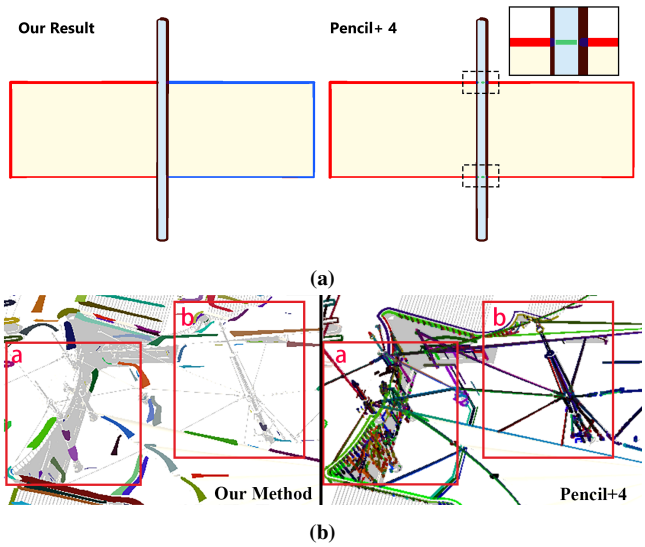


Figure 13: *Limitations: (a) Our method wrongly partitions the rectangle into two strokes due to occlusion by a stick (top left) while Pencil+4 preserves the integrity well (top right); (b) Our method falsely clusters contour features of thin objects (the red rectangle regions, bottom left) and Pencil+4 again yields more reasonable results (bottom right).*

References

[BCGF10] BÉNARD P., COLE F., GOLOVINSKIY A., FINKELSTEIN A.: Self-similar texture for coherent line stylization. In *Proceedings of the 8th International Symposium on Non-Photorealistic Animation and Rendering* (New York, NY, USA, 2010), NPAR '10, Association for Computing Machinery, p. 91–97. URL: https://doi.org/10.1145/1809939.1809950, doi:10.1145/1809939.1809950. 1

[BH19] BÉNARD P., HERTZMANN A.: Line drawings from 3d models: A tutorial. *Foundations and Trends® in Computer Graphics and Vision 11*, 1-2 (2019), 1–159. doi:10.1561/0600000075.

[BHK14] BÉNARD P., HERTZMANN A., KASS M.: Computing smooth surface contours with accurate topology. *ACM Transactions on Graphics 33*, 2 (2014), 1–21. doi:10.1145/2558307.

[BJC*12] BÉNARD P., JINGWAN L., COLE F., FINKELSTEIN A., THOLLOT J.: Active strokes: Coherent line stylization for animated 3d models. In *NPAR 2012 - 10th International Symposium on Non-photorealistic Animation and Rendering* (Annecy, France, 2012), NPAR 2012 - 10th International Symposium on Non-photorealistic Animation and Rendering, ACM, pp. 37–46.

[Ble90] BLELLOCH G.: *Pre x sums and their applications*. Tech. rep., Citeseer, 1990. 4

[BOA09] BILLETER M., OLSSON O., ASSARSSON U.: Efficient stream compaction on wide simd many-core architectures. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), HPG '09, Association for Computing Machinery, p. 159–166. 3

[CF09] COLE F., FINKELSTEIN A.: Fast high-quality line visibility. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games* (Boston, Massachusetts, 2009), Proceedings of the 2009 symposium on Interactive 3D graphics and games, Association for Computing Machinery, p. 115–120. 2

[CF10] COLE F., FINKELSTEIN A.: Two fast methods for high-quality line visibility. *IEEE Transactions on Visualization and Computer Graphics 16*, 5 (2010), 707–717. doi:10.1109/TVCG.2009.102. 3, 4

[CM02] CARD D., MITCHELL J. L.: Non-photorealistic rendering with pixel and vertex shaders. *Direct3D ShaderX: vertex and pixel shader tips and tricks* (2002), 319–333. 2

[CS16] CARDONA L., SAITO S.: Temporally coherent and artistically intended stylization of feature lines extracted from 3d models. *Computer Graphics Forum 35*, 7 (2016), 137–146. doi:https://doi.org/10.1111/cgf.13011. 2

[DiV13] DIVERDI S.: A brush stroke synthesis toolbox. In *Image and Video-Based Artistic Stylisation*, Image and Video-Based Artistic Stylisation. 2013, pp. 23–44. 7

