# Rendering Generalized Cylinders using the A-Buffer

by

Ivan Neulander

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Rendering Generalized Cylinders using the A-Buffer

Ivan Neulander

Master of Science

Graduate Department of Computer Science

University of Toronto

1997

## Abstract

A variety of objects modelled in computer graphics can be efficiently approximated with generalized cylinders, particularly when they are viewed at a relatively small scale. In this thesis we present a unique way of rendering generalized cylinders using polygon-based projective rendering: a rendering meta-primitive called the *paintstroke*. Paintstrokes allow for the concise modelling and efficient dynamic tessellation of generalized cylinders, making direct use of their screen-space projections so as to minimize the number of polygons required to construct their images. The resulting savings in vertex transformations, rasterization overhead, and edge antialiasing more than repay the cost of the tessellation. Used in conjunction with our A-Buffer polygon renderer, paintstrokes achieve a good balance of speed and image quality when drawn at small to medium scales, generally surpassing other methods for rendering generalized cylinders.

# Acknowledgments

A number of people have contributed to the research and the writing that went into this thesis. I wish to thank my supervisor, Michiel van de Panne, for suggesting an initial direction for my research, for giving me the guidance and motivation to complete it, and for tirelessly reviewing (as evidenced by the dreaded "sea of green") and improving upon my thesis drafts. Thanks are also due to my second reader, James Stewart, for the many subtle observations and insightful suggestions from which the thesis has benefited.

The rest of my colleagues at DGP deserve credit as well, particularly Fabrice Neyret, who has always eagerly and deftly answered my spur-of-the-moment graphics-related questions, and has occasionally even participated in debugging my code! I am also grateful to my friend and fellow graduand, Nick Torkos, for finding a practical application for my rendering software in his impressive quadruped animations.

Finally, I am forever indebted to my parents for their love, support, and encouragement, without which this thesis would never have materialized.

*I gratefully acknowledge the NSERC scholarship that has funded my work toward this thesis.*

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

The ability to make pictures with a computer has entertained, enlightened, and challenged us for decades. From its humble beginnings in the 1950's, the study of computer graphics has spawned a multi-billion dollar industry and remains one of the most rapidly expanding areas of computer science. One of the many reasons behind its growing popularity is the power it provides to rapidly and precisely translate a complex specification into a realistic image. This task can be divided into two basic phases: building the specification, or *modelling*, and generating the image, or *rendering*.

The modelling and rendering processes are linked by the concept of a *rendering primitive*, which is a member of the limited class of objects that can be directly rendered by the computer hardware. All other types of objects need to be composed of these basic building blocks, into which they are ultimately decomposed before they (or rather, their constituent primitives) are to be rendered. Examples of 3-D rendering primitives in a typical graphics system are points, lines, and triangles.

While a scene description could be fashioned purely out of rendering primitives, its construction would typically be an onerous task for the modeller. Many types of objects, particularly those with curved surfaces, can only adequately be approximated using a large number of finely distributed primitives, making them painstakingly slow and error-prone for a human to construct. A better approach is to invoke a computer program to translate the object's representation expressed in some more convenient form (such as a parametric surface defined using control points) into rendering primitives. This translation can either be done during the modelling phase, whereby the primitives output by the translator are directly incorporated into the model that is passed to the renderer, or it can be done as a preface to the rendering phase, in which case the model contains the compact description, which is automatically decomposed into primitives at rendering time. The latter alternative has the advantage of being user-transparent: the modeller treats the compact representation of the object as though it were a rendering primitive; its translation into actual primitives happens behind the scenes. Because this type of object is not a true primitive, but it behaves like one, we call it a *meta-primitive*.[1]

In this thesis we will develop and explore a meta-primitive called the *paintstroke*, designed for rendering generalized cylinders using a polygon-based projective rendering system. We define a generalized cylinder as the surface produced by extruding a circle along a path through space, allowing the circle's radius to vary along the path. During the extrusion, the circle's orientation is such that the plane it spans is always orthogonal to the path. We will show that, in addition to providing a convenient and succinct representation for generalized cylinders, paintstrokes can offer significant advantages over comparable methods in rendering these surfaces.

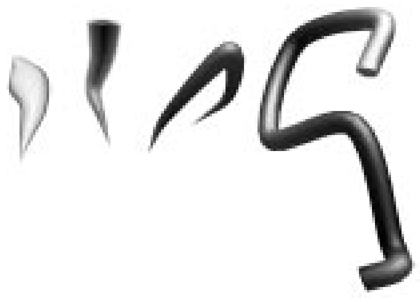


Figure 1.1: A few sample paintstrokes rendered using our algorithm.

[1]Having distinguished between the two, we will frequently use the term 'primitive' to denote a 'meta-primitive', allowing the context to indicate which meaning is intended.

## 1.1 The Purpose of the Paintstroke Primitive

### 1.1.1 Motivation

The work presented in this thesis is motivated largely by the observation that (1) a significant portion of objects we see around us are thin and roughly tubular in shape, especially when viewing natural phenomena; and (2) the current methods for rendering high quality images of such objects at small to medium scales are not as efficient as they could be. Our goal in designing the paintstroke was to furnish the user with an effective means of rendering these objects at a variety of scales, providing a good balance of rendering speed and image quality for tubes between one and ten pixels in screen-projected thickness, which is where other methods generally fail to do so.

### 1.1.2 Applications

Because they are limited to modelling generalized cylinders, paintstrokes are not suitable for designing arbitrary objects. This specialization, however, allows for highly optimized rendering that consumes less time, memory, and bandwidth than more general methods. Thus, paintstrokes can serve as inexpensive building blocks for highly complex geometry. Combined in large numbers, they can be used to efficiently render a variety of detailed natural phenomena such as fur, hair, branches, twigs, and pine needles. Simpler structures like wires, hoses, and pipes are equally suitable.

By taking advantage of their view-dependent tessellation scheme, paintstrokes can very inexpensively approximate volumetric opacity and Fresnel effects, making them useful in rendering water streams, icicles, and wisps of smoke, to name a few examples. This has traditionally been difficult to accomplish with other projective-rendering methods, necessitating the expensive solution of ray-tracing. Finally, a global shading function can be used to approximate self-shadowing for globally convex objects uniformly layered with paintstrokes, offering a very inexpensive and reasonably effective alternative to more sophisticated methods like shadow-mapping [Wil78, RSC87].

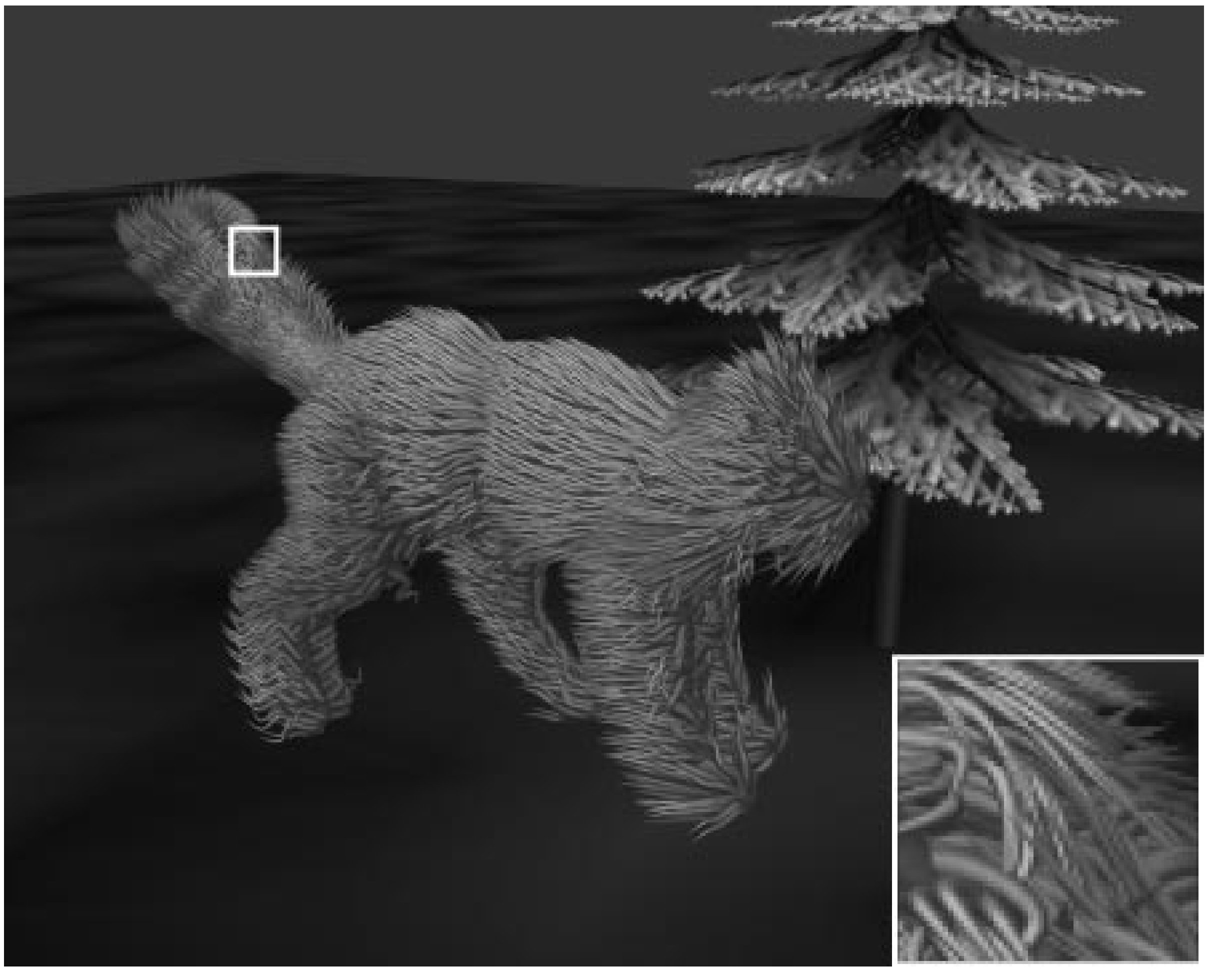


Figure 1.2: An example of high geometric detail that is captured with paintstrokes.

## 1.2 Features of Paintstrokes

As mentioned, the paintstroke is a dynamically tessellated polygon-based meta-primitive. There are two basic phases to rendering it: (1) tessellate it into polygons, and (2) render the polygons. Chapter 3 is devoted to the first phase, and Chapter 4 to the second. Although polygon-based projective rendering was by no means the only choice for the paintstroke's infrastructure, it is well-suited to the scope and applications of the primitive. As an alternative, ray-tracing would be a considerably slower way to render polygons at the scale for which paintstrokes are intended.[2] Abandoning polygons for implicit or

[2]At much smaller scales, it would become viable, since it would eliminate the inefficiency of touching pixels multiple times (when rendering opaque objects), which becomes the major drawback of projective rendering.

parametric surfaces would, on the other hand, involve working with a far more complex (and probably more computationally expensive) representation than that of our approach.

### 1.2.1 Paintstroke Tessellation

When rendering a curved surface using polygon primitives, at some point the surface must be tessellated into polygons. Because this process consumes time, it is often done only once, storing the tessellated polygons in place of the curved surface they represent. Thus, the tessellation contributes nothing to the rendering time, having been performed as a pre-processing step. Such an approach is known as *static tessellation*.

Paintstrokes are not rendered in this way. We maintain a compact descriptive model of the generalized cylinder, as will be explained in Chapter 3. The tessellation occurs every time the model is rendered, following its transformation into eye-space. Although this *dynamic tessellation* contributes to the overall rendering cost, it capitalizes on important symmetries and view-invariances of the generalized cylinder, which permit its screen projection to be accurately tiled with only a small number of relatively large polygons. The resulting savings in vertex transformations, rasterization overhead, and A-Buffer fragment blending more than compensate for the tessellation cost. Furthermore, by continually adjusting the granularity of their tessellation, paintstrokes smoothly adapt their level of detail to the scale at which they are rendered.

### 1.2.2 Polygon Rendering

Since paintstrokes are ultimately rendered as polygons, a fast, high-quality rendering engine for polygons is essential to our approach. Due to the small scale at which paintstrokes may be drawn, the problem of aliasing needs to be addressed, both along the edges and near specular highlights. Our solution was to implement an A-Buffer algorithm [Car84] with adaptive Phong supersampling. In addition to fast and accurate edge antialiasing, the A-Buffer also allows for precise transparency blending and reasonable antialiasing of interpenetrating (or touching) surfaces, both of which are difficult to achieve with standard Z-Buffer implementations. The adaptive Phong shader dynamically varies the number of Phong samples per pixel, depending on a number of parameters, including surface specularity and per-pixel normal variation. Both of these techniques are detailed in Chapter 4.

While our polygon renderer has several features specifically geared toward paintstrokes, it is a nevertheless general-purpose rendering engine. Among other things, it allows the modeller to arbitrarily combine paintstrokes with standard polygons, set camera orientations and change lighting parameters. This has enabled us to generate many useful images and animations that go beyond simply testing the rendering of paintstrokes.

## 1.3 Scope

Because paintstrokes capture the full geometry of a scene, they are ideally suited to a sufficiently large scale that allows all of their geometric detail to be seen. Although intended for rendering tubes one to ten pixels in thickness, they can accommodate a much wider variety of scales. Thicknesses ranging from a fraction of a pixel to hundreds are possible. Despite this flexibility, their usefulness at these extremes is limited. At very small scales, aliasing problems begin to exceed the paintstroke's antialiasing capabilities and image quality suffers. In addition, the per-pixel rendering speed diminishes due to high oversampling of the shading function. At this scale, other techniques that simplify or altogether eliminate the underlying geometry (such as texture-mapping or volumetric textures) are reasonable alternatives.

At the same time, there is a limit to using paintstrokes to model objects having large screen coverage. Due to the simplicity of their shape and colour attributes, they are generally not suitable for close-up images, unless the object being modelled happens to be a generalized cylinder with a simple lengthwise colour variation. Thus, the perfect scale for paintstrokes is an intermediate one, where the full geometry of a scene is visible, but the discrepancy of each paintstroke's appearance with that of the object it represents is not too conspicuous.

As an example, consider using paintstrokes to model tree branches in a forest. If viewed from an airplane flying high overhead, this scene would be better rendered with another technique, such as texture mapping. The geometry of the branches appears at such a fine scale that it can be adequately represented as just a colour variation across

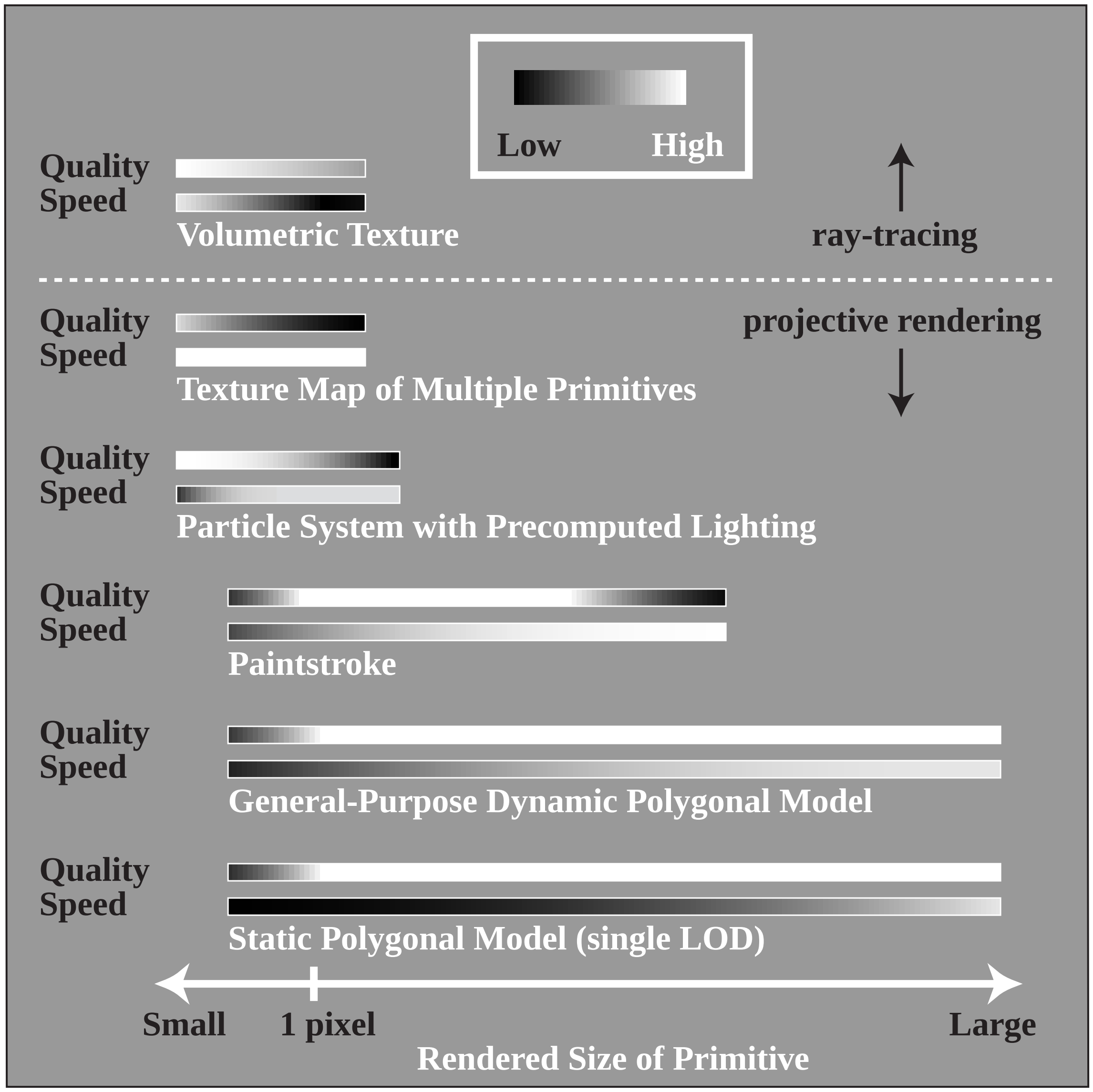


Figure 1.3: Approximate speed/quality characteristics of various rendering methods applied to tubular objects.

a large scale model. On the other hand, if the scene is viewed by a person passing through the forest, the paintstroke model will be appropriate for the majority of the visible trees. Only the ones close to the viewer would need to be rendered with a more general technique.

