Graz University of Technology

Institute for Computer Graphics and Vision

Master's Thesis

Non-Photorealistic Line Rendering for Handheld Augmented Reality

---

Peter Mohr

Graz, Austria, October 2013

Thesis supervisors

Univ.-Prof. Dipl.-Ing. Dr. techn. Dieter Schmalstieg

Dipl.-Ing. Dr. techn. Denis Kalkofen

MSc. Dr. Eduardo Veas

Abstract

Silhouettes play an important role for visualizing three-dimensional objects. Only a few graphical elements are required to convey their form and relative position. This is why silhouettes are often applied in abstract graphics like technical drawings or illustrations. Silhouettes are often drawn using various line styles to indicate additional information like occlusion or depth. This property can now then be used to visualize complex environments in mobile augmented reality with the help of stylized silhouettes in an easier perceptible way. The goal of this thesis is thus the realization of automatic calculation and visualization of stylized silhouettes of real objects in a mobile augmented reality application. There exist elaborate algorithms and data structures for the automatic creation of stylized silhouette drawings. However, these algorithms were developed for traditional desktop systems and are not suited for mobile devices. In the course of this work different algorithms and methods for efficient silhouette calculation of 3D models were examined. Beyond that, new algorithms for the creation of continuous strokes from individual silhouette edges and also their stylization with textures are described in this thesis. Furthermore an implementation of these methods is discussed, which evaluates them in terms of quality and speed on a smartphone and a modern desktop PC system. Finally, use cases of the implemented mobile augmented reality system are described and silhouette overlays on real world objects are shown, using different line styles.

Keywords. Augmented Reality, silhouette detection, mobile, stylization

Kurzfassung

Silhouetten spielen eine wichtige Rolle bei der Visualisierung dreidimensionaler Objekte. Sie erlauben es mit wenigen graphischen Elementen deren Forum und relative Lage darzustellen. Daher finden sie häufig Anwendung in abstrakten Grafiken wie etwa in technischen Zeichnungen oder Illustrationen. Silhouetten werden darüber hinaus häufig in verschiedenen Linienarten gezeichnet, um so weitere Informationen wie Kantenverdeckungen oder Tiefe kodieren zu können. Diese Eigenschaft lässt sich nun ausnutzen um komplexe reale Umgebungen, mit Hilfe von stilisierten Silhouetten in einer mobilen Augmented Reality Visualisierung, einfacher wahrnehmbar zu machen. Ziel dieser Arbeit ist daher die automatische Berechnung und Darstellung stilisierter Silhouettenkanten realer Objekte in einer mobilen Augmented Reality Anwendung umzusetzen. Zur automatischen Erstellung stilisierter Silhouettenzeichnungen existieren aufwendige Algorithmen und Datenstrukturen welche für traditionelle Desktop-Systeme entwickelt wurden und sich für mobile Geräte nicht eignen. Im Zuge dieser Arbeit wurden daher Algorithmen und Methoden zur effizienten Berechnung von Silhouetten auf 3D-Modellen untersucht. Darüber hinaus beschreibt diese Arbeit neuartige Algorithmen zur Erzeugung von zusammenhängenden Kantenzügen aus einzelnen Silhouettenkanten sowie zu deren Stilisierung mit Hilfe von Texturen. Außerdem wird eine Implementierung der Methoden diskutiert, die sie unter dem Aspekt ihrer Leistung auf einem Smartphone und einem aktuellen PC-System evaluiert. Zum Abschluss beschreibt diese Arbeit Anwendungsfälle des implementierten Systems und zeigt dabei die Überlagerung von realen Objekten mit ihren virtuellen Silhouetten in verschiedenen Stilen.

Stichwörter. Augmented Reality, Silhouettenerkennung, Stilisierung

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Non-photorealistic rendering (NPR) is an area of research which investigates algorithms to automatically generate illustrative renderings. There are numerous applications from technical illustration to medical visualizations. A large part of the computer graphics research deals with photorealistic renderings where the goal is to mimic the reality as faithfully as possible. NPR on the other hand tries to generate abstract renderings with the goal to emphasize those features that maximize information for the given goal, and omit those deemed unnecessary. NPR techniques like stippling, hatching, water and oil color imitation or toon shading aim to automatically create an artistic rendering.

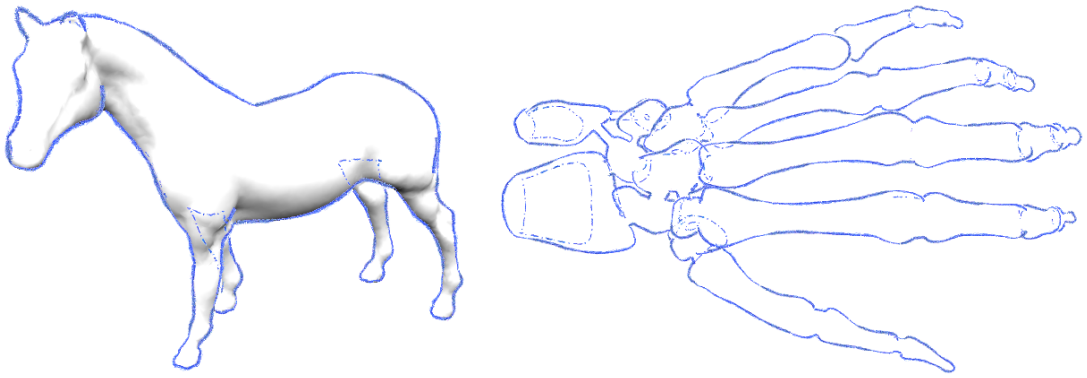


Figure 1.1: Silhouette rendering of complex shapes with stylized lines. The style was created using a colored pencil stroke. Hidden lines have a different (dot-dash) texture. Both images are produced by the framework implemented in this thesis.

This thesis focuses on one important part of NPR, the rendering of stylized lines. Technical illustrations like the mechanical parts in Figure 1.3 should convey information of the shape, which is done using feature lines like the silhouette and other distinctive creases

and borders. The power of line drawings is that they can convey a lot of information about the shape and structure of an object with very few distinctive strokes. Figure 1.1 shows two objects rendered with stylized lines.

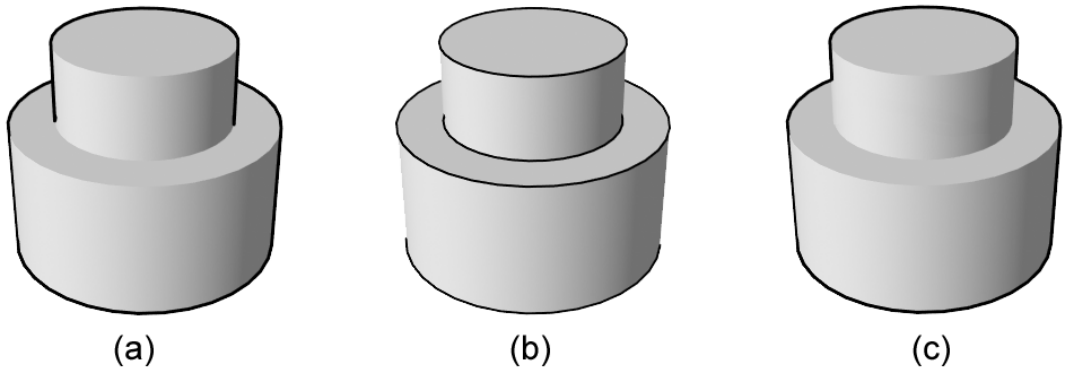


Figure 1.2: Important feature lines: silhouettes (a), sharp creases (b) and contour (c).

Figure 1.2 shows the most important feature line types. The silhouette of an object is located at the locations where the surface normal is perpendicular to the view direction (Figure 1.2a). Sharp creases also provide important cues for the structure of the object. While the silhouette is different for every viewpoint, crease edges are static because they only depend on the geometry (Figure 1.2b). The contour (Figure 1.2c) separates the object from the background. It is similar to the definition of the silhouette, except that no internal edges are considered.

Artists have used the technique of line drawings to convey the 3D shape long before the age of computers, and algorithms have been developed to automatically determine the important feature lines. A recent study by Cole et al. [CGL+08] shows that lines drawn by artists are mostly consistent with renderings generated by current line drawing algorithms. Cole et al. asked several artists to draw the same collection of natural and artificial 3D objects as a line graphic without shading. The drawings where scanned and then multiple evaluations were performed. The results show that approximately 75% of all drawn lines are consistent between the different artists. Further analysis showed, that the combined lines generated by four common image and object space line rendering algorithms cover 86% of the artists lines.

Common to almost all line drawings is the presence of the objects silhouette (see Figure 1.2a), which is the set of edges which define the outline of an object. The silhouette line shows the basic shape of the object and also separates it from the background or other objects. While other feature lines like sharp creases or borders can be determined in a

precomputation step, the silhouette needs to be recomputed every time the viewpoint changes. Fast silhouette detection is an important area for research because it is one of the limiting factors for interactive line drawing applications. Current desktop computers with dedicated graphics hardware can achieve interactive frame rates for line renderings of detailed meshes. Isenberg et al. [IHS02] have created a NPR framework, which features a silhouette rendering pipeline among other NPR visualization techniques. They achieved interactive frame rates on hardware with a fraction of the computing power of a current consumer personal computer (PC).

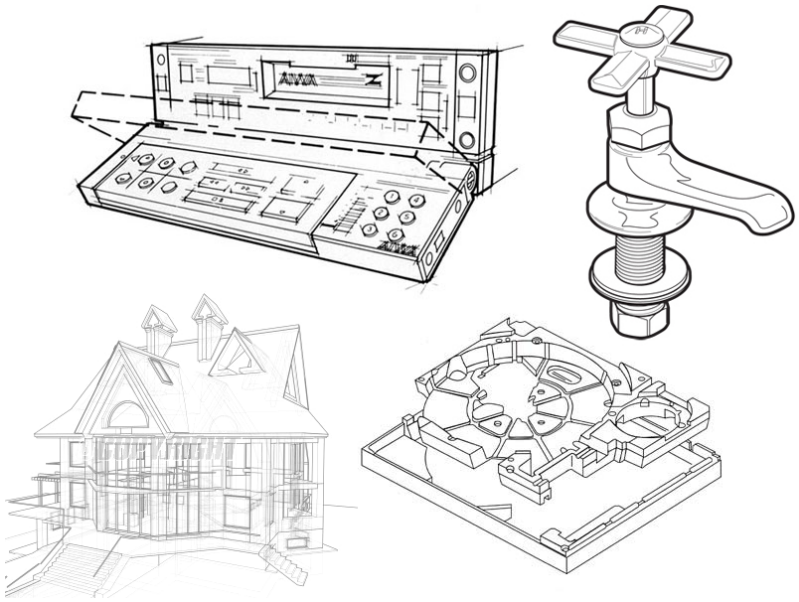


Figure 1.3: Different examples of line drawings. The silhouettes and crease lines convey information of the basic shape of the objects.

Line stylization is an important topic for line rendering applications. Stylization is used to enhance the display of line drawings. A simple line itself may vary in thickness and color, but it is desirable to have more options to change the appearance of the line. Line stylization options include varying thickness, color and transparency as well as wavy lines or artistic strokes like open ended ink strokes. Isenberg et al. [IHS02] demonstrate some these line variations in their framework. Line illustrations can be used to emphasize the overall shape of an object. In combination with an augmented reality system, the real world image can be enhanced to show the user an augmented version of an object of interest. For example, a large complex machine or component can be outlined without obstructing the view of the real object. It is also possible to show how the machine looks on the backside with dashed or more transparent lines. Furthermore the internal workings

of the machine can be displayed. The scope of application for an AR stylized line drawing system covers for example maintenance, training for operators, marketing or interactive user manuals (imagine an IKEA or Lego interactive construction manual, like the mockup in Figure 1.5).

1.1 Mobile Augmented Reality

Figure 1.4: Handheld AR example: The invisible train, a multi-user Augmented Reality application for handheld devices. [WPLS05]

With increasing hardware capabilities of smartphones and tablets on the market, mobile devices become an active area of augmented reality (AR) research. Current handheld devices already feature multi-core processors and also, more importantly, integrated graphics hardware capable of displaying complex 3D content at interactive frame rates. In most cases the devices also have an integrated camera which can be used by tracking algorithms to register a 3D object in the real world (Figure 1.4). The provided touchscreen can be used to implement an intuitive user interface. Another, very important aspect of mobile devices is their widespread availability. Smartphones and tablet computers are produced for the mass market, and sometimes also subsidized by mobile telephone providers which results in a relatively low price. AR applications with Virtual Reality Glasses or a CAVE

system are nice, but very expensive and not available to everyone. AR applications on mobile devices however can target a much greater audience.

This thesis aims to bring a silhouette rendering application to mobile devices. There are several constraints to handheld devices, for example they still have lower processing speed and limited memory in comparison to a desktop computer. In addition, the graphics hardware has a reduced feature set i.e. there is no geometry shader which would be particularly useful for silhouette visualization. To deal with the named limitations, it is necessary to carefully choose and optimize the algorithms used for silhouette detection and visibility determination.

1.2 Goal

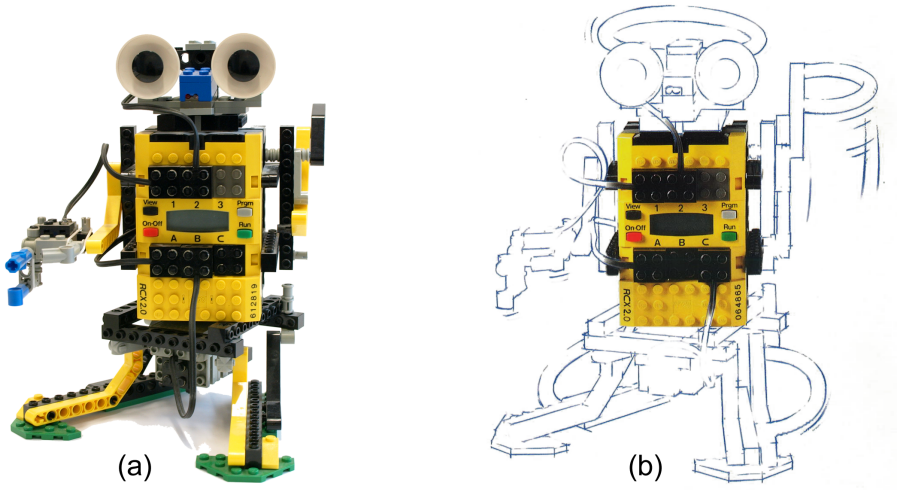


Figure 1.5: Illustrative line renderings: normal view of a Lego robot (a). Stylized lines can be used to outline various parts (b).

The goal of this thesis is to generate a silhouette for a large mesh on a handheld device at interactive frame rates. In the scope of this thesis a mesh is considered large if it has more than 200.000 polygons. The actual meshes used in this thesis range from 4.000 to 250.000 polygons.

Problems: It is not feasible to load such large meshes unmodified on a mobile device since it would consume too much memory. Additionally the silhouette detection and calculation would be too slow. Silhouette detection algorithms deliver a set of edges which are arbitrarily ordered. To allow stylization, the edges need to be concatenated to continuous strokes.

Proposed solutions: It is necessary to reduce the amount of polygons of the mesh handed to the mobile device. This step can be carried out on a normal desktop computer and the resulting low polygon mesh is then send to the mobile device. This kind of distributed computing approach swaps static, memory and processing intensive operations out to improve performance. Additionally the silhouette detection process must be efficient to achieve interactive frame rates. The proposed solution is to narrow down the search region for silhouettes as much as possible. In a process called seeding, a set of polygons is marked as start points for a local search. The focus of this thesis lies on fast silhouette generation, stylization and visibility determination on a mobile device. Other distinctive mesh features require no per-frame computation and can therefore be calculated offline in a preprocessing step.

Besides this introductory chapter, the thesis is structured into five further chapters. Chapter 2 summarizes related research work in the fields of silhouette detection algorithms, line visibility and stylization. The chapter provides an overview of existing algorithms for silhouette extraction in image and object space. Their benefits and drawbacks are briefly discussed, and special attention is paid to the intended use on a mobile device.

Chapter 3 describes the structure of the silhouette rendering engine. Based on algorithms discussed in chapter 2 a concept is created which fulfils the requirements stated in the motivation of the thesis. The chapter starts the introduction of the half-edge data structure, which represents the mesh in the form of so-called half-edges. The structure provides efficient mechanisms to navigate on the mesh which facilitates neighborhood search operations. Next, different seeding algorithms are developed with special attention to fit the environment of a mobile device. Problems of the algorithms are discussed in detail, especially in relation to performance and visual quality. The last part of the chapter deals with two related topics, stroke generation and stylization. Continuous strokes are required to enable coherent stylization of silhouette lines. An efficient method for line visibility determination using shader programs is introduced.

Based on the concept introduced in chapter 3, chapter 4 describes some implementation details of the silhouette rendering engine. Several aspects are described in detail and source code snippets are provided. A short introduction demonstrates how arbitrary line styles can be integrated in the system with very few effort. The functionality of the concept is first tested on a prototype application on a desktop computer. Changes made to the code to port the prototype to the mobile device are documented.

Chapter 5 shows the results of the implementation. The developed silhouette detection

algorithms are compared to each other in several evaluation runs, both on the PC and the mobile device. The processing time of the optimized algorithms is measured against the basic brute force algorithm. The quality of the silhouette algorithms is compared to the ground truth provided by the brute force algorithm. The performance of the algorithms is also tested with different mesh resolutions to determine the scalability. The discussion section sums up the evaluation and the results for each algorithm are discussed in detail.

The concluding chapter 7 gives a short summary of the concepts and findings of this thesis. The findings of the usage examples provide a basis for future improvements of the methods.

Chapter 2

Related Work

Contents

There are numerous approaches to silhouette extraction and visualization which greatly vary in their properties. Basically silhouettes can be extracted either in image space or in object space. Image space algorithms are usually fast because modern graphics hardware supports most of the necessary operations, but it is difficult to stylize the resulting silhouette edges. Their speed depends mostly on the resolution of the output image. Object space approaches deliver silhouette edges in the form of 3D vertices and edges which can be easily stylized and modified, but their speed is dependent on the polygon count of the rendered object and the calculations are usually done on the CPU. The speed penalty of object space approaches can be reduced by applying various optimisations, mostly by reducing the amount of geometry the algorithm has to process in each frame. Hybrid approaches usually combine object and image space algorithms.

2.1 Crease edge extraction

Apart from silhouette edges there are other significant feature lines of an object, e.g. sharp creases and border edges. Creases are found by comparing the angle between the two faces of an edge to a threshold value. A cube is a good example for sharp creases, because all faces are joined at a 90 degree angle. Figure 2.1a shows the crease edges on a cube. Border edges on the other hand have only one adjacent face and appear on single faces or

non-closed mesh surfaces, as can be seen on the partial torus in Figure 2.1b. Crease and border edges can be identified in a preprocessing step and are not part of the silhouette detection problem.

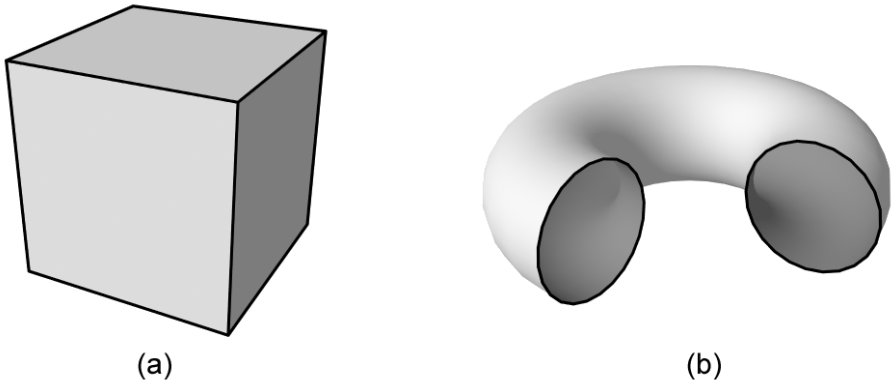


Figure 2.1: Creases and borders: The faces of a cube are joined at a 90 degree angle and form sharp creases (a). Border edges have only one adjacent face (b).

2.2 Silhouette Extraction

The silhouette of an object is located at the point where the dot product between the surface normal at this point and the view vector is zero, which means the two vectors are perpendicular to each other. In most practical real time applications the object is composed of discrete triangles, so a silhouette edge is defined by an edge shared between a front facing and a back facing triangle. The simple brute force object space algorithm described in section 2.2.2 finds the silhouette in this way. Other, more advanced algorithms produce better silhouettes at the expense of computation time.

2.2.1 Image Space Silhouette detection

Image space silhouette detection algorithms operate on one or more image buffers to find silhouette edges. This is usually done by rendering the 3D object to an accessible buffer and then applying an edge detector to find discontinuities in the rendered image. Since the edge detection would produce a lot of noise on a color shaded and textured rendering, the detection is usually done in the depth buffer [ST90]. Figure 2.2 illustrates this basic process. This method is very fast on accelerated graphics hardware since all the computations can be done in shader programs which run in parallel [MBC02].

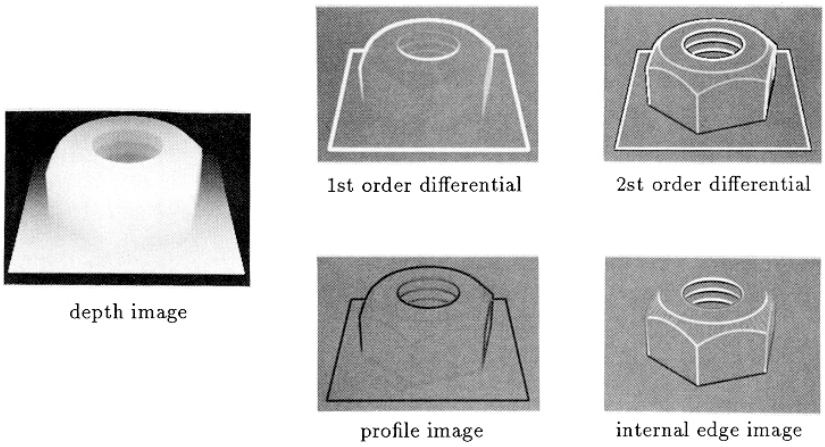


Figure 2.2: Silhouettes derived from the depth buffer. Image adapted from [ST90]

An improvement to Saitos [ST90] method is described in [Her99], where a normal buffer is used to additionally detect C1 discontinuities. The combination of edges found in both buffers yields a good result (see Figure 2.3).

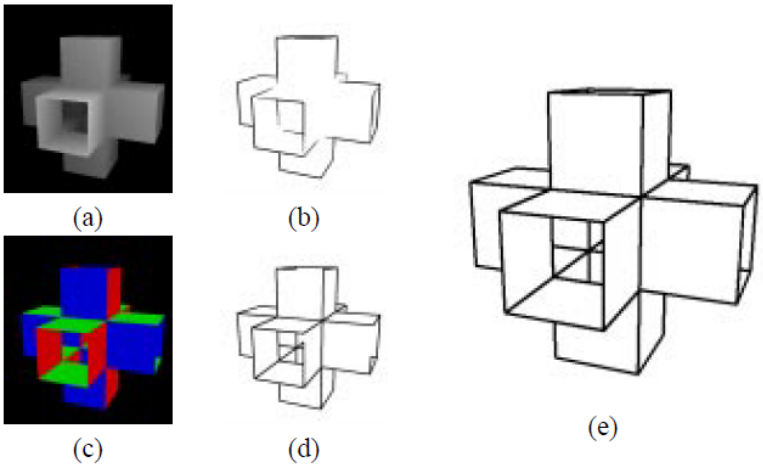


Figure 2.3: Enhanced edge detection in image space: Depth buffer (a), edges derived from the depth buffer (b), normal map (c), edges derived from the normal buffer (d) and the combined edge image (e). Adapted from [Her99]

Since image space algorithms operate on a rendered buffer the complexity depends only on the number of pixels in the image, which usually does not change between frames. The algorithms also can be easily implemented, especially when the computations are done in shader programs.

One important quality of image space algorithms is that they are also independent

regarding the type, resolution and quality of the 3D object, because they operate only on the 2D pixel image. This makes them agnostic to the mesh. Additionally there is no need for special mesh data structures, e.g. stored neighborhood information. The precision of image space algorithms is naturally limited to pixel precision since they operate on a rendered image. In most cases the precision is sufficient because the output of the algorithm is displayed in the same resolution in pixels. One downside of image space algorithms is that there is no easy and fast way to get a three dimensional description of the detected silhouettes. This is required for stylizing connected silhouette edges. One possible way to gain a analytical description out of the detected image space silhouette edges is to read back the pixel data from the graphics hardware and then fit curves or line segments to the pixel outline. Loviscach [Lov02] describes an approach which fits Bézier curves to the pixel silhouette (Figure 2.4). Unfortunately both reading back data from the graphics hardware and the curve fitting process are rather slow which makes this algorithm difficult to use at interactive frame rates.