[GDS04] GRABLI S., DURAND F., SILLION F.: Density measure for line-drawing simplification, 2004 6-8 Oct. 2004 2004. 1

[Goo03] GOOCH B.: Silhouette extraction. *Course Notes for Theory and Practice of Non-Photorealistic Graphics: Algorithms, Methods, and Production Systems 6* (2003), 1–10. 2

[GTDS04] GRABLI S., TURQUIN E., DURAND F., SILLION F. X.: Programmable style for npr line drawing. In *Proceedings of the Fifteenth Eurographics conference on Rendering Techniques* (Norrköping, Sweden, 2004), Proceedings of the Fifteenth Eurographics conference on Rendering Techniques, Eurographics Association, p. 33–44. 8

[GTDS10] GRABLI S., TURQUIN E., DURAND F., SILLION F. X.: Programmable rendering of line drawing from 3d scenes. *ACM Transactions on Graphics 29*, 2 (2010), 1–20. 1, 5, 8

[GVH07] GOODWIN T., VOLLICK I., HERTZMANN A.: Isophote distance: a shading approach to artistic stroke thickness. In *Proceedings of the 5th international symposium on Non-photorealistic animation and rendering* (San Diego, California, 2007), Proceedings of the 5th international symposium on Non-photorealistic animation and rendering, Association for Computing Machinery, p. 53–62. 1

[Har07] HARVILL A.: Effective toon-style rendering control using scalar fields., 2007. 2

[Har10] HARRIS M.: *State of the Art in GPU Data-Parallel Algorithm Primitives*. Tech. rep., Nvidia, 2010. 7

[HLW93] HSU S. C., LEE I. H. H., WISEMAN N. E.: Skeletal strokes. In *Proceedings of the 6th Annual ACM Symposium on User Interface Software and Technology* (New York, NY, USA, 1993), UIST '93, Association for Computing Machinery, p. 197–206. URL: https://doi.org/10.1145/168642.168662, doi:10.1145/168642.168662. 7

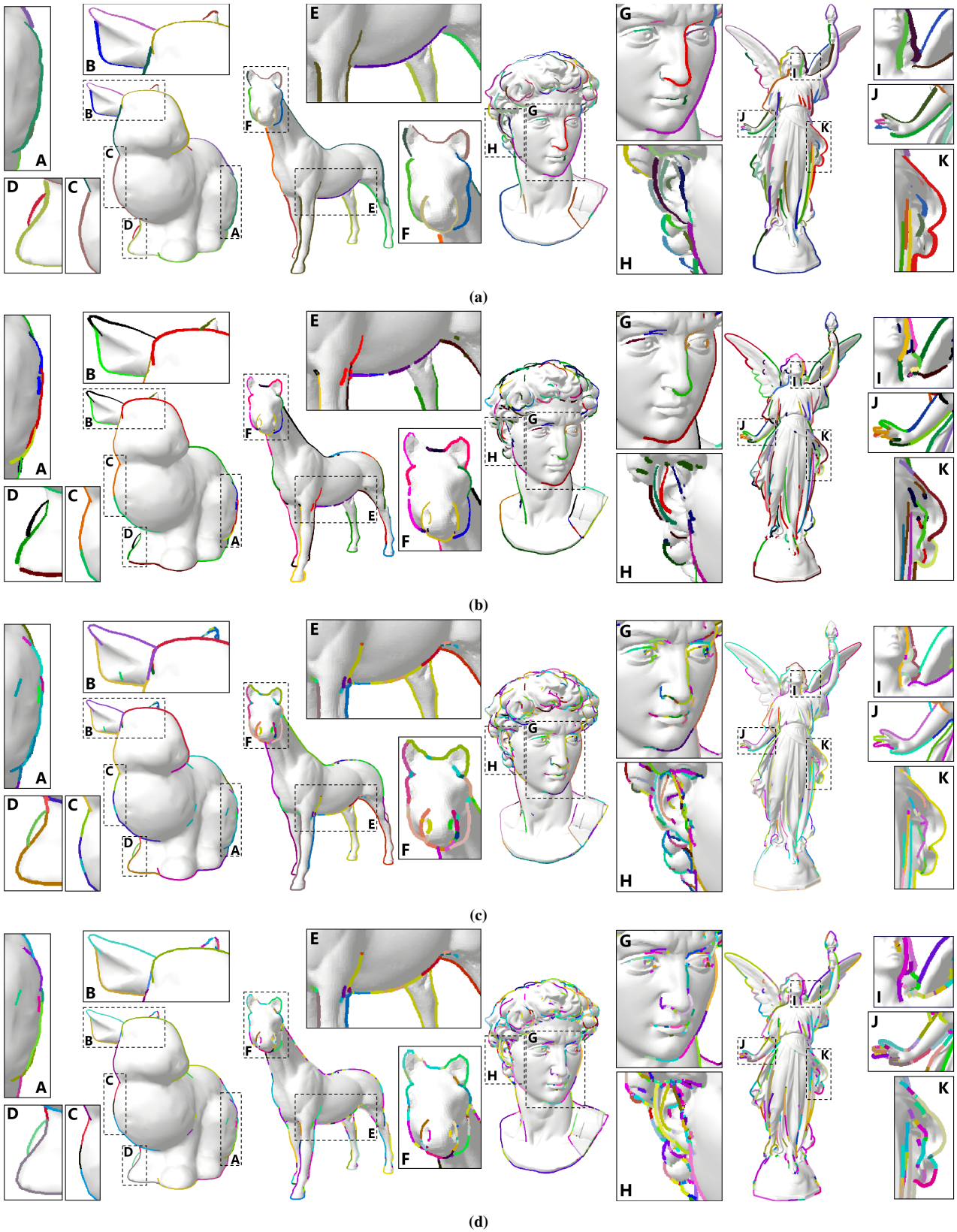
[IHS02] ISENBERG T., HALPER N., STROTHOTTE T.: Stylizing silhouettes at interactive rates: From silhouette edges to silhouette strokes. *Comput. Graph. Forum 21*, 3 (2002), 249–258. URL: https://doi.org/10.1111/1467-8659.00584, doi:10.1111/1467-8659.00584. 3