Figure 1.3 gives some insight into the range of scales at which paintstrokes and comparable rendering methods are most useful. It also depicts the relationship between these scales and the corresponding image quality and rendering speed. Although sparse in details, it gives a general sense of each method's scope and allows for some basic speed/quality comparisons between them.

Because no single chart could summarize all rendering scenarios, we have made a number of assumptions in formulating ours. The measure of image quality shown has been normalized to the highest quality achievable using standard projective rendering. Features such as radiosity, glossy reflection, and refraction are not considered. Rendering speed is expressed in per-pixel rather than per-primitive terms. This explains why the speed of several methods shown drops off at smaller scales.

The two methods most similar in scope to paintstrokes are general-purpose dynamic polygonal models (of which paintstrokes are a *specific* instance) and particle systems. Volumetric textures become reasonable alternatives at very small scales. A detailed discussion of these methods will be the main thrust of Chapter 2. Finally, after we have examined the paintstroke primitive in depth, we will compare it with these methods. That will be the topic of Chapter 5.

# Chapter 2

# Alternative Methods for Rendering Tubes

In this chapter we examine some possible alternatives to paintstrokes, most of which were introduced in Chapter 1. We do not limit our modelling domain strictly to generalized cylinders, but consider all objects that are reasonably approximated by generalized cylinders at some scale. This considerably broadens the scope (and usefulness) of our model.

## 2.1 Overview of Polygon-Based Models

### 2.1.1 Static vs. Dynamic Models

The simplest type of polygonal model is one that is *static*, consisting of a fixed set of polygons. These may be directly specified by the modeller, generated by a tessellation routine applied to a non-polygonal (usually parametric or implicit) surface, or derived by modifying an existing polygonal surface. Any work needed to obtain these polygons is done during a pre-processing phase, so that it does not consume rendering time. In contrast, *dynamic* models create or modify the polygonal representation of a model during rendering. Normally, this is done either by tessellating the model from a parametric or implicit surface, or by simplifying or refining an existing polygonal model into the one that is rendered.

### 2.1.2 Static Polygonal Models

The most obvious advantage of a static model, aside from its simplicity, is the processing time saved by *not* modifying its representation during rendering. Because the cost of polygonizing a static model is not included in the cost of rendering it, this type of model normally consists of a very efficient polygon mesh[1] constructed by slow but high-quality algorithms. Quite often, such algorithms will provide a great deal of flexibility in setting error tolerances that determine the allowable deviation of the output polygonization from the original surface. A good example is the work on simplification envelopes by Cohen *et al.* [CVM$^+$96]. Their approach computes a pair of implicit surfaces (the inner and outer envelopes) that define the allowable boundaries of the simplified polygonal mesh. Given a user-specified polygonal mesh as input, the output is a coarser mesh it that is sandwiched between the simplification envelopes, thereby satisfying the user-defined error tolerances while preserving global topology. Many other surface simplification algorithms that achieve similar results are catalogued by Heckbert and Garland in [HG94].

Although there are cases where static tessellation is the best choice, greater efficiency and flexibility can usually be attained with dynamic models. This is because of several important limitations, to be discussed below. In Chapter 5, we will argue that for rendering generalized cylinders, a dynamic tessellation scheme like the one used by paintstrokes is decidedly superior to static tessellation.

One limitation of static tessellation is that it cannot handle any deformation of the model during rendering, since this would involve modifying the arrangement and shapes of the underlying polygons—a process that is normally achieved through dynamic re-tessellation. Since many computer-generated animations portray a large number of deformable objects, animators seldom rely solely on static models; they use dynamic tessellation in much of their rendering.

Another disadvantage of the static model is that its way of describing surface—as a set of polygons—often fails to be concise. For example, representations that specify a parametric surface using a mesh of control points can usually provide a more precise

[1]By this we mean that the mesh provides a very good approximation to the underlying surface using relatively few polygons.

description of the model using much less data. Such representations can only be used with dynamically tessellated models, and generally have far more modest storage and bandwidth requirements than do static models.

Finally, a static model does not lend itself to effective level-of-detail adjustment when rendered in an animation. Because such a model is always drawn at a single level of detail, the speed (or conversely, quality) at which it is rendered suffers when it is viewed at a variety of screen sizes, since its level of detail can only be ideally suited to a single scale. This deficiency is frequently redressed by pre-computing multiple static models of an object at various levels of detail, and selecting the appropriate one based on an estimate of the model's screen-projected size. Although this multiresolution approach is common [HG94, HG97] and works quite well for still images, it poses a critical problem for animation: Transitions between levels of detail are discontinuous, often causing severe popping artifacts. These artifacts can be mitigated by compositing the images at the higher and the lower levels of detail during a transition, but this solution is expensive and not entirely effectual—not only does it entail rendering the object twice, but it also requires the rendered images to be alpha-blended.[2]This is clearly impractical in situations where a large number of objects are continually shifting levels of detail, as when travelling through a complex landscape populated with small objects (e.g. blades of grass) whose distance from the viewer is in constant flux.

Another issue is the tradeoff in choosing the number of levels of detail at which an object is to be represented. Using too few will result in a poor match between the ideal number[3] of polygons for a given scale and the number actually used in the model. Hence, much of the time, either the rendering will be slower than it could be, or the image quality will be substandard. Moreover, differences between successive levels of detail will be large, exacerbating the popping during transitions. On the other hand, using too many levels of detail can cause frequent switching between levels, which, as described

[2]Note that there *are* efficient ways to eliminate this popping artifact by gradually altering the polygonal structure of a model between successive levels of detail. However, because this modification occurs during rendering, by our definition, it does not apply to static models; we shall discuss this technique in §2.1.4, under the rubric of dynamic models.

[3]Such an ideal number would depend on the type of tessellation used, the way of measuring how accurately the tessellated model approximates a desired object (which is often somewhat subjective), and an error tolerance.

above, is undesirable.[4] Moreover, maintaining a large number of models at various levels of detail further contributes to the excessive storage requirements of static tessellation.

Lastly, the number and arrangement of polygons within a static model cannot be adjusted to suit a particular viewing position. Some dynamic models, such as paintstrokes, exploit this view-dependency to achieve the same image quality using fewer polygons than a static model, even when the latter is tessellated at the optimum granularity for its scale. This result is borne out in Chapter 5, where we present a detailed comparison between paintstrokes and statically tessellated generalized cylinders.

### 2.1.3 Dynamic Tessellation

Because dynamic tessellation consumes rendering time, the tessellation speed can have a noticeable impact on the overall cost of rendering a scene. Thus, the time devoted to re-tessellating objects must be balanced against any savings afforded by their revised polygonizations.

Most dynamic tessellation algorithms are for general-purpose parametric surfaces such as NURBS and Bézier patches, which are applicable to a wide variety of models. In addition to these general algorithms, we will examine a technique by Jim Blinn [Bli89] that is specialized for tessellating circular tubes, as is the paintstroke.

#### Tessellation Methods for General Parametric Surfaces

Nonuniform rational B-splines (NURBS) are a popular way of representing arbitrary continuous curves using a set of control points and a knot vector. A NURBS surface is a two-dimensional extension of this curve, defined as the Cartesian product of one NURBS curve with another, and specified using a mesh of control points spanning the extent of the desired surface. NURBS curves have two main advantages over their non-rational counterparts such as Bézier and Hermite curves. First, they are able to exactly represent quadric curves (e.g. ellipses and hyperbolas), which the latter can only approximate. And second, they are invariant under the perspective transformation, whereas the latter are not. Specifically, if the control points of a NURBS are perspective-transformed, the

[4]Given that the model is stored at sufficiently many levels of detail, the popping will eventually cease to be a problem. It will, however, be replaced by prodigious demands on storage and bandwidth.

NURBS constructed from the transformed points is the true perspective projection of the original curve. These advantages translate directly to NURBS surfaces, which can be used for exact representations of spheres, cylinders, and (of interest to us) generalized cylinders. Likewise, the NURBS surface is invariant under a perspective transformation. A third advantage is the existence of efficient trimming algorithms which allow the modeller to trim a NURBS surface with a parametric curve.

Because NURBS are defined as a ratio of B-splines, evaluations tend to be expensive, requiring division operations [Sil90]. This is a major disadvantage of NURBS, one that is not shared by non-rational splines. Although a number of optimizations have been proposed to reduce the cost of evaluating and tessellating NURBS surfaces [SC88, Sil90, AES94], non-rational spline surfaces are still cheaper to tessellate in most cases. Unless a highly accurate representations of elliptical or hyperbolic solids is needed (which would require a large number of control points to be adequately approximated with non-rational alternatives), or the invariance under the perspective transform is particularly valuable, it is generally more efficient to use non-rational parametric surfaces.

A wide variety of non-rational parametric surfaces are used in modelling. They are based on families of B-Splines, Bézier curves, Hermites, and others. Although each of these classes of curves has unique modelling characteristics, they are all equivalent from a rendering standpoint. That is to say that a curve belonging to any one of them can be expressed in terms of any other. For example, a B-spline can be expressed as a Bézier curve with a different set of control points. This equivalency allows these curves, and analogously, the surfaces based on them, to be rendered using a single algorithm, regardless of what class of spline was used to model them.

Among the more efficient evaluation algorithms are forward difference methods, which are best suited to (parametrically) evenly spaced evaluation points, and subdivision methods, which can be used to produce a progressively refined mesh of evaluation points. For the purpose of dynamic tessellation, the latter approach is the more useful. For Bézier curves, a particularly efficient subdivision method exists, based on the de Casteljau algorithm [Far88]. This involves averaging the positions of pairs of control points to create new points that approximate the spline. Given a Bézier segment of 4 control points, the subdivision generates two subsegments each with 4 control points (derived from the averaged points), one of them shared between the subsegments. This is illustrated in Figure 2.1. Either or both of the subsegments can be subdivided in the same way in order to refine the approximation.

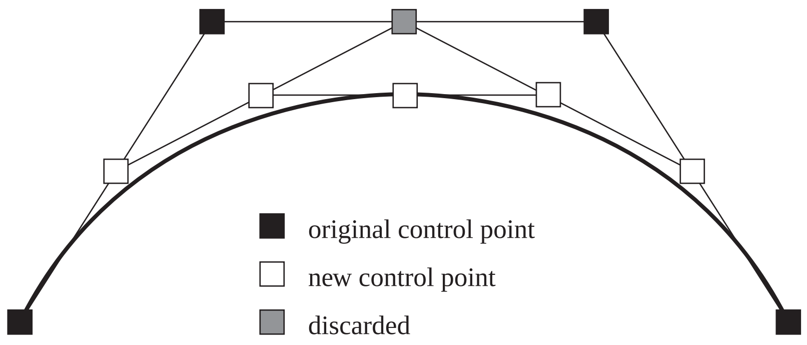


Figure 2.1: Efficient subdivision of a Bézier curve segment.

Because the only operations involved are additions and divisions by two, the subdivision is very inexpensive. Moreover, as suggested by Robert Beach [Bea91], this technique allows efficient, curvature-dependent, adaptive subdivision. A suitable test for subdivision is whether the four control points of a Bézier segment are approximately collinear. If not, the segment is divided in half as described above, and the process is recursively applied to each half. This algorithm is easily extended to two dimensions to produce a mesh of evaluation points that can serve as polygon vertices.

**Blinn's Optimal Tubes**

Jim Blinn [Bli89] describes a view-adaptive tessellation scheme for Gouraud-shaded cylinders that he calls *optimal tubes*, and an extension to handle constant-radius generalized cylinders. Blinn's approach is similar to our own, in that he applies a view-dependent tessellation to minimize the number of polygons required to produce a high-quality image. Because Gouraud shading does not interpolate normals, breadthwise subdivisions are used to capture shading information at visually significant points on a cylinder's surface. Assuming a single point-source light and incorporating the Lambertian (i.e. diffuse and ambient, but not specular) shading model, Blinn polygonizes the region of the cylinder visible to the viewer into rectangular strips, with boundaries along the cylinder's silhouette lines and along the lines where the illumination “significantly” changes. The latter occur at the two shadow lines and at the line of maximum (Lambertian) illumination, where the surface normal is coincident with the central light vector (i.e. halfway between the shadow lines). An example of this arrangement is shown in Figure 2.2.

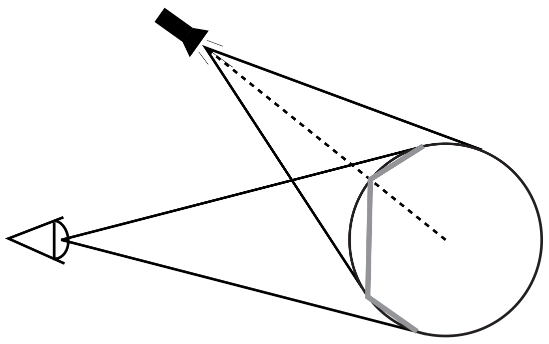


Figure 2.2: Sample polygonization of an optimal tube.

As an extension to this tessellation scheme, [Bli89] also presents a method of properly joining a pair of optimal tubes, eliminating the cracks that would ordinarily appear at the joint. This is needed when concatenating a series of tubes into a constant-radius generalized cylinder. However, no algorithm for subdividing the latter into the former is given.

### 2.1.4 Dynamic Surface Simplification and Refinement

Surface simplification and refinement differs from tessellation in that it is applied to existing polygonal meshes in order to produce new ones. Efficient simplification of detailed polygonal models is the primary goal of these techniques, which may be used in generating static as well as dynamic models. In the latter case it is also necessary to provide an inverse transformation that converts the simplified models back into the more complex ones. This transformation is called surface refinement. In conjunction with surface simplification, it allows a model's level of detail to continuously vary over a range of scales.

The notion of surface simplification and refinement can be extended beyond just the modification an object's polygonal mesh: It can involve switching between different types of rendering primitives. An example of such an approach is the tree-rendering algorithm by Weber and Penn [WP95]. Based on the desired level of detail, the algorithm dynamically selects points or lines to replace the more expensive polygons that are used to model the leaves and branches of a tree. At a sufficiently small scale, the use of these cheaper primitives can considerably expedite the rendering without compromising image quality.

Algorithms for surface simplification and refinement are usually quite general in nature, being geared toward different classes of topology (e.g. manifold, degenerate) but not to shapes as specific as generalized cylinders. Although their generality is for the most part an advantage, their speed and output quality can be surpassed in very specific cases by specialized dynamic tessellation algorithms, such as the paintstroke's.

### Progressive Meshes

Hugues Hoppe's work on view-dependent progressive meshes [Hop97] presents a fast surface simplification and refinement algorithm that makes local adjustments to the granularity of a model's polygonal (triangular) mesh based on its eye-space transformation. Parts of the model that lie outside of the view frustum or that face away from the viewer are simplified to very coarse levels, whereas regions near the silhouette are refined into a fine mesh. The screen-projected size of the model is another determinant of overall mesh granularity.

As in Hoppe's earlier paper [Hop96] on this subject, the simplifications and refinements to the polygonal mesh are implemented using two basic transformations: the vertex split, and the edge collapse. The former serves to refine a model by dividing a vertex into two, thereby forming a new edge and a new polygon. The latter removes an edge and replaces it with a single vertex, thereby deleting a polygon and simplifying the mesh. As shown in Figure 2.3, these operations work as inverses to one another, allowing an original model to be simplified to arbitrarily few polygons and then refined through the same number of steps back to its original complexity.

Popping artifacts that tend to arise in level-of-detail transitions are addressed using a technique called the *geomorph*, which performs vertex split and edge collapse operations gradually, moving a pair of vertices together or apart over a number of frames. This

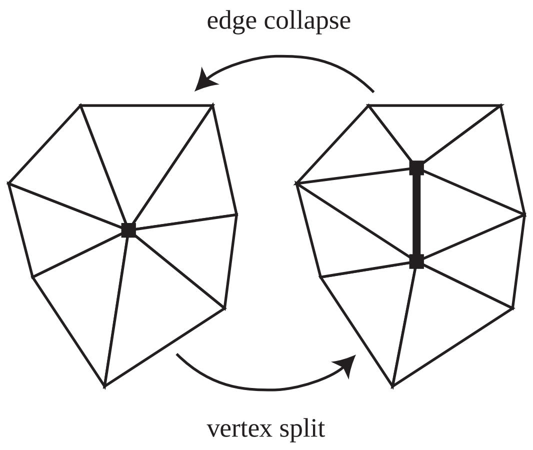


Figure 2.3: The vertex split and edge collapse operations.

creates a smooth visual transition at a much lower cost than the alpha-blending approach described in §2.1.2.

Much to this technique's advantage, it exploits the temporal coherence of animations by reusing (and therefore amortizing the cost of modifying) a mesh over many frames. According to the author, the cost of this dynamic re-polygonization accounts for less than 15% of the total rendering time on a graphics workstation.