Figure 2.4: Stylization using fitted curves: A simulated ink brush was used to stylize the silhouettes. Image taken from [Lov02]

2.2.2 Object Space Silhouette Detection

To achieve more stylization options it is necessary to compute an objects silhouette in 3D coordinates. Object space algorithms operate directly on the input mesh instead of image buffers.

For image space algorithms the visibility of the silhouette lines is determined automatically through the graphics hardware when the mesh is rendered. Object space algorithms usually have a separate step for computing visibility.

Vertex	VFB	VFB	VFB	VFB
1	**211**	300	**411**	500
2	**311**	500	x00	x00
3	**411**	500	x00	x00
4	500	x00	x00	x00
5	x00	x00	x00	x00

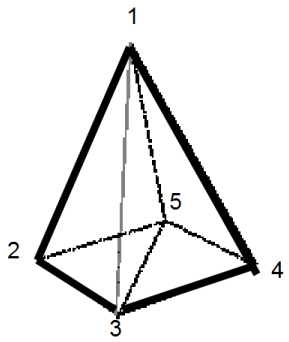


Figure 2.5: The edge buffer: Edges store additional flags for front facing (F) and back facing (B) faces adjacent to an edge. Image taken from [BS00]

The brute force algorithm simply follows the definition of a silhouette edge. It iterates over all edges of the polygonal mesh and checks the two adjacent faces of each edge if they are front or back facing. If the edge has one front face and one back face, then it is a silhouette edge. Figure 2.6 shows an example. The underlying data structure of the mesh is important for the performance of this algorithm. For example, Buchanan and Sousa [BS00] use an edge buffer structure which stores additional information in the edges (Figure 2.5). The algorithm they present tests for each face whether it is front or back facing. It then updates either the front or back indicator bit of all the face's edges using XOR. After this first pass only the edges with both bits set are silhouette edges and are rendered. Another suitable data structure is the half-edge structure which stores adjacency and neighborhood information in so-called half-edges. This data structure allows iterating over the edges of the mesh and gives direct access to the adjacent face of each edge. Regardless of the underlying data structure this simple algorithm will always find all silhouette edges and works in orthographic and perspective projection. It is also convenient to implement since the only calculations to be carried out are dot products of face normals and the view vector.

On the downside the algorithm is not suitable for larger meshes because it needs to traverse the whole mesh and calculate the dot products for every frame. The computation time per frame is linear to the number of edges in the mesh. Since only a small amount of the edges are silhouette edges, this algorithm is quite inefficient. It is difficult to achieve interactive frame rates for larger meshes even when a efficient data structure for the mesh is used. The silhouette found by this simple algorithm is likely to have artefacts, such as edge clusters or jagged lines, due to the finite accuracy of the polygonal mesh. The problem is that the original edges are used as silhouette edges. Figure 2.7 shows some

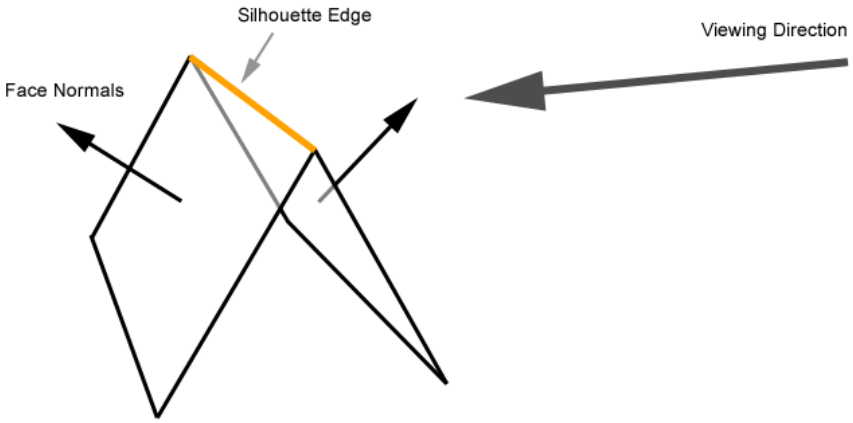


Figure 2.6: Object space silhouette detection: An edge is a silhouette edge if the normal of one adjacent face points away from the viewer and the other face normal points towards the viewer.

of the artefacts that might occur. There are methods to eliminate these artefacts, as described in [IHS02]. The artefact removal step is carried out after silhouette detection in a separate step.

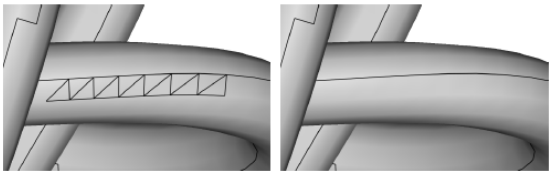


Figure 2.7: Edge clusters: This artefact can occur because of numerical instabilities when the faces are almost parallel to the viewing direction. Image taken from [IHS02]

Hertzmann and Zorin [HZ00] describe an subpolygon approach where the silhouette edges are interpolated across the faces. The algorithm uses the vertex normals instead of the face normals to approximate the polygonal mesh as a smooth surface. A silhouette edge is found by calculating the dot product between the view vector and each vertex normal of the face. If the dot products at the two vertices of an edge have a different sign, then there is a zero crossing between them. The position of the crossing where the view vector is perpendicular to the normal is then linearly interpolated on the edge. Figures 2.8 and 2.9 illustrate the result of this operation. This algorithm produces a more accurate silhouette and is less prone to artefacts like edge clusters and zigzag lines. This also makes it very suited for further line stylization. Like the aforementioned simple silhouette

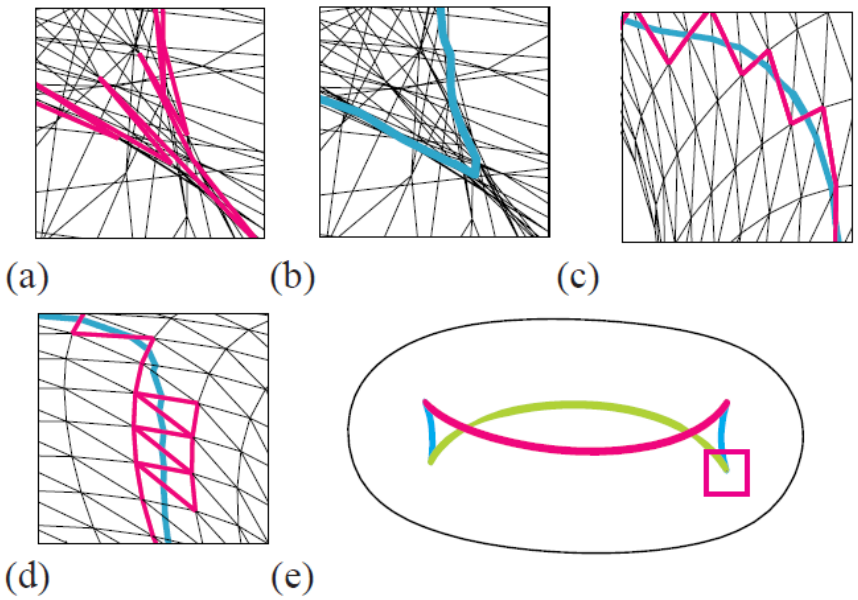


Figure 2.8: Silhouette interpolation: Artefacts occur at faces positioned almost parallel to the view direction. The interpolation method yields smooth results. Image taken from [HZ00]

algorithm the subpolygon algorithm also needs to iterate through all the faces of the mesh. Moreover, it calculates three dot products for each face instead of two. A simple speedup is to store the dot product values for each vertex once it is calculated and use the result again for the neighboring faces.

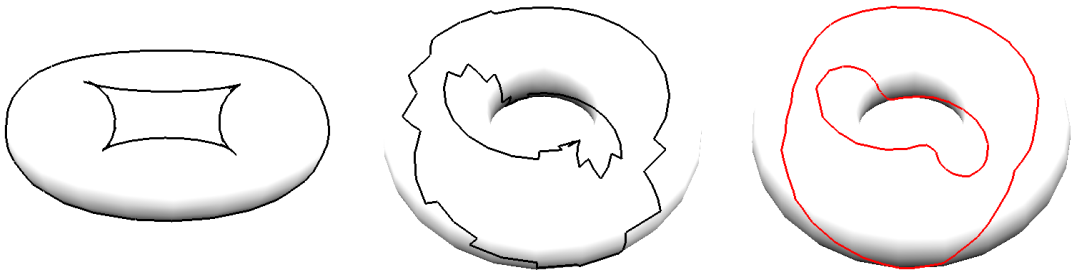


Figure 2.9: Comparison between the simple and the sub-polygon silhouette detection algorithm. The sub-polygon silhouette does not show any artefacts and is much smoother than the non-interpolated version.

Both of the previous described algorithms need to traverse the whole mesh in every frame to compute the silhouette, thus interactive frame rates are only possible for small

meshes.

Markosian et al. [MKT+97] presents a method to speed up the silhouette finding process by selecting only a few candidate faces randomly for the initial computation. If a silhouette is found the algorithm continues to search in the face neighborhood and tries to follow the silhouette. Additionally the algorithm uses the fact that the silhouette for the next frame is usually spatially close to the current silhouette. This also speeds up the search for silhouette edges. Since only a fraction of the faces respectively edges of a mesh contribute to the silhouette this approach provides a speedup also for large meshes. As a negative, this approach is not guaranteed to find all silhouette edges.

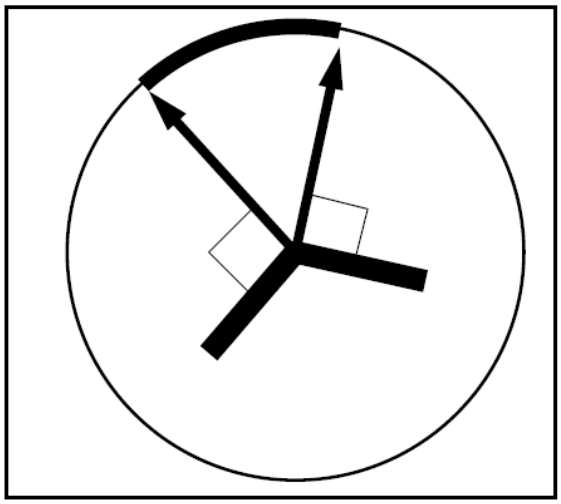


Figure 2.10: 2D representation of the Gauss map: The projection of the normals of two connected faces form an arc on the circle. Image taken from [GSG+99]

There has been more research to speed up the silhouette detection by adding a preprocessing step to optimize the detection speed. Gooch et al. [GSG+99] present a method where the face normals are projected onto a gaussian sphere. The projection of the normals of two adjacent faces form an arc on the sphere, which represents an edge (see Figure 2.10). The view plane maps to a plane through the center of the gaussian sphere, but only if it is an orthographic projection. Now the silhouette edges can be detected by intersecting the view plane with the arcs on the sphere. If one arc is intersected it means that the corresponding edge has one front and one back facing polygon respectively to the view direction and is therefore a silhouette edge. This method eliminates the need to check each face if it is front or back facing. Additionally the data can be stored in a hierarchical way to limit the number of intersection tests. Gooch uses a octahedron or icosahedron

as a starting point and successively subdivides it. The arcs are stored at the leafs of the hierarchical structure.

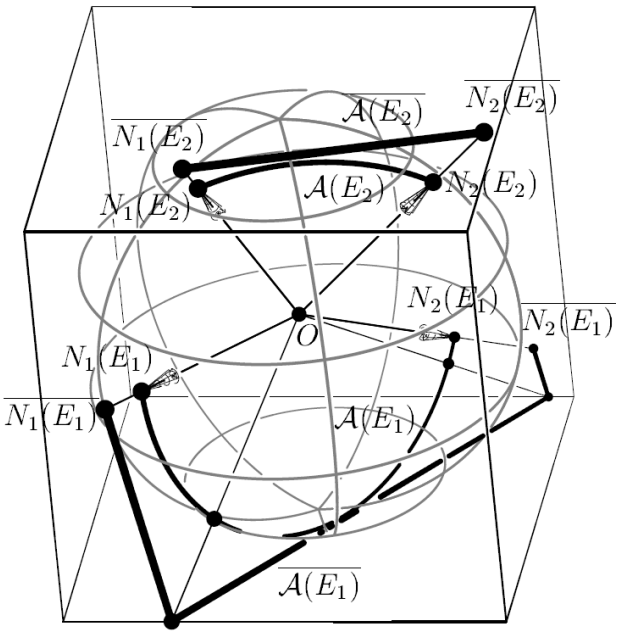


Figure 2.11: The arcs on the gaussian sphere are mapped to a circumscribing cube. Image taken from [BE99]

Benichou and Elber [BE99] propose a similar approach where they map the arcs of the gaussian sphere to line segments on a circumscribing cube (Figure 2.11). The cube surface is subdivided into a grid so intersection tests are only needed for grid segments which contain the mapped viewplane. Similar to Gooch, their method also works only in orthographic projection. To overcome this limitation, Hertzmann and Zorin [HZ00] present an approach which is also based on the properties of a dual representation in 4D. To find silhouette edges, the plane created by the viewpoint in 4D is intersected with the eight sides of a hypercube. Similar to Gooch, the sides of the hypercube are decomposed into octrees for speedup. This method works for both orthographic and perspective projection.

A different method for speeding up the silhouette detection is described by Sander et al. [SGG+00a]. The approach also uses hierarchical grouping to speed up the detection process. The mesh edges and attached faces are stored in a tree. The branches contain face clusters and additional information in the form of two cones created by the face clusters normals (Figure 2.12). The cones define the possible positions of the viewpoint where the face cluster is either entirely front facing or back facing. If a cluster is entirely front or

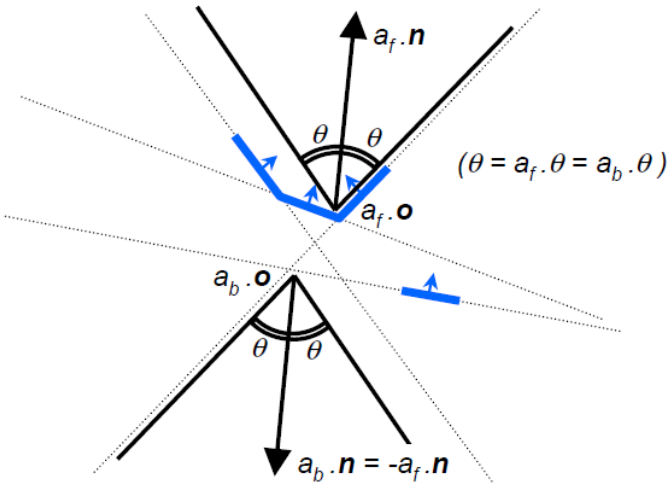


Figure 2.12: Faces and edges are stored in a tree structure. The normals of a branch form cones. Entire clusters of entirely front or back facing polygons can be ignored during silhouette calculation. Image taken from [SGG+00a]

back facing, the whole subtree can be disregarded for the silhouette search.

Other than the stochastic approach by Markosian [MKT+97] the precomputation methods reduces the number of faces and edges which have to be traversed, respectively, but it is still guaranteed that all of the silhouette edges are found. The limitation of preprocessing takes effect when animation of the mesh is required. Once the mesh and therefore the face or vertex normals changes, the precomputed data becomes invalid and needs to be recalculated. Since the preprocessing step is usually expensive in terms of runtime and memory, it is difficult to reach interactive frame rates for a changing mesh.

2.2.3 Hybrid Approaches

Hybrid approaches incorporate both object space and image space procedures. The output is a pixel silhouette image similar to the pure image space algorithms. Rustagi [Rus89] presents a simple technique which makes use of the stencil buffer. The object is translated up, down, left and right for a given offset and rendered each time and the stencil buffer is incremented by one. The resulting stencil has a value of 4 where the object is originally located and the contour has a value of three or two. The contour can then be rendered using an appropriate stencil function. Another interesting method which utilizes the depth buffer is described by Rossignac [RvE92]. First, the polygons of the object are rendered to the depth buffer. Then the object is translated away from the camera by an offset and

rendered again in wireframe mode with thicker lines. Because of the offset the inner lines are hidden by the depth buffer and only the contour lines remain. Additional feature lines can be added by transforming the object closer to the camera.

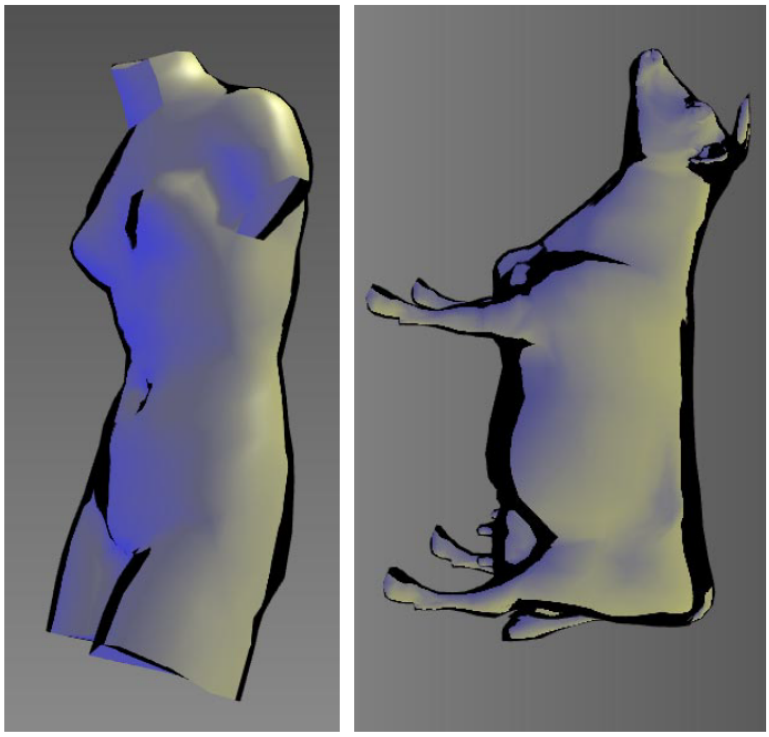


Figure 2.13: Silhouette rendering with the use of environment maps: Areas perpendicular to the view vector reflect a darker shade, thus it gives the illusion of a silhouette. Image taken from [GSG+99]

Gooch [GSG+99] also presents an alternative method for silhouette rendering by employing a special environment map. This method darkens the areas on the model where the normal vector is almost vertical to the view vector. These areas indicate the presence of a silhouette. The results of this technique look different and more artistic than other methods, as can be seen in Figure 2.13.

To sum up, hybrid silhouette algorithms deliver a rasterized pixel silhouette similar to image space algorithms, but allow more control over the visual result, e.g. line width and color. The computation time is also similar to pure image space methods, which means the algorithms usually produce interactive frame rates even on weaker hardware such as current mobile devices. Due to the nature of the output there is still no fast way to stylize the found silhouette.

Line Visibility Determination

Object space algorithms find silhouettes regardless of the visibility in the rendered image, which raises the need for visibility checking. For most cases it is desirable to remove the hidden silhouette lines, but in some cases it is also necessary to render hidden lines in a different style, e.g. with a dashed stroke to mimic a technical illustration. Almost similar to silhouette detection, visibility determination can be performed in image space, object space or in both as a hybrid solution.

A simple approach to visibility checking is to use the depth buffer. The mesh is rendered only to the buffer and then the silhouette lines are drawn. Invisible lines are automatically removed through the normal depth buffer function. The result can be enhanced by rendering the silhouette lines as polygons like billboards, which can be textured. Unfortunately the depth buffer will remove the parts of the silhouette lines which lie inside of the mesh so they will appear thinner than internal silhouettes. This method also makes it difficult to apply different styles to visible and hidden silhouette lines since it does not produce an analytical description for the line visibility. As a positive, the approach is easy to implement and very fast since it makes direct use of the graphics hardware.

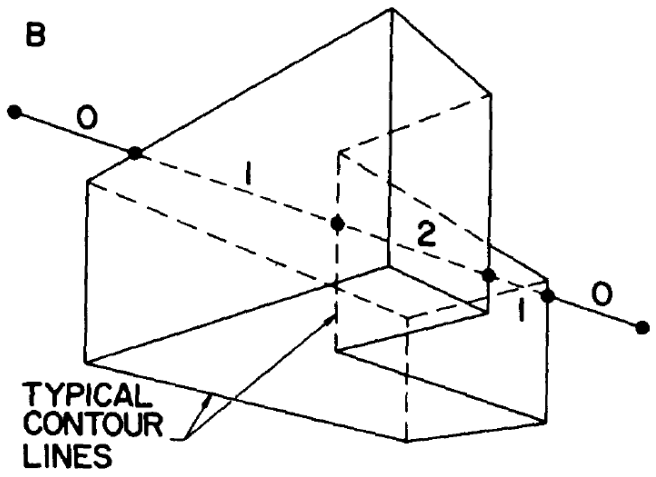


Figure 2.14: Quantitative invisibility (QI): Line visibility detection approach by Appel. Image taken from [App67]

Line visibility detection in object space is a more complicated task. Typically the silhouette edges are split at the point where the visibility changes. Appel [App67] determines the number of front facing polygons which lie in front of a silhouette line from the current

viewpoint. This is called quantitative invisibility (QI), see Figure 2.14. The method is often used, e.g. by Markosian [MKT+97] et al. or Hertzmann and Zorin [HZ00]. Basically the QI value at one vertex is determined by casting a ray from the viewpoint to the vertex and counting the intersected faces which are front facing. When following the connected silhouette edges, the QI value changes when an edge intersects another silhouette edge in 2D projection. After the calculation every silhouette segment has a QI value. Only edges with a value of zero (zero faces occlude the edge) are visible. Markosian et al. [MKT+97] optimize this approach by using a relative QI value. This reduces the amount of raytracing operations, since some silhouettes are already marked invisible because of their relative position. Hertzmann and Zorin [HZ00] also use a modified version of this method for their subpolygon method described earlier. They first divide the subpolygon silhouette edges at positions where the visibility changes. Then a simple silhouette using the existing edges of the mesh is calculated and the visibility is determined with Appel's approach. The visibility of the subpolygon segments can then be determined by looking at the visibility of the normal silhouette edges nearby.

The result of object space visibility determination algorithms is an analytic description of the silhouette with visibility information for each segment. With this information it is easily possible to stylize the silhouette lines e.g. with textured quads and use different styles for visible and invisible silhouette parts. Because of the involved complex operations object space approaches usually need more computation time than image space algorithms, but they produce silhouette data which is usable for later stylization.

Object space visibility determination has a high precision but is computationally expensive. For most applications only pixel accuracy is needed. With the combination of object space and image space methods, fast and sufficiently precise visibility tests are possible. Northrup and Markosian [NM00] use an ID buffer reference image [KMN+99] to determine silhouette visibility. They render the extracted silhouette edges and mesh triangles into the buffer where each element has a unique color. The buffer is then read back from the graphics card for further processing. Basically a silhouette edge is visible if its color shows up in the ID buffer.

Isenberg et al. [IHS02] use only the depth buffer for visibility checking. Similar to [KMN+99] the depth buffer is read back from the graphics card. The buffer image is then sampled along the silhouette edges. Due to the instability issues which arise if only one pixel is tested, they check the 8-neighborhood of the center pixel, see Figure 2.15. For performance reasons only every nth pixel is tested along the line. If one depth value in

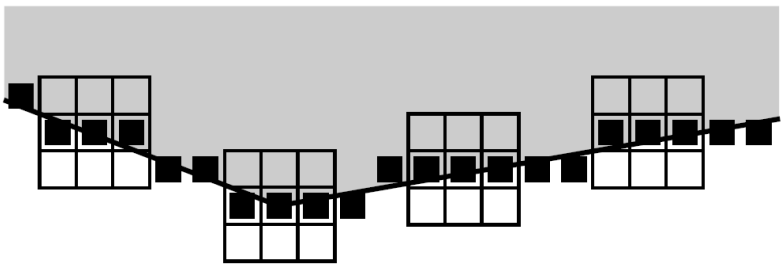


Figure 2.15: The depth buffer is sampled along the silhouette edge to determine the visibility. A 8-neighborhood is used to avoid instabilities. Image taken from [IHS02]

the 8-neighborhood is farther away from the viewpoint than the silhouette segment, the silhouette is visible in this point. The number of skipped pixels along a segment can be adjusted, the more pixels are skipped the lesser the accuracy but the speed increases. The result of these hybrid algorithms is again an analytic description of the silhouette with visibility information. The segments can be concatenated to strokes and then stylized with textures.