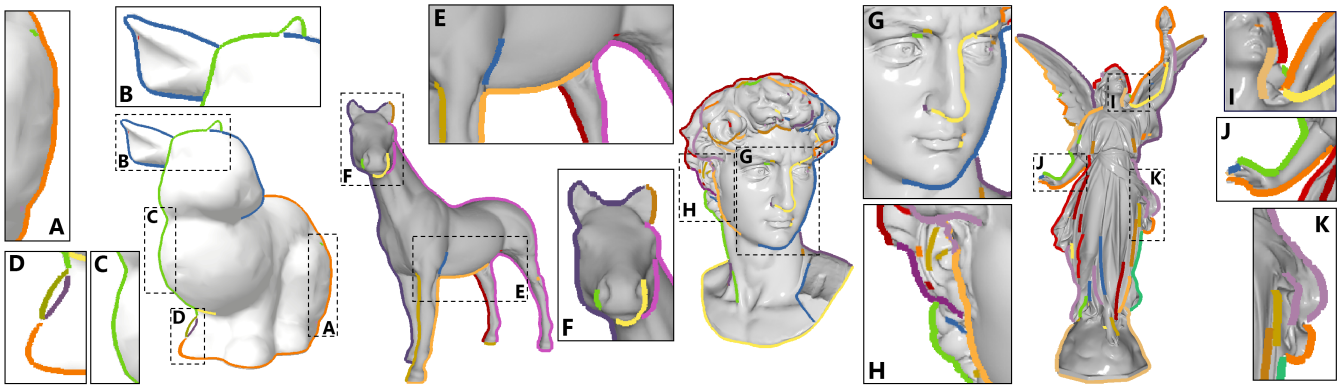
[LFHK21] LIU D., FISHER M., HERTZMANN A., KALOGERAKIS E.: Neural strokes: Stylized line drawing of 3d shapes, October 2021. 2

[LGJLC05] LEWINER T., GOMES JR J. D., LOPES H., CRAIZER M.: Curvature and torsion estimators based on parametric curve fitting. *Computers & Graphics 29*, 5 (2005), 641–655. 6

[LNHK20] LIU D., NABAIL M., HERTZMANN A., KALOGERAKIS E.: Neural contours: Learning to draw lines from 3d shapes, June 2020. 2

[McG04] MCGUIRE M.: Observations on silhouette sizes. *Journal of Graphics Tools 9*, 1 (2004), 1–12. doi:10.1080/10867651.2004.10487594. 3