## 2.2 Particle Systems

Particle systems are a departure from the canon of representing and rendering objects as surfaces. Many types of material have extremely complex surfaces, which would be difficult to model and slow to render using a surface-based representation. Fire, smoke, and clouds are common examples. Particle systems provide a more efficient way of working with these types of objects, as well as many others.

A particle system describes an image using a (typically) large number small simple objects called particles. In most implementations, including the seminal work by Bill Reeves [Ree83] and Reeves and Blau [RB85], these objects are tiny spheres or cubes, whose motion through space is described using implicit equations which additionally incorporate an element of randomness. Reeves simplifies the rendering of particles by treating them as point light sources. This eliminates the issue of visibility, since a pair of overlapping particles both contribute equally to the colour intensity over the overlapping region.[5] Because of their simplicity, particles permit fast rendering and motion blur. The latter is useful not only for animation, but also for rendering elongated objects, which can be represented as the (appropriately configured) motion-blurred trail left behind a particle.

To render a circular tube with particle systems, one would use motion-blurred spherical particles moving along a desired path. If the diameters of the particles can vary during the motion, arbitrary generalized cylinders can be constructed. In order to resolve visibility and apply proper shading, we would need a more complex particle model than the one used in [RB85], which handles both of these in an *ad hoc* fashion suited only to particular models. Specifically, we would require $z$-values and surface normals (or some approximations thereof), neither of which are used by Reeves and Blau.

Implementing motion blur requires integrating a particle's position over time, which is an expensive operation if performed explicitly. Fortunately, the simplicity of a spherical particle allows simple approximations to be used in lieu of a true position-time integral, expediting the rendering process considerably. Two such methods are brush extrusions, and the polyline approach.

### 2.2.1 Brush Extrusions

Brush extrusions, as described by Turner Whitted [Whi83], approximate the temporal integration of a particle—the brush tip—by rendering it at multiple discrete positions along a path, without using motion blur. Provided that the sampling frequency is sufficiently high, this concatenation of discrete tip images (samples) produces an apparently smooth and continuous trail. This method is well suited for real-time user interaction, using an input device such as a mouse or stylus. The user drags the brush tip along some path on the screen, as one would do with a paintbrush, and it leaves behind a nicely rendered antialiased trail (provided, of course, that the tip image is antialiased).

Although the brush extrusions described in [Whi83] are generated by a constant, precomputed brush tip image which moves only in the $x$-$y$ plane, this limitation still permits

[5]Of course, this is only an approximation. In reality, the closer particle should make the greater contribution, all other things being equal.

reasonably accurate rendering of constant-radius 3-D tubes under weak perspective and near-directional lighting, using a pre-rendered sphere as the tip. A more flexible alternative, which would allow variable-radius tubes under strong perspective and point-source lighting, would be to dynamically re-render the brush tip as it moves along the path. A compromise would be to pre-render the tip at various sizes and store them in memory. This would permit perspective effects and radius variation, albeit limited by the maximum stored radius of the tip. It would still, however, require a near-directional light source, since the sphere's shading would need to be constant.

### 2.2.2 Cone Spheres

In [Max90] Nelson Max presents a way to approximate generalized cylinders that is similar in spirit to Whitted's brush extrusions. Instead of rendering a sphere at multiple points along a path, it is rendered at fewer and more widely separated points. Each adjacent pair of rendered spheres are then joined using a tight-fitting truncated cone, as illustrated in Figure 2.4. Whereas the cones form the main body of the resulting solid, the purpose of the spheres is to provide smooth "elbow" joints connecting adjacent cones. This approach is argued to be more efficient than brush extrusions because fewer overlapping images need to be used to render a continuous-looking tube.

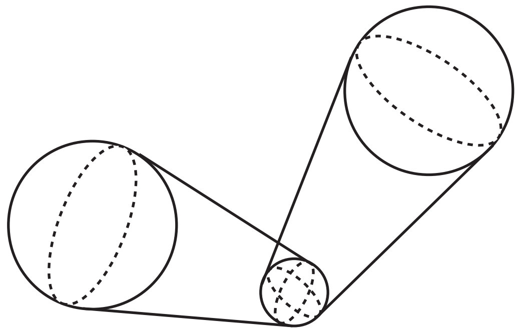


Figure 2.4: A pair of cone-spheres.

As we shall see in Chapter 3, cone-spheres are quite similar in nature to paintstrokes, although, unlike the latter, they are not explicitly polygonized. The screen-projected cone profiles (which are polygonal) and spheres are rendered using a scanline algorithm with an *ad hoc* antialiasing mask function. While Phong shading is applied to the cones, it is not used directly for the spheres. Instead, the shaded intensities of the two extended cones that enclose a sphere are blended to derive its shading. This creates reasonably smooth-looking highlights over a set of cone spheres approximating a curved tube.

### 2.2.3 Polylines with Precomputed Shading

The polyline approach is a common method of rendering very thin constant-radius tubes: The tube's shape is approximated using a set of line segments, whose widths correspond to the screen-projected thickness of the tube. Usually, this method is applied to circular extrusions (i.e. constant radius generalized cylinders), although it could in principle be used for variable-radius tubes as well, by modulating the thickness of the line segments used—as long as the maximum thickness remains small.

The shading of a polyline model can be performed efficiently using a clever approximation based on the work of Kajiya and Kay [KK89]. The shaded colour assigned to each line segment is derived from the approximate integral of the Phong function around the circumference of the (straight) tube represented by the segment. This approximation is validated by the observation that at small scales, a shaded tube appears to have a single uniform colour, since the eye cannot discern the variation in brightness across its breadth. It can be shown that at any point on the tube, the Phong integral is a function of only two scalars: the angle $\theta$ between the tube's tangent vector at the point and the vector from the point to the light source, and the angle $\phi$ between the tube's tangent and the vector from the point to the viewer. This is illustrated in Figure 2.5. $\theta$ alone is sufficient to specify the diffuse component of the shading, while both angles are needed for the specular. Because its domain has only two dimensions, the shading function for polylines can be precomputed over a range of quantized values and stored in a 2-dimensional array, allowing fast table-lookup operations to be used in place of traditional Phong sampling. Moreover, by using analytically computed integrals rather than point sampling, this approach elegantly circumvents the problem of spatial aliasing that is inherent in using traditional (i.e. sampling-based) Phong shading with models having high per-pixel normal variation, such as thin tubes.

A common application for polylines is in rendering hair and, to a lesser extent, fur.

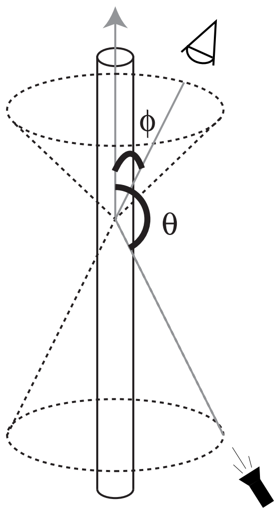


Figure 2.5: The two angles that specify a polyline cylinder's reflectance.

Examples of the former include [LTT91, RCI91], which make effective use of the pre-computed shading model in dealing with the high degree of specularity exhibited by hair. Hair is particularly suitable for the polyline approach because its constituents are extremely thin in proportion to their length. For instance, when rendering human hair from several metres of distance using a typical field of view and resolution, the hairs will be less than a pixel in thickness and possibly dozens of pixels in length. The length is significant because it largely determines how well the hair would be represented using a global texture map, as described in the following section. For very short hairs (which tend to occur in fur, rather than in human hair) texture-mapping entire clumps of hair becomes a superior alternative to rendering the individual hairs, yielding comparable image quality at far greater speed.

## 2.3 Global Texture-Mapping Methods

A fast way to render a large group of objects at a small scale is to pre-render them from one or more viewing angles and convert the resulting images into texture maps. At rendering time, these textures are applied to relatively large polygons, which effectively replace the finely detailed geometry represented in the texture map. For many real-time applications, this is the only viable way of rendering a geometrically rich model (such as a tree) at a small scale, because mapping a texture onto a polygon is so much faster than rendering a large number of tiny objects. Used under the right circumstances, this technique can produce high image quality at unparalleled speed. The ideal scenario for it is one where the entities comprising the texture are sufficiently distant from the viewer that their parallax and occlusion effects are negligible.

An example of this approach is the common technique called billboarding [NDW93], which has been used to inexpensively render trees and other complex objects, when viewed from a distance. At small scales, a tree's constituent leaves, twigs, and branches project to sufficiently small screen-space images that a texture map gives an adequate approximation of their true geometry. The texture is mapped onto a polygon that is continually rotated to face the viewer, ensuring that the tree image is always orthogonal to the viewing direction.[6] The polygon's shape needn't conform to the tree's outline; alpha values are used to "hide" parts of the polygon that are outside of the tree's image.

More advanced methods combine moderate geometric complexity with texture-mapping to preserve global aspects of the simplified geometry. The work on multiresolution surface viewing by Andrew Certain *et al.* [CPD$^+$96] allows for the relative proportion of geometric and colour (i.e. texture) detail to be specified as a rendering parameter. Moreover, the amount of detail in each can be progressively refined in an efficient way, using wavelets. Although the ability to adjust texture detail is generally irrelevant for rendering with texture-mapping hardware (in the paper it was used principally as a variable form of image compression), the progressive geometry refinement is a useful feature if a desired frame rate needs to be attained at the possible expense of image quality. When navigating a scene of variable complexity, each object can be allotted a fixed amount of time for image refinement, resulting in a steady rendering speed with variable image quality.

A successful attempt to improve the quality of texture-mapped images at larger scales is described by Jonathan Shade *et al.* in their paper on hierarchical image caching [SLS$^+$96]. Their method involves building an object hierarchy of a complex scene and

[6]The rotation is necessary to prevent the polygon (and the textured image it contains) from being distorted by parallax when viewed from the side.

caching the rendered image of each node in the hierarchy for use in subsequent frames. The cached image is reused by texture-mapping it onto a quadrilateral that is drawn in place of the original geometry. The hierarchical approach ensures that a large number of small distant objects are clustered into a single texture map, preventing a proliferation of small polygons.

What distinguishes this technique from the others is that it applies an error metric to determine how well a cached image continues to approximate its associated geometry as the viewer moves about the scene. When the quality of the approximation dips below a given threshold, the cached image is replaced by a freshly rendered one based on the viewer's new position. According to the authors, this permits a roughly tenfold speed increase over plain view frustum culling when rendering walkthroughs of a complex outdoor scene, with a minimal reduction in image quality.

## 2.4 Volumetric Textures

The basic idea behind volumetric textures is to replace volumes of complex repetitive geometry with sampled distributions of its density and reflectance behaviour, called *texels*. Using a volumetric ray-tracer, these distributions can then be efficiently rendered at a cost that is invariant to the amount of detail stored in the texel (provided that its size and resolution remain constant). This invariance makes volumetric rendering somewhat akin to texture-mapping in providing a near-constant rendering time for scenes of arbitrary complexity. Unlike texture-mapping, however, volumetric textures correctly handle parallax and occlusion effects, since texels are truly three-dimensional in nature.

Early work by Kajiya and Kay [KK89] produced impressive results in rendering fur, using an *ad hoc* reflectance model based on the cylinder.[7] This work has been generalized by Fabrice Neyret [Ney95b] to permit more general reflectance models based on the normal distributions of ellipsoids, and to allow hierarchies of multiple texel resolutions that minimize cost and spatial aliasing in the spirit of mip-mapping. Further extensions by Neyret include methods to deform texels so as to permit some basic forms of animation

[7]Forms of this reflectance model have found use in other rendering methods, such as the polyline technique described in §2.2.3.

[Ney95a], though this is still not as flexible as with conventional geometric models.

Although usable at a wide range of scales, volumetric textures entail the considerable computational overhead of volumetric ray-tracing. Accordingly, they offer an effective alternative to the above projective rendering methods only if (1) the general rendering requirements strain or exceed the capacity of projective rendering (e.g. soft shadows, accurate reflections and refractions, volume opacity effects), or (2) the amount of per-texel detail is sufficiently high to take advantage of their near-constant rendering time.

## 2.5 Summary

As our sampling shows, there are a variety of ways to render tubular objects. Whereas some methods are sufficiently general to model tubes of arbitrary cross-section, others are specialized for plain or generalized cylinders, as is the paintstroke. Moreover, some methods are suited for large-scale rendering while others are only useful at very small scales. As we shall see in Chapter 5, the paintstroke primitive, although limited in scope, can usually do a better job in rendering generalized cylinders within its intended range of scales than all the competing methods presented here.

# Chapter 3

# The Paintstroke: A Generalized Cylinder Primitive

In this chapter we examine the structure and properties of the paintstroke primitive, and develop its dynamic tessellation algorithm. We begin by discussing the way in which paintstrokes are modelled and represented within our projective rendering framework. The remainder of the chapter is largely devoted to a detailed account of their tessellation. We conclude with an overview of the special rendering effects that are possible with paintstrokes, and briefly explain how these are achieved.

## 3.1 Representation

The essential properties of a paintstroke can be succinctly described with a one-dimensional parametric function, $\mathbf{ps}(T)$. The components of this function are visual attributes that vary along the length of the paintstroke: position, radius, colour, opacity, and reflectance. All components but the radius are themselves vectors, consisting of related scalar subcomponents. The overall $\mathbf{ps}$ function appears in Equation 3.1 below, with the components ordered as listed above.

$$\mathbf{ps}(T) = \begin{bmatrix} \mathbf{pos}(T) \\ rad(T) \\ \mathbf{colour}(T) \\ \mathbf{op}(T) \\ \mathbf{refl}(T) \end{bmatrix} \tag{3.1}$$

The $\mathbf{ps}(T)$ function is defined using a series of $n \geq 2$ control points, $\{\mathbf{cp}_0, \mathbf{cp}_1, \ldots, \mathbf{cp}_{n-1}\}$. Each of these is an arbitrary constant vector bearing a value of $\mathbf{ps}(T_i)$ at regular intervals of $T_i$.[1]The control points are used by the rendering algorithm to generate simple, visually appealing interpolant functions for all the components. A sample paintstroke with variation in radius and colour appears in Figure 3.1. The white dots indicate the positions of the control points.

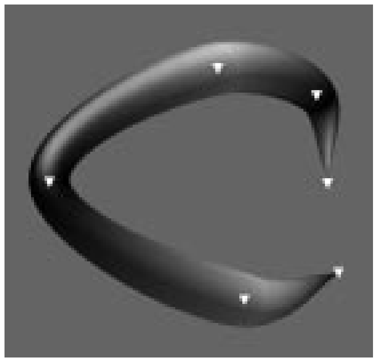


Figure 3.1: Sample rendered paintstroke with control points indicated.

In our implementation, the $\mathbf{pos}(T)$ and $rad(T)$ components are piecewise-cubic splines. These provide a reasonable degree of continuity and flexibility, yet can be efficiently generated, evaluated, integrated, and differentiated. The remaining components of $\mathbf{ps}(T)$ are piecewise-linear interpolants. While they lack the smoothness of splines, our experience has shown that the eye is considerably less attuned to derivative discontinuities in these latter components than to those of the position or radius.

At this point, we introduce a re-parametrization of $\mathbf{ps}(T)$, which will be more useful in dealing with piecewise functions: we define $\mathbf{ps}_m(t)$ as the section of $\mathbf{ps}(T)$ where $T \,\epsilon\, [a, b]$ such that $\mathbf{ps}(a) = \mathbf{cp}_m$ and $\mathbf{ps}(b) = \mathbf{cp}_{m+1}$. The new parameter, $t \,\epsilon\, [0, 1]$, ranges over a single section of a paintstroke between a pair of neighbouring control points:

$$\mathbf{ps}_m(0) = \mathbf{cp}_m \tag{3.2}$$

$$\mathbf{ps}_m(1) = \mathbf{cp}_{m+1} \tag{3.3}$$

[1]Besides capturing these discrete values of $\mathbf{ps}(T)$, the control points contain some additional information that is used by the global lighting algorithm to simulate the self-shadowing of convex objects consisting of paintstrokes. We shall ignore this additional information until §3.3.3, in which we present the global shading algorithm.

As a notational shorthand, we omit the subscript in the new parametrization if the indices of the bounding control points are implied by the context. For example, we write $\mathbf{ps}(t)$ in reference to $\mathbf{ps}_m(t)$, where the value of $m$ is implicit. We will make frequent use of this shorthand form when we discuss a single section of a paintstroke that is bounded by an arbitrary pair of adjacent control points. From this point on, our use of the term *section* in the context of paintstrokes will be restricted to the portion of a paintstroke between two adjacent control points. We will use the term *segment* to refer to a subset of a section.

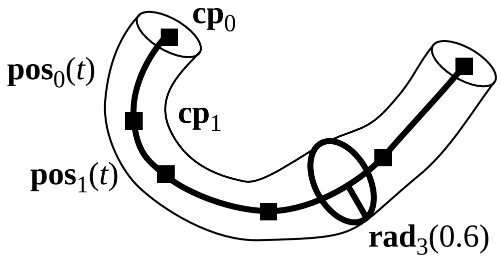


Figure 3.2: Paintstroke with piecewise position and radius components shown.

Defining all the components of a paintstroke at the same control point is potentially inefficient if some components exhibit greater variation than others—consider an elaborately coloured paintstroke with a simple underlying geometry, most of whose control points are introduced to store colour information, not geometry. This could be rectified by modifying our implementation to use a separate set of control points for each component or group of components with similar complexity. An alternative approach, discussed in Chapter 6, is to use a one-dimensional texture to encode non-positional information.

Having presented the general structure of $\mathbf{ps}(t)$ and its components, we now shift our focus to the latter. We examine the components of an arbitrary paintstroke section $\mathbf{ps}_m(t)$, deriving each component's interpolant function from the control point values assigned by the modeller.