2.3 Line Stylization

Line stylization describes the process of applying a style to a line. It requires an input of connected edges, forming a stroke. The algorithms described in section 2.2.2 already provide line segments in object space, but they are not connected. Isenberg et al. [IHS02] describe the line concatenation as part of their work. They define a silhouette stroke as a series of connected silhouette segments. The process to get clean silhouette strokes can be divided into four steps: removal of redundant edge segments which could cause artefacts, visibility determination, concatenation of segments to form strokes and artefact removal after the projection back to 2D. Isenberg et al. use the winged edge data structure which provides connectivity information to concatenate silhouette segments to strokes. They begin at a random silhouette segment and search for connected edges in both directions. The beginning or end of a stroke is a segment which has only one connected segment. The use of the winged edge structure eliminates the additional processing needed to find the connectivity, since it provides this information as part of the data structure.

Once the silhouette segments are concatenated to strokes, the stylization is possible. The usual approach is to generate a triangle strip with continuous texture coordinates along the stroke, as can be seen in Figure 2.16.

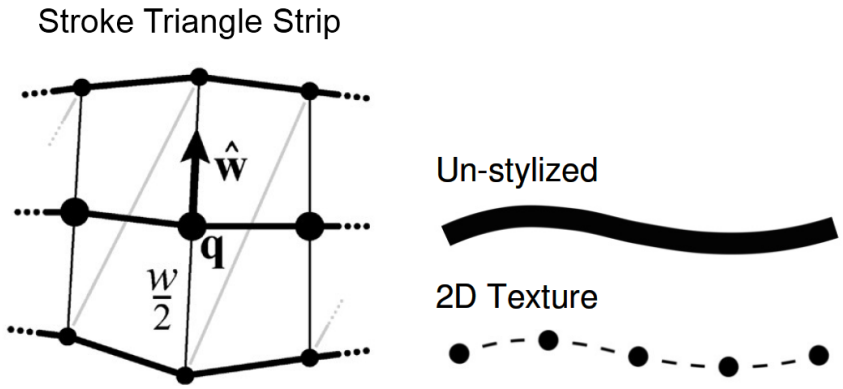


Figure 2.16: Strokes can be represented as textured triangle stripes or similar OpenGL primitives. Adapted from [Fin05]

Different line styles can be applied with textures and alpha maps. In his course notes, Finkelstein has collated a comprehensive intro to the stylization topic [Fin05]. The line style can take many forms, from pencil to watercolor or oil brush types, wiggled or dashed like in technical drawings. Figure 2.17 shows a few examples of the possible line styles.

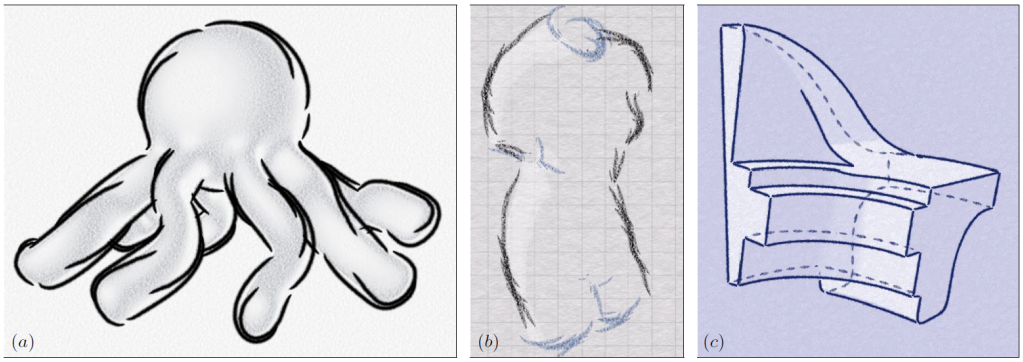


Figure 2.17: Three examples of the many possible line styles. Image adapted from [KDMF03]

Chapter 3

Concept

Contents

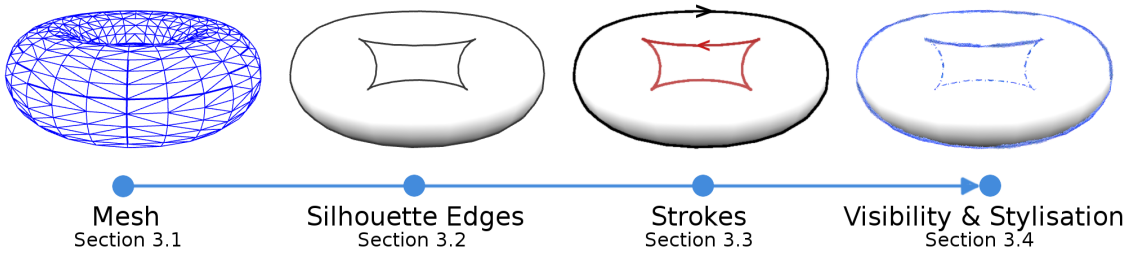


Figure 3.1: Pipeline for Rendering Stylized Strokes.

The goal of the implementation in this thesis is to generate stylized line drawings on a mobile device. The application should be able to process medium to large meshes at interactive frame rates. Given the limited capabilities of mobile devices compared to desktop computers, a number of performance improvements need to be considered. The general approach of generating a silhouette line drawing is outlined in Figure 3.1 and can be described as follows:

1. Mesh preparation: Load the mesh, generate a half-edge data structure with neighborhood information.
2. Silhouette edge detection: A silhouette detection algorithm identifies the silhouette edges of the current frame.
3. Stroke generation: All silhouette edges are concatenated to form continuous strokes.
4. Stylization & visibility determination: Different styles are applied to visible and hidden silhouette lines.

3.1 Mesh Representation

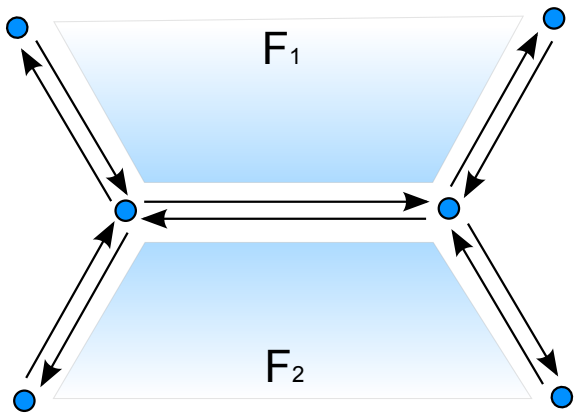


Figure 3.2: The Half-Edge Structure: An edge consists of two directed half-edges. Each half-edge has a vertex as origin and one adjacent face. Additionally the references of the incoming and outgoing half-edges are stored. Furthermore each half-edge knows its so-called twin, i.e. the opposite half-edge.

The representation of the mesh is important for the efficiency of object space silhouette detection. For the implementation of the silhouette renderer a half-edge data structure was chosen. The half-edge structure enables efficient navigation of the mesh because it provides various iterators for walking through the mesh. Figure 3.2 shows how the half-edge mesh is structured. The structure consists of vertices, faces and half-edges. In contrast to other mesh representations there are no full edges, but each edge in the mesh is represented by two directed half-edges. Every half-edge has a pointer to the previous half-edge, to the next half-edge and to its twin. Furthermore the starting point and the containing face is stored. This information enables useful mesh iterators. Figure 3.3 shows some examples of possible neighborhood operations which can be useful for optimized silhouette extraction.

- Face-Face Iterator: Iterate through all faces adjacent to the given face (Fig. 3.3a)
- Face-Halfedge Iterator: Iterate trough the half-edges which span the given face (Fig. 3.3b)
- Face-Vertex Iterator: Iterate through all vertices of the given face (Fig. 3.3c)
- Vertex-Vertex Iterator: Iterate trough all vertices which are directly connected to the given vertex (Fig. 3.3d)
- Vertex-Face Iterator: Iterate through all faces the given vertex belongs to (Fig. 3.3e)
- Vertex-Halfedge Iterator: Iterate through all half-edges which are connected to the given vertex

The connectivity information and the possible operations on a half-edge based mesh make it very easy to find e.g., the two faces which share one edge or find connected edges. This is used in the object space silhouette detection algorithm described in section 4.3.

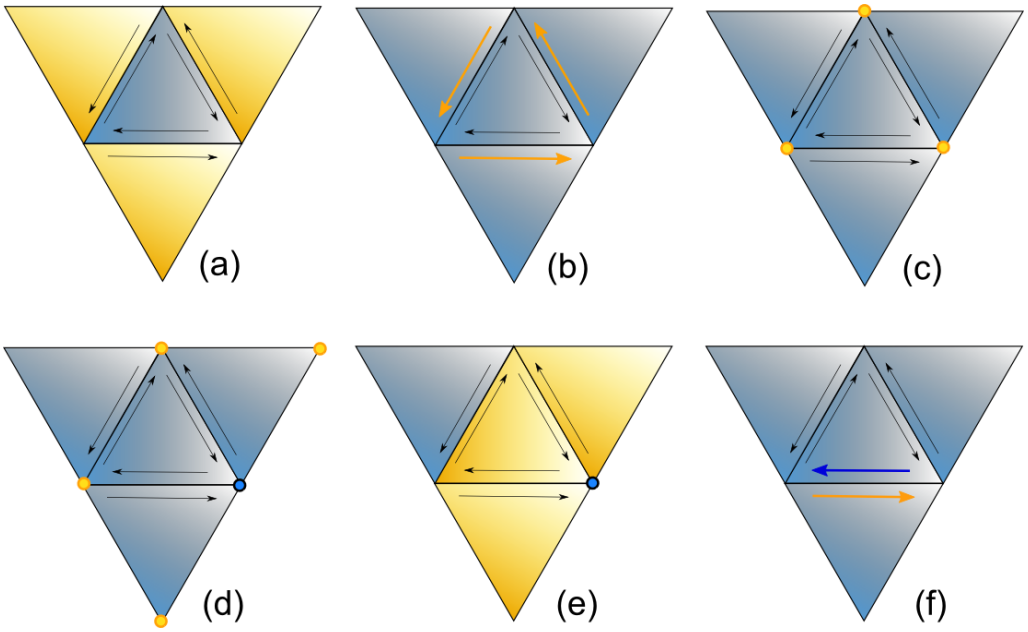


Figure 3.3: Some possible navigation operations on the half-edge data structure: face-face iterator (a), face-halfedge iterator (b), face-vertex iterator (c), vertex-vertex iterator (d), vertex-face iterator (e) and get twin half-edge (f)

3.2 Seed Edge Selection

The brute force approach for silhouette edge extraction described in section 2.2.2 is not appropriate for large amounts of polygons but it is guaranteed to find all silhouette edges on a mesh. The algorithm needs to iterate over all faces respectively edges of the mesh and test them. Since only a fraction of the faces in a mesh contribute to the silhouette it is beneficial to test a small subset of the faces. If a silhouette part is found, a neighborhood search is commenced. The following approaches describe different ways to get an initial set of seed edges.

3.2.1 Randomisation

A possible approach to speedup the silhouette detection process is to use randomized seed points and edges. Markosian et al. describe such an approach in [MKT+97]. Only a small portion of the edges in a mesh contribute to the silhouette, typically about $O(\sqrt{n})$ edges where n is the total amount of edges in the mesh [SGG+00b]. Markosian et al. randomly select a small fraction of the edges in the mesh and check if they belong to a silhouette. When a silhouette edge is found, they examine adjacent edges to find the complete silhouette. Large silhouettes, in terms of edge count, are more likely to be found. The probability that a silhouette is detected is proportional to its length. The algorithm does not guarantee that all silhouettes are found, but it is more likely that the longer silhouettes are detected. Under the premise to achieve interactive frame rates this drawback is acceptable, since the longer a silhouette is, the more important it is for the appearance of the rendering.

The process can be optimized further by creating a sorted candidate list of edges. Markosian et al. sort the edges according to their dihedral angle. This increases the probability that silhouette edges are found. An additional speedup can be gained by taking interframe coherence into account, as described in section 3.2.2.

According to Markosian et al., the performance of the silhouette detection on large meshes is five times higher than with the trivial silhouette detection.

The half-edge data structure is used to perform all necessary neighborhood searches. The approach implemented for this thesis generates a set of random numbers every frame which are used as indices for the seed faces. Different from Markosian et al. the faces are used because the silhouette edges are interpolated over the mesh using the sub-polygon method described in [HZ00]. The number of seed faces can be defined in percent of the total faces of the mesh. For every seed face the algorithm determines if it contains a

silhouette. If the face contains a silhouette it is likely that adjacent faces also contribute to a silhouette. The algorithm then tries to follow the silhouette in both directions until it reaches either the end of the silhouette or encounters an already visited face. Details of the implementation are described in section 4. Figure 3.4 shows the general process flow of the randomisation algorithm.

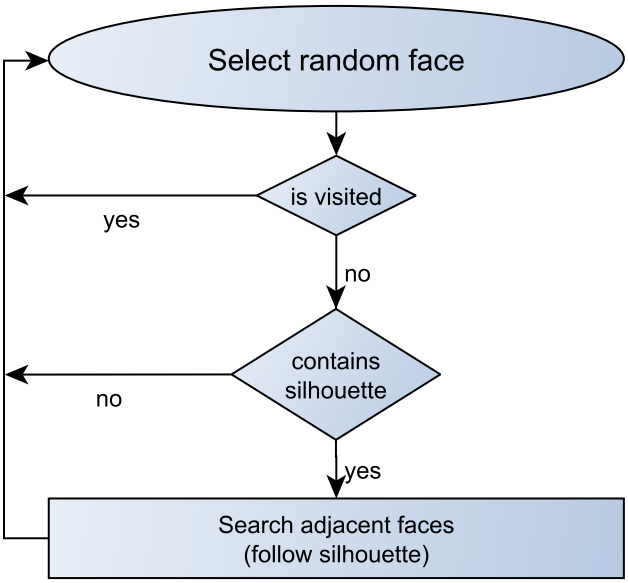


Figure 3.4: Finding seed faces with the randomisation approach: In a set of randomly selected faces each face is first tested if it has already been processed. If not, a preliminary fast silhouette test determines if the face is a silhouette candidate. If a silhouette face is found, the silhouette edge is computed. The algorithm now follows the silhouette in both directions using an adjacency search. When the search terminates, the next random face in the set is processed.

3.2.2 Interframe Coherence

Based on the assumption that a viewpoint only changes in small steps, which is equivalent to a smooth camera movement, one can also exploit spatial coherence. Hereby, the silhouette edges of the new frame are likely to be in the vicinity of the previous silhouette edges. Markosian et al. test all silhouette edges of the previous frame, to check if they are silhouette edges again. Furthermore, they pick a small amount from the silhouette edges of the previous frame and begin a limited neighborhood search to further improve the detection rate. The adjacent edges are traced either away or towards the viewpoint.

The termination criterion is either a silhouette edge is detected or the maximum number of adjacent edges is reached. The threshold can be a predefined value.

In this thesis the spacial coherence approach is also used with slight modifications. Figure 3.6 shows the approach used in this thesis. In the initialisation phase the simple brute-force algorithm is used to mark the initial set of silhouette faces. The brute-force method is used because it is guaranteed to find all silhouette faces. The next phase exploits the fact that the silhouette of the next frame is likely in the vicinity of the previous silhouette. Figure 3.5 demonstrates this connection.

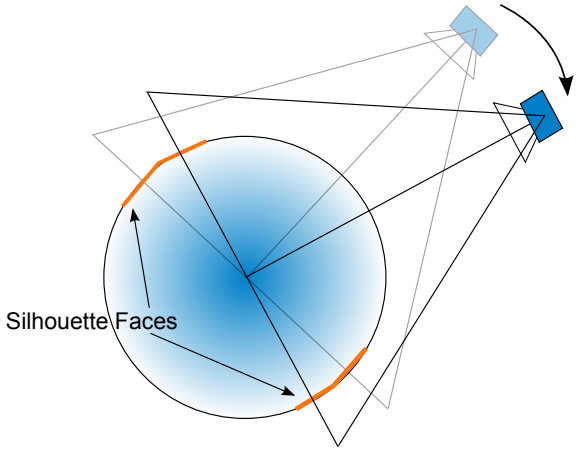
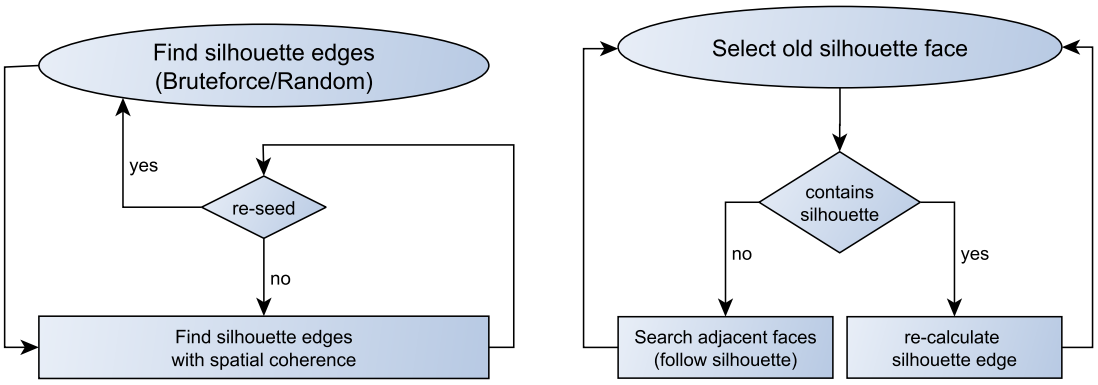


Figure 3.5: Interframe coherence: If a smooth camera movement is assumed and the new camera position is in the vicinity of the old position, it is likely that the new silhouette is not far from the silhouette of the previous frame.

The assumption is only true if there is a small camera movement between frames. In an AR application there is usually a smooth transition of the camera, since it is usually moved around by a human operator. However, if the frame rate is too low and to many frames are skipped the silhouette could be outside the search boundary of the previous frame. There are two simple straightforward solutions to remedy this problem. The first approach is described by Markosian et al. in [MKT+97]. They select a subset of the previously marked silhouette edges and perform an additional extended neighborhood search around these edges. In the case of this thesis, which operates primarily on faces, the neighborhood of a subset of the faces containing a silhouette of the previous frame is included in a extended search. The second, simpler approach is to perform a full silhouette search every n passed frames. This can be done by using a brute force approach since it will detect all silhouette edges in the current view. The detected silhouette will then serve as seed for the next iterations of the interframe coherence search. Figure 3.6(a) shows the connections for this

approach. The trigger to re-seed the silhouette faces can be the number of passed frames as described before, but it is also possible to make the decision dependent on the number of detected silhouette faces. For instance, if the number of detected silhouette edges significantly drops compared to the previous frame, it would indicate that the camera has moved to much and a re-seeding step is necessary. The performance can be increased by using the randomisation approach for silhouette detection instead of the brute-force approach, however it is not guaranteed that it will detect all silhouettes. For this work two versions of the coherence algorithm were implemented, which differ slightly in their behavior. The first algorithm (referred to as Coherence A in the evaluation section) only uses the silhouette faces of the previous frame and runs a limited neighborhood search. This is a very fast approach, but it is not robust since it can loose track of the silhouette easily. The second variant of the coherence approach (Coherence B in the evaluation section) selects seed faces based on their history. Every time a face is recognized as a silhouette face, a counter is increased. The counter is decreased when the face is no longer a silhouette face. This makes the algorithm more stable as it does not loose track of silhouette faces quickly. Additionally instead of the limited neighborhood search, the algorithm will try to follow a silhouette once a silhouette face is detected.



(a) Interframe coherence overview

(b) Processing step

Figure 3.6: Interframe coherence approach: The overall process is shown in (a). A re-seeding step can be triggered based on the elapsed frames or if the detection rate drops below a certain threshold. The coherence search (b) iterates over the silhouette faces of the previous frame and determines whether it contains a silhouette or not. If a silhouette is found the algorithm proceeds to the next frame, else the neighborhood of the face is searched for silhouette faces.

Camera Tracking

The performance of the silhouette detection can be further improved when not only the position of the previous silhouette edges is taken into account, but also the camera movement. If the camera movement is known prior to the silhouette detection step, the set of edges for the search can be reduced even more. The typical AR application uses a tracking system to position virtual elements in a real world image. The tracking system delivers the position of the camera relative to a marker, hence it is possible to determine the motion direction of the camera. Knowing the motion of the camera, the direction in which the silhouette "'moves"' can be calculated. Instead of performing a complete neighborhood search, performance could be increased by limiting the search to the direction of movement. In Figure 3.5 the search direction is towards the faces in the movement direction of the silhouette. Faces which lie on the other side can be omitted in the silhouette detection, which would provide speedup. The camera position is available through the tracking system and has no further impact on the performance since it is always necessary to reference the object in the real world image.

3.2.3 Image based Seed Edges

A different, image space based approach, is seed edge selection through processing the current camera image with an edge detection operator. For example, the canny edge detector can be applied to the camera image to detect edges. Figure 3.7 shows an image filtered with the canny edge detector.

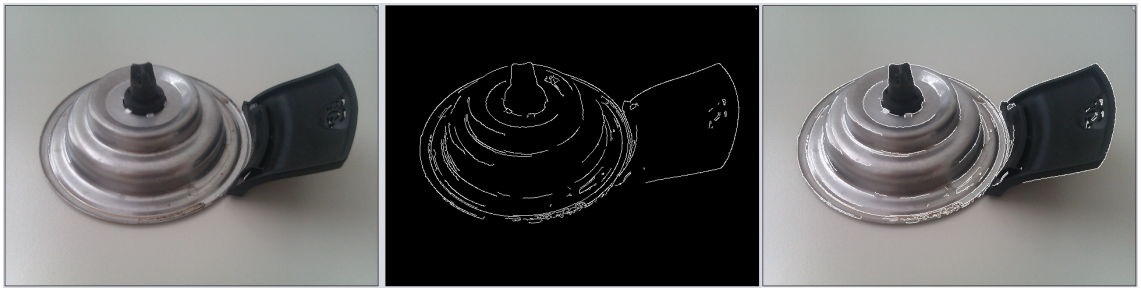


Figure 3.7: Canny edge detector: A multi-stage edge detector which is able to detect various edges in images. The first image shows the unmodified color image, the middle image shows the edges detected by the canny edge detector. The last image shows an overlay of the original and the detected edges. It can be seen that the edges are likely to contribute to a silhouette.

The edges detected in image space can be used as seed hints for the silhouette detection

process. There are two different thinkable approaches to this technique. One way is to detect the edges in the camera image. Since the edge detection works in image space, the detector will certainly detect edges in the background, at shadow borders and texture features on the object. A simple way to remove these unwanted edges is to mask out the background for example with the depth buffer of the rendering pass. Now all remaining pixels in the image can be used as seed points. Another way to use image space seed points is to use the output of a normal render pass as input image. This way the image is always perfectly referenced to the mesh and it contains no background. Since the image can be rendered in a definable style, the edge detector can easily be parametrized to extract a optimal amount of distinctive edges. The edge detector can be implemented on the GPU using a shader program for maximum performance. An example shader program can be found in the book "'GPU Gems 2"' [PF05].

3.2.4 Precomputed Silhouettes

Promising seed faces can be identified through a precomputation step. The silhouette is calculated for a number of predefined camera positions and the contributing faces are stored in a map. When the camera is moved interactively, the closest precomputed viewpoint is determined and the according set of faces is used as a seed for the silhouette detection. The viewpoints for the precomputation step are calculated as multiples of the two angles Θ and Φ of the spherical coordinate system shown in Figure 3.8. Equation 3.3 shows how the camera position is calculated.

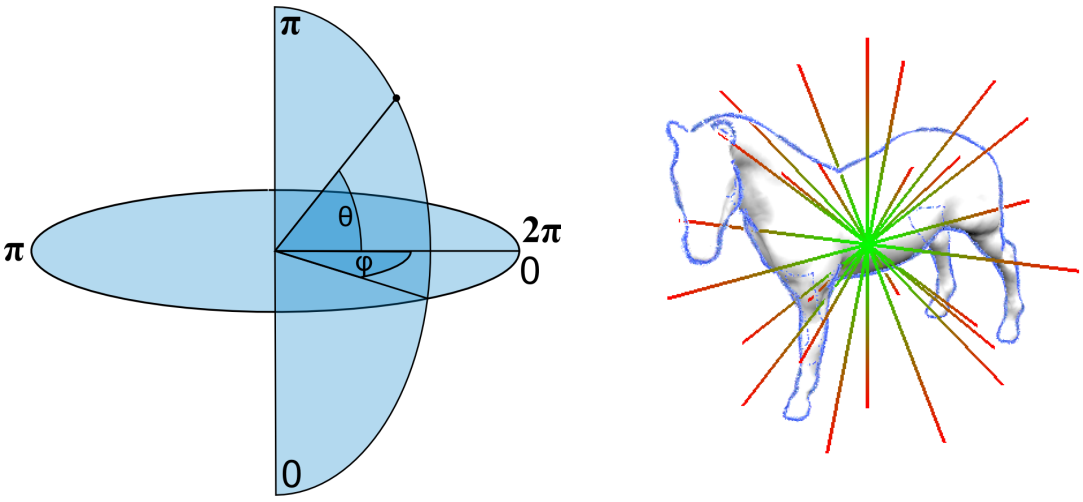
$$Camera_x = r \cdot \sin\Phi \cdot \cos\Theta \tag{3.1}$$

$$Camera_z = r \cdot \sin\Phi \cdot \sin\Theta \tag{3.2}$$

$$Camera_y = r \cdot \sin\Phi \tag{3.3}$$

During the precomputation process the angles Φ and Θ are incremented with a predefined step size $\Delta_\Phi = \frac{\pi}{steps_\Phi}$ and $\Delta_\Theta = \frac{2\pi}{steps_\Theta}$. With $steps_\Phi$ and $steps_\Theta$ the number of viewpoints is defined as $steps_\Phi \cdot steps_\Theta$.