(e) *Active Strokes*

Figure 14: *(continued) Comparison of contour stroke quality among five approaches: four examples are presented for each approaches and from top to bottom are respectively our approach, Pencil+4, Line Art, Freestyle and Active Strokes. Dotted rectangles on the model are zoom-in regions whose larger versions are placed around the models.*

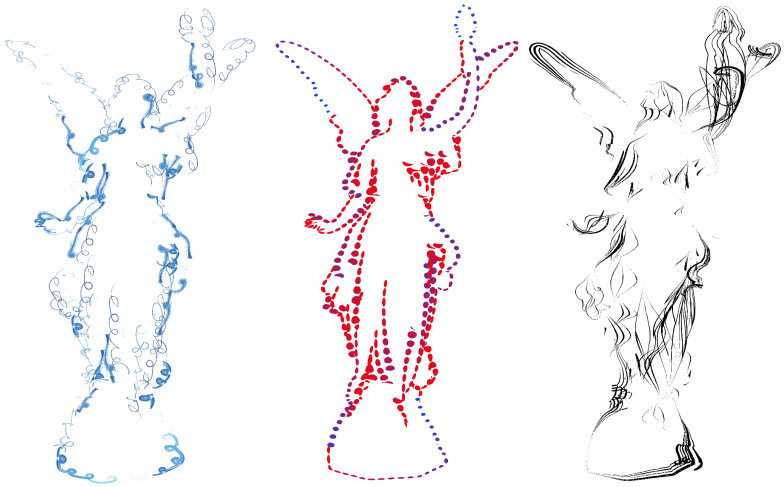


Figure 15: *Stylization with texture mapping: three types of stroke patterns are depicted for the Lucy model.*

[MH04] McGuire M., Hughes J. F.: Hardware-determined feature edges. In *Proceedings of the 3rd international symposium on Non-photorealistic animation and rendering* (2004), Proceedings of the 3rd international symposium on Non-photorealistic animation and rendering, pp. 35–47. 2, 3

[Mor66] Morton G. M.: A computer oriented geodetic data base and a new technique in file sequencing. 5

[ND04] Nienhaus M., Döllner J.: Sketchy drawings. In *Proceedings of the 3rd international conference on Computer graphics, virtual reality, visualisation and interaction in Africa* (Stellenbosch, South Africa, 2004), Proceedings of the 3rd international conference on Computer graphics, virtual reality, visualisation and interaction in Africa, Association for Computing Machinery, p. 73–81.

[NM00] Northrup J., Markosian L.: Artistic silhouettes: A hybrid approach. In *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering* (2000), pp. 31–37. 3

[PSM*13] Peciva J., Starka T., Milet T., Kobrtek J., Zemcik P.: Robust silhouette shadow volumes on contemporary hardware. In *GraphiCon'2013* (2013), pp. 56–59. 3

[RC99] Raskar R., Cohen M.: Image precision silhouette edges. In *Proceedings of the 1999 symposium on Interactive 3D graphics* (Atlanta, Georgia, USA, 1999), Proceedings of the 1999 symposium on Interactive 3D graphics, Association for Computing Machinery, p. 135–140. 2

[Sel03] Selinger P.: Potrace: a polygon-based tracing algorithm. *Potrace (online), http://potrace. sourceforge. net/potrace. pdf (2009-07-01)* (2003). 2, 5

[SHG*] Sengupta S., Harris M., Garland M., et al.: Efficient parallel scan algorithms for gpus. 4, 5

[ST90] Saito T., Takahashi T.: Comprehensible rendering of 3-d shapes. *SIGGRAPH Comput. Graph. 24*, 4 (1990), 197–206. doi:10.1145/97880.97901. 2

[WKS07] Wächter C., Keller A., Stich M.: Efficient and robust shadow volumes using hierarchical occlusion culling and geometry shaders, 2007. 3

[Wri90] Wright W. E.: Parallelization of bresenham's line and circle algorithms. *IEEE Computer Graphics and Applications 10*, 5 (1990), 60–67. 2, 3, 4

Figure 16: *Stylization with toon shading for two monsters: each row shows the final render (left) and stylized strokes (right).*

[Wyl79] WYLLIE J. C.: *The Complexity of Parallel Computations*. PhD thesis, Cornell University, 1979. 2, 5

[XFZ16] XIONG X., FENG J., ZHOU B.: Real-time image vectorization on gpu. In *VISIGRAPP (1: GRAPP)* (2016), pp. 143–150. 2