### 3.1.1 Position

$$\mathbf{pos}(t) = \begin{bmatrix} pos_x(t) \\ pos_y(t) \\ pos_z(t) \end{bmatrix} \tag{3.4}$$

The function $\mathbf{pos}(t)$ defines the path that the paintstroke segment follows through $\mathbb{R}^3$ (in our case, eye-space) using a piecewise Catmull-Rom spline. Given the eye-space position values $\mathbf{p}_{m-1}$, $\mathbf{p}_m$, $\mathbf{p}_{m+1}$, and $\mathbf{p}_{m+2}$ at the (parametrically evenly-spaced) control points, the Catmull-Rom spline extends from $\mathbf{p}_m$ to $\mathbf{p}_{m+1}$, according to the equation

$$\mathbf{pos}_m(t) = \frac{1}{2} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{p}_{m-1} \\ \mathbf{p}_m \\ \mathbf{p}_{m+1} \\ \mathbf{p}_{m+2} \end{bmatrix} \cdot \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix} \tag{3.5}$$

As Figure 3.3 shows, a Catmull-Rom spline is equivalent to a Hermite curve, such that the tangent vector at each inner control point joins the two surrounding control points. Because the first and last control point of the paintstroke must be interpolated[2], we double them. This produces linear interpolants for the first and last segments. A

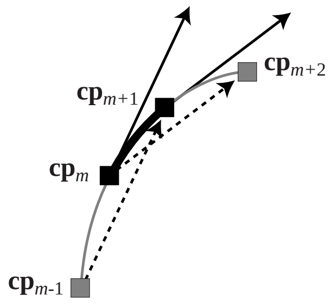


Figure 3.3: The Catmull-Rom spline.

future version of our algorithm will likely also incorporate a Bézier spline representation, which permits more intuitive modelling.

[2]Notice that a Catmull-Rom spline segment only interpolates between the middle two of the four control points that specify it.

### The Eye-Space Coordinate System

Our eye-space coordinate system, shown in Figure 3.4, has the viewer at the origin and looking toward the positive $z$-axis. Orthogonal to the $z$-axis lies the *projection plane*, whose distance from the viewer along the positive $z$-axis is called the *projection distance* and denoted by $d_{proj}$.

Given an arbitrary paintstroke section, the unit vector extending from the viewer in the direction of $\mathbf{pos}(t)$ is called the *view vector* and denoted by $\mathbf{view}(t)$. It is used extensively in the rendering process, and we shall refer to it throughout this chapter. Observe that $\mathbf{view}(t) = \frac{\mathbf{pos}(t)}{\|\mathbf{pos}(t)\|}$.

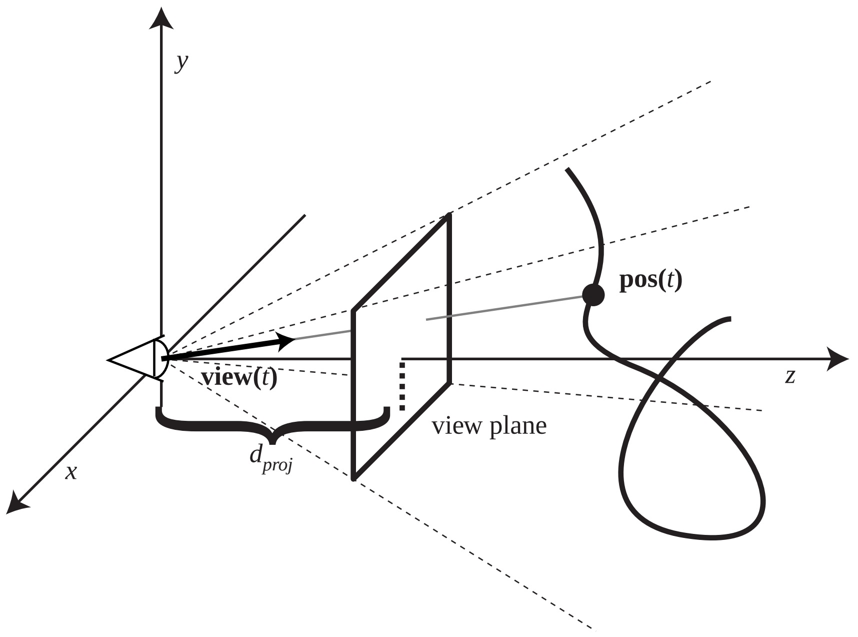


Figure 3.4: The geometry of our eye-space coordinate system.

## 3.1.2 Radius

The function $rad(t)$ defines the thickness of the segment, measured orthogonally to the tangent vector of the path, $\mathbf{pos}'(t)$. It is expressed in the same units as $\mathbf{pos}(t)$ and can take any nonnegative value; however, using a value that exceeds the paintstroke's radius of curvature yields unsightly folds in the surface.

The radius function is defined as a piecewise Catmull-Rom spline, precisely like each

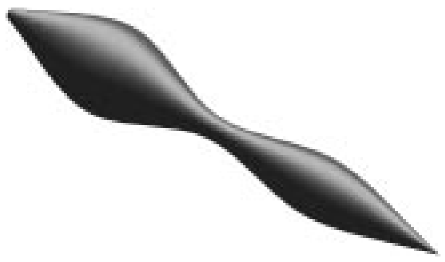


Figure 3.5: Variation in the radius component of a paintstroke.

component of $\mathbf{pos}(t)$. The model supplies a set of radii $\{r_0, r_1, \ldots, r_{n-1}\}$ corresponding to the paintstroke's radii at the $n$ control points. As with the positional interpolant, the first and last control points are doubled.

$$rad_m(t) = \frac{1}{2} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix} \begin{bmatrix} r_{m-1} \\ r_m \\ r_{m+1} \\ r_{m+2} \end{bmatrix} \cdot \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix} \tag{3.6}$$

### 3.1.3 Colour

$$\mathbf{colour}(t) = \begin{bmatrix} colour_r(t) \\ colour_g(t) \\ colour_b(t) \end{bmatrix} \tag{3.7}$$

The $\mathbf{colour}(t)$ function is expressed in terms of a red, green, and blue component, denoted respectively by $colour_r(t)$, $colour_g(t)$, and $colour_b(t)$. These components vary independently along the path of the segment. An alternative colour representation in terms of hue, saturation, and colour value (HSV), would allow easier modelling and provide more intuitive interpolation, albeit at the expense of (nonlinear) conversions to RGB space.

Our implementation only permits colour variation *along* the paintstroke, and not around its girth—the latter is considerably more involved, being view-dependent and nonlinear in screen-space. In this regard it is similar to texture-mapping, a feature that is discussed in Chapter 6 as a potential enhancement to paintstrokes.

A colour value is assigned at each control point, and the $\mathbf{colour}(t)$ function interpolates linearly between these values, along the spine of the paintstroke. Given the set

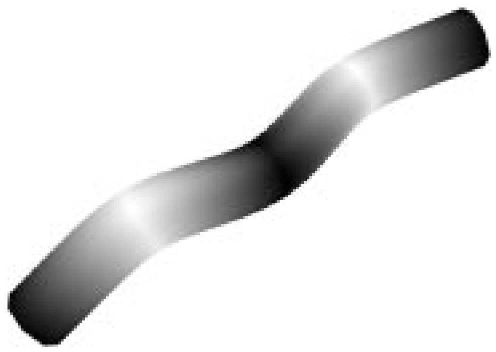


Figure 3.6: Variation in the colour component of a paintstroke.

of $(r, g, b)$ colour points $\{\mathbf{c}_0, \mathbf{c}_1, \ldots, \mathbf{c}_{n-1}\}$, the equation for the interpolated colour value between control points $\mathbf{c}_m$ and $\mathbf{c}_{m+1}$ is

$$\mathbf{colour}_m(t) = (1-t)\mathbf{c}_m + t\mathbf{c}_{m+1} \tag{3.8}$$

### 3.1.4 Opacity

$$\mathbf{op}(t) = \begin{bmatrix} op_{min}(t) \\ op_{max}(t) \\ op_{centre}(t) \\ op_{edge}(t) \end{bmatrix} \tag{3.9}$$

A segment's opacity, represented by $\mathbf{op}(t)$, can vary both along its length and across its breadth. The lengthwise opacity is modulated according to the segment's orientation relative to the viewer, with the maximum opacity, $op_{max}(t)$, attained when the paintstroke's path is collinear with the view vector, and the minimum $op_{min}(t)$ when the two vectors are orthogonal. This corresponds to the intuitive notion that looking *along* the paintstroke should yield the greatest opacity, as shown in Figure 3.8. Section 3.3.1 provides a concrete example of this, and also explains how we interpolate between $op_{min}(t)$ and $op_{max}(t)$ to obtain the paintstroke's lengthwise opacity at a given value for $t$.

In addition to the lengthwise opacity modulation, a paintstroke allows the modeller to independently set opacity values for the centre and the edges of its screen-projected image. These values are represented by the functions $op_{centre}(t)$ and $op_{edge}(t)$, respectively. Any point on the surface of the paintstroke thus has an additional opacity value that depends

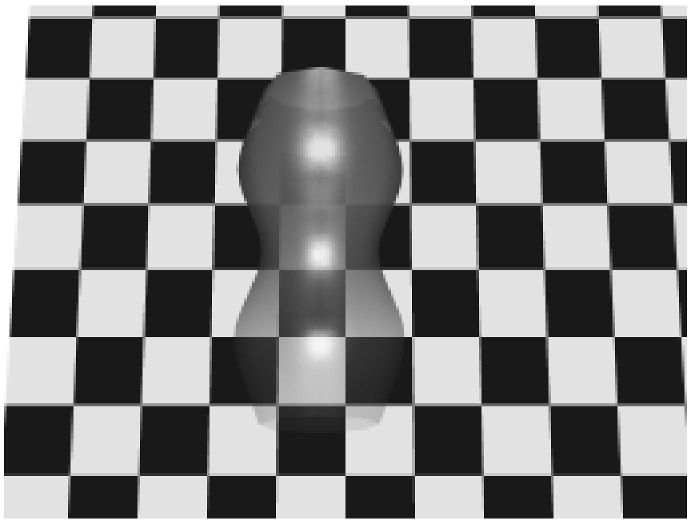


Figure 3.7: Simple variation in the opacity of a paintstroke.

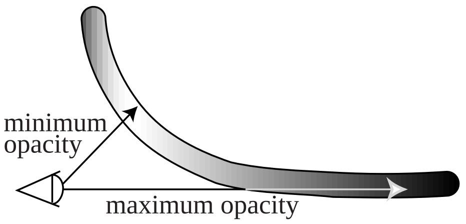


Figure 3.8: Intuition for lengthwise opacity variation.

on its position between the centre and the nearest edge. This breadthwise opacity value is multiplied by the lengthwise opacity described above to yield an overall opacity.

Each component of the $\mathbf{op}(t)$ vector is interpolated linearly along the segment, as with the colour. Given the set of opacity vectors $\{\mathbf{o}_0, \mathbf{o}_1, \ldots, \mathbf{o}_{n-1}\}$ corresponding to the $n$ control points, the interpolation is given by

$$\mathbf{op}_m(t) = (1-t)\mathbf{o}_m + t\mathbf{o}_{m+1} \tag{3.10}$$

### 3.1.5 Reflectance

$$\mathbf{refl}(t) = \begin{bmatrix} refl_{k_a}(t) \\ refl_{k_d}(t) \\ refl_{k_s}(t) \end{bmatrix} \tag{3.11}$$

The function $\mathbf{refl}(t)$ contains the ambient, diffuse, and specular reflectance coefficients of a segment. These are used in the local shading model, which we shall describe in §4.3.6. Like the other $\mathbf{ps}(t)$ components, the reflectance function is user-defined. However, it can also be automatically adjusted by the global shading algorithm, as discussed in §3.3.3. Given the set of reflectance vectors $\{\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{n-1}\}$, the reflectance interpolant is

$$\mathbf{refl}_m(t) = (1-t)\mathbf{r}_m + t\mathbf{r}_{m+1} \tag{3.12}$$

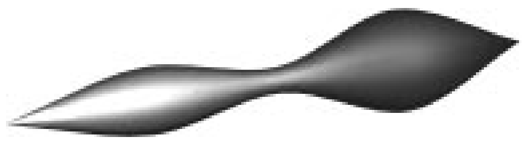


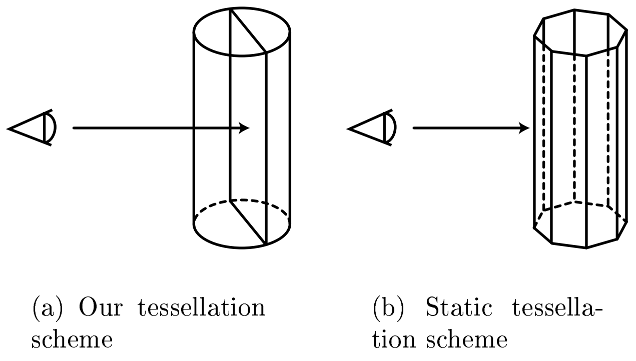
Figure 3.9: Variation in the reflectance component of a paintstroke.

## 3.2 Tessellating the Paintstroke

Traditional tessellation schemes subdivide a surface into a set of world-space or eye-space polygons. Whereas the arrangement of these polygons is pre-determined, the arrangement of their screen-space projections is view-dependent. The tessellation of a paintstroke is in a sense the opposite: it yields a view-dependent set of eye-space polygons whose screen-space projections are pre-determined. The paintstroke in effect directly polygonizes the screen-projection of a generalized cylinder—not the full eye-space surface. This is what makes the name "paintstroke" appropriate to our primitive: an artist drawing a three-dimensional tube with a single stroke of the paintbrush capitalizes on the simplicity of this object's screen projection, as does our tessellation scheme. Although the view-dependency of a paintstroke's polygonization entails dynamic tessellation, it allows the paintstroke to be drawn from any viewing angle with a very small number of polygons. As we shall see in Chapter 5, this more than makes up for the cost of the retessellation.

Some of the assumptions behind our methodology rely on weak perspective[3], particularly with respect to surface normals. Because paintstrokes are intended for relatively small-scale rendering, which typically implies a moderate viewing distance, this assump-

[3]By this we mean that the perspective projection of an object is very similar to an orthonormal one.


(a) Our tessellation scheme

(b) Static tessellation scheme

Figure 3.10: Example of our dynamic tessellation compared to traditional static tessellation.

tion is not at all unreasonable. But even under fairly strong perspective, we have found paintstrokes to produce adequate results.

In the following sections, we shall examine the algorithm that performs the dynamic tessellation. Figure 3.11 indicates the various stages of the tessellation process that will be examined, shown in the order in which they are performed. Our discussion will generally proceed in this order, with two exceptions: all optional steps are discussed at the end, and the Inflection Point constraint is explained *after* the other Lengthwise Subdivision constraints, because an understanding of the latter is needed to see the purpose of the former.

### 3.2.1 Geometric Transformations and Interpolant Generation

The first step in the tessellation process transforms all the geometric data stored in the control points $\{\mathbf{cp}_0, \mathbf{cp}_1, \ldots, \mathbf{cp}_{n-1}\}$ from world-space to eye-space. This data consists of the **pos** and $\mathbf{N}_{gl}$ components. The latter is called the *global normal* vector and is used for global shading effects, discussed in §3.3.3.

Each section of the paintstroke, bounded by control points $\{\mathbf{cp}_0, \mathbf{cp}_1\}, \{\mathbf{cp}_1, \mathbf{cp}_2\}$, $\ldots$,$\{\mathbf{cp}_{n-2}, \mathbf{cp}_{n-1}\}$, is rendered individually. All the sections are processed in the same manner, except that the first and last one may be closed off with a polygonal endcap, as explained in §3.2.5.