During interaction, the current position of the camera is used to determine the closest precomputed viewpoint. The faces contributing to the precomputed silhouette are then used as seed faces to start a neighborhood search. Once a silhouette face is found, the algorithm traces the silhouette. All processed faces are marked as visited, so that the



(a) Spherical coordinate system

(b) Example of predefined camera viewpoints

Figure 3.8: Definition of a point in space with spherical coordinates Θ, Φ ans the radius (a). The unit sphere is divided into an array of viewpoints using increments of both Θ and Φ.

algorithm can ignore these faces in the next step to minimize redundant checks.

3.3 Stroke Generation

Stroke generation is a step required for the stylization of the silhouette. In this thesis a stroke is defined as a concatenation of silhouette edges. The silhouette of an object can be composed of multiple silhouette strokes. A stroke can have a start and an end or it can be a loop. A stroke also has no discontinuities, it is always a continuous sequence of edges. Figure 3.9 displays the relationships between silhouette edges and strokes. The silhouette definition used here includes inner feature lines, additional to the objects contour. For clarity, the contour defines the objects shadow, whereas the silhouette also contains inner lines which follow from the definition of the silhouette (transition between front and back facing polygons).

It is important for the stylization process that the silhouette edges are in the correct order within a stroke. The algorithms for silhouette extraction described earlier deliver a list of edges which contribute to the silhouette of a given mesh. However, the sequence of the edges is usually unknown. The approach described here generates ordered silhouette strokes in three steps:

1. Find silhouette edges using one of the algorithms described in section 2.2.2.

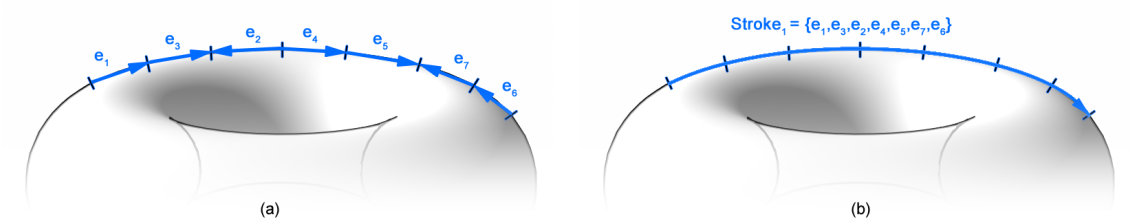


Figure 3.9: From unconnected edges to continuous strokes: A stroke is an ordered list of connected silhouette edges.

2. Concatenate all silhouette edges to continuous strokes.

3. Sort the edges in a way that all edges within a stroke face the same direction.

In this thesis a few variants (see section 3.2) of the sub-polygon method are used to determine the silhouette edges. In the initial prototype phase the brute force variant was used, because it is fast to implement and guaranteed to detect all silhouette edges. To concatenate the edges correctly, some kind of neighborhood information needs to be available. The half-edge data structure uses throughout this thesis aids in the process of detecting silhouette edges and is also very helpful for the concatenation process if the silhouette edges are actual edges on the mesh. The usage of the sub-polygon method however introduces a new problem: The detected edges are newly created entities which do not belong to the actual mesh. The edges can be seen as a 3D overlay on the mesh. The problem is now that the edges are not stored in the half-edge structure and are therefore missing the neighborhood information. To remedy this problem, the created silhouette edges are stored in a special data structure which allows the inclusion of additional neighborhood information. The information in this structure is like an anchor which ties the silhouette edges to the mesh. Figure 3.10 shows a graphical representation of the silhouette edge data structure and how it relates to the underlying mesh. The structure to hold the silhouette edges contains the following information:

- Start and end vertex. The vertices are located on two different edges of a mesh face. Their position is determined by the sub-polygon interpolation method.

- Edge length. The length of the edge is stored here because it is required for the stylization process.

- Start and end half-edge. The reference to the half-edges of the face where the silhouette edge lies on. This information references the edge to the mesh.

- Edge ID. This is a unique number identifying the edge.

The silhouette detection process generates silhouette edges in the form of this data structure and stores them in an appropriate collection. Additionally, the process also stores some information in the mesh which is required for the concatenation. If a silhouette edge is found on a face, a reference to the generated silhouette edge is stored in each of the two half-edges which contribute to the edge, i.e. the half-edges where the newly created vertices lie on.

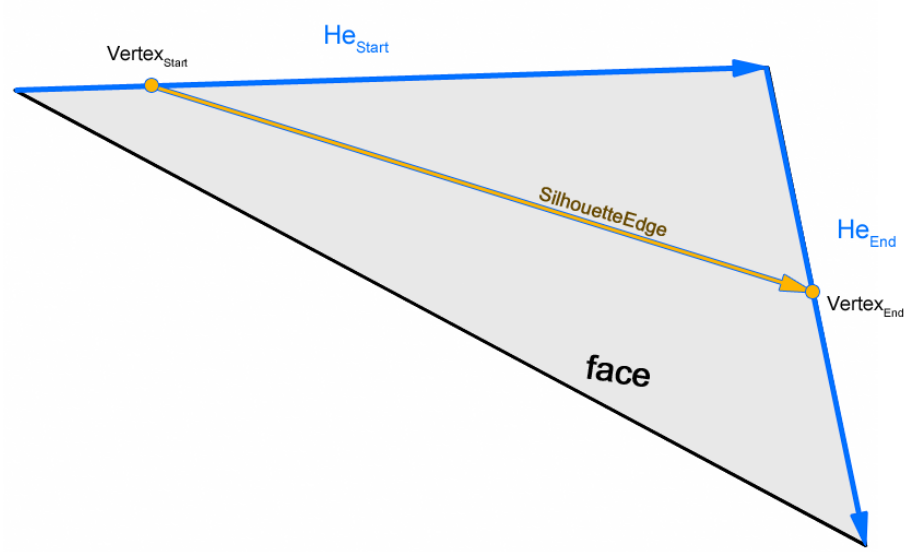


Figure 3.10: Silhouette edge data structure: A *SilEdge* stores a start and end vertex, a reference to the start and end half-edge, an Id and its length.

In the next step the edges are concatenated to form strokes. A stroke can either be open or a closed, forming a loop. The concatenation process loops over all detected edges. For every edge, it checks if it has already been added, and if not a new Stroke is created and the edge is added to the stroke. A stroke is represented with an appropriate data structure which holds an ordered list of silhouette edges. For each added edge, the algorithm looks at the attached half-edges and checks if they have a silhouette edge reference. This is a recursive function which terminates if there are no more edges with neighbors or if the neighboring edge has already been visited. An already visited edge usually indicates that the stroke is a loop. Figure 3.11 shows the stroke forming process. The algorithm terminates when there are no more edges to process. The result is a list of strokes, where each stroke contains a continuous, ordered list of silhouette edges. The final step in the stroke generation process is to make sure that all edges within a stroke face in the same

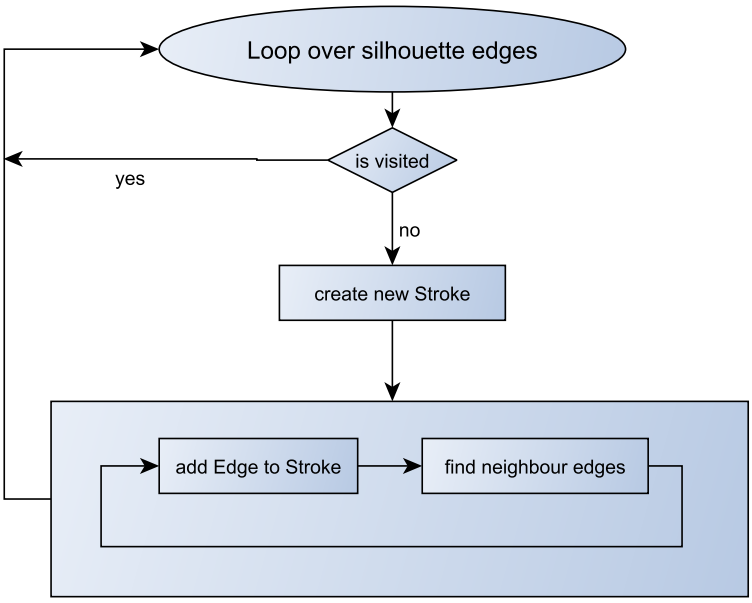


Figure 3.11: Stroke generation process: The stroke building algorithm starts at an arbitrary silhouette edge and uses the neighborhood information of the edge data structure to find adjacent silhouette edges. These edges form a continuous stroke. One stroke is completed if there are no more adjacent edges, or a loop is detected.

direction. Figure 3.12 shows the problem. The edges are all in the correct order, but it is possible that some of them face in opposite directions. This poses a problem for the stylization process, where continuous texture coordinates are assigned. The final step in

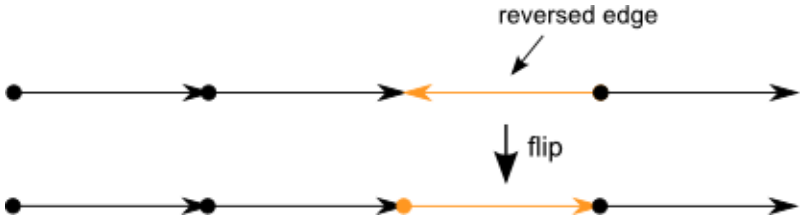


Figure 3.12: Edge sorting: Edges facing in the wrong direction are detected and flipped.

the stroke generation process is the correction of the edge directions. Figure 3.12 shows the edges before and after the correction. The algorithm loops over all edges of each stroke and checks for the case that the ID of the next edge is identical to the ID of the current edge. If such an edge is detected, the references to the previous respectively next edge are exchanged.

The final result of the stroke generation process is a set of strokes, where each stroke

consists of an ordered list of silhouette edges which face in the same direction. The strokes themselves are stored in an unordered collection for later processing in the stylization step.

3.4 Stroke Stylization & Visibility

The stylization of silhouette strokes is an integral part of the line rendering process. Stylization describes the process of applying different visual attributes to a line. A normal line in 3D space is simply the connection between two vertices. There is only minimal control over its graphical appearance, for example its color. Stylized lines are desirable because their appearance can be altered in many ways to fit to any scenario. Controllable attributes are for example:

- Line thickness
- Opacity
- Color
- Pattern
- Texture

In addition, it should be possible to make the stylization attributes dependent on parameters like the visibility of the stroke or the distance to the camera. This should be possible at pixel accuracy along the stroke.

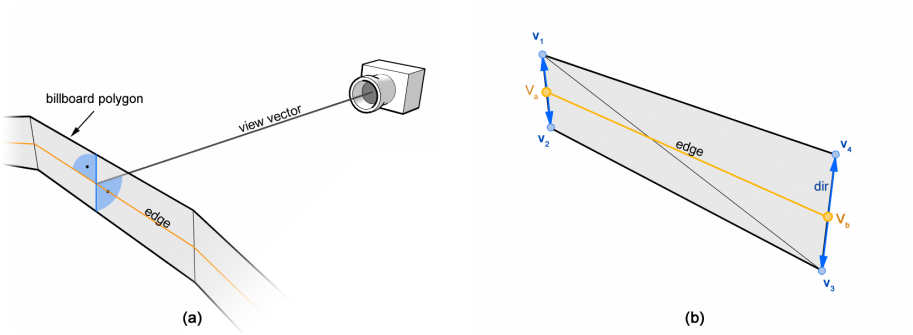


Figure 3.13: Billboard polygon calculation: The *dir* vector is perpendicular to the view vector and the edge (a). The *dir* vector is used to calculate the positions of the vertices v_1 to v_4, which form the billboard polygon.

To make stylization possible in the way described above, some preparations of the silhouette edge data are required. The prerequisites are that the silhouette edges are

already combined into continuous, correctly aligned strokes. The steps necessary to achieve this are described in the previous section 3.3. Furthermore, a standard line in OpenGL does not provide any advanced options to control its appearance. To be able to use stylization, the silhouette lines are represented as a polygon chain. The technique used in this thesis is billboarding, Figure 3.13a shows how it is done. A flat polygon, in this case a quad is placed at the exact position of a silhouette edge using the start and end vertex position of the edge. This polygon is then rotated so that it always faces the camera. Figure 3.13b shows how the vertices of the quad are calculated. The cross product between the normalized camera vector and the normalized edge vector yields the direction vector. The four vertices of the billboard quad are determined by vector addition respectively subtraction to the vertices of the edge. The following formula is used to calculate the direction vector:

$$\overline{dir} = \left|\left|\overline{cam}\right| \times \left|\overline{edge}\right|\right| \cdot stroke\_size \tag{3.4}$$

The camera vector $\overline{cam}$ is the vector from a vertex of the edge and the edge vector $\overline{edge}$ is formed by the vertices of the edge. The result of the cross product is also normalized and then scaled with $stroke\_size$, which defines the width of the quad and thereby the stroke thickness. To apply a texture to the quad, appropriate texture coordinates need to be generated. There are some problems to consider when generating texture coordinates on a stroke. One thing to keep in mind is that there are usually multiple silhouette strokes, which also differ in their length. Also the length of the strokes depends on the mesh. The stroke texture should maintain its scale for every stroke, and possibly also between frames. Figure 3.14 shows an example of how the stroke texture should look like. Another thing to consider is that the silhouette edges within a stroke are not uniformly sized. The texture coordinate calculation needs to take these facts into account so that the final result is a consistently scaled textured stroke.

There are several methods to achieve a uniform scale over all strokes in the image. In this thesis the longest stroke of the frame is determined and used as a base scale. A user-definable value is used to calculate a scale term which is used for the calculation of the texture coordinates, which is a fraction of the length of the longest stroke. Equations 3.6 to 3.7 describe the calculation, where tx_size is a user-definable value which controls the width of the polygon quad, L is the length of the longest stroke and tx_{u_s}, tx_{u_e} are the start and end texture coordinates in the u direction.

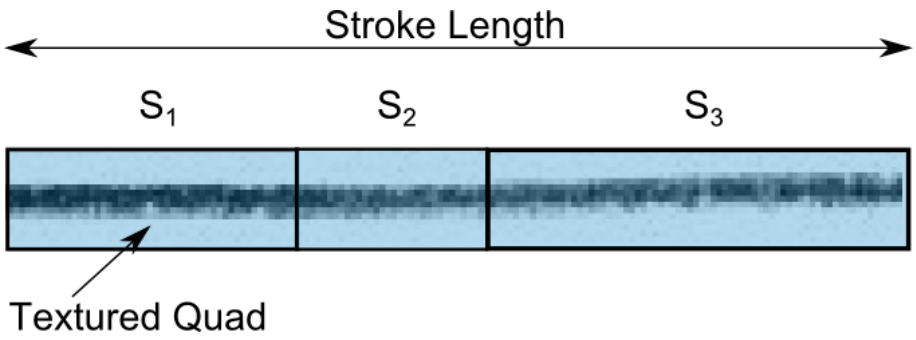


Figure 3.14: Quads with applied texture: The texture is evenly scaled along the differently sized silhouette segments S_1, S_2 and S_3. The scale is determined by the overall length of the stroke, or a fraction of the stroke length.

$$scale = \frac{tx\_size}{L} \tag{3.5}$$

$$tx_{u_s} = tx_{u_{e-1}} \tag{3.6}$$

$$tx_{u_e} = tx_{u_s} + l * scale \tag{3.7}$$

The automatic texture coordinate generation is the foundation of the stylization step. Important properties like opacity, color or shape of the line can be defines using textures with alpha channel. Figure 3.15 shows an example of a texture set.

Figure 3.15: Example texture set: This is a hand-drawn texture set of three different line styles. In this example a blue felt-tip pen was used to draw the lines on normal paper. The drawing was then digitized and the background removed.

The texture set in Figure 3.15 contains three different hand-drawn line styles. The process for creating such texture sets is described in detail in section 4. The textures used for stylization can be of any origin, be it hand-drawn, extracted from photographs or computer generated. It is also possible to use wavy or scribbled lines. The opacity is controlled by the alpha channel of the texture image.

A very important problem for silhouette rendering is visibility determination. One goal of this thesis is to achieve a fast and accurate visibility check which also works on mobile platforms. The challenge on mobile platforms is to deal with the limited capabilities concerning shader units and the overall lower performance than desktop units. This thesis aims to bring a fast visibility determination algorithm to mobile devices without sacrificing the quality of the rendition. It is important to note that the visibility determination is not simply a hidden line removal problem, which could be solved with OpenGL buffers. For stylization, the hidden lines should be marked as hidden rather than being removed. This way, hidden lines can be rendered in a different style than visible lines, for example one could use a dashed, semitransparent line style for hidden lines and a thicker, continuous style for visible lines. Figure 3.16 shows an enlarged part of a silhouette rendering with different line styles.

Figure 3.16: Line stylization: Visible silhouette lines are rendered as continuous colored pencil streaks and hidden lines are represented as dashed lines.

There are three main methods to control the visibility along the silhouette. Figure 3.17 shows the different approaches.The simple approach is to change the style per stroke segment, so that the precision is dependent on the geometric resolution of the stroke, as shown in Figure 3.17a. This is a very fast solution but in most cases the accuracy is too low and there would be clearly visible artefacts. The other two possibilities achieve acceptable accuracy for stylization changes: One way is to calculate the position of a style change, e.g. a visibility change, in object space and insert new vertices into the stroke at these points (Figure 3.17b). The second approach, which is implemented in this thesis, is an image space approach. Here the style is applied with a shader program, no new

geometry is required (Figure 3.17c).

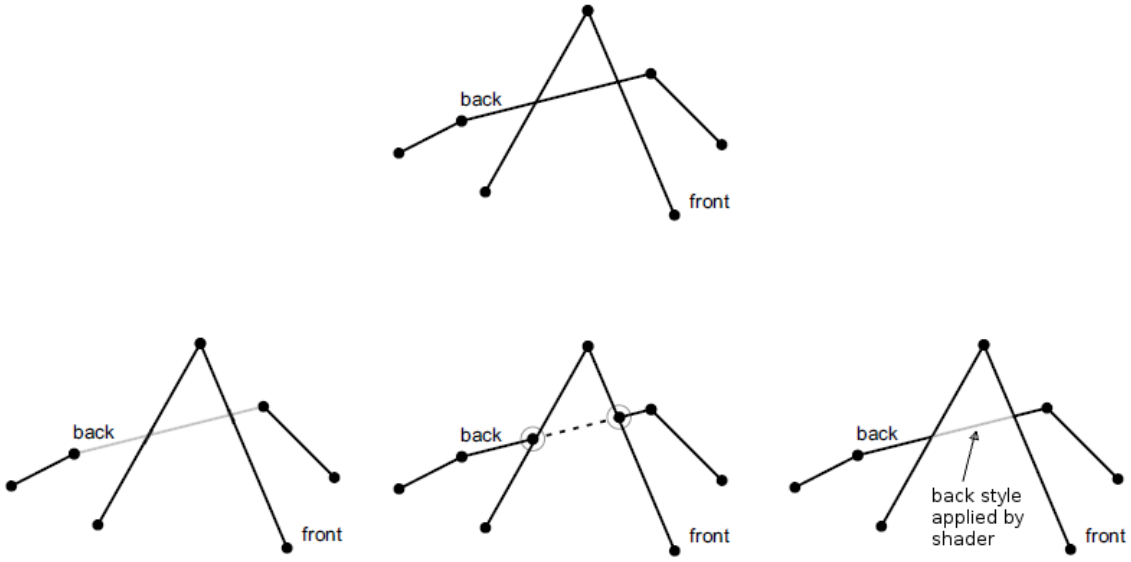


Figure 3.17: Different approaches for visibility changes. The entire segment is marked as hidden (a). New vertices are inserted at points where the visibility of the silhouette edge changes (b). The visibility is determined in image space and the hidden segment is stylized using a shader program (c). Images adopted from [IB06]

In this thesis the approach shown in Figure 3.17c is implemented. It has adequate accuracy and is also fast because it can be done using a shader program. The geometric approach (Figure 3.17b) is not feasible for the use on mobile devices, because it requires either some sort of raytracing, which is computationally expensive, or access to the depth buffer of the graphics hardware. Reading back data from the graphics card is usually a slow process. The approach of this thesis leaves the computation of the visibility entirely on the graphics hardware. The following steps are required to compute the line visibility:

1. Generate lookup vectors
2. Render the mesh to a texture with a depth shader program enabled
3. Render the silhouette quads with the stylization shader program enabled

The algorithm makes use of the depth buffer, similar to [IHS02]. However, the existing silhouette lines are not modified, but a shader program is used to render different line styles depending on the visibility of the line. This approach allows pixel-accurate visibility determination. In the first render pass the mesh is rendered with a shader program, which emulates the depth buffer, to a framebuffer object so that the silhouette shader can access

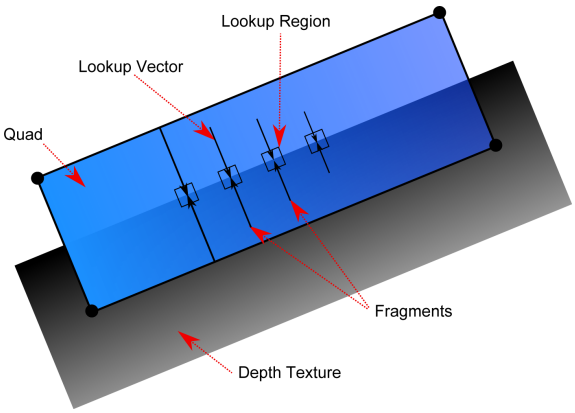


Figure 3.18: Visibility check: Lookup vectors are encoded into the vertex colors. The fragment shader evaluates a 8-neighborhood of the target location to determine visibility.

the rendered depth buffer as a texture. This first render pass is necessary because the real depth buffer is not accessible on current mobile devices. The shader program used to generate the depth texture can be found in the source code section (A).

In the second pass the silhouette quads are rendered to the normal framebuffer with the activated silhouette shader. The silhouette shader uses information coded into the fragment color to determine a lookup position in the depth texture. Figure 3.18 shows how the silhouette shader works. To minimize artefacts which can occur because of numerical imprecision, a 8-neighborhood of each tested pixel is used to determine the visibility.

The lookup position is calculated from the fragment's texture coordinates and the interpolated lookup vector. From this position the 8-neighborhood of depth values are read from the depth texture. If one of the depth map values is less than the current fragment depth, then the fragment is regarded visible and the according stroke style is assigned. Figure 3.18 depicts this operation. In short, the values in the depth texture at the position of the silhouette line determine the visibility of the quad pixel row perpendicular to the line.

Chapter 4

Implementation

Contents

The silhouette rendering system developed as part of this thesis is written in the Java programming language. The system was first implemented as a prototype on the PC platform, to evaluate overall performance and for the ease of debugging. Thereafter, the application for the Android operating system was adapted from the first prototype. Details of the differences between the PC-based and the mobile application are described in section 4.7. The mesh representation is a half-edge structure loosely based on a project by Claude Fuhrer and Lukas Feuz (Bern University of Applied Sciences). The visualization is done in OpenGL using parts of the Lightweight Java Game Library (LWJGL).

The meshes can be obtained by reconstructing real world objects. Some meshes used throughout this thesis were created using a 3D scanner, other objects were reconstructed using multiple photographs with Autodesk 123D Catch.

4.1 Texture Generation

This is a quick overview of how to generate a new brush style for the framework. The tools used are for example a piece of paper, a pencil, a scanner and an image editing software like Gimp. The hand-drawn approach is just one example, of course the creation can be done directly in an image editor. Figure 4.1 shows the steps for creating a new style.