A section is handled as follows. First, the polynomial coefficients of the interpolants

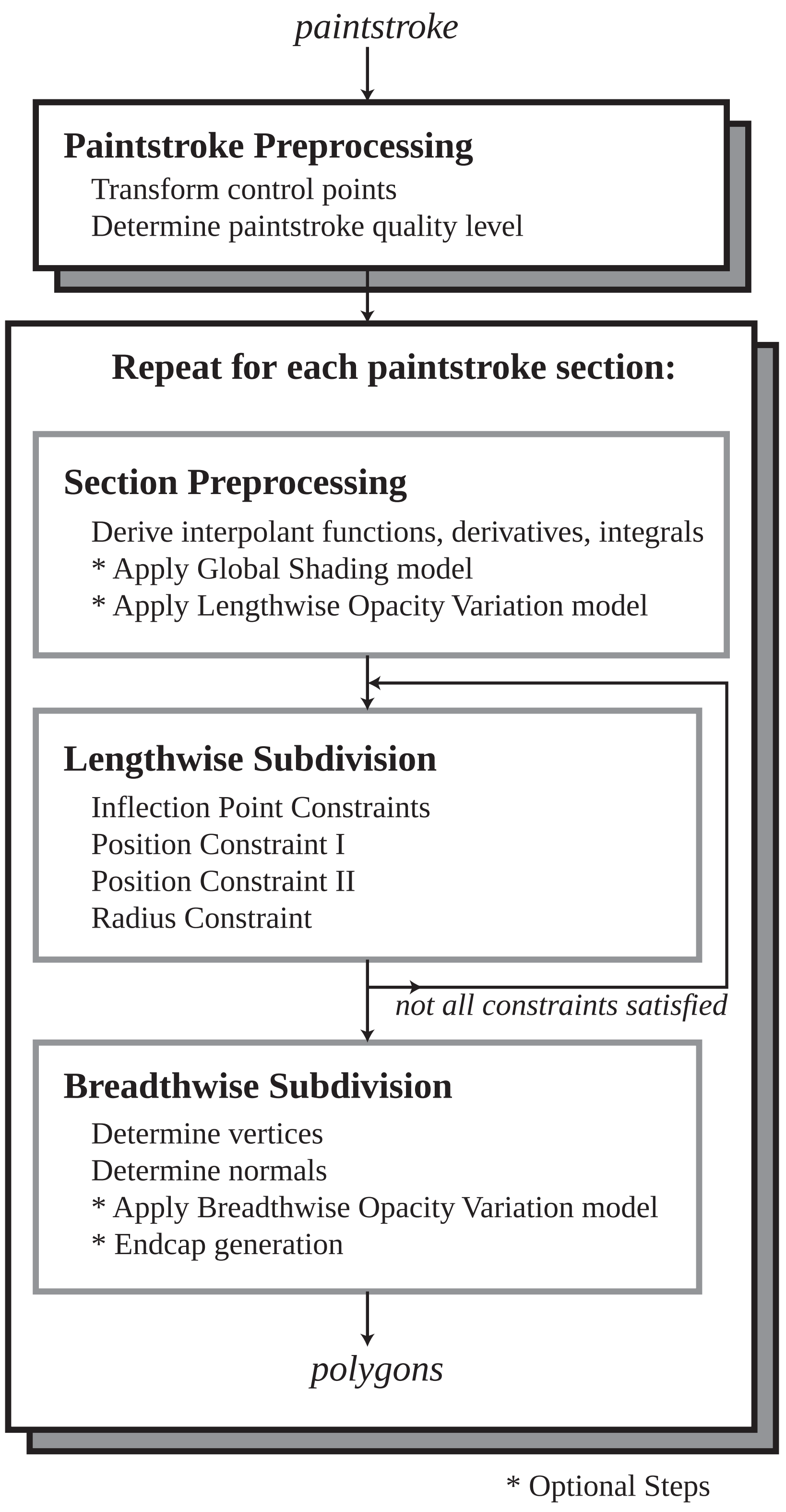

paintstroke
Paintstroke Preprocessing
Transform control points
Determine paintstroke quality level
Repeat for each paintstroke section:
Section Preprocessing
Derive interpolant functions, derivatives, integrals
* Apply Global Shading model
* Apply Lengthwise Opacity Variation model
Lengthwise Subdivision
Inflection Point Constraints
Position Constraint I
Position Constraint II
Radius Constraint
not all constraints satisfied
Breadthwise Subdivision
Determine vertices
Determine normals
* Apply Breadthwise Opacity Variation model
* Endcap generation
polygons
* Optional Steps

Figure 3.11: Stages of paintstroke tessellation.

$\mathbf{pos}(t)$ and $rad(t)$ are computed, as well as for their derivatives and antiderivatives. These coefficients can be precomputed for $rad(t)$ and reused each time the section is drawn in subsequent frames. The $\mathbf{pos}(t)$ component needs to be re-generated for every frame because the viewing transformations affect this component of the control points.

The following two phases will subdivide the section, first along its length, and then along its breadth. The lengthwise subdivision breaks the section into segments, each of which is then divided along its breadth to ultimately produce a set of polygons. That completes the tessellation.

### 3.2.2 Lengthwise Subdivision

Once the piecewise interpolants have been generated, the section between each pair of adjacent control points is recursively subdivided in half into pairs of segments until the *subdivision criterion* for all the segments has been satisfied. Whenever a segment is subdivided, the split occurs at the parametric midpoint, i.e. at $\mathbf{ps}(0.5)$. The two halves are then recursively subdivided in the same manner until no further subdivisions are required.

A paintstroke segment $\mathbf{ps}(t)$ for $t \epsilon [a, b], a < b$ is either subdivided or advances to the next phase, depending on the behaviour of its $\mathbf{pos}(t)$ and $rad(t)$ components. In order to advance, it must be approximately linear in $\mathbf{pos}(t)$ and $rad(t)$[4], because it is subsequently drawn as a truncated cone—which *is* linear in these components. There are two constraints based on the former and one on the latter. If there are inflection points in any component of $\mathbf{pos}(t)$ or $rad(t)$, these need to be dealt with as described in §3.2.2 below, since the constraints we use are only valid on interpolants with monotonic derivatives (within the specified segment).

#### Position Constraint I

The first position constraint is based on the angle $\theta$ between the two-dimensional tangent vectors $\mathbf{pos}'_{scr}(a)$ and $\mathbf{pos}'_{scr}(b)$. These are the $(x, y)$ screen-space projections of the derivative $\mathbf{pos}'(t)$ at the segment's endpoints, $t = a$ and $t = b$. If $\theta$ exceeds a threshold

[4]Note that it is already linear (by definition) in the other components.

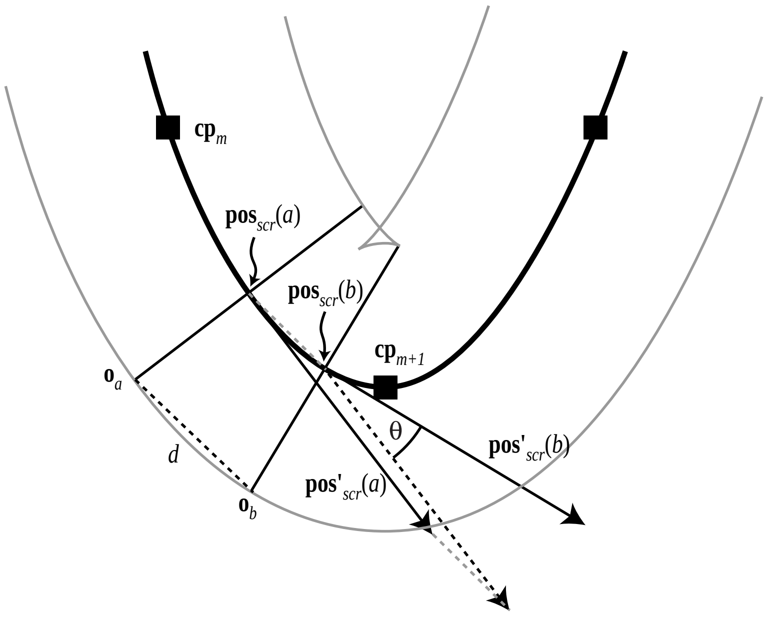


Figure 3.12: The elements of Position Constraint I.

value denoted by $\theta_{max}$, the constraint forces a subdivision. $\theta_{max} \epsilon \, (0^0, 90^0)$ is a function of the segment's maximum length, as defined below, and a user-adjustable tolerance parameter $tol_\theta$.

The values $\mathbf{pos}'_{scr}(a)$ and $\mathbf{pos}'_{scr}(b)$ are obtained by analytically differentiating the function $\mathbf{pos}_{scr}(t)$, the screen-projection of the paintstroke's path. Because our projection point lies at the origin of the eye-space coordinate system and the projection plane is orthogonal to the $z$-axis, determining $\mathbf{pos}_{scr}(t)$ to within a constant offset requires only three items: the eye-space position $\mathbf{pos}(t)$, the projection distance $d_{proj}$ (as shown in Figure 3.4), and the scaling difference between screen-space and eye-space coordinates.[5] The latter two are effectively combined into the nonzero projection constant $c_{proj}$, used in the formulas below. The constant offsets *offset*$_x$ and *offset*$_y$ depend on the viewport position; they are not computed since they vanish when $\mathbf{pos}_{scr}(t)$ is differentiated.

$$\mathbf{pos}_{scr}(t) \quad = \quad \begin{bmatrix} c_{proj} \, \frac{pos_x(t)}{pos_z(t)} + \mathit{offset}_x \\ c_{proj} \, \frac{pos_y(t)}{pos_z(t)} + \mathit{offset}_y \end{bmatrix} \tag{3.13}$$

[5]We assume an equal scaling for the $x$ and $y$ directions.

$$\mathbf{pos}'_{scr}(t) = \begin{bmatrix} c_{proj} \dfrac{pos'_x(t)pos_z(t) - pos'_z(t)pos_x(t)}{pos_z^2(t)} \\ c_{proj} \dfrac{pos'_y(t)pos_z(t) - pos'_z(t)pos_y(t)}{pos_z^2(t)} \end{bmatrix} \tag{3.14}$$

The next step is to normalize $\mathbf{pos}'_{scr}(a)$ and $\mathbf{pos}'_{scr}(b)$, and then compute their dot product, which yields $\cos\theta$.[6] If this value is negative, we know that $\theta$ exceeds the maximum value of $\theta_{max}$ ($90^0$) so we subdivide the segment without any further work. Otherwise, we need to determine $\theta_{max}$. We begin by finding the segment's maximum length $d$, defined as the straight-line distance along the outside of a curved screen-projected paintstroke segment. If the segment is straight, either side can be used, since their lengths are equal. The value of $d$ is computed as the distance between the two *outside points* $\mathbf{o}_a$ and $\mathbf{o}_b$, which are the points lying on the outside boundary of the segment at $t = a$ and $t = b$, respectively. The various elements of Position Constraint I are shown in Figure 3.12.

The outside point $\mathbf{o}_a$ is computed by displacing the position point $\mathbf{pos}_{scr}(a)$ by one of the two vectors perpendicular to $\mathbf{pos}'_{scr}(a)$, namely

$$[pos_{scr}{}'_y(a), -pos_{scr}{}'_x(a)]^\top \text{ or } [-pos_{scr}{}'_y(a), pos_{scr}{}'_x(a)]^\top \tag{3.15}$$

which has been normalized and scaled by the screen-projected radius $c_{proj}rad(a)/pos_z(a)$. To determine which of the two perpendicular vectors points toward the outside of the paintstroke's curvature, we apply a test based on $\mathbf{pos}''_{scr}(a)$, recognizing that the second derivative vector always points toward the centre of curvature. The same algorithm is applied to obtain $\mathbf{o}_b$.[7] Once the outside points have been determined, it is trivial to compute $d^2$, the square of the distance between them. We use this value to derive $\cos^2\theta_{max}$, as per the equation

$$\cos^2\theta_{max} = \frac{d^2}{d^2 + tol_\theta^2} \tag{3.16}$$

Figure 3.13 provides a geometric interpretation for $\theta_{max}$ as a function of $d$ and $tol_\theta$. The effect of this function is to enforce a strict angular tolerance for long segments

[6]Because we normalize the derivative vectors, we can omit the multiplications by $c_{proj}$ in Equation 3.14. This yields the vector $\mathbf{pos}'_{scr}(t)/c_{proj}$ instead of $\mathbf{pos}'_{scr}(t)$, but the two are equal when normalized.

[7]Although the first segment requires computing both $\mathbf{o}_a$ and $\mathbf{o}_b$, subsequent segments reuse the second point from the previous segment, so that only one outside point per segment needs to be computed.

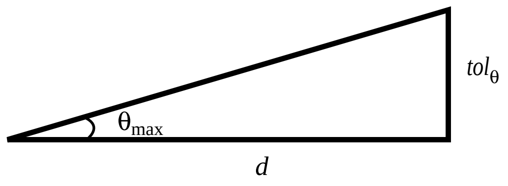


Figure 3.13: Geometric interpretation of $\theta_{max}$.

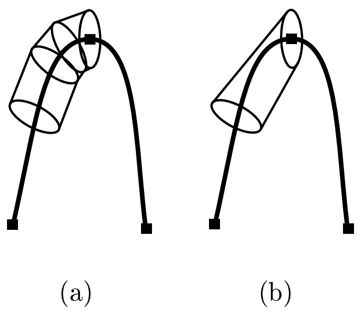


Figure 3.14: Thickness distortion resulting from inadequate lengthwise subdivision.

(where $d$ is large), but to relax the tolerance for short ones. As a result, the lengthwise subdivision granularity adapts to the screen-projected length of the paintstroke segment, and it does so in a manner that can be modified by tuning the positive parameter $tol_\theta$.

Finally, the test that decides whether this constraint causes a subdivision is the following, with a false value triggering the subdivision:

$$\cos^2\theta \geq \cos^2\theta_{max} \tag{3.17}$$

Given that $\theta \,\epsilon\, (0^0, 90^0)$, the above relation is equivalent to the more intuitive (but also more expensive) test condition, $\theta \leq \theta_{max}$.

Aside from the obvious purpose of maintaining a smooth silhouette for a paintstroke segment, Position Constraint I also keeps the segment's thickness from being distorted by sharp bends in its screen-projected path. Segments with even a slight eye-space curvature will, from certain viewing angles, exhibit a sharp curvature in their screen-space projection. This is an important consideration because any segment with greatly differing endpoint tangents appears significantly thinner than the curved tube it represents. An example of this phenomenon is given in Figure 3.14.

**Position Constraint II**

The second position constraint maintains a desired degree of linearity in the $z$-component of $\mathbf{pos}(t)$. This is necessary to ensure that a curved segment is adequately subdivided even when viewed from an angle that makes its screen projection close to linear. In such a situation, Position Constraint I would allow the entire segment to be rendered without any subdivision, regardless of its true (i. e. eye-space) curvature. Although the rendered image would have the correct shape, it would fail to express the true variation in the surface normals along the segment.

To implement this constraint, we begin by computing over the interval $[a, b]$ the exact average values of $pos_z(t)$ and its linear interpolant $\frac{t-a}{b-a}\,[pos_z(b) - pos_z(a)] + pos_z(a)$. The absolute difference between the two is a measure of $pos_z(t)$'s nonlinearity. This value is then scaled by $\frac{d_{proj}}{pos_z(a)}$, a factor representing the foreshortening effect of the perspective transformation at $\mathbf{pos}(a)$, given the projection distance $d_{proj}$ (shown in Figure 3.4).[8] Finally, the perspective-adjusted nonlinearity measure is bounded by the parameter $tol_z$, specified by the user. The formula for this constraint simplifies to the following:

$$\frac{d_{proj}}{pos_z(a)} \left| \frac{1}{b-a} \int_a^b pos_z(t)\,dt \; - \; \frac{pos_z(a) + pos_z(b)}{2} \right| < tol_z \tag{3.18}$$

where the integral is easily obtained from the precomputed coefficients of the (quartic) antiderivative to $pos_z(t)$.

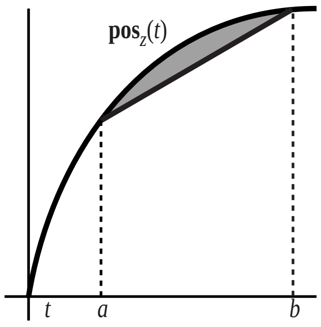


Figure 3.15: Geometric interpretation of $\int_a^b pos_z(t)\,dt \; - \; \frac{b-a}{2}[pos_z(a) + pos_z(b)]$.

[8]Applying the perspective factor for $\mathbf{pos}(a)$ to the entire interval $[a, b]$ is a reasonable simplification because, at the screen-projected size that paintstrokes are intended for, the perspective effect should be very similar at both endpoints of a segment.

The shaded region in Figure 3.15 represents the raw difference between the integrals of $pos_z(t)$ and its linear interpolant over the region $[a, b]$. We will discuss shortly how to cope with an inflection point in the interval. To obtain the measure of nonlinearity (prior to the perspective adjustment) we must divide this raw difference by $b - a$. We do this to essentially normalize the interval $[a, b]$, whose length does not consistently correspond to the length of the paintstroke segment.[9] Contrary to our initial intuition, this does not counteract the view-adaptive nature of this constraint, since enlarging a paintstroke (by uniformly scaling its control points) would have no effect on the parametric distance $b - a$. Thus the normalization of this interval only serves to treat segments of different physical lengths "equally", by not biasing the nonlinearity measure with a high value for large parametric intervals that may correspond to small physical distances. Note that the perspective-adjusted nonlinearity measure is sensitive both to true enlargement of the paintstroke and to the perspective-induced enlargement of its screen projection as it approaches the viewer.

### The Radius Constraint

The radius constraint ensures a smooth lengthwise variation in the radius of a segment. It is precisely analogous to Position Constraint II, relying on the perspective-adjusted average difference between the $rad(t)$ function and its linear interpolant. The simplified equation for this constraint is

$$\frac{d_{proj}}{pos_z(a)} \left| \frac{1}{b-a} \int_a^b rad(t)\,dt \;-\; \frac{rad(a) + rad(b)}{2} \right| < tol_{rad} \tag{3.19}$$

### Inflection Point Constraints

**Simple Inflection Points** Position Constraint II and the Radius Constraint both require that the spline interpolants to which they apply have monotonic derivatives with respect to parameter $t$ over the entire segment $t \,\epsilon\, [a, b]$. In order to satisfy this condition, we must ensure that $pos_z(t)$ and $rad(t)$ have no inflection points in the open interval

[9]If this is not clear, consider a paintstroke with three control points, the first two close together and the third one far away. The parametric distances between the first two and last two control points are the same: one.

$t \,\epsilon\, (a, b)$. This is both necessary and sufficient to satisfy the condition. Since the interpolants are cubics, finding an inflection point within a segment—there can only be one—amounts to finding the zero of the interpolant's second derivative within the interval $(a, b)$. If one is found, a subdivision is carried out at the value of $t$ where it occurs, producing a pair of subsegments which do not contain the inflection point in their open intervals. The purpose of the foregoing requirement is exemplified in Figure 3.16(a), which shows how a function with a non-monotonic derivatives can confound the average-value constraints. The average value of the function, based on its (signed) integral, is close to that of its linear interpolant—yet, clearly, the function is far from being linear. By forcing a subdivision at the inflection point, we create two subsegments whose derivatives are monotonic within their respective intervals, thus removing the anomaly.