1. Draw some lines in the desired style. This can be done on paper or directly in an image editor (for example Gimp).
2. Remove the background so that only the stroke remains.
3. Separate the strokes into individual images and store them as a PNG file with alpha channel. These files can now be used as brush styles.
4. Create a stroke pack with the following line of code: `stroke_packs.add(new StrokePack("visible.png", "hidden.png", stroke_width, tex_size));`.

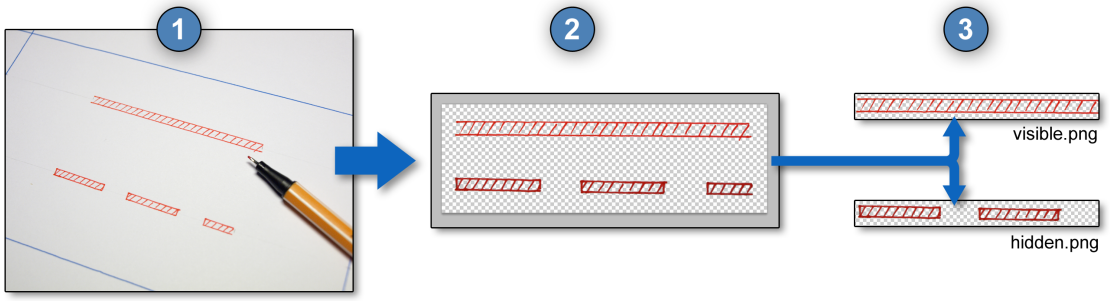


Figure 4.1: Creating a new style: Draw a line style (1) on paper or in an image editing software. Make the background transparent (2). Save each line style in a texture with alpha channel (3). The style can now be used in the framework.

4.2 User Interface

The user interface for the mobile application consists of a main menu bar at the bottom of the screen. Figure 4.2 shows a mockup of the interface. The menu is activated using the menu button on the mobile device, which is a default behavior on Android devices.

The menu is structured as following:

- Render Options

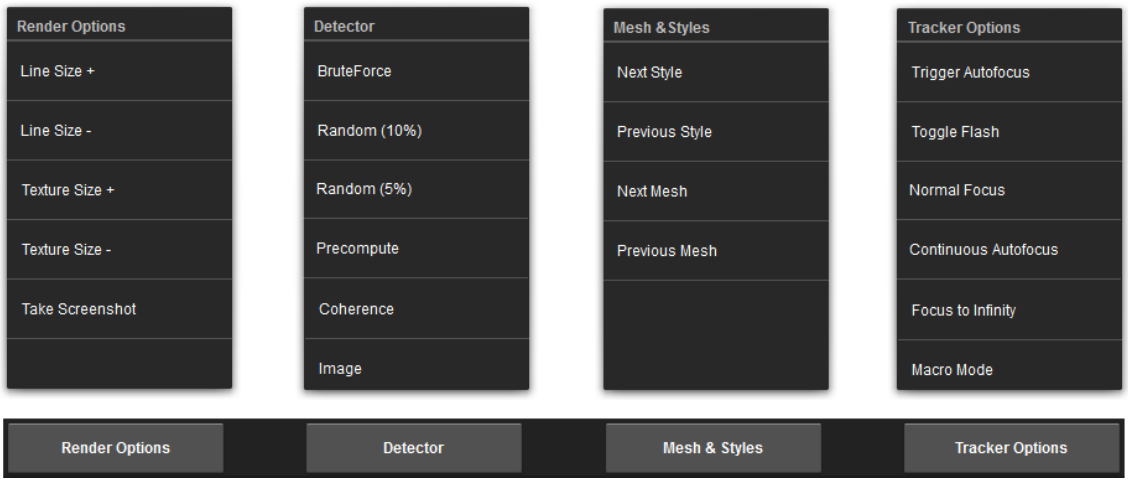


Figure 4.2: Structure of the user interface on the mobile device

  - Line Size +/-: Changes the thickness of the stroke
  - Texture Size +/-: Changes the texture stretching along the stroke
  - Take Screenshot: Captures the current frame to an image on the SD card

- Detector
  - BruteForce etc.: Change the seeding algorithm

- Mesh & Styles
  - Previous/Next Style: Switches through predefined style sets
  - Previous/Next Mesh: Switches between predefined meshes

- Tracker Options
  - Trigger Autofocus: The camera autofocus is triggered once
  - Toggle Flash: Activates the integrated front LED of the device if present
  - Normal Focus: Focus stays unchanged
  - Continuous Autofocus: The camera always focuses if the image is out of focus
  - Focus to Infinity
  - Macro Mode

4.3 Silhouette Detection

The silhouette detection step is carried out in object space. The PC-based prototype is capable of using the simple brute-force algorithm for testing and evaluation purposes and also the sub-polygon approach described in [HZ00]. The half-edge structure simplifies the process through its neighborhood queries. The brute force algorithm iterates over all edges, or more specifically the half-edges of the mesh and finds the two adjacent faces. This is done by using the `getTwin()` method of the half-edge class. Once the faces have been found, the dot product of each of the face normals with the view vector is calculated. If the products differ in their signs, the edge can be rendered as a silhouette line. Listing A.1 shows a short excerpt from the brute force algorithm. Note that here the silhouette is directly passed to OpenGL. Other possible solutions include marking the half-edges as silhouettes or create a new list of edges which can be postprocessed in another step.

The output of this algorithm often produces artefacts and jagged silhouette lines as described in chapter 2. Figure 4.3b shows how the silhouette looks from an angle different to the viewpoint used for detection. Since only existing edges are used, the quality depends on the mesh resolution. Other artefacts which might occur are described in [IHS02]. To avoid these artefacts, the sub-polygon method by Hertzmann et al. [HZ00] was implemented. The sub polygon method uses the vertex normals instead of face normals to

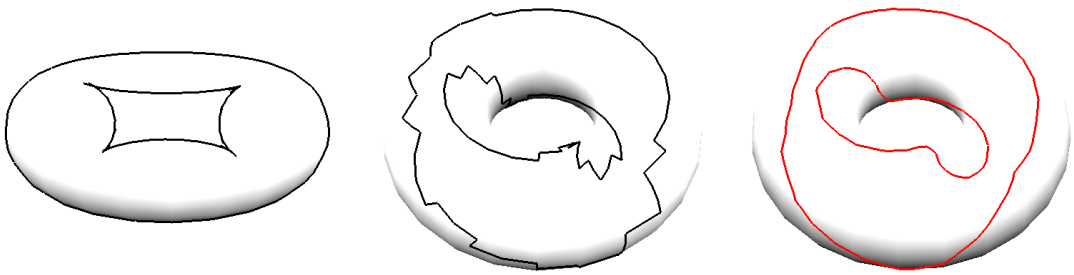


Figure 4.3: Comparison between the simple and the sub-polygon silhouette detection algorithm. The quality of the silhouette depends on the mesh resolution.

emulate a smooth surface. For every face in the mesh the dot products of each vertex normal with the view vector are calculated. Then the products are compared regarding their sign. If the dot products of two vertices of an edge have a different sign, then there is a zero crossing between them. This point denotes the position of the silhouette edge. The comparison is done for all three edges of the face. If there are dot products with different signs, then the algorithm finds both zero crossings. Figure 4.4 shows how the sub-polygon

algorithm works.

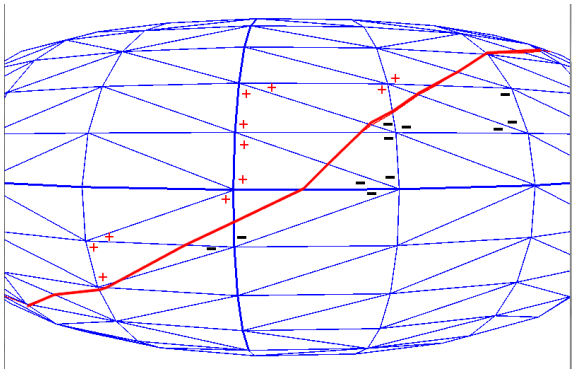


Figure 4.4: Sub-Polygon Silhouette: Positive dot products are shown as + signs, negative products as - signs. The zero crossings mark start and end points of the silhouette edges.

The implementation of the sub-polygon approach yields much smoother silhouettes, as can be seen in Figure 4.3c. Listing A.2 shows an abbreviated version of the function which extracts the sub-polygon silhouette. Note that this implementation is also a brute force variant which checks all faces for every frame. Again the silhouette lines are rendered directly.

The main loop iterates over all faces of the mesh. For each face, a `FaceVertexIter(face)` iterates over all vertices of the face. For each vertex, the dot product of the vertex normal and the view vector is calculated and stored as an attribute of the vertex. This improves performance, since almost all vertices are shared between faces, so there is no need to recalculate the dot products for every face. To avoid unnecessary operations the calculation for the face can be stopped if all dot products have the same sign, which means the face is entirely front or back facing. If one dot product differs in its sign, there is a silhouette edge on the face. The start and end points of the silhouette edge are computed by interpolating the position of the zero crossing along the edges of the face [HZ00]. Once the start and end points have been found, the resulting silhouette edge can be either rendered directly or stored in an appropriate data structure.

4.4 Stroke Concatenation and Sorting

The silhouette detection step described in section 4.3 yields all silhouette edges for the current frame, unfortunately the edges are not yet connected. In the following step the edges are concatenated to form continuous strokes which can be stylized. Again, the connectivity information of the half-edge data structure is used to find continuous edges.

Listing A.3 shows the functions for this step.

The connected edges are then sorted so that they face the same direction. This is important for the textured stylization. The method `sortStrokes()` in listing A.4 iterates over the silhouette edge list and flips the edges if necessary.

4.5 Silhouette Rendering

To be able to stylize the silhouette using different textures, billboard style quads are generated along the strokes. Due to the lack of a geometry shader on the current mobile platform, the quads are generated for every frame on the CPU. To provide correct texture scaling, the texture coordinates are generated with a scale factor which is proportional to the longest connected silhouette stroke. This is an arbitrary measurement, it is also possible to choose a fixed value or make the factor user definable. Listing A.5 shows the quad and texture coordinate generation part of the `drawPolygonalStrokes()` method. Alongside the texture coordinates, more information needed for the visibility determination is coded into the color values of the vertices. This is described in detail in section 4.6.

4.6 Visibility Determination

Listing shows the code for the silhouette shader. The inputs of the shader are three textures: the depth texture and two different stroke textures. It is of course possible to add further attributes to each texture, e.g. transparency information or additional color information. The lookup position is calculated from the fragment's texture coordinates and the interpolated lookup vector. From this position the surrounding depth values d0-d8 are read from the depth texture. If one of the depth map values is less than the current fragment depth, then the fragment is regarded visible and the according stroke texture is assigned. Figure 3.18 depicts this operation.

4.7 Implementation for Android

This section deals with the differences encountered while porting the PC-based prototype to the Android platform on a handheld device. The hardware used initially is a HTC Desire Z running Android version 2.3. The Processor is a Qualcomm MSM7230 Snapdragon with 800 MHz. Further evaluation was done on a ASUS Transformer tablet computer with a NVIDIA Tegra3 processor. The Android platform supports OpenGL ES 2.0, which

consists of a subset of the normal OpenGL and is optimized for mobile devices. Since the prototype is written in the Java programming language it was very straightforward to compile the application for Android. Aside some platform specific file reading operations, the rendering code required the most effort to adapt because of the differences to OpenGL ES. Mesh rendering is done via vertex buffers, listing A.6 shows the mesh preparation method.

The same procedure is also used to create the silhouette polygons. The determination of the vertices, texture coordinates and lookup vectors works the same way as shown in listing A.5, however the rendering portion is different on OpenGL ES. Listing A.7 shows the part of the silhouette rendering function where the information is prepared for the graphics hardware.

Chapter 5

Evaluation

Contents

The goal of this evaluation is to compare the discussed algorithms in terms of speed, scalability and quality. The brute force algorithm serves as provider for the ground truth, since it will always detect all silhouette edges. The quality of an algorithm is measured on the number of correct silhouette edges it detects. Scalability is evaluated with different input meshes which vary in the number of polygons. All tests are carried out on both PC and tablet computer.

5.1 Test Setup and Evaluation Procedure

The algorithms described in chapter 3 were evaluated on a desktop computer and a tablet computer. Table 5.1 shows the specifications of the used hardware. The code running on both setups is substantially the same to ensure a fair comparison. The algorithm code is mostly unchanged between the setups, necessary changes apply only to the application framework itself. For all test runs the same mesh files were used. Table 5.2 lists the properties of the different mesh files used for the evaluation. The mesh collection (see Figure 5.1) includes 3D-scans of real objects and artificial objects in different resolutions.

The test run consists of a fixed camera sweep around the object, with a fixed number of frames (400 frames). Figure 5.2 shows an illustration of the camera sweep. In every frame the processing time consumed by the silhouette detection algorithm under test is

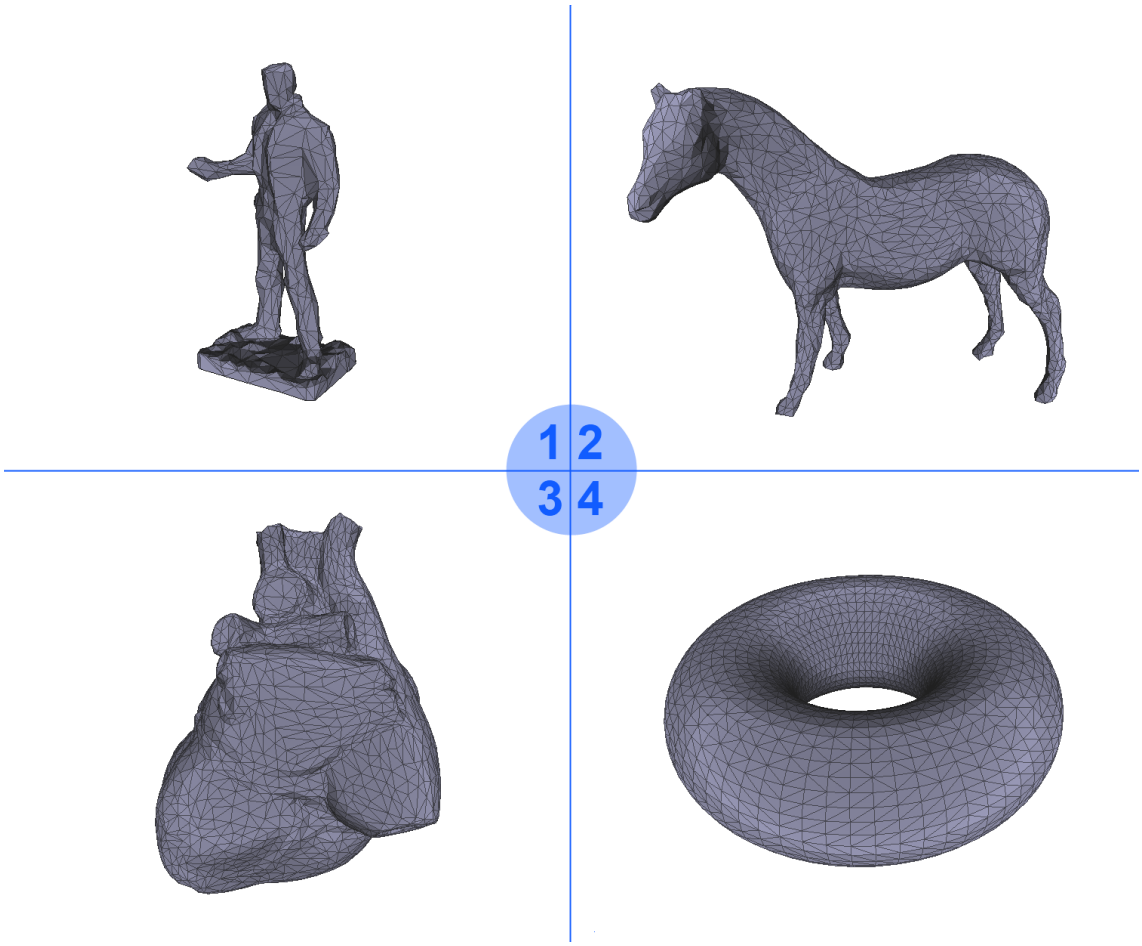


Figure 5.1: Evaluated meshes: Scan of a Terminator action figure (1), model of a horse (2), scan of a heart model (3), torus (4)

	Asus Transformer Prime TF201	Desktop Computer
Operating System	Android 4.0.3	Windows 8 (64 bit)
CPU	1.3 GHz Quad-Core Nvidia Tegra 3	3.0 GHz Intel Core2 Quad Q9650
Memory	1 GB	4 GB
Graphics Hardware	520 MHz ULP GeForce (Tegra 3)	NVIDIA GeForce GTX275

Table 5.1: Specification of the used hardware

Mesh	Polygons	Half-edges	Vertices
Arnold	1589	4834	809
Horse	3464	10392	1734
Horse2	21334	64002	10669
Heart	5008	15024	2508
Torus 2K	2304	6912	1152
Torus 3K	3072	9216	1536
Torus 4K	4608	13824	2304
Torus 8K	8192	24576	4096

Table 5.2: Properties of the used meshes

measured and logged. To ensure a precise measurement, the Java system function `System.nanoTime()` was used to measure the run time.

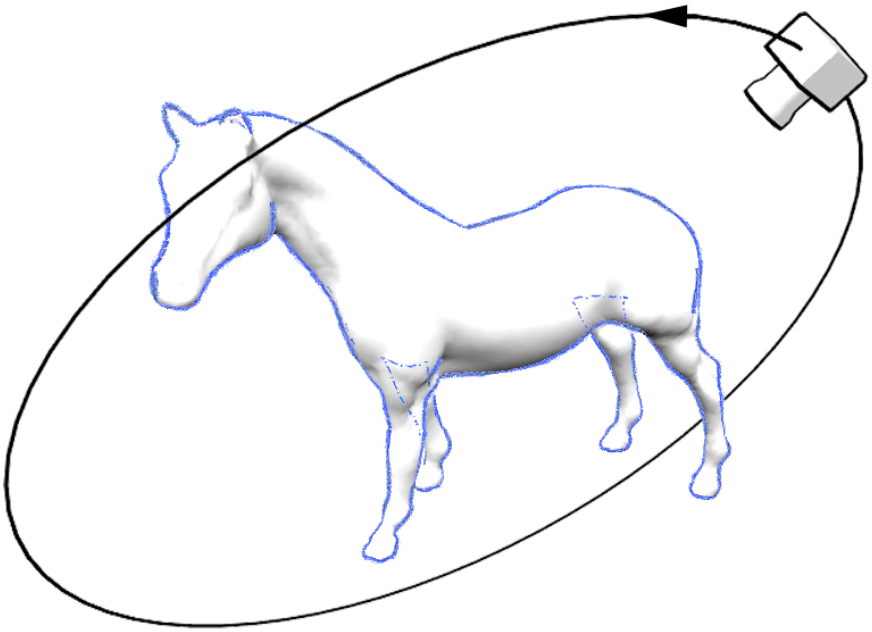


Figure 5.2: Illustration of the camera sweep used for the evaluation procedure. The camera will circle the object completely in 400 steps (frames).

The quality of the silhouette detection can be assessed by the amount of detected silhouette edges. The ground truth is provided by the brute force detection algorithm which is guaranteed to find all silhouette edges, since this algorithm always checks the entire mesh. A logging framework captures the processing time and the number of detected silhouette edges for every frame of the test run. This data is written to a log file for further evaluation.

5.2 Results

This section shows the test results obtained by the method described in section 5.1. The different algorithms are tested on all meshes from table 5.2 and furthermore on a mobile device and a PC (see table 5.1). Some renderings of the evaluated meshes can be seen in Figure 5.3.

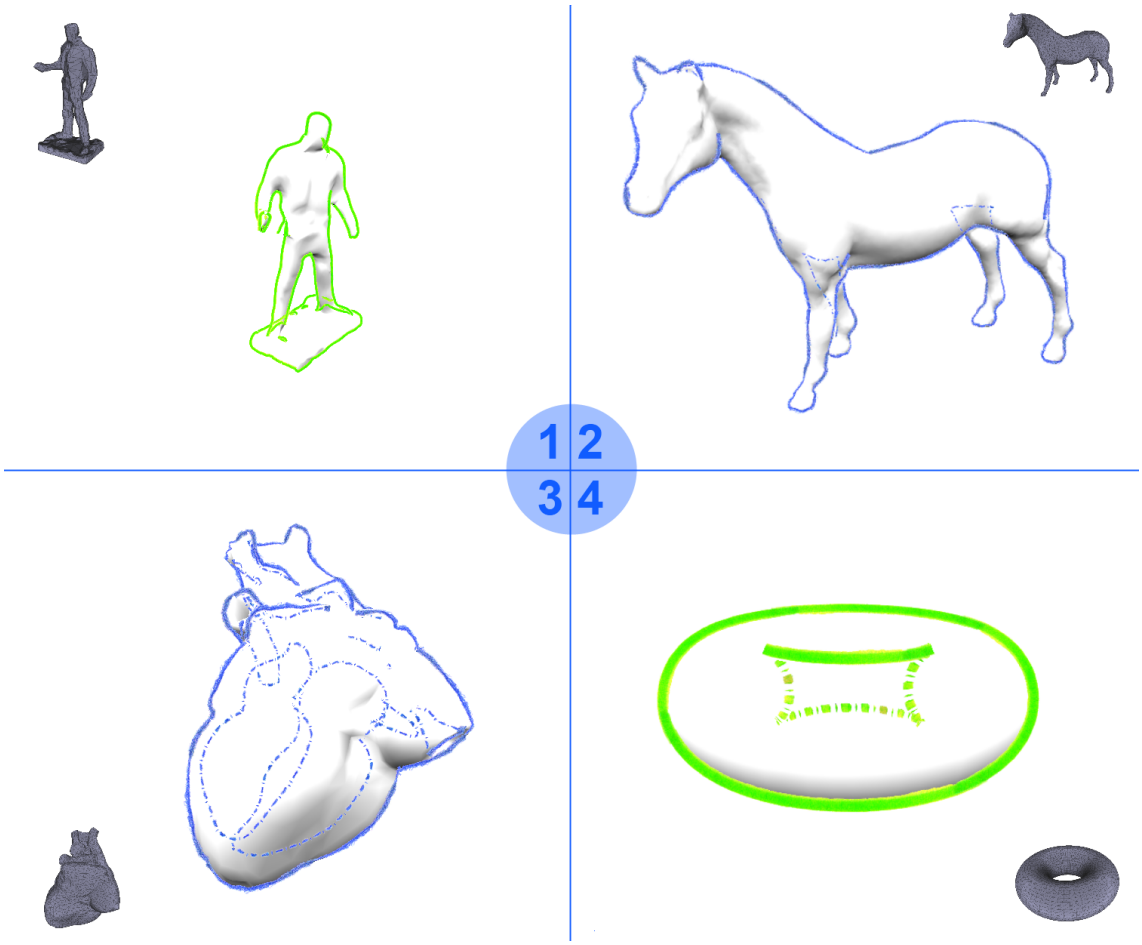


Figure 5.3: Stylized line renderings of the evaluation meshes. Highlighter brush style (1 and 4), blue colored pencil style (2 and 3)

Heart Mesh

The heart mesh is a 3D scan of a plastic heart model with a polygon count of approximately 5000 faces. An image of the mesh is depicted in Figure 5.1(3). The data of the evaluation runs on the PC and the mobile device are shown in Figures 5.4 and 5.5. On

the PC the speedup gained by the random and precompute algorithms is marginal, the coherence(B) approach even looses to the brute force approach. The overhead introduced by the algorithms diminishes the gain, especially on low polygon meshes. The picture on the mobile device is different, here the random, precompute and coherence(B) approaches are faster then the brute force approach. The coherence(A) approach is always fastest, but at the cost of quality. As a note, the PC benchmark may not be convincing, but as the mesh resolution increases a larger performance gain can be seen. A total time of 160ms for 400 frames equals to 2500 frames per second. Concerning the quality of the detection the brute force approach determines the ground truth because it is guaranteed that it will detect all silhouette edges. Figure 5.4 also shows that the quality of the random and precompute and coherence(A) approaches is very close to the brute force algorithm. The lowest quality silhouette is generated by the coherence(B) approach, because it looses some edge segments as the camera is moved. This is an expected behavior, and ways to improve the performance are discussed in section 5.3.

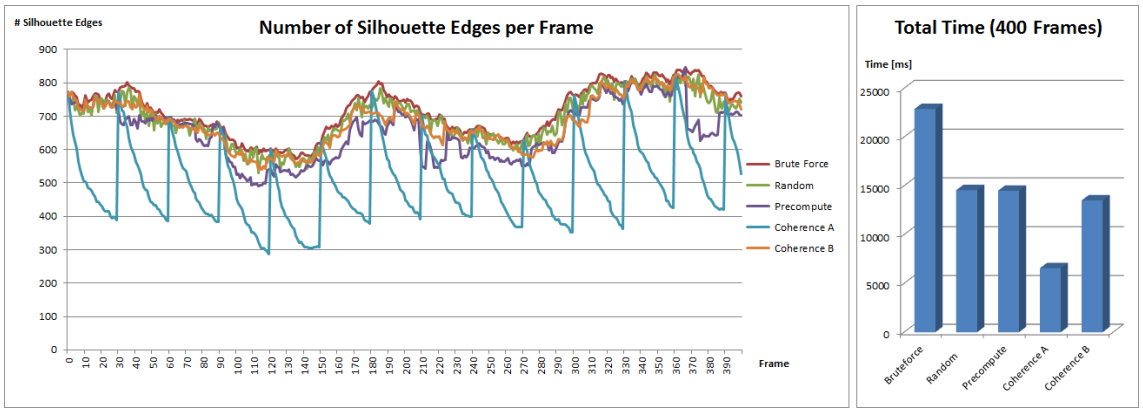


Figure 5.4: Heart mesh (5008 polygons) performance log on the mobile device.

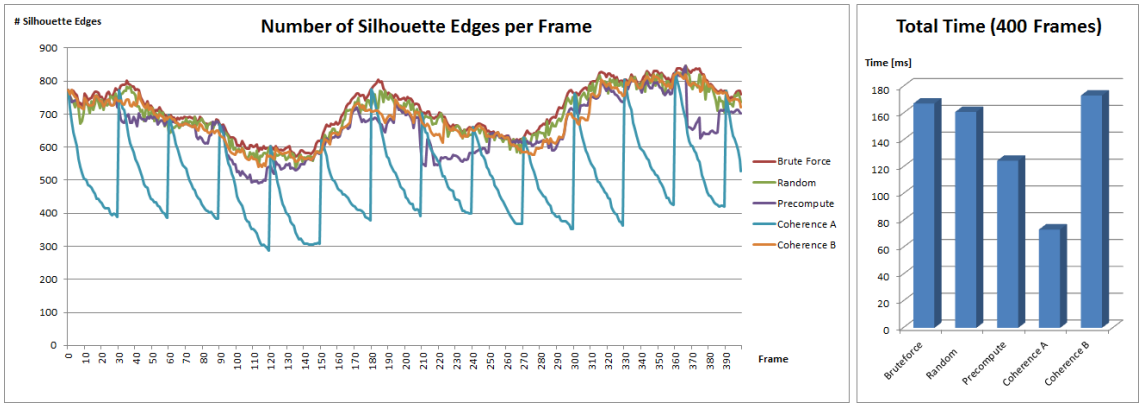


Figure 5.5: Heart mesh (5008 polygons) performance log on the PC.

Horse Mesh

The horse mesh shown in Figure 5.1(2) consists of approximately 3400 polygons. It is created by an artist and features mostly smooth surfaces but also some small details like the ears. Figures 5.6 and 5.7 show the evaluation data of the performed test runs. On the PC, the difference between the algorithms is minimal, similar to the evaluation of the heart mesh. On the tablet computer the brute force approach is slowest, followed by the random, precompute and coherence(B) approaches which have similar performance. The silhouette quality charts show that in this case the random and precompute algorithm have almost the same quality of the brute force approach, whereas the coherence(A) algorithm tends to loose silhouette edges as the camera moves. The results for the horse mesh are as expected similar to the heart mesh. The performance gain is better for high polygon meshes. The Horse2 mesh has the same shape as the Horse mesh, but has a polygon count of 21334. Figures 5.8 and 5.9 show the evaluation results for the Horse2 mesh. On both platforms the advanced algorithms are faster than the brute force algorithm.

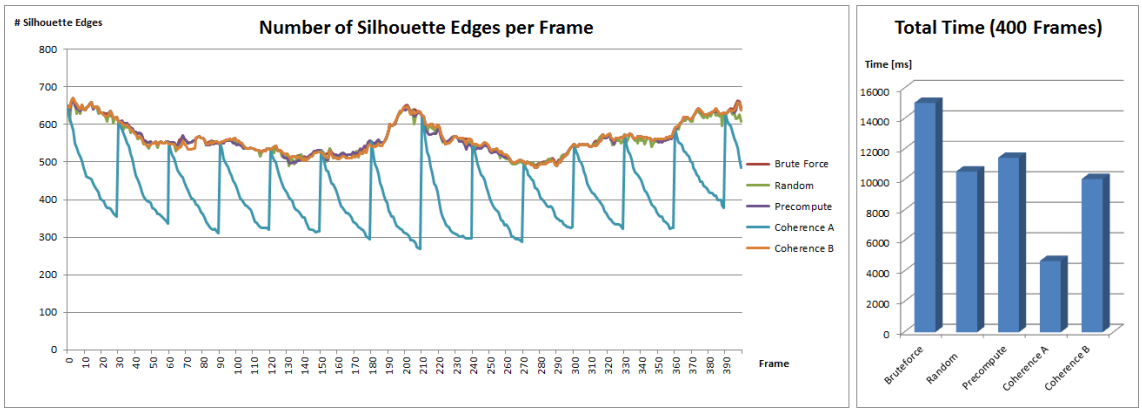


Figure 5.6: Horse mesh (3464 polygons) performance log on the mobile device.

Arnold Mesh

The Arnold mesh is a scan of an Terminator action figure which has been reduced to about 1600 polygons. This mesh has the lowest polygon count in the evaluation. The results (Figures 5.10 and 5.11) are similar to the horse and heart meshes. The overhead introduced by the advanced algorithms results in lower performance on low polygon meshes.

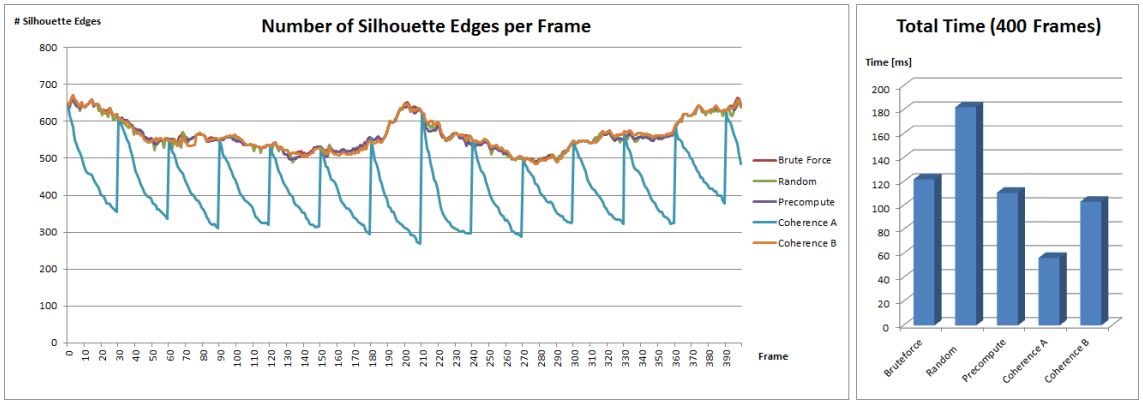


Figure 5.7: Horse mesh (3464 polygons) performance log on the PC.

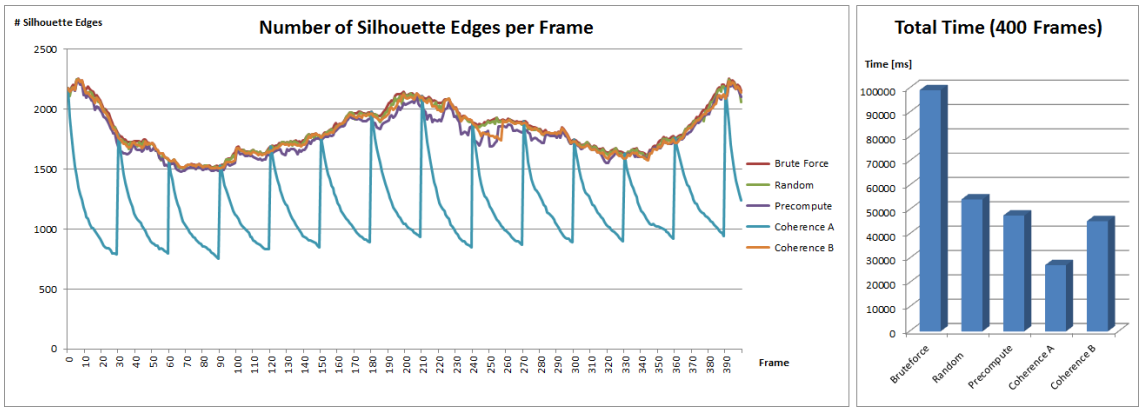


Figure 5.8: Horse2 mesh (21334 polygons) performance log on the mobile device.

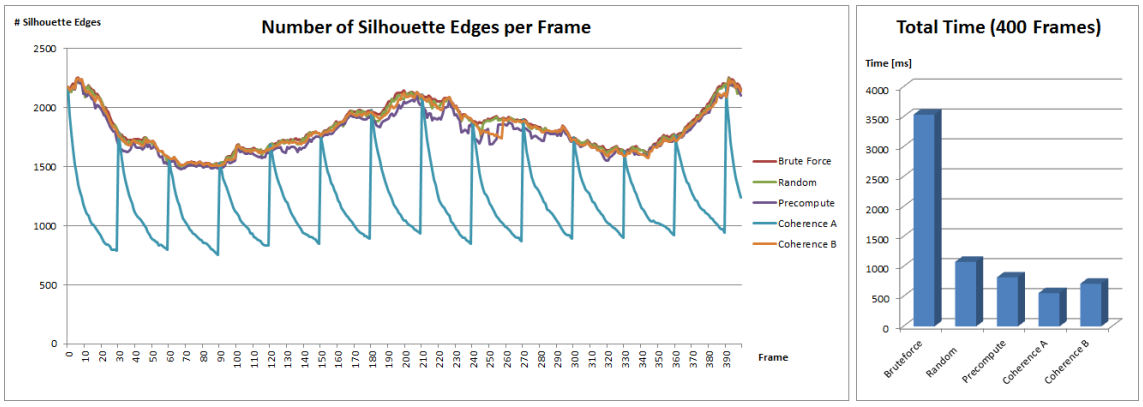


Figure 5.9: Horse2 mesh (21334 polygons) performance log on the PC.

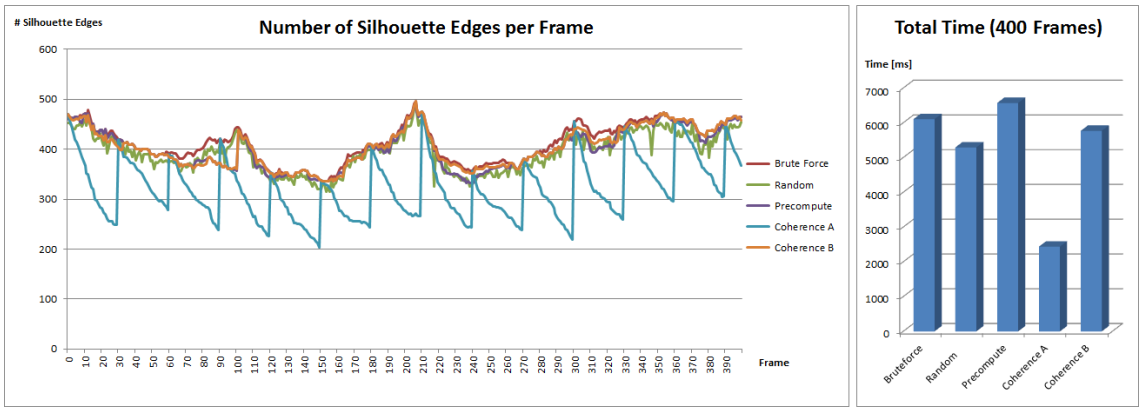


Figure 5.10: Arnold mesh (1589 polygons) performance log on the mobile device.

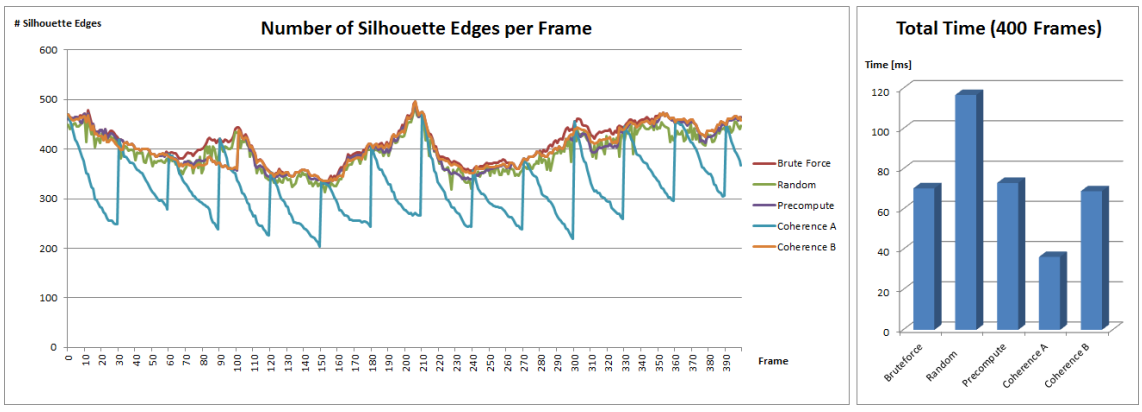


Figure 5.11: Arnold mesh (1589 polygons) performance log on the PC.

Torus Meshes

Finally, the torus meshes are artificial objects generated by a 3D modelling tool. The meshes come in different resolutions ranging from 2000 to 8000 polygons but the shape of the tori stays the same. These meshes have been evaluated mainly to show the algorithm performance in relation to the mesh resolution. The results of the comparison are discussed in section 5.3.

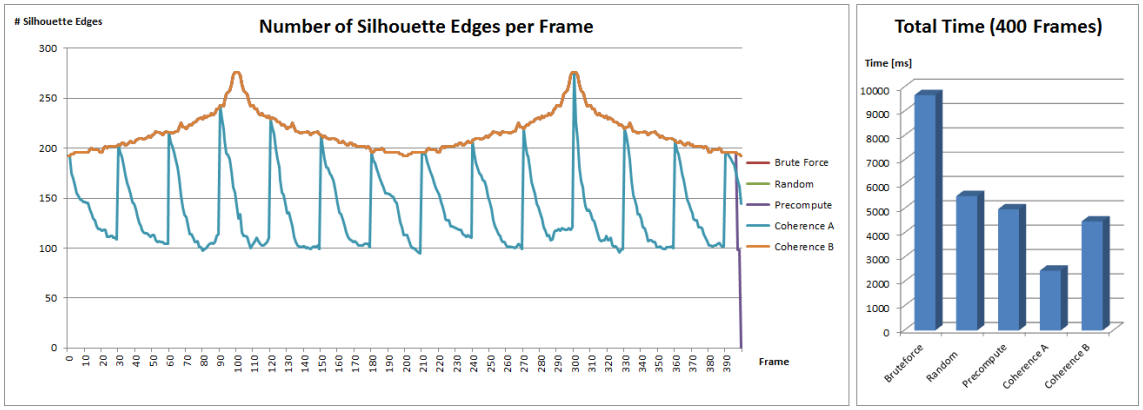


Figure 5.12: Torus 2K mesh (2304 polygons) performance log on the mobile device.

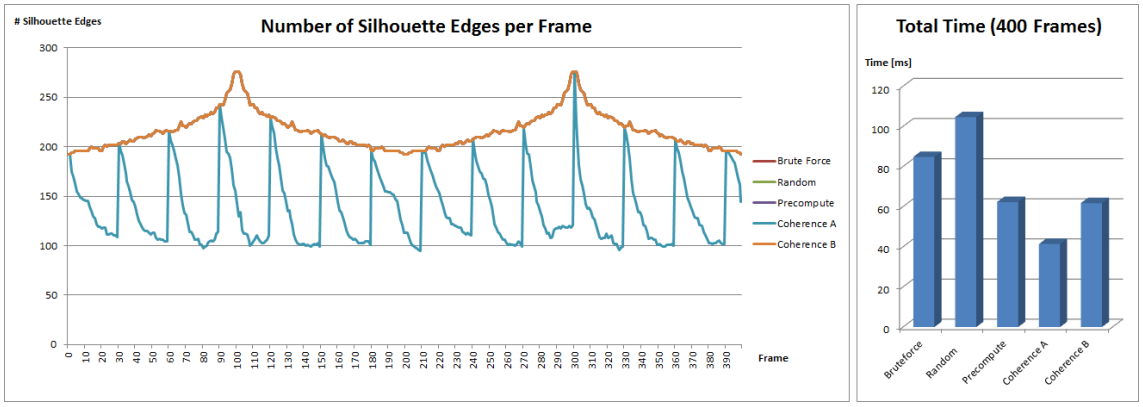


Figure 5.13: Torus 2K mesh (2304 polygons) performance log on the PC.

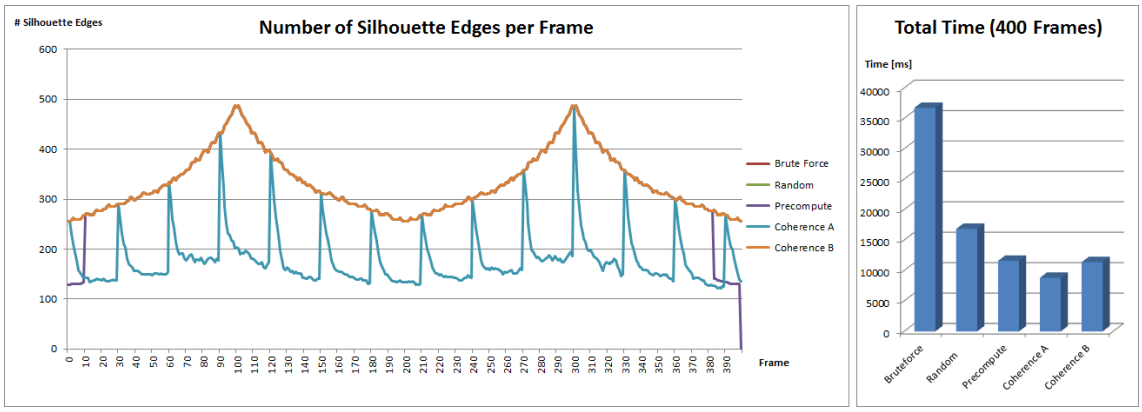


Figure 5.14: Torus 8K mesh (8192 polygons) performance log on the mobile device.

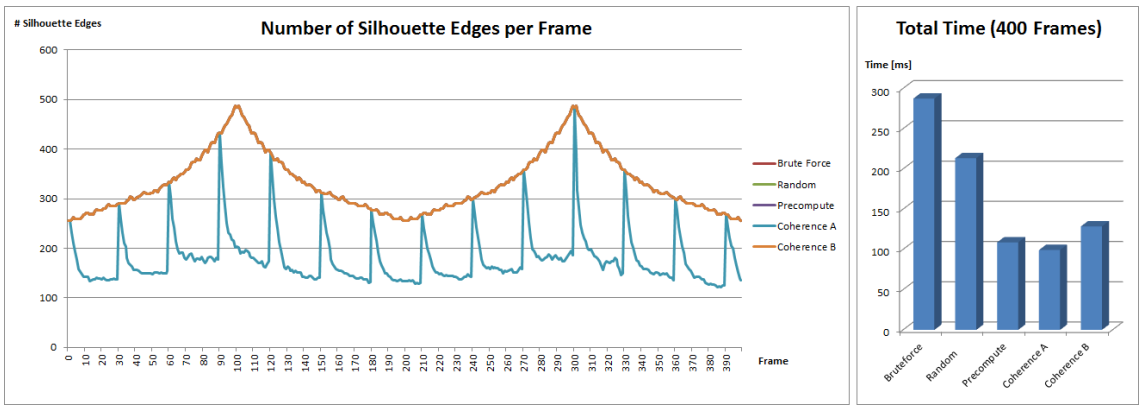


Figure 5.15: Torus 8K mesh (8192 polygons) performance log on the PC.

5.3 Discussion

The evaluation of the implemented algorithms both on the PC and the mobile device showed a clear performance win compared to the basic brute force algorithm. In this section the results are discussed in detail, starting with an analysis of overall algorithm performance in relation to mesh resolution.

5.3.1 Mesh resolution impact

The comparison of algorithm performance in relation to the mesh resolution clearly shows that the implemented algorithms for silhouette detection yield a performance improvement compared to the brute force approach. Figure 5.16 shows a chart of the average processing times captured on the mobile device. The mesh files used are tori with different amounts of polygons (torus 2K, 3K, 4K and 8K, see table 5.2).

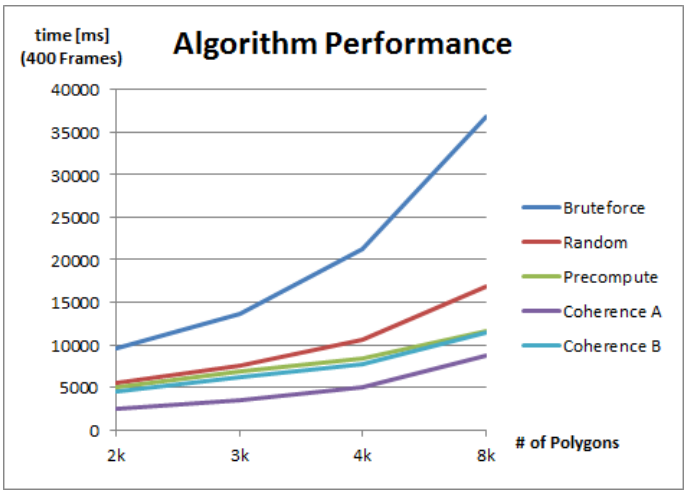


Figure 5.16: Plot of the processing time of the different algorithms for meshes with increasing resolution

The brute force algorithm (blue line) shows an almost linear behavior, which was expected since it has to iterate over all faces of the mesh. The other algorithms have a much better performance with increasing polygon count. The random approach is set to work on a 10% set of the total polygons in this test run, so with the increasing mesh resolution more polygons are taken into account. The precomputed silhouettes approach yields a good performance and is very similar to the coherence(B) algorithm. The coherence(A) algorithm is fastest, but with a very low quality (see section 5.2).

5.3.2 Random and Precompute approach

According to the captured test data, both random and precompute approach perform very well regarding processing time and silhouette quality. It is notable that the randomisation approach has a very good quality, i.e. the number of detected silhouette edges is in most cases similar to the brute force approach. Figure 5.16 shows that the randomisation approach also has a good performance which is only slightly dependent on the total number of polygons in the mesh. In the configuration used for the evaluation the algorithm was set to use a randomly selected set of 10% of the mesh faces. This parameter can be used to fine-tune the algorithm, but the selected percentage proofed to be good (a range between 5% and 20% was tested).

The precomputed silhouettes approach shows similar results as the randomisation approach, but it scales differently with the mesh resolution. In most cases the quality is similar to the randomisation and brute force approach. Glitches can occur if the precomputed viewpoints are too sparse. The precomputation step itself does not contribute to the runtime performance and it is possible to store the results of the precomputation in combination to the mesh file. The only drawback of having a large number of viewpoints is the memory usage, but this can usually be neglected. It is notable that the performance of the precomputation approach scales very good with the resolution of the mesh, as can be seen in Figures 5.16 and 5.15. The performance and quality of this algorithm can be manipulated using the number and position of the viewpoints for precomputation. In the evaluation run a number of 40 evenly distributed viewpoints is used (8 rotation and 5 inclination steps).