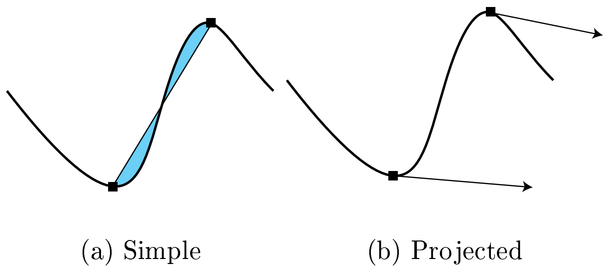

(a) Simple (b) Projected

Figure 3.16: A function with non-monotonic derivatives.

**Projected Inflection Points** Another type of inflection point that may arise is one within the screen-projected path of a segment. As can be seen in Figure 3.16(b), this can cause a nonlinear path segment to have equal tangents at the endpoints, thereby erroneously satisfying Position Constraint I. Precisely locating this type of inflection point is quite expensive, given the nonlinearity of the perspective projection and the resulting complexity of the projected curve. However, if we approximate the perspective projection with a simple orthonormal one (i.e. we discard the $z$-component), the curve becomes more tractable. This simplification provides a much faster means of detecting and locating the approximate inflection points, and is very accurate at the relatively small scales that paintstrokes are suited to.

Abbreviating the segment's eye-space positional components $pos_x(t)$ and $pos_y(t)$ to $x(t)$ and $y(t)$, we find the values of $t$ that make either $\frac{d^2y}{dx^2}$ or $\frac{d^2x}{dy^2}$ equal to zero. We reason as follows:

$$\frac{dy}{dx} = \frac{dy}{dt}\frac{dt}{dx} \tag{3.20}$$

$$= \frac{\frac{dy}{dt}}{\frac{dx}{dt}} \tag{3.21}$$

$$\frac{d^2y}{dx^2} = \frac{\frac{d^2y}{dt^2}\frac{dx}{dt} - \frac{d^2x}{dt^2}\frac{dy}{dt}}{\left(\frac{dx}{dt}\right)^3} \tag{3.22}$$

Similarly,

$$\frac{d^2x}{dy^2} = \frac{\frac{d^2x}{dt^2}\frac{dy}{dt} - \frac{d^2y}{dt^2}\frac{dx}{dt}}{\left(\frac{dy}{dt}\right)^3} \tag{3.23}$$

Setting either $\frac{d^2y}{dx^2}$ or $\frac{d^2x}{dy^2}$ to zero yields the same quadratic in terms of the polynomial coefficients for $x(t)$ and $y(t)$. Note that although one of these derivatives may be undefined due to a vanishing denominator, they can never *both* be undefined. That is because the values of $\frac{dx}{dt}$ and $\frac{dy}{dt}$ cannot both vanish at the same point, unless a pair of consecutive control points are identical in position—an illegal condition that, if present, is eliminated when the control points are read in. Given that

$$x(t) = x_3t^3 + x_2t^2 + x_1t + x_0 \tag{3.24}$$

$$y(t) = y_3t^3 + y_2t^2 + y_1t + y_0 \tag{3.25}$$

the inflection points are obtained from the zeros of the quadratic

$$I(t) = 3(x_2y_3 - x_3y_2)t^2 + 3(x_1y_3 - x_3y_1)t + x_1y_2 - x_2y_1 \tag{3.26}$$

If $I(t)$ has distinct zeros, either (or both) of them lying in the interval $[0, 1]$ are inflection points. If the two zeros are identical, they are not inflection points, since they do not represent a sign change in $\frac{d^2y}{dx^2}$ (or in $\frac{d^2x}{dy^2}$).

Because solving the roots of $I(t)$ is fairly expensive operation, our algorithm avoids doing so wherever possible. For example, when the coefficient of $t^2$ in $I(t)$ is several orders of magnitude smaller in absolute value than the coefficient of $t$, we ignore the quadratic term, approximating $I(t)$ with the resulting linear equation. If $I(t)$ cannot be simplified in this way, we evaluate $I(0)$, $I(1)$, and, if necessary, $m$ and $I(m)$, where $m$ is the (easily obtained) value of $t$ that yields an extremum. Based on these values we can determine whether any roots fall within $[0, 1]$, without explicitly solving them. If they do—which is sufficiently rare in practice to make the above tests worthwhile—then we solve for them.

### 3.2.3 Breadthwise Subdivision

After a segment has been sufficiently shortened by lengthwise subdivision, it is finally tessellated into polygons along its breadth. This involves dividing it into a ring of polygons which tile the truncated cone that the segment represents. The tessellation is view-dependent—the divisions occur relative to the centre and edges of the segment, *as they appear to the viewer.* Although the polygons are ultimately rendered using a perspective projection, the method used to tessellate them assumes an orthographic projection. The inaccuracy of this assumption is negligible at the paintstroke's intended range of scales. Moreover, this inaccuracy is visually much less significant (and is independent of) the shading and cross-sectional inaccuracies discussed below.

The specifics of a paintstroke's breadthwise tessellation depend on its *quality level.* Each paintstroke bears one of three possible quality levels, numbered 0, 1, and 2. This quantity is determined at an early preprocessing stage (indicated in Figure 3.11), based on several user-defined parameters discussed below. As shown in Figure 3.17, the number of each quality level represents $\log_2 N$, where $N$ is the number of polygons tiling the side of the segment that is closest to the viewer. Hence, a quality-zero segment is tessellated into a single polygon that always faces the viewer, a quality-one segment into two on each side (the viewer's side and the one opposite to it), and a quality-two segment into four on each side. For quality-one and quality-two paintstrokes, the side opposite the viewer is often hidden and can thus be safely ignored, saving considerable rendering time. This important optimization will be discussed shortly.

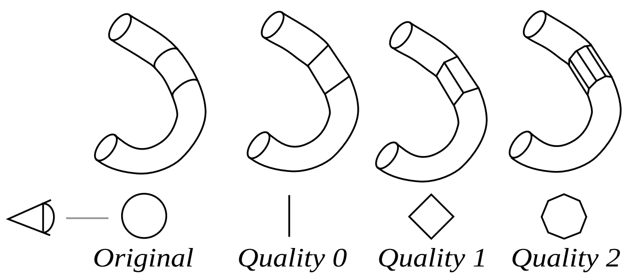
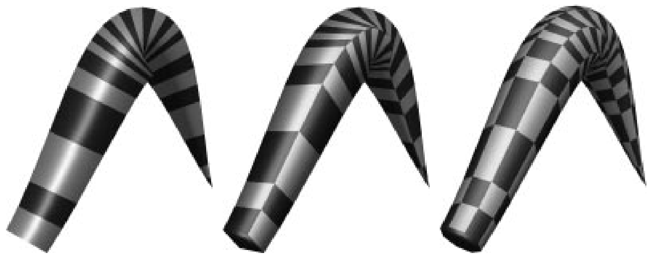


Figure 3.17: Breadthwise tessellation schemes for the three levels of quality.



(a) Quality 0 (b) Quality 1 (c) Quality 2

Figure 3.18: Paintstrokes generated at the three rendering quality levels.

### Rendering Quality

A paintstroke's quality level can be set to vary according to its maximum screen-space thickness. This feature is generally useful, although transitions in quality level are not always seamless (as are the changes in the paintstroke's polygonization from scene to scene *within* a given level of quality). For this reason, all segments of a paintstroke share the same quality level. As the following descriptions indicate, higher quality levels yield higher image quality. We have made no attempt to quantify this image quality, since it really depends on a number of factors, including the paintstroke's orientation and reflectance, the lighting, and the user's need for a *precise* (as opposed to imprecise but consistent) image.

**Quality Level 0** The tessellation of quality-zero segments is the simplest: the entire segment becomes a single quadrilateral with vertices along the edges of the paintstroke, corresponding to the silhouette of the generalized cylinder. This scheme yields the smallest number of polygons, and the greatest savings over a general-purpose tessellation method. However, it also yields the poorest rendering quality in several regards: (1) The shading is inaccurate, being based on the linear interpolation of high curvature over a single polygon. (2) A quality-zero segment disappears when viewed head-on, i.e. when the tangent of its path, $\mathbf{pos}'(\mathrm{t})$, is collinear with the view vector. (3) The self-occlusion effect accompanying high screen-space curvature—seen as a fold in the surface—is inaccurate. (4) Paintstrokes of this quality level do not support breadthwise opacity variation, as this feature requires a minimum of two polygons along the breadth of the paintstroke. Despite their limitations, quality-zero paintstrokes are still very useful at a small scale, where the above deficiencies are largely irrelevant.

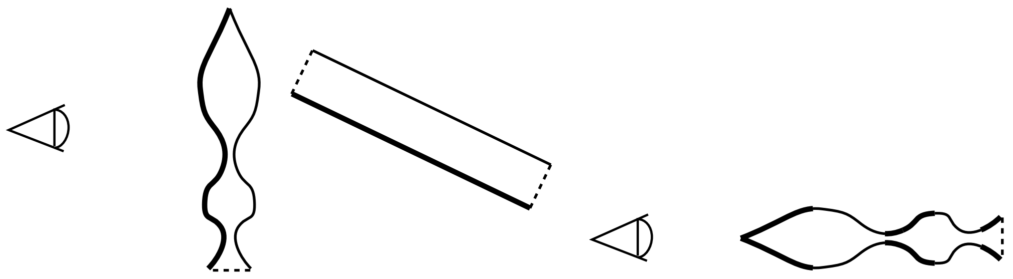
**Quality Level 1** At this level, the viewer's side of the segment is divided into two equal-sized quadrilaterals that have a common edge along the middle of the segment. The same is done, if necessary, with the opposite side. Interpolating normals across two polygons rather than one greatly improves the appearance of a shaded segment, because of a more accurate normal distribution, and also improves the screen-space fold at high curvature. Furthermore, paintstrokes of this quality level no longer disappear when viewed head-on, although they may reveal their quadrilateral cross-section if their path is sufficiently straight.

**Quality Level 2** Quality-two paintstrokes produce the highest quality images, both in their shading and in their appearance when viewed head-on. However, because they generate four polygons per segment, their savings over a general tessellation scheme are less pronounced. They are best suited to rendering at larger scales, where high image quality is essential.

### Eliminating the Segment's Hidden Side

On the side of a segment facing the viewer, breadthwise subdivision generates a semi-ring of 1, 2, or 4 vertices around each endpoint, depending on the level of quality used. If both sides of a segment may be visible, then a full ring of 2, 4, or 8 vertices is generated[10], and any backfacing polygons are removed through faceculling at a later stage. Whether the full ring is visible to the viewer depends on two criteria: the orientation of the segment relative to the view vector, and behaviour of the radius derivative. Figure 3.19 provides a geometric intuition for this dependency. For a given segment $\mathbf{ps}(t), t \,\epsilon\, [a, b]$, the full ring is generated if and only if the following condition holds at $t = a$ and $t = b$

$$\left(\frac{\mathbf{pos}'(t)}{\|\mathbf{pos}'(t)\|} \cdot \mathbf{view}(t)\right) rad'(t) > tol_{ring} \tag{3.27}$$



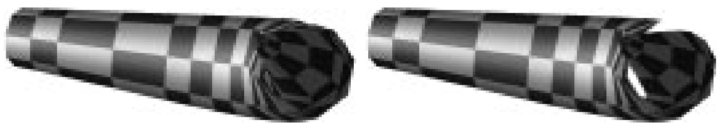
(a) Only one side visible (b) Both sides visible

Figure 3.19: Paintstroke orientation and radius derivative determine side visibility.

The nonnegative constant $tol_{ring}$ can be tuned to achieve a desired level of strictness in eliminating a partially hidden side. For example, if $tol_{ring} = 0$, then the full ring of polygons will be used whenever there is *any* variation in the radius and the segment's path is not perfectly orthogonal to the view vector—that is, whenever the opposite side is at all visible, even if the resulting image is so similar as to be indistinguishable from one created with just the semi-ring. Positive values for $tol_{ring}$ will cause the semi-ring to be used in place of the full ring when the side opposite the viewer is visible. We have found that $tol_{ring} \approx 2$ works well in practice. This value causes semi-rings to be used in

[10]The vertices along the edges are shared by both sides; this is why their number does not double.

place of full rings only when the half of the of the paintstroke further from the viewer is just barely visible. Thus it minimizes the polygon count without noticeably degrading the image.



(a) $tol_{ring} = 2$ (b) $tol_{ring} = 20$

Figure 3.20: A paintstroke rendered using two values for $tol_{ring}$, one reasonable and the other excessive.

### Determining the Polygon Vertices

To obtain a paintstroke polygon's vertices, we first determine their displacements from a point on the central path of the segment. These displacements are view-dependent vectors which all originate at $\mathbf{pos}(t)$, radiating outward as shown in Figure 3.21. We refer to them as the **out** vectors: $\mathbf{out}_{edge}(t)$ points to one of the segment's two lengthwise silhouette edges, while $\mathbf{out}_{centre}(t)$ reaches the breadthwise centre of the segment. The other two vectors, $\mathbf{out}_{mid_1}(t)$ and $\mathbf{out}_{mid_2}(t)$ are linear combinations of $\mathbf{out}_{edge}(t)$ and $\mathbf{out}_{centre}(t)$ that point to the angular midpoint between the centre and each edge. The full set of **out** vectors is depicted in Figure 3.22.

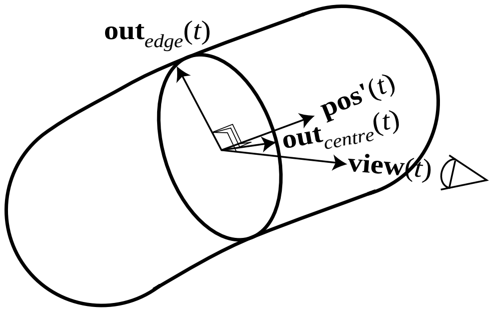


Figure 3.21: The view-dependent **out** vectors along the centre and edge of a segment.

$$\mathbf{out}_{edge}(t) = rad(t)\frac{\mathbf{view}(t) \times \mathbf{pos}'(t)}{\|\mathbf{view}(t) \times \mathbf{pos}'(t)\|} \tag{3.28}$$

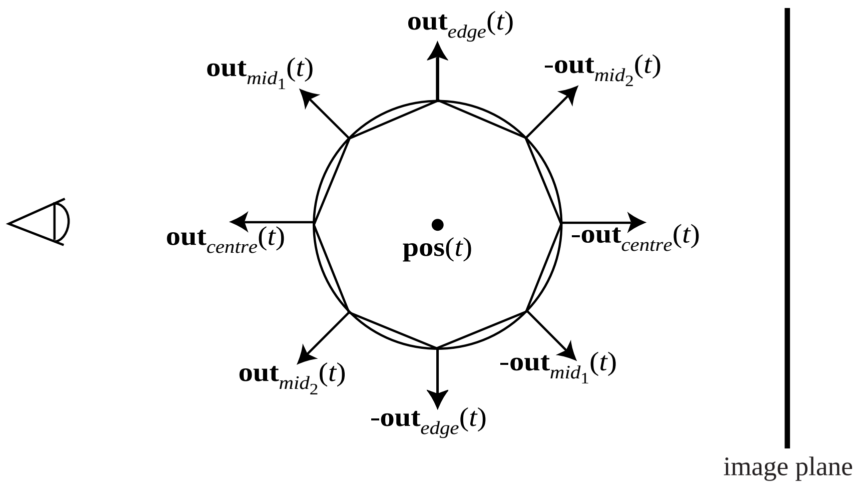

$\mathbf{out}_{edge}(t)$
$\mathbf{out}_{mid_1}(t)$
$-\mathbf{out}_{mid_2}(t)$
$\mathbf{out}_{centre}(t)$
$\mathbf{pos}(t)$
$-\mathbf{out}_{centre}(t)$
$\mathbf{out}_{mid_2}(t)$
$-\mathbf{out}_{mid_1}(t)$
$-\mathbf{out}_{edge}(t)$
image plane

Figure 3.22: The complete set of **out** vectors relative to the given viewing direction.

$$\mathbf{out}_{centre}(t) = rad(t)\frac{\mathbf{out}_{edge}(t) \times \mathbf{pos}'(t)}{\|\mathbf{out}_{edge}(t) \times \mathbf{pos}'(t)\|} \tag{3.29}$$

$$\mathbf{out}_{mid_1}(t) = \frac{1}{\sqrt{2}}\mathbf{out}_{centre}(t) + \frac{1}{\sqrt{2}}\mathbf{out}_{edge}(t) \tag{3.30}$$

$$\mathbf{out}_{mid_2}(t) = \frac{1}{\sqrt{2}}\mathbf{out}_{centre}(t) - \frac{1}{\sqrt{2}}\mathbf{out}_{edge}(t) \tag{3.31}$$

Vertices along the edges are now computed as $\mathbf{pos}(t) + \mathbf{out}_{edge}(t)$ and $\mathbf{pos}(t) - \mathbf{out}_{edge}(t)$. For quality-zero paintstrokes, these are the only vertices used. For higher quality levels, the centre vertex on the side of the segment facing the viewer is given by $\mathbf{pos}(t) + \mathbf{out}_{centre}(t)$, and the one one on the opposite side by $\mathbf{pos}(t) - \mathbf{out}_{centre}(t)$. For quality-two paintstrokes, the remaining four vertices are computed in the same manner, although the position of the vertices relative to the edges is inverted on the side opposite the viewer (i.e. if $\mathbf{pos}(t) + \mathbf{out}_{mid_1}(t)$ is between the centre of the viewer's side and one edge, then $\mathbf{pos}(t) - \mathbf{out}_{mid_1}(t)$ is between the centre of the opposite side and the *other* edge). The vertices are computed at both endpoints of the segment, yielding a ring (or semi-ring, if only the viewer's side is visible) of quadrilaterals.