5.3.3 Interframe coherence approach

The version A of the interframe coherence approach did not perform as well as expected concerning the quality of the silhouette. It tends to loose too much silhouette edges over the test run. The problem here lies in the fact that only the nearest neighborhood of the seed faces is included in the silhouette search. This of course can be done very fast, but if the current silhouette has moved out of the search range determined by the previous silhouette detection step, the algorithm does not find the new silhouette edge. This error propagates through the next frames and the quality of the silhouette decreases further. The evaluation used a configuration where the initial seed for the coherence search was done by the brute force approach. The re-seeding took place every 30 frames. Figure 5.17 shows a detailed partial log of a test run. The re-seeding times can be identified by

the peaks in the edge detection rate. After re-seeding a continuous loss of edges occurs, which propagates through the following frames until the next re-seeding step. Possible

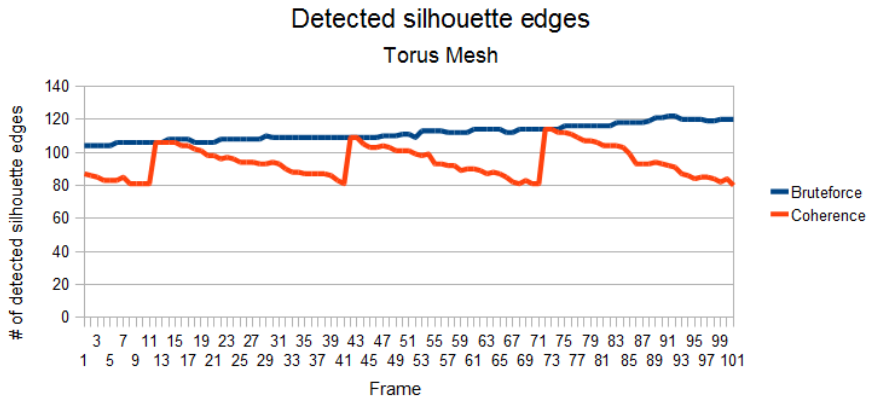


Figure 5.17: Plot of the detected silhouette edges by the brute force (blue) and the coherence (orange) algorithm. The effect of the re-seeding is clearly visible.

improvements could include a higher frequency of re-seeds, a larger search window and possibly the use of a different algorithm for re-seeding. A higher re-seed frequency would increase the quality, but of course the performance of the algorithm would be lower. Instead of the brute force search it would be possible to use one of the faster algorithms (for example the random or precompute approach). A larger search neighborhood around the seed faces would also improve the quality of the coherence algorithm, but again this would lower the overall performance. The version B of the coherence approach showed a better behavior with performance and quality similar to the precompute approach.

5.3.4 Image based approach

The image based selection of seed edges (image based approach) has some major drawbacks. First, it can not detect all silhouette edges because hidden silhouettes are invisible in the rendered image. Second, it needs additional render passes for rendering the mesh with color-coded normals, with face IDs and for edge detection. These render passes do not affect the performance greatly, but they still decrease the overall performance since the mesh has to be rendered multiple times. Third, the most severe drawback is the slow memory access for reading back data from the graphics hardware to the CPU. The color-coded face IDs which are determined by the image space edge detection need to be read from a texture. This process is slow on the PC and extremely slow on the tested mobile device. Figure 5.18 shows the processing time of the image based approach compared to the other tested algorithms. The test results show that the image based approach is far slower than the brute force algorithm and additionally offers less quality.

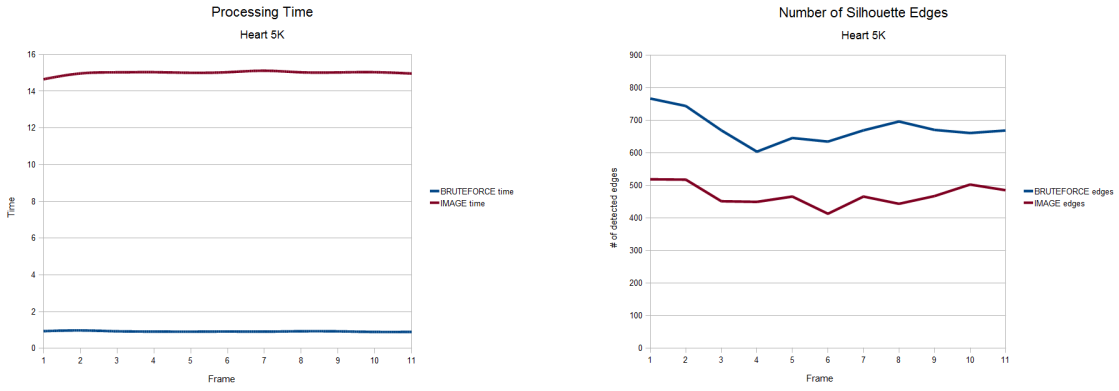


Figure 5.18: Plot of the performance of the image based approach compared to the brute force approach. The main reason for the poor performance is the slow memory access to the graphics hardware. The data was captured from a test run on the PC.

Chapter 6

Application

In this chapter the results of the AR line rendering framework are presented. The images are captured from the OpenGL framenbuffer on the mobile device and reflect exactly the screen output on the device. Two AR setups were used to demonstrate the line renderer. The setups can be seen in Figure 6.1. For both examples a natural feature tracking target was used.

(a) (b)

Figure 6.1: AR setups: An educational heart model (a) and a large metal valve mounted on a plate (b).

For every setup, experiments with different styles and rendering methods were conducted. The following images show how the system can render line illustrations in different ways. One rendering method is the normal overlay of stylized lines without occluding the real object. The second method is to add the rendering of the mesh in addition to the stylized lines. This approach partially occludes the real object, which in some cases improves

the display of the stylized lines.

Figure 6.2 shows the AR setup with the valve. In this example only the top handwheel has been used to augment the scene. The line style is a simple white brush for the visible parts and a white dashed style for the hidden lines. The colors of the real handwheel and valve are very dark, so the white lines are well suited for this example as they are clearly visible.

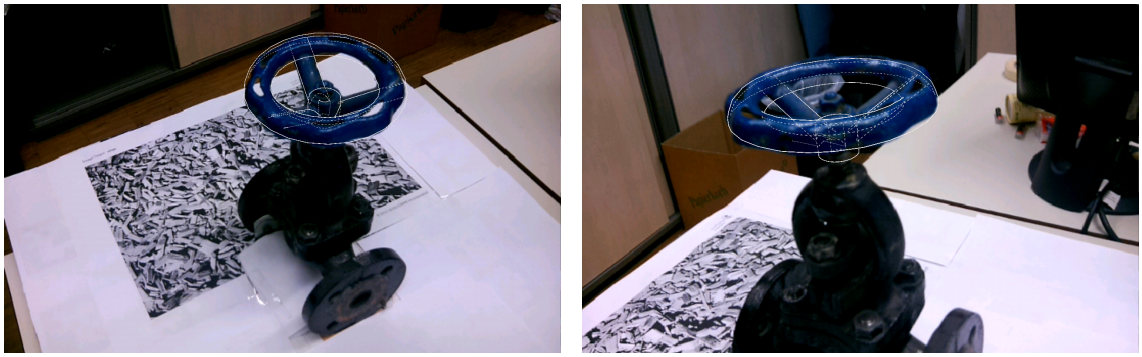
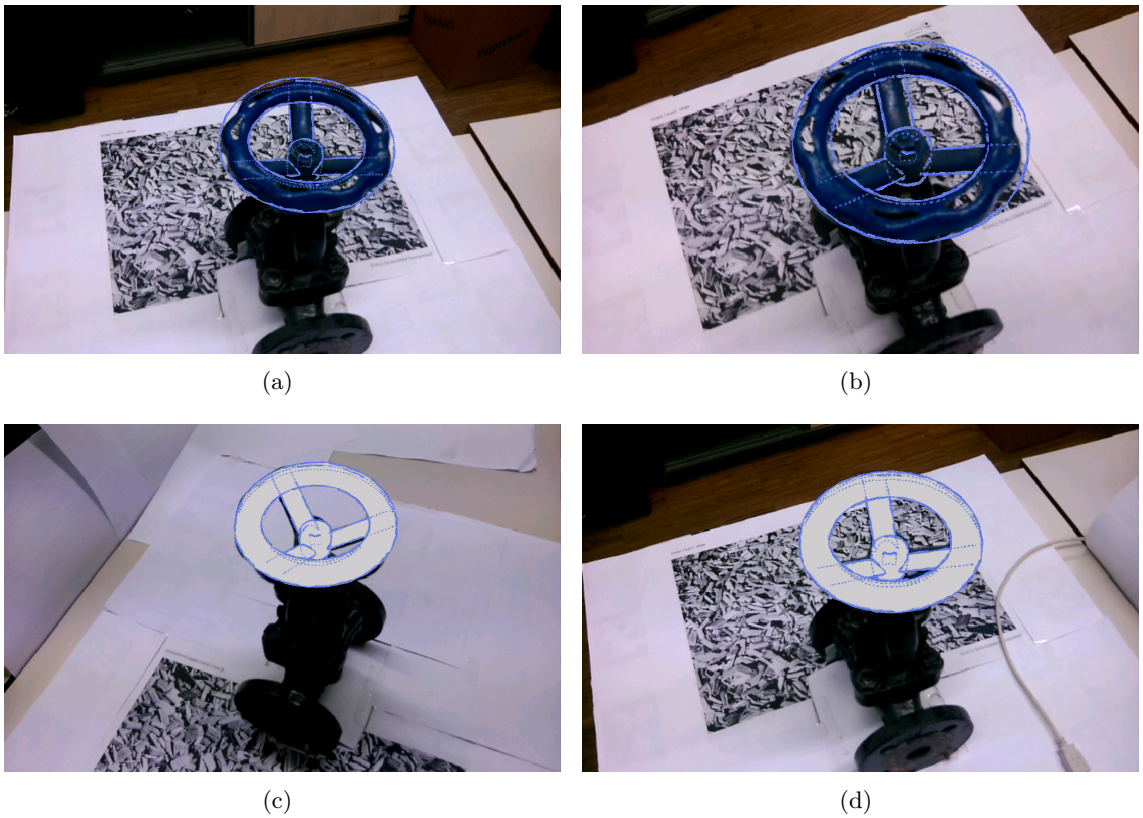


Figure 6.2: AR scene of the valve with a line rendering of the top handwheel. A white brush is used to stylize the lines. The dark colors of the valve make the lines clearly visible.

The next example shows the valve setup augmented with the pencil brush style and different and rendering methods. The overlay of the mesh occludes part of the real object, a sort of diminished illustrative AR line rendering. Figure 6.3 shows the screenshots for this experiment. The scene was captured in different angles with line-only and occluded rendering. The blue color of the real handwheel makes it difficult to see the stylized lines. The diminished version with occluded handwheel shows the lines clearer and gives the scene a hand-drawn look, which could be useful in the illustration of AR manuals. The look resembles that of the Lego robot shown in Figure 1.5b.

Figure 6.4 shows the valve scene with more different line styles. The brushes were derived from a bright text marker (Figure 6.4a and c) and a wiggly line drawn with a fountain pen (Figure 6.4b and d). Both styles appear differently in the normal rendering and in the diminished illustrative AR line rendering. Both brushes are not suited for technical illustration and yield a more artistic look, the example was made to show that arbitrary line styles are possible with the framework.

Figure 6.5 shows screenshots from the heart model scene. The line rendering was done using a white brush, a black brush and a colored pencil brush. The stylization of hidden lines can clearly be seen at the center of the heart where the heart ventricles lie



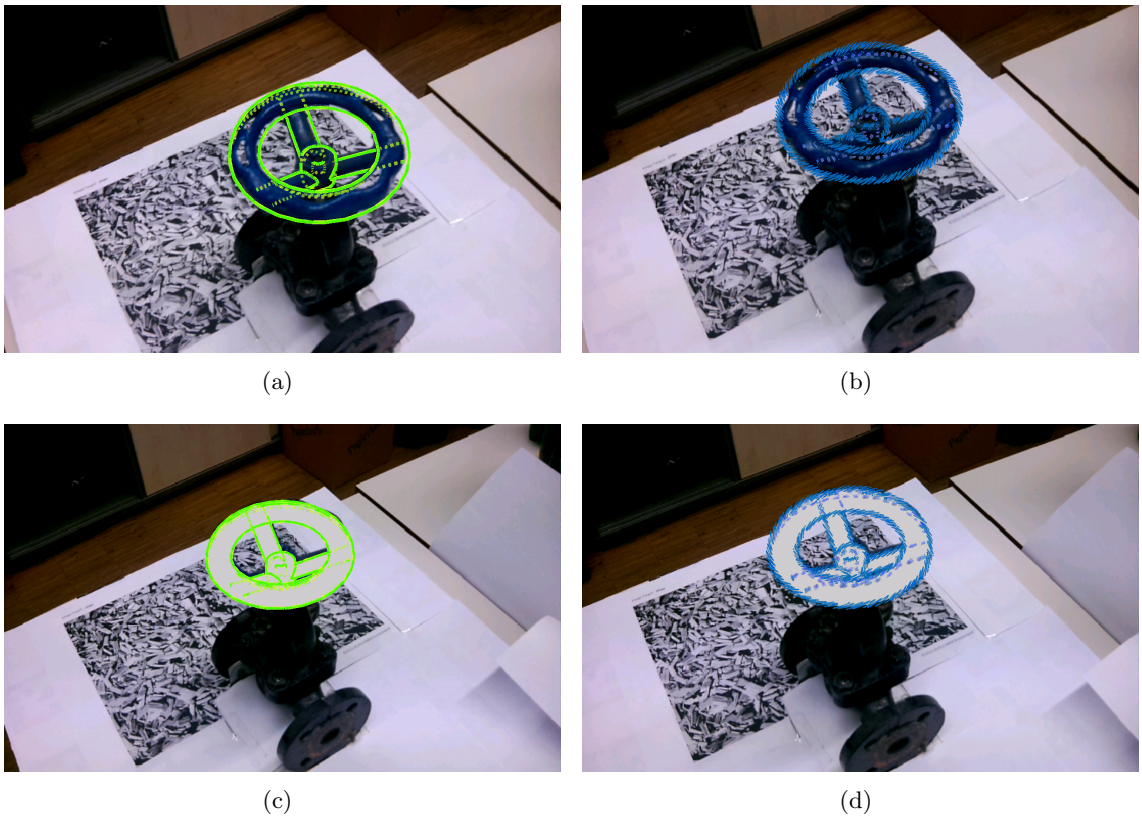
(a) (b)

(c) (d)

Figure 6.3: The valve scene augmented in different ways: For the lines a colored pencil brush was used, the hidden lines are rendered with a dashed pencil line. Image (a) and (b) show the valve from different angles with stylized lines only. The diminished AR line illustration (c and d) show the scene with the occluded handwheel.

underneath. The brushes used in Figure 6.5a and c emulate a technical illustration, while the pencil brush renders a hand-drawn image. In this experiment all brushes are clearly visible in both rendering methods. The pencil brush in Figure 6.5b adds a bit of clutter to the image.

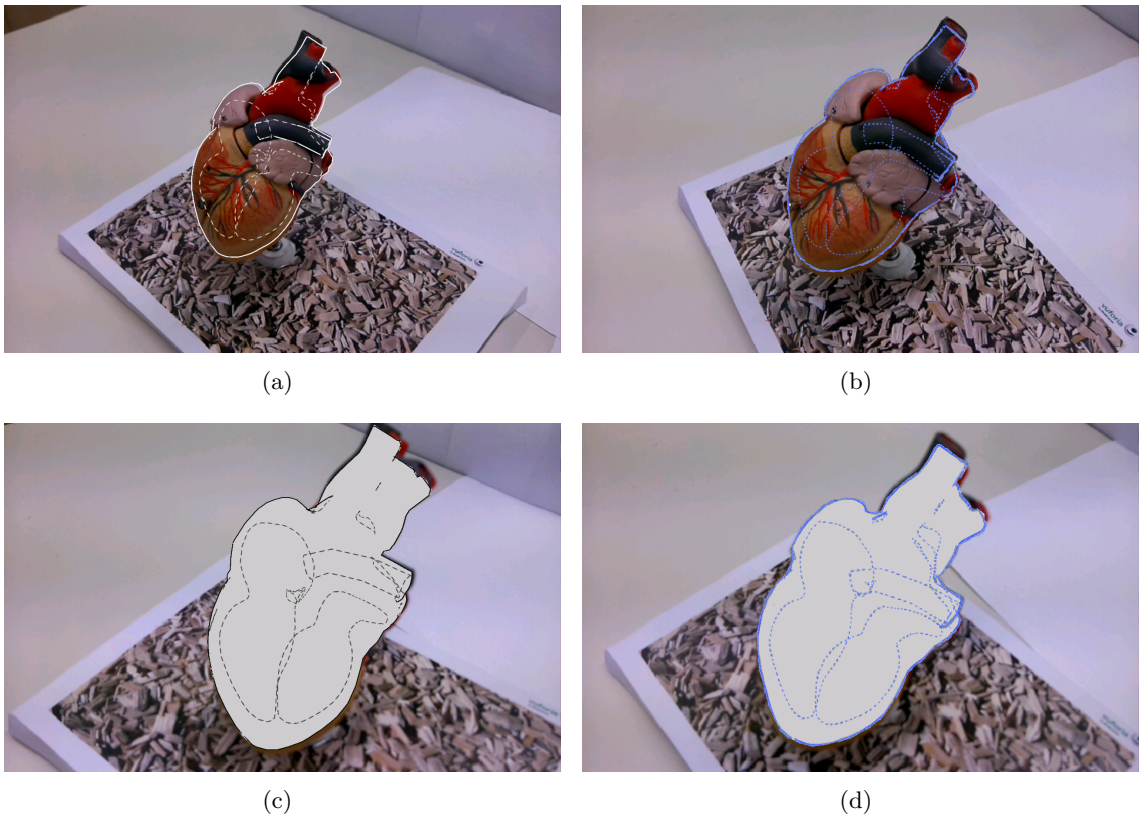
Figure 6.6 shows the heart model illustrated with other possible line styles. It can be seen that not all brushes are suited for the purpose of creating a technical illustration, nevertheless interesting effects can be created. Concludingly Figure 6.7 shows some example renderings of the PC based version of the framework.



(a) (b)

(c) (d)

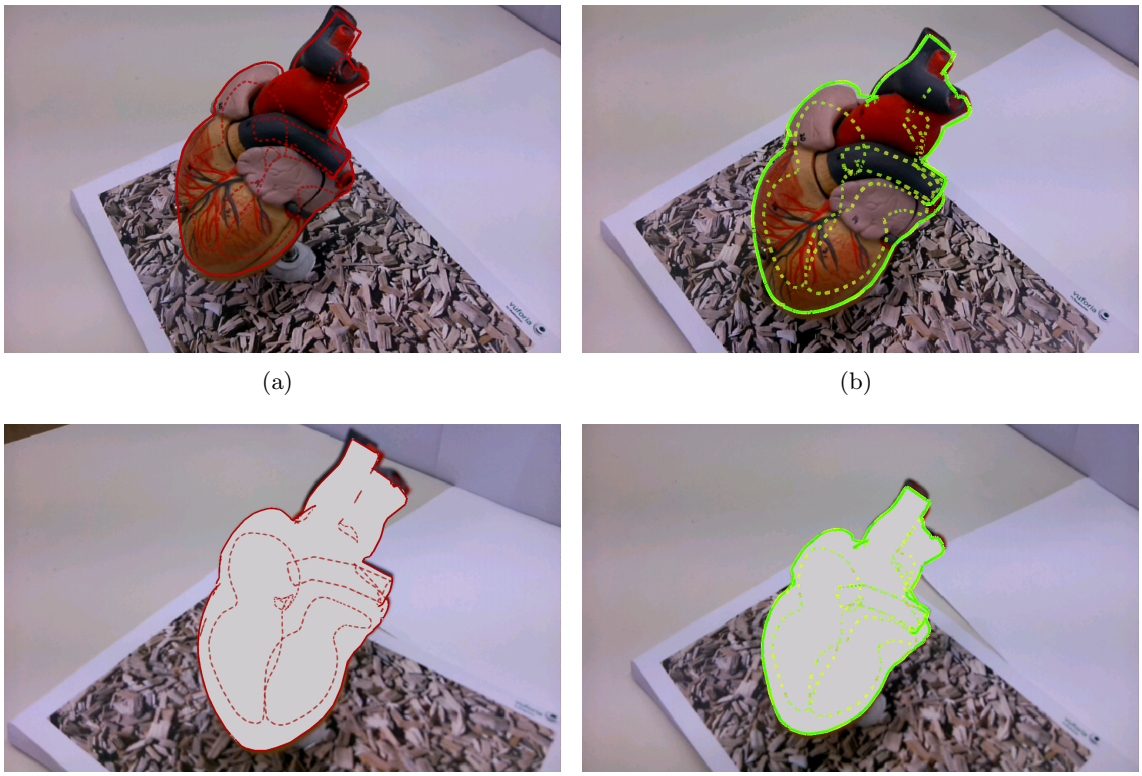
Figure 6.4: Application of different line styles and rendering methods: The brush in (a) and (c) is derived from a green bright text marker and the brush in (b) and (d) is a wiggly line created with a fountain pen. Each style was applied as a normal rendering and as a diminished reality illustration. The bright text marker stands out in the dark colors of the valve in the normal rendering (a), but it is not so visible in the diminished version (c). On the contrary the normal rendering of the wiggly lines appears cluttered while the diminished illustration creates an interesting look.



(a) (b)

(c) (d)

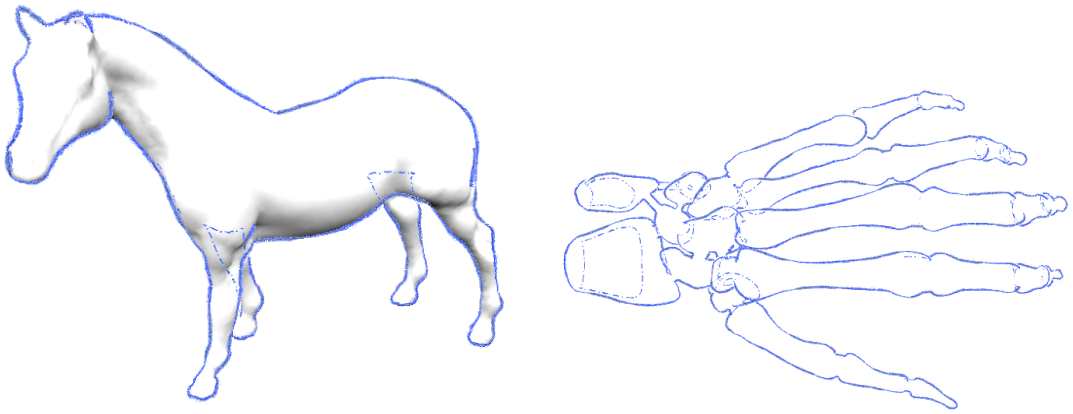
Figure 6.5: AR scene with an educational heart model: The heart was scanned with a 3D scanner and referenced on the natural feature tracking target. The model was rendered using different styles and rendering methods. In image (a) a white brush was used to stylize the silhouette lines. The white lines can clearly be seen on the darker background. An inverted brush was used in (c) because the mesh overlay is white. The result looks like a technical illustration. In (b) and (d) the colored pencil brush was used for of the lines. The visibility can be described as good in both images, where the diminished AR line rendering looks more like a hand drawn overlay. In this scene the whole object of interest was occluded by the illustration.



(a) (b)

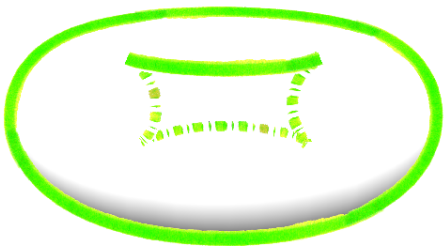
(c) (d)

Figure 6.6: The heart model illustrated with different styles and rendering methods: The brush style in (a) and (c) is a simple red line brush. Since the basic colors in the real world object are also reddish, this style is not a good fit for the normal rendering in this case. This style works better on bright objects or in the diminished AR illustration. The marker brush was also applied to the line rendering of the heart (b and d). In the normal rendering the brush stands out against the backdrop. Similar to the valve example the marker brush is too bright for the diminished illustration (d). Additionally the marker brush is to wide and does not show the details of the heart model well.

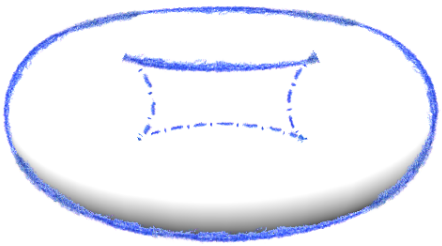


(a)

(b)

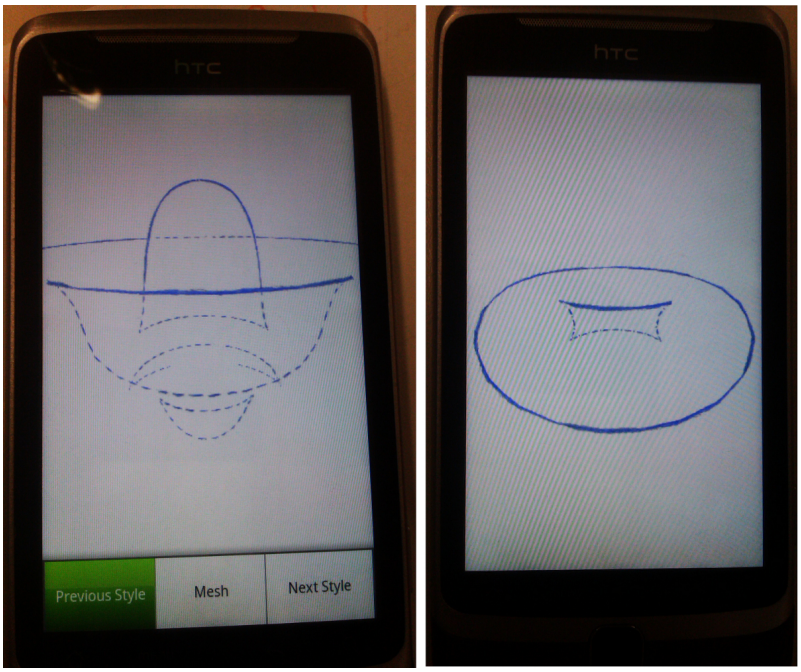


(c)



(d)

Figure 6.7: Images rendered with the PC version of the framework: These are VR images, showing different models and line styles. Horse mesh with pencil brush (a), skeleton of a hand with pencil brush (b), torus with pencil and marker brush (c and d).