### 3.2.4 Computing the Normals

Vertex normals for paintstroke polygons are readily obtained from the **out** vectors discussed in the previous section.[11] In fact, if a segment has no variation in radius, the

[11]As with the vertex positions, this determination is based on an orthographic projection.

**out** vector corresponding to a given vertex *is* the normal for that vertex. In the general case, each normal vector is equal to its corresponding $\mathbf{out}(t)$ vector plus an adjustment vector $\mathbf{adj}(t)$ in the direction of $\mathbf{pos}'(t)$, whose norm is determined by the derivatives of the radius and position. The **out** vectors define the breadthwise normal variation of a paintstroke (which is equivalent to that of a plain cylinder), while the **adj** vector represents the lengthwise normal variation, determined by the behaviour of the paintstroke's radius.

$$\mathbf{adj}(t) = -\frac{rad'(t)}{||\mathbf{pos}'(t)||^2}\,\mathbf{pos}'(t) \tag{3.32}$$

Assuming that the vector **out** has been normalized, we can justify this formula by considering the truncated cone in Figure 3.23 and reasoning as follows. Note that the vector **out** can represent any one of the **out** vectors of a paintstroke, since they are all orthogonal to the path $\mathbf{pos}'(t)$, represented by $\Delta\mathbf{pos}$.

$$\begin{aligned}
\frac{||\mathbf{adj}||}{||\mathbf{out}||} &= \frac{\Delta rad}{||\Delta\mathbf{pos}||} && (3.33)\\
||\mathbf{adj}|| &= \frac{\Delta rad}{||\Delta\mathbf{pos}||} && (3.34)\\
\mathbf{adj} &= ||\mathbf{adj}||\,\frac{\Delta\mathbf{pos}}{||\Delta\mathbf{pos}||} && (3.35)\\
&= \frac{\Delta rad}{||\Delta\mathbf{pos}||}\,\frac{\Delta\mathbf{pos}}{||\Delta\mathbf{pos}||} && (3.36)\\
&= \Delta rad\,\frac{\Delta\mathbf{pos}}{||\Delta\mathbf{pos}||^2} && (3.37)\\
\lim_{b-a\to 0}\mathbf{adj}(t) &= rad'(t)\,\frac{\mathbf{pos}'(t)}{||\mathbf{pos}'(t)||^2} && (3.38)
\end{aligned}$$

### Breadthwise Distribution of Normals

In projective rendering, the normals in the interior of a polygon are usually derived by bilinearly interpolating each component of the normals across the polygon's screen-space projection. This is the case with our polygon renderer.[12] A result of this bilinear inter-

[12]As we shall see in §4.3, our interpolation scheme is not an exact bilinear interpolation, but an approximation to it. For the purpose of this discussion, however, we can ignore this detail.

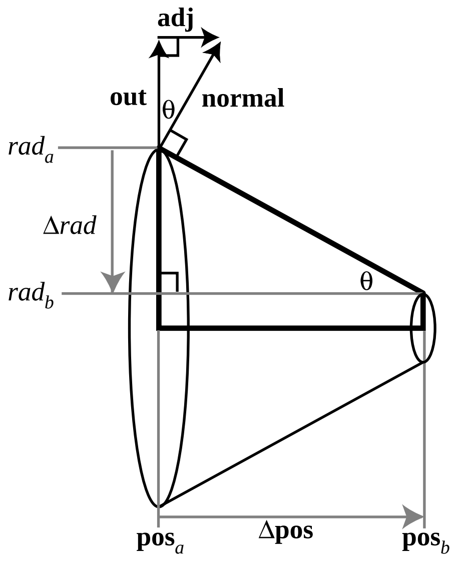


Figure 3.23: A cross-sectional view of a truncated cone.

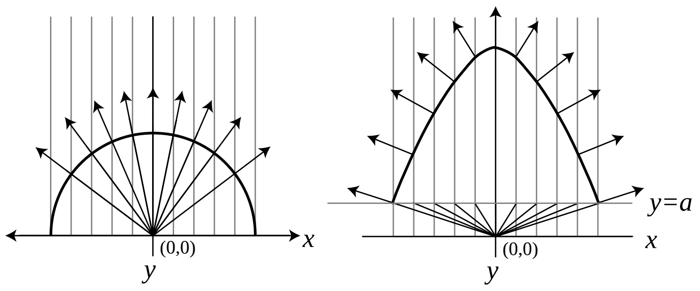
polation is that the rate of change (of direction) of an interpolated normal with respect to interpolation distance is smallest at the edges and greatest somewhere in the interior of a polygon. However, as Figure 3.24 illustrates, this is a very poor approximation of a generalized cylinder's breadthwise normal distribution. When a large amount of curvature is interpolated over a single polygon, the resulting image appears to have a ridge at the centre (see Figure 3.18) because the normals at that point are varying most rapidly instead of least rapidly, as they should for a true generalized cylinder. As one would expect, the more polygons are used to express a paintstroke's breadthwise normal variation, the better the approximation becomes. It is for this reason that shaded paintstrokes of higher quality levels have a rounder appearance than those of lower ones.

Any distribution of normals can be associated with a surface whose normals form the same distribution. The surface corresponding to a generalized cylinder's breadthwise normal distribution is a circular extrusion, or a cylinder. In contrast, the breadthwise normal distribution produced by interpolating over a single paintstroke polygon is that of a parabolic extrusion—a nontrivial fact, though easily justified. Referring to Figure 3.24(b), which represents a cross-sectional view of the polygon with the $x$-axis denoting the direction of interpolation and the $y$-axis the central normal vector, we reason as follows: the normal at any value of $x$ has slope $a/x$ where $a > 0$ represents the

$nudgefactor$, a value that determines the range of the normals to be interpolated. Thus, the tangent of the curve $f(x)$ whose normal at point $x$ is $a/x$ must be $-x/a$, yielding the equations

$$f'(x) = -\frac{x}{a} \quad (3.39)$$

$$f(x) = -\frac{x^2}{2a} + c \quad (3.40)$$



(a) Cylinder (b) Polygon

Figure 3.24: Breadthwise distributions of normals for a true cylinder and a linearly interpolated polygonal representation.

## Nudge Factors

For segments of quality zero, the normals along the lengthwise edges of a polygon are artificially nudged toward the normals of the centre vertices (even though the latter do not appear in a quality-zero segment). This is needed because the true edge normals are coplanar, so the subsequent interpolation between the edges would never have the (required) perpendicular component in the central direction—at the middle of the polygon, the normal would simply vanish instead of pointing orthogonally to the edges. The amount by which the edge normals are shifted toward the centre normal, specified by the nudge factor, determines both the range and distribution of the normals. A large nudge factor produces a smaller range but improves the cylindrical appearance of the distribution by reducing the height of the parabola whose shape it approximates. As the nudge factor approaches zero, the derivative of the normal's direction with respect to interpolation distance approaches infinity, causing severe spatial aliasing in the shading model.[13] A well-chosen value for the nudge factor produces a reasonable range of normals that do not vary too quickly at the centre, and are thus not prone to this type of aliasing.

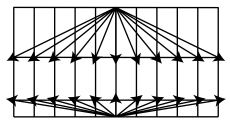
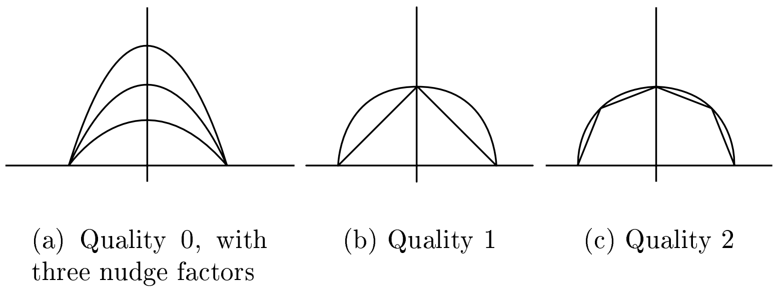


Figure 3.25: Normals interpolated using a small (bottom) and a larger (top) nudge factor.

For paintstrokes of quality level one or two, no explicit nudge factors are used, since they are already implied by the $90^0$ (for quality one) or $45^0$ (for quality two) of breadthwise normal variation across each polygon. The greatest difference in normal distributions is between levels zero and one. Accordingly, paintstrokes that automatically adjust their level of quality to their screen-projected size can produce popping artifacts when making a transition between these two levels. Transitions between levels one and two, while perceptible, are far less conspicuous. Naturally, the smoothness of any quality level transition also depends on the paintstroke's reflectance and its orientation relative to the light source and the viewer.



(a) Quality 0, with three nudge factors

(b) Quality 1

(c) Quality 2

Figure 3.26: Breadthwise normal distributions and their implied surface shapes for paintstrokes.

[13]Of course, the same type of aliasing will occur with a true cylinder, whose normals along the *edges* vary rapidly with respect to interpolation distance.

### 3.2.5 Endcap Generation

A paintstroke of quality one or two can be terminated at either end with an endcap, provided that the radius is greater than zero. This feature is made optional by classifying paintstrokes as either *open* (without endcaps) or *closed* (with endcaps), and allowing the model to specify the type of paintstroke used. The construction of the endcaps for quality-one and quality-two paintstrokes is shown in Figure 3.27. The essentially flat geometry of quality-zero paintstrokes precludes (and eliminates the need for) endcaps. A triangle fan is used for the endcaps in order to allow the central point to assume a normal parallel to the paintstroke's tangent vector $\mathbf{pos}'(t)$. A improved implementation for generating smoother endcaps at large sizes is outlined in Chapter 5.

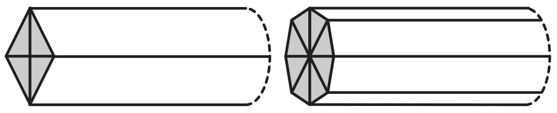

(a) Quality 1 (b) Quality 2

Figure 3.27: Endcap construction.

### 3.2.6 Problems with High Screen Curvature

It is the modeller's responsibility to ensure that the radius of a paintstroke does not exceed its radius of curvature in world-space (or equivalently, in eye-space). But even a well-behaved paintstroke that satisfies this requirement may, when transformed into screen-space, have a projected $(x, y)$-path whose radius of curvature is easily exceeded by the

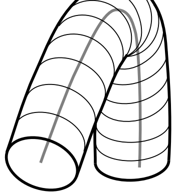


Figure 3.28: A paintstroke with high screen-projected curvature.

paintstroke's projected radius. This occurs when the direction of a curved paintstroke's path approaches that of the view vector, as exemplified in Figure 3.28.

This situation can give rise both to concave and complex (specifically, bowtie) polygons at all three quality levels. The latter are caused by the intersection of edges along the endpoints of a segment, and the former can occur when these edges do not quite intersect, but their endpoints are close. Since most polygon rendering algorithms, including ours, work only with simple convex polygons, these degenerate polygons pose a problem.

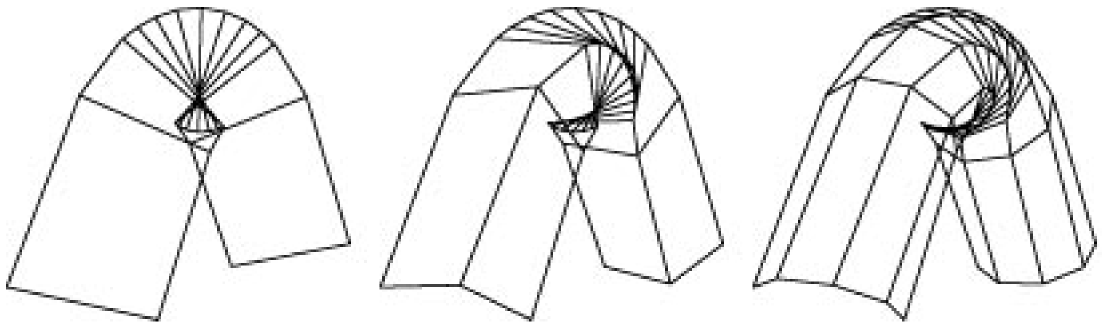

(a) Quality 0 (b) Quality 1 (c) Quality 2

Figure 3.29: Tessellation meshes producing bowtie polygons at all three levels of rendering quality for a paintstroke of sharp screen-space curvature.

Our implementation solves the problem in the traditional way, by splitting the offending polygon into an equivalent pair of triangles. As Figure 3.30 illustrates, splitting bowtie polygons can result in T-junctions, which many rendering systems cannot reliably handle [NDW93]. Hardware-based rendering engines typically use fixed-point arithmetic which may fail to represent the point in the middle of the T-junction as lying on the line joining the two points on either side of it. This can result in cracks intermittently opening up in the junction. In contrast, our polygon renderer handles T-junctions without difficulty because it represents vertex positions as double-precision floating-point values, rather than using the less accurate fixed-point representation.

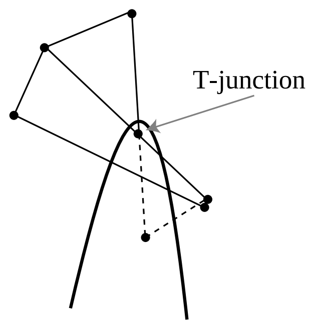


Figure 3.30: T-junction produced by splitting a bowtie polygon.

## 3.3 Special Rendering Effects

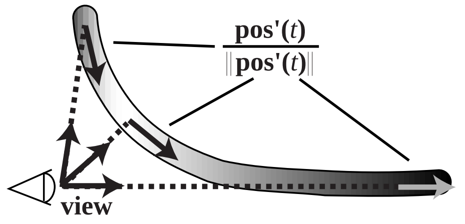
### 3.3.1 Lengthwise Opacity Variation

As mentioned in §3.1.4, the lengthwise opacity of a paintstroke segment varies according to the values of $op_{min}(t)$ and $op_{max}(t)$. The latter opacity is applied when the segment is viewed head-on, whereas the former is used when it is viewed orthogonally to its path. For intermediate cases, an opacity value is interpolated between these extremes, based on the dot product of the normalized tangent vector and the view vector.

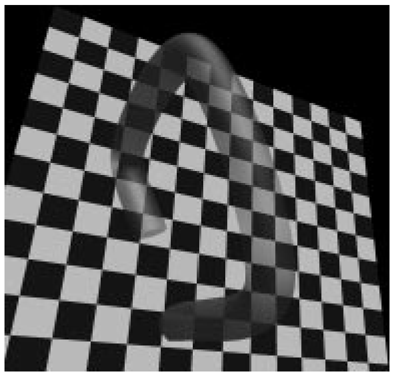
$$opacity_l(t) = \left|\mathbf{view}(t) \cdot \frac{\mathbf{pos}'(t)}{\|\mathbf{pos}'(t)\|}\right| op_{max}(t) + \left(1 - \left|\mathbf{view}(t) \cdot \frac{\mathbf{pos}'(t)}{\|\mathbf{pos}'(t)\|}\right|\right) op_{min}(t) \tag{3.41}$$

By exploiting this opacity interpolation, it is possible to simulate volume opacity, which varies according to the distance that penetrating light rays travel through a material. However, since we are basing the opacity on just a tangent vector, rather than any measure of distance, this effect is only a crude approximation, whose accuracy could be arbitrarily wrong. Nevertheless, the effect produces good results in practice, and is far less expensive than computing true volume opacity.

In order to quantify the accuracy of our opacity interpolation in a typical example, we have computed the true volume opacity of the bent tube shown in Figure 3.32. The light penetration distance $d$ is determined by the constants $r_1 = 10$, $r_2 = 11$, $L = 10$,



(a) The geometry behind our approximation



(b) The result

Figure 3.31: Lengthwise opacity variation simulating volume opacity.

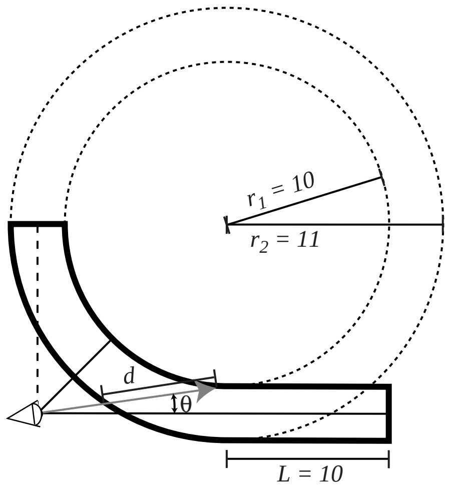


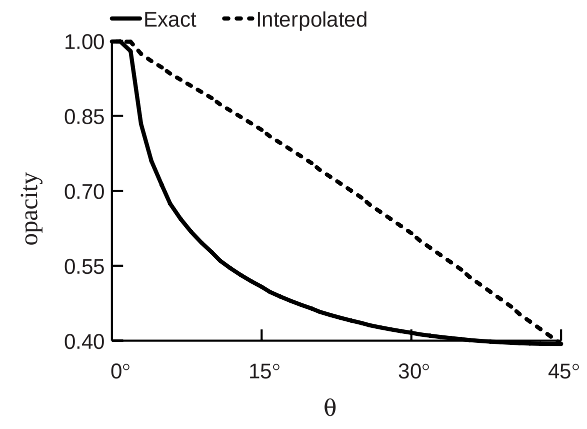
Figure 3.32: Tube used in the opacity comparison.

and by the parameter $\theta \,\epsilon\, [0^0, 45^0]$. We compute the true opacity using the formula

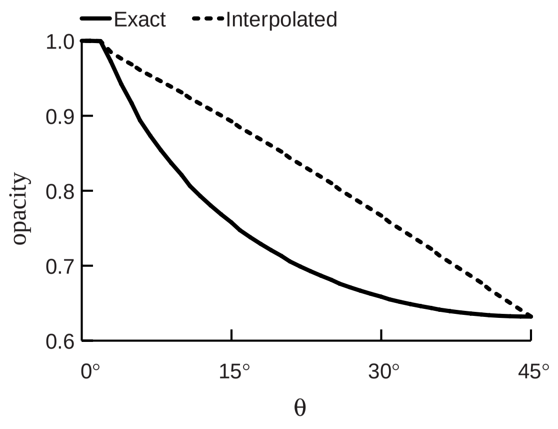
$$opacity = 1 - e^{-\rho\, d} \tag{3.42}$$

where $\rho$ represents the material density, which is assumed constant over the tube. We ran the comparison several times, using different values for $\rho$.