(a)

(b)

Figure 6.8: The silhouette rendering application was installed on different handheld devices. The HTC Desire Z smartphone is shown in (a) where it displays the torus with a pencil style in VR. The more powerful ASUS Transformer Prime Tablet, which was also used in the evaluation, shows an AR rendering of the valve scene with a wiggly line style.

Chapter 7

Conclusion and Future Work

In this thesis a non-photorealistic line rendering framework for handheld augmented reality was created. Several algorithms and methods for fast silhouette detection in object space were developed and implemented. The half-edge data structure for mesh representation was used to provide efficient search mechanisms. A rendering pipleline for fast visibility determination using shader programs was created. The framework was tested on a normal PC and a mobile device running Android OS.

In section 3.1 the use of a half-edge data structure was presented. The structure proved to be useful for the task, as it provides efficient navigation methods and fast iterators to walk over the geometry. The neighborhood information included in the data structure was helpful for all local search operations after the seeding process. The only drawback is that because of the underlying map data structure, access on individual elements of the mesh is slower than for example a simple array based structure.

Section 3.2 introduced several methods to increase the performance of the object space based silhouette detection. In general, the efficiency of the silhouette detection can be improved by reducing the amount of input polygons for the silhouette calculation. This procedure is called seeding or seed face selection. The developed algorithms focus on the selection of promising silhouette face candidates which serve as starting points for a close range neighborhood search to find the silhouette. Some algorithms did not perform in a satisfactory manner while others showed a very good performance. On the one hand, the image based seed face selection fails to meet the speed requirement because of the slow graphics memory access, and it is also not able to detect some hidden silhouettes. The interframe coherence approach performs very fast, but it fails at the quality requirement. Improvements of the algorithm are possible but not without compromising the speed. On

the other hand, the randomisation approach proved to be fast and also generates a good quality silhouette compared to the ground truth provided by the brute force search. Also the new precomputed silhouettes approach showed a satisfying behavior, being fast and also accurate, and it also scales very good with increasing mesh resolution.

A method for generating continuous strokes from single silhouette edge segments was presented in section 3.3. The approach relies on the neighborhood information provided by the half-edge data structure which eases the process of finding adjacent edges. The algorithm outputs a number of so-called silhouette strokes, where each stroke represents a continuous part of the objects silhouette. The stroke generation step makes it possible to apply a style to the silhouette.

In section 3.4 the methods which were used to apply stylization to the silhouette were presented. The generated continuous strokes were rendered using a billboard approach which supplies textured polygons to the rendering pipeline. The use of shader programs made it possible to achieve an efficient visibility determination on silhouette lines. This was done by encoding additional information like lookup coordinates into the vertex attributes. The shader program then used a depth map of the object to determine the line visibility at pixel accuracy.

The important details of the implementation were described in section 4, where an overview of the used data structures and the particular differences between the PC and the mobile platform implementation was given. The application code for both platforms was written in the Java programming language which made it possible to transfer the code between PC and mobile implementation with almost no changes.

Section 5 presented the evaluation results of the different seeding algorithms. All methods were compared against the brute force approach which also provides the ground truth for the silhouette quality. It could be shown that some seeding mechanisms improved the performance of silhouette notably. Tests were performed on a PC and a mobile device. There were no unexplainable differences between the algorithm performance between the platforms, given that the mobile device does not have the same computation power and the amount of memory as it is available on a PC.

In the course of this thesis efficient silhouette detection algorithms were identified which deliver object-space silhouettes in the form of continuous, parametrized strokes. The visibility determination using shader programs also proved to be fast. Additional work could involve a port of the system to another programming language like C++, which is also natively available on the Android platform. It would also be interesting to

investigate a more efficient system for handling the silhouette geometry data, which is currently newly generated every frame and uploaded to the graphics hardware. Future mobile devices may provide more options like a geometry shader, which would enable more efficient methods for vertex handling. Furthermore it would be nice to create a more integrated user interface, possibly a screen overlay, so that the user can tweak stylization parameters while seeing a real time feedback of his changes.

Appendix A

Source Code

Listing A.1: Simple Brute Force Silhouette Detection

```
Iterator<Halfedge> he_it = mesh.getHalfedgeIter();
while(he_it.hasNext())
{
  Halfedge he = he_it.next();
  if(he.isVisited())
  {
    continue;
  }
  Face f1 = he.getFace();
  Face f2 = he.getTwin().getFace();
  Vertex3f v1 = he.getOrigin();
  Vertex3f v2 = he.getTwin().getOrigin();

  he.setVisited(true);
  he.getTwin().setVisited(true);

  Vector3f n1 = f1.getNormal();
  Vector3f n2 = f2.getNormal();

  Vector3f cam = new Vector3f(cam_r * cam_lx, cam_r * cam_ly, cam_r *
      cam_lz);
  cam.sub(v1);

  float d1 = cam.dot(n1);
  float d2 = cam.dot(n2);

  if(d1 * d2 < 0)
```

```
    {
28    GL11.glBegin(GL11.GL_LINES);
      GL11.glVertex3f(v1.x, v1.y, v1.z);
30    GL11.glVertex3f(v2.x, v2.y, v2.z);
      GL11.glEnd();
32  }
  }
```

Listing A.2: Brute Force Sub-Polygon Silhouette Detection

```
1 /*
   * Brute force search for silhouette edges over the entire mesh
3  * For each vertex the dot product with the camera vector is calculated and
       stored.
   * For each face, the dot products of its vertices are compared regarding
      their sign.
5  * If the dot products have different signs along an edge, there is a
      silhouette point.
   * The vertices of the silhouette edge are interpolated along the face
      edges.
7  */
  public void drawSilhouetteBfInterpolate()
9 {
    Iterator<Face> f_it = mesh.getFaceIter();
11  while(f_it.hasNext())
    {
13    Face f = f_it.next();
      FaceVertexIter fv_it = new FaceVertexIter(f);
15    while(fv_it.hasNext())
      {
17      Vertex3f v = fv_it.next();
        if(v.isVisited())
19      {
          continue;
21      }
        Vector3f cam = new Vector3f(cam_r * cam_lx, cam_r * cam_ly, cam_r *
            cam_lz);
23      Vector3f n = v.getNormal();
        cam.sub(v); // cam - v
25      v.setDotp(cam.dot(n));
        v.setVisited(true);
27    }
```

```
29      fv_it = new FaceVertexIter(f);
        Vertex3f v1 = fv_it.next();
31      Vertex3f v2 = fv_it.next();
        Vertex3f v3 = fv_it.next();
33
        float d1 = v1.getDotp();
35      float d2 = v2.getDotp();
        float d3 = v3.getDotp();
37
        // All dot products have the same sign - no silhouette edge
39      if(d1 * d2 > 0 & d1 * d3 > 0 & d2 * d3 > 0)
        {
41        continue;
        }
43
        Vertex3f v12 = new Vertex3f(v2.x, v2.y, v2.z);
45      v12.sub(v1);

47      Vertex3f v13 = new Vertex3f(v3.x, v3.y, v3.z);
        v13.sub(v1);
49
        Vertex3f v23 = new Vertex3f(v3.x, v3.y, v3.z);
51      v23.sub(v2);

53      float dist12 = v1.distance(v2);
        float dist13 = v1.distance(v3);
55      float dist23 = v2.distance(v3);

57      v12.divideVertex(dist12);
        v13.divideVertex(dist13);
59      v23.divideVertex(dist23);

61      float step_v1_v2 = dist12 / (Math.abs(d1) + Math.abs(d2));
        float step_v1_v3 = dist13 / (Math.abs(d1) + Math.abs(d3));
63      float step_v2_v3 = dist23 / (Math.abs(d2) + Math.abs(d3));

65      float scale_factor;
        Vertex3f start_vertex;
67      if((d1 * d2) < 0)
        {
69        GL11.glBegin(GL11.GL_LINES);
```

```
71        scale_factor = Math.abs(d1) * step_v1_v2;
          start_vertex = v1.add(v12.mult(scale_factor));
73        GL11.glVertex3f(start_vertex.x, start_vertex.y, start_vertex.z);

75        if((d1 * d3) < 0)
          {
77          scale_factor = Math.abs(d1) * step_v1_v3;
            start_vertex = v1.add(v13.mult(scale_factor));
79          GL11.glVertex3f(start_vertex.x, start_vertex.y, start_vertex.z);
          }
81        else
          {
83          scale_factor = Math.abs(d2) * step_v2_v3;
            start_vertex = v2.add(v23.mult(scale_factor));
85          GL11.glVertex3f(start_vertex.x, start_vertex.y, start_vertex.z);
          }
87
          GL11.glEnd();
89      }
        else
91      {
          if((d1 * d3) < 0)
93        {
            GL11.glBegin(GL11.GL_LINES);
95          // start vertex on edge v0_v4
            scale_factor = Math.abs(d1) * step_v1_v3;
97          start_vertex = v1.add(v13.mult(scale_factor));
            GL11.glVertex3f(start_vertex.x, start_vertex.y, start_vertex.z);
99
            // end vertex on edge v2_v4
101         scale_factor = Math.abs(d2) * step_v2_v3;
            start_vertex = v2.add(v23.mult(scale_factor));
103         GL11.glVertex3f(start_vertex.x, start_vertex.y, start_vertex.z);
            GL11.glEnd();
105       }
        }
107   }
    }
```

Listing A.3: Concatenation of continuous silhouette edges

```
  public void buildStrokes()
```

```
{
  Iterator<SilhouetteEdge> se_it = sil_edges.iterator();
  while(se_it.hasNext())
  {
    SilhouetteEdge edge = se_it.next();
    if(!edge.isVisited())
    {
      Stroke s = new Stroke();
      int prev = edge.getStart_he().getTwin().getSilEdge().getId();
      int next = edge.getEnd_he().getTwin().getSilEdge().getId();
      recFwAdd(edge, s, prev);
      strokes.add(s);
    }
  }
}

/*
 * recursive directional helper function for concatenating silhouette edges
     .
 */
public void recFwAdd(SilhouetteEdge e, Stroke s, int prev)
{
  if(e.isVisited())
  {
    return;
  }
  else
  {
    e.setVisited(true);
    s.addEdge(e);
    if(e.getStart_he().getTwin().getSilEdge() != null & !e.getStart_he().
        getTwin().getSilEdge().isVisited() & e.getStart_he().getTwin().
        getSilEdge().getId() != prev)
    {
      recFwAdd(e.getStart_he().getTwin().getSilEdge(), s, e.getId());
    }
    else if(e.getEnd_he().getTwin().getSilEdge() != null & !e.getEnd_he().
        getTwin().getSilEdge().isVisited() & e.getEnd_he().getTwin().
        getSilEdge().getId() != prev)
    {
      recFwAdd(e.getEnd_he().getTwin().getSilEdge(), s, e.getId());
    }
  }
```

```
40 }
```

Listing A.4: Edge direction sorting

```
  /*
2  * Iterates over all strokes and for each strokes silhouette edges and
       corrects the direction
   * (switches start/end vertex and HE)
4  */
  public void sortStrokes()
6 {
    Iterator<Stroke> st_it = strokes.iterator();
8   while(st_it.hasNext())
    {
10    Stroke s = st_it.next();
      Iterator<SilhouetteEdge> se_it = s.getEdges().iterator();
12    SilhouetteEdge e = se_it.next();
      int id = e.getId();
14    while(se_it.hasNext())
      {
16      e = se_it.next();
        if(e.getEnd_he().getTwin().getSilEdge() != null & e.getEnd_he().
            getTwin().getSilEdge().getId() == id)
18      {
          e.switchDir();
20      }
        id = e.getId();
22    }
    }
24 }
```

Listing A.5: Quad and texture coordinates generation

```
  public void drawPolygonalStrokes()
2 {
    float s_l = getLongestStroke();
4   Vertex3f cam = new Vertex3f(-cam_r * cam_lx, -cam_r * cam_ly, -cam_r *
        cam_lz);
    cam.normalize();
6
    Iterator<Stroke> st_it = strokes.iterator();
8   while(st_it.hasNext())
```

```
    {
10    Stroke s = st_it.next();
      float s_lenght = s_l;
12    float c_length_tx = 0;
      float step = texture_size / s_lenght;
14
      Iterator<SilhouetteEdge> se_it = s.getEdges().iterator();
16    while(se_it.hasNext())
      {
18      SilhouetteEdge e = se_it.next();
        Vertex3f v1 = e.getStart_vertex();
20      Vertex3f v2 = e.getEnd_vertex();
        Vertex3f v12 = new Vertex3f(v2.x, v2.y, v2.z);
22      v12.sub(v1); //he1
        v12.normalize();
24
        Vertex3f dir = Vertex3f.cross(v12, cam);
26      dir.normalize();
        dir.divideVertex(stroke_size);
28
        float tx_start = c_length_tx;
30      float tx_end   = c_length_tx + e.getLength() * step;
        c_length_tx = tx_end;
32
        updateMatrices();
34      GLU.gluProject(v1.x, v1.y, v1.z, modelView, projView, viewport,
            projArray);
        int v1x = (int) Math.round(projArray.get());
36      int v1y = (int) Math.round(projArray.get());
        float depth1 = projArray.get();
38      projArray.clear();

40      GLU.gluProject(v2.x, v2.y, v2.z, modelView, projView, viewport,
            projArray);
        int v2x = (int) Math.round(projArray.get());
42      int v2y = (int) Math.round(projArray.get());
        float depth2 = projArray.get();
44      projArray.clear();

46      GLU.gluProject(v1.x + dir.x, v1.y + dir.y, v1.z + dir.z, modelView,
            projView, viewport, projArray);
        int x1 = (int) Math.round(projArray.get());
48      int y1 = (int) Math.round(projArray.get());
```

```
        projArray.clear();
50      GLU.gluProject(v1.x - dir.x, v1.y - dir.y, v1.z - dir.z, modelView,
            projView, viewport, projArray);
        int x2 = (int) Math.round(projArray.get());
52      int y2 = (int) Math.round(projArray.get());
        projArray.clear();
54      GLU.gluProject(v2.x - dir.x, v2.y - dir.y, v2.z - dir.z, modelView,
            projView, viewport, projArray);
        int x3 = (int) Math.round(projArray.get());
56      int y3 = (int) Math.round(projArray.get());
        projArray.clear();
58      GLU.gluProject(v2.x + dir.x, v2.y + dir.y, v2.z + dir.z, modelView,
            projView, viewport, projArray);
        int x4 = (int) Math.round(projArray.get());
60      int y4 = (int) Math.round(projArray.get());
        projArray.clear();
62
        [...]
64
        float tx1 = (float)x1/viewportWidth;
66      float ty1 = (float)y1/viewportHeight;

68      float tx2 = (float)x2/viewportWidth;
        float ty2 = (float)y2/viewportHeight;
70
        float tx3 = (float)x3/viewportWidth;
72      float ty3 = (float)y3/viewportHeight;

74      float tx4 = (float)x4/viewportWidth;
        float ty4 = (float)y4/viewportHeight;
76


78      GL11.glBegin(GL11.GL_QUADS);
        GL11.glColor4f(ivec1x, ivec1y, depth1, 1.0f);
80      GL11.glTexCoord4f(tx1, ty1, 0.0f, tx_start);
        GL11.glVertex3f(v1.x + dir.x, v1.y + dir.y, v1.z + dir.z); //TR
82      GL11.glColor4f(ivec2x, ivec2y, depth1, 1.0f);
        GL11.glTexCoord4f(tx2, ty2, 1.0f, tx_start);
84      GL11.glVertex3f(v1.x - dir.x, v1.y - dir.y, v1.z - dir.z); //TL

86      GL11.glColor4f(ivec3x, ivec3y, depth2, 1.0f);
        GL11.glTexCoord4f(tx3, ty3, 1.0f, tx_end);
88      GL11.glVertex3f(v2.x - dir.x, v2.y - dir.y, v2.z - dir.z); //BR
```

```
        GL11.glColor4f(ivec4x, ivec4y, depth2, 1.0f);
90      GL11.glTexCoord4f(tx4, ty4, 0.0f, tx_end);
        GL11.glVertex3f(v2.x + dir.x, v2.y + dir.y, v2.z + dir.z); //BL
92      GL11.glEnd();
      }
94  }
}
```

Listing A.6: Mesh Initialisation for OpenGL ES

```
1 public void initMesh(Mesh mesh)
{
3   float coords[] = new float[mesh.getNumFaces() * 3 * 3];
    int count = 0;
5   Iterator<Face> iter = mesh.getFaceIter();
    while(iter.hasNext())
7   {
      Face f = iter.next();
9     FaceVertexIter fv_it = new FaceVertexIter(f);
      while(fv_it.hasNext())
11    {
        Vertex3f v = fv_it.next();
13      coords[count] = v.x;
        coords[count] = v.y;
15      coords[count] = v.z;
        count+=3;
17    }
    }
19  ByteBuffer vbb = ByteBuffer.allocateDirect(coords.length * 4);
    vbb.order(ByteOrder.nativeOrder());
21  meshVB = vbb.asFloatBuffer();
    meshVB.put(coords);
23  meshVB.position(0);
}
```

Listing A.7: Silhouette quads: generation of texture coordinates and lookup vectors

```
//Vertex coordinates
2 silQuadVBPos = ByteBuffer.allocateDirect(vCoords.length * mBytesPerFloat).
    order(ByteOrder.nativeOrder()).asFloatBuffer();
silQuadVBPos.put(vCoords).position(0);
4
```

```
//Color attribute with encoded lookup vector
silQuadVBCol = ByteBuffer.allocateDirect(cCoords.length * mBytesPerFloat).
    order(ByteOrder.nativeOrder()).asFloatBuffer();
silQuadVBCol.put(cCoords).position(0);

//Texture coordinates (for texture mapping and lookup)
silQuadVBTexc = ByteBuffer.allocateDirect(tCoords.length * mBytesPerFloat).
    order(ByteOrder.nativeOrder()).asFloatBuffer();
silQuadVBTexc.put(tCoords).position(0);


// draw
glVertexAttribPointer(silPositionHandle, 3, GL_FLOAT, false, 0,
    silQuadVBPos);
glEnableVertexAttribArray(silPositionHandle);

glVertexAttribPointer(silColorHandle, 4, GL_FLOAT, false, 0, silQuadVBCol);
glEnableVertexAttribArray(silColorHandle);

glVertexAttribPointer(silTexCoordHandle, 4, GL_FLOAT, false, 0,
    silQuadVBTexc);
glEnableVertexAttribArray(silTexCoordHandle);

glDrawArrays(GL_TRIANGLES, 0, vCoords.length/3);
```

Bibliography

[App67] Arthur Appel. The Notion of Quantitative Invisibility and the Machine Rendering of Solids. In Solomon Rosenthal, editor, *Proceedings of the 22nd ACM National Conference 1967 (Washington, D.C., USA)*, pages 387–393. Thompson Books, 1967.

[BE99] Fabien Benichou and Gershon Elber. Output Sensitive Extraction of Silhouettes from Polygonal Geometry. In Bob Werner, editor, *Proceedings of the 7th Pacific Conference on Computer Graphics and Applications, Pacific Graphics 1999 (PG 1999, October 5–7, 1999, Seoul, Korea)*, pages 60–69, IEEEAdr, 1999. IEEEPub.

[BS00] John W. Buchanan and Mario Costa Sousa. The Edge Buffer: A Data Structure for Easy Silhouette Rendering. In Jean-Daniel Fekete and David Salesin, editors, *NPAR2000*, pages 39–42, New York, 2000. ACM Press.

[CGL+08] Forrester Cole, Aleksey Golovinskiy, Alex Limpaecher, Heather Stoddart Barros, Adam Finkelstein, Thomas Funkhouser, and Szymon Rusinkiewicz. Where do people draw lines? *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 27(3), August 2008.

[Fin05] Adam Finkelstein. Stylization of line drawings. SIGGRAPH 2005 Course Notes, 2005.

[GSG+99] Bruce Gooch, Peter-Pike J. Sloan, Amy A. Gooch, Peter Shirley, and Richard Riesenfeld. Interactive Technical Illustration. In Jarek Rossignac, Jessica Hodgins, and James D. Foley, editors, *Proceedings of the 1999 Symposium on Interactive 3D Graphics*, pages 31–38, New York, 1999. ACM Press.

[Her99] Aaron Hertzmann. Introduction to 3d non-photorealistic rendering: Silhouettes and outlines. *SIGGRAPH 99 Course Notes (Course on Non-Photorealistic Rendering)*, 1999.

[HZ00] Aaron Hertzmann and Denis Zorin. Illustrating Smooth Surfaces. In Judith R. Brown and Kurt Akeley, editors, *SIGGRAPH2000*, CGPACS, pages 517–526, New York, 2000. ACM SIGGRAPH.

[IB06] Tobias Isenberg and Angela Brennecke. G-strokes: A concept for simplifying line stylization. *Computers & Graphics*, 30(5):754 – 766, 2006.

[IHS02] Tobias Isenberg, Nick Halper, and Thomas Strothotte. Stylizing Silhouettes at Interactive Rates: From Silhouette Edges to Silhouette Strokes. *EUROGRAPHICS2002*, 21(3):249–258, September 2002.

[KDMF03] Robert D. Kalnins, Philip L. Davidson, Lee Markosian, and Adam Finkelstein. Coherent stylized silhouettes. *ACM Trans. Graph.*, 22(3):856–861, July 2003.

[KMN+99] Michael A. Kowalski, Lee Markosian, J. D. Northrup, Lubomir Bourdev, Ronen Barzel, Loring S. Holden, and John F. Hughes. Art-Based Rendering of Fur, Grass, and Trees. In *SIGGRAPH99*, CGPACS, pages 433–438, New York, 1999. ACM Press/ACM SIGGRAPH.

[Lov02] Jörn Loviscach. Rendering artistic line drawings using off-the-shelf 3-d software. In Isabel Navazo Alvaro and Philipp Slusallek, editors, *EUROGRAPHICS2002Abstr*, pages 125–130, Oxford, UK, 2002. Eurographics Association, Blackwell Publishers.

[MBC02] Jason L. Mitchell, Chris Brennan, and Drew Card. Real-time image-space outlining for non-photorealistic rendering. In *ACM SIGGRAPH 2002 conference abstracts and applications*, SIGGRAPH '02, pages 239–239, New York, NY, USA, 2002. ACM.

[MKT+97] Lee Markosian, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein, and John F. Hughes. Real-Time Nonphotorealistic Rendering. In Turner Whitted, editor, *SIGGRAPH97*, CGPACS, pages 415–420, New York, 1997. ACM Press/ACM SIGGRAPH.

[NM00] J. D. Northrup and Lee Markosian. Artistic Silhouettes: A Hybrid Approach. In Jean-Daniel Fekete and David Salesin, editors, *NPAR2000*, pages 31–37, New York, 2000. ACM Press.

[PF05] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005.

[Rus89] Pramod Rustagi. Silhouette Line Display From Shaded Models. *IRIS Universe*, 1989(9):42–44, Fall 1989.

[RvE92] Jarek R. Rossignac and Maarten van Emmerik. Hidden Contours on a Frame-Buffer. In *Proceedings of the 7th Eurographics Workshop on Computer Graphics Hardware*, pages 188–204, 1992.

[SGG+00a] Pedro V. Sander, Xianfeng Gi, Steven J. Gortler, Huges Hoppe, and John Snyder. Silhouette Clipping. In Kurt Akeley, editor, *SIGGRAPH2000*, CGPACS, pages 327–334, New York, 2000. ACM SIGGRAPH, ACM Press.

[SGG+00b] Pedro V. Sander, Xianfeng Gu, Steven J. Gortler, Hugues Hoppe, and John Snyder. Silhouette clipping. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, pages 327–334, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[ST90] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. *SIGGRAPH Comput. Graph.*, 24(4):197–206, September 1990.

[WPLS05] Daniel Wagner, Thomas Pintaric, Florian Ledermann, and Dieter Schmalstieg. Towards massively multi-user augmented reality on handheld devices. In *Proceedings of the Third international conference on Pervasive Computing*, PERVASIVE'05, pages 208–219, Berlin, Heidelberg, 2005. Springer-Verlag.