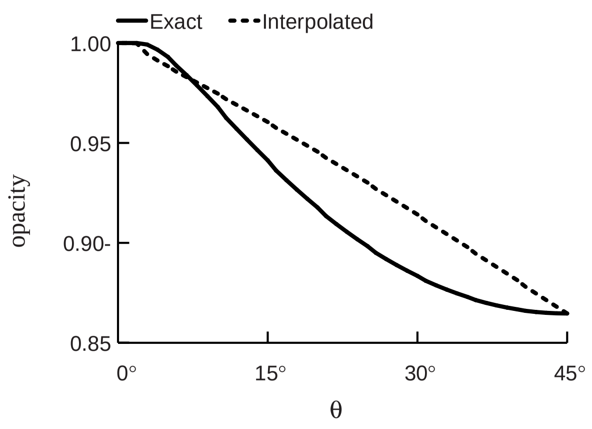
We have constructed this comparison so that the interpolant endpoints $op_{min}(t)$ and $op_{max}(t)$ are set to the true minimum and maximum opacity values (computed using Equation 3.42), which occur at $\theta = 0^0$ and $\theta = 45^0$. As the results in Figure 3.33



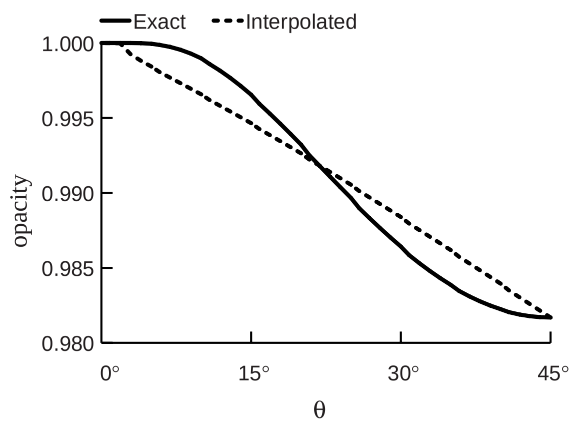
(a) $\rho = 0.5$



(b) $\rho = 1.0$



(c) $\rho = 2.0$



(d) $\rho = 4.0$

Figure 3.33: Exact vs. interpolated opacity values for $\theta \,\epsilon \,[0^0, 45^0]$.

illustrate, our approximation works best for fairly high values of $\rho$, at which the opacity becomes insensitive to the relatively long penetration distances introduced by $L$ (since penetration distance is not captured by our interpolation model).

### 3.3.2 Breadthwise Opacity Variation

The surface normals spanning the breadth of a paintstroke provide a simple and useful way of modulating the opacity across its breadth. This effect is achieved in each ring of polygons comprising a paintstroke segment by storing a dot product of the normal at each front-facing vertex with view vector. All the dot products within the segment are then divided by the maximum dot product, which occurs at the centre vertex. The quotient is stored for each vertex $\mathbf{v}_i$ as the parameter $s_i$. Given the vertex normal $\mathbf{N}_i$, the equations for $s_i$ and for the final opacity, $o_i$, are

$$s_i = \frac{\mathbf{N}_i \cdot \mathbf{view}}{\max_j(\mathbf{N}_j \cdot \mathbf{view})} \tag{3.43}$$

$$o_i = (1 - s_i)op_{edge} + s_i\, op_{centre} \tag{3.44}$$

This value is multiplied by the lengthwise opacity value from the previous section, to yield an overall opacity at each vertex.

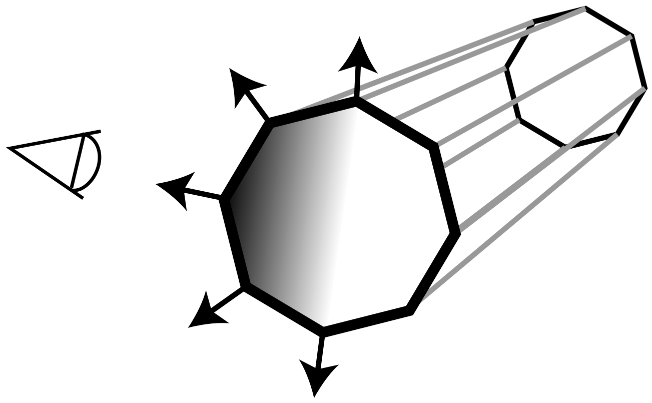


Figure 3.34: Implementation of breadthwise opacity variation.

A simpler way to implement this type of opacity variation would be to assign each vertex a fixed value for $o_i$, based on the position of the vertex along the width of the paintstroke. Thus, for a quality-one paintstroke, the vertex at the centre would have a value of one, and the vertices along the edges would have values of zero. Although this approach seems initially appealing because it is simple and inexpensive, it breaks down when a full ring of polygons becomes visible, as in Figure 3.20. In this case, the vertices along the top and bottom of the tapered segment (corresponding to the "edge" polygons, though, from this angle, they are nowhere near the edges) would have an opacity value of $op_{edge}$ while vertices along the left and right would have opacity values of $op_{centre}$. Our solution guarantees that the opacity will depend on the angle of the surface normal relative to the viewer, eliminating this anomaly.

Breadthwise opacity variation can be used to produce fuzzy paintstrokes (by using a high value for $op_{centre}$ and a low value for $op_{edge}$) or to simulate the Fresnel effect for streams of water or icicles (by doing the reverse). Applications for the former include modelling wisps of hair or blobs of smoke. An example of the latter is shown in Figure 3.36.

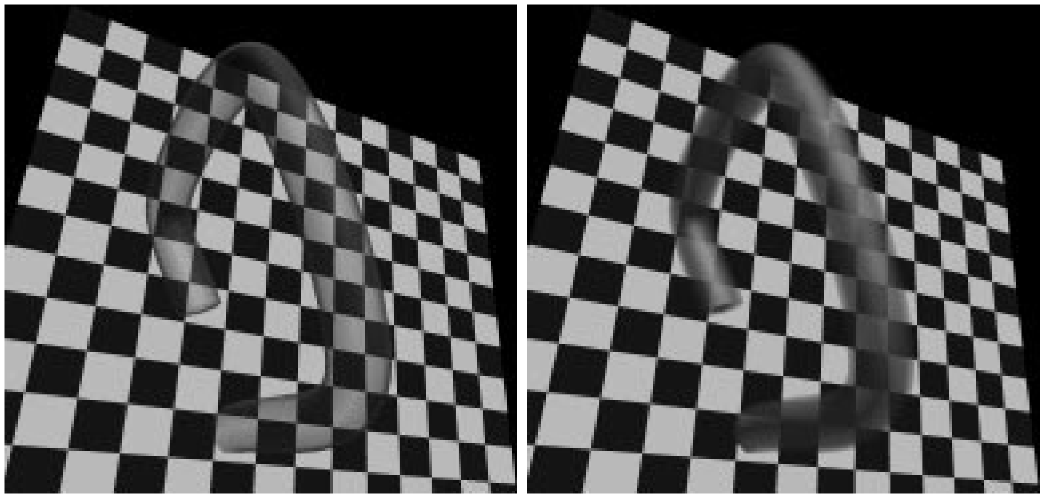


Figure 3.35: Two types of breadthwise opacity variation.

### 3.3.3 Global Shading Algorithm

The surface normals derived in §3.2.4 enable us to apply accurate local shading to each individual paintstroke. However, they fail to take into account the shadows that paintstrokes can cast onto themselves and other paintstrokes. While this problem could be rectified by explicitly computing shadows for all paintstrokes, as with shadow buffering [Wil78], this approach would significantly increase rendering time and memory requirements. Our solution, while not as general as true shadow calculation, produces good results for homogeneous layers of paintstrokes covering a roughly convex shape. It is

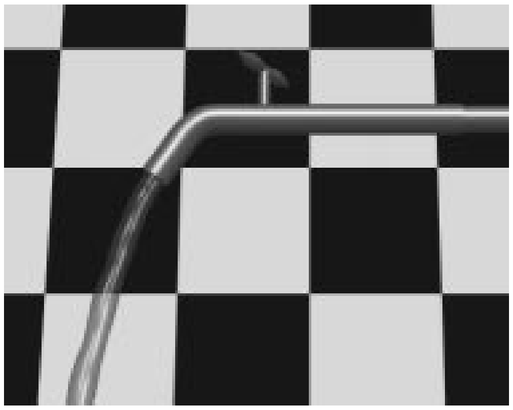


Figure 3.36: Breadthwise opacity variation used to simulate the Fresnel effect in a stream of water.

similar in spirit to the one proposed by Reeves and Blau [RB85].

Each control point of a paintstroke has associated with it a global normal $\mathbf{N}_{gl}$ and a global depth value $d_{gl}$, as shown in Figure 3.37. The former indicates the direction of the global surface to which the control point belongs, and the latter the relative depth from that surface, expressed as a value between zero (on the surface) and one (maximally distant from the surface). Note that this is unrelated to the true position of the control point—this model is based only on the global normal and depth value. The global normal value is entered by the user in world-space and is automatically transformed into eye-space (as is the position vector) when a paintstroke is rendered. The depth values are constant.

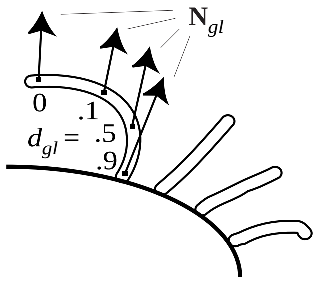


Figure 3.37: Global normals and depth values assigned to the control points of a paint-stroke.

At any control point, the estimated amount of light penetration, $p$, relative to a light direction $\mathbf{L}$ is given by the following equation, with $\alpha$ and $\beta$ as defined below. The vectors

$\mathbf{L}$ and $\mathbf{N}_{gl}$ are assumed normalized.

$$\begin{aligned} p &= \frac{1}{2}\left[\sqrt{(\alpha\beta)^2 - \alpha^2 + 1} - \alpha\beta\right] \\ \alpha &= 1 - d_{gl} \\ \beta &= \mathbf{N}_{gl} \cdot \mathbf{L} \end{aligned} \tag{3.45}$$

This penetration value, ranging between zero and one, has a specific geometric interpretation. As illustrated in Figure 3.38, we construct a unit sphere centred at the origin. The position of the control point in this model is defined to be $(1 - d_{gl})\mathbf{N}_{gl}$, which always lies within the sphere. Now we extend a line segment in the direction of the light vector $L$, joining some point on the surface of the sphere to the control point within. The length of this line segment represents the penetration value at the control point. Since the length could vary from 0 to 2, we multiply it by $\frac{1}{2}$ in order to normalize it.

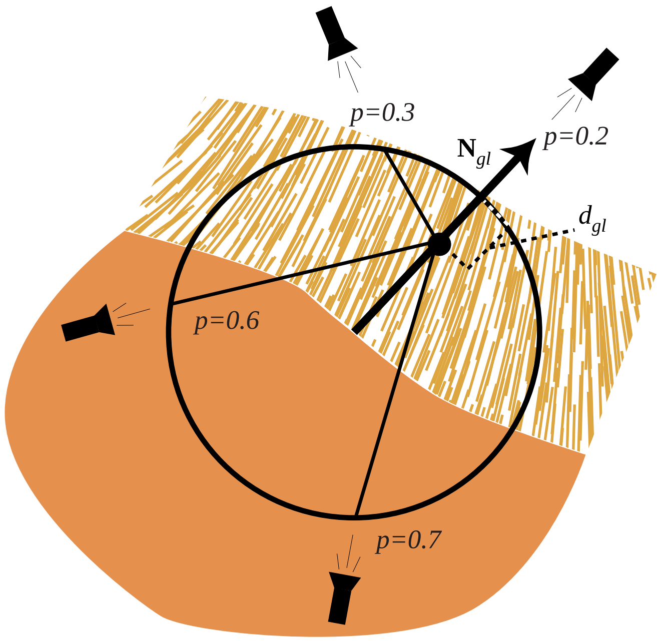


Figure 3.38: Penetration values at various light angles for a given global normal and depth.

When the penetration at a given control point $\mathbf{cp}_i$ is determined, the reflectance vector $\mathbf{r}_i$ for that control point is scaled down according to a negative exponential function involving its penetration value $p_i$ and a user-defined material density factor $\rho$.

$$\mathbf{r}_i := \mathbf{r}_i \, e^{-p_i \rho} \tag{3.46}$$

This global shading model works well when a large number of control points are uniformly distributed over a convex volume. This is usually the case with fur and foliage, so this method is particularly useful in modelling these.

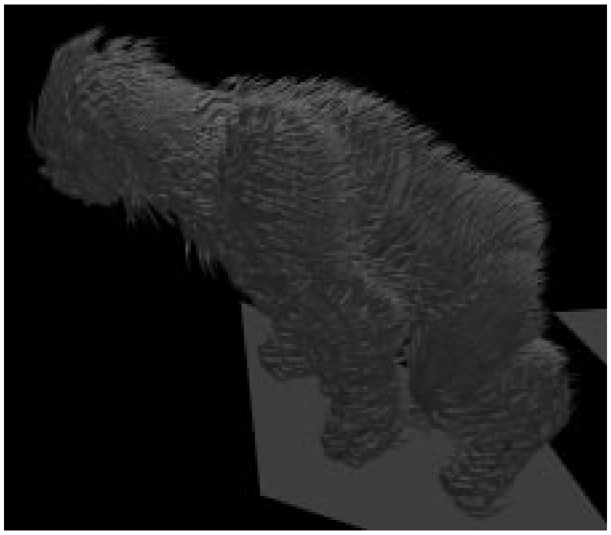


Figure 3.39: Example of a global shading effect.

## 3.4 Summary

In this chapter we have discussed three important aspects of the paintstroke primitive: its representation, its tessellation, and the unique rendering features that are made possible by the above. We have also examined some of the strengths and limitations of the different quality levels of paintstrokes, a topic that will be revisited in Chapter 5.

# Chapter 4

# Rendering Polygons Using the A-Buffer

In the preceding chapter, we have examined the overall structure of paintstrokes, and how they are tessellated into polygons. This chapter explains how our rendering engine converts these and other polygons into screen images. Unless otherwise noted, all polygons discussed in this chapter are of the simple and convex variety—an assumption that greatly simplifies the task of rendering them and sets the stage for a number of significant algorithmic optimizations. As we have seen in Chapter 3, complex or concave polygons can always be decomposed into simple convex ones, so they can still be rendered, albeit with some extra work.

## 4.1 Overview of the A-Buffer

The A-Buffer is a framework that provides an efficient way to represent and composite rasterized images with subpixel accuracy. It accomplishes this through a mechanism that appropriately blends their colour and opacity values according to subpixel coverage, effectively applying a box filter over each pixel. This filtering provides fast, high-quality antialiasing of the resulting image. In a typical projective-rendering graphics pipeline, the A-Buffer's scope lies within the last stage of the screen-space phase, where rasterization occurs.

In the A-Buffer framework, polygons are rendered in two stages: first they are rasterized into a set of *fragments*—simplified subpixel resolution images, each covering a single pixel. A more rigorous definition of fragments will be presented shortly. Once all the polygons have been rasterized, the fragments over each pixel are *blended*, or composited, to generate a final colour value that is assigned to that pixel. Thus, polygons are not completely rendered one at a time, as they are with most other renderers; they *all* have to be rasterized before a single pixel can be drawn to the screen.[1] As mentioned, the A-Buffer performs spatial antialiasing by applying a pixel-sized box filter to each fragment. Thus, each fragment is treated as an individual supersampled subpixel-resolution image (filtered independently of the others) that contributes to the final colour of the pixel it covers, and only that pixel.

There are numerous implementations of the A-Buffer in circulation. While most of these are software-based, some have been designed to work in hardware [SS93], though these are rarely found in practice. Ours is based on the original implementation by Loren Carpenter [Car84], although it fundamentally differs in two aspects: the way that fragments are generated, and the way that intersecting fragments are blended. Similar work based on using bitmaps to approximate pixel coverage, as described by Fiume and Fournier in [FFR83], predates the A-Buffer. Their approach, however, does not keep track of multiple fragments per pixel, and as a result, blended pixels are dependent on the order in which polygons are processed. Consequently, this solution provides less accurate results for overlapping surfaces, and as presented cannot adequately handle transparency, although it is faster and more memory-efficient than the A-Buffer.

Thus far we have referred to the A-Buffer only in the context of rendering polygons. Although that is indeed its most common application, the A-Buffer can be used in imaging a variety of non-polygonal primitives (e.g. lines, ovals, and even text) using different methods of rasterization (e.g. scanline, ray-tracing, bilinear interpolation). This variety of incarnations exists because the A-Buffer does not impose a particular rasterization algorithm, but only specifies the *format* of the fragments it processes. Hence, any rasterization scheme can in principle be made to work within the framework of the A-Buffer. In fact, more than one could be used for different sets of primitives within the same scene, where the fragments are generated in different ways, but processed by a single A-Buffer

[1]It is noteworthy that the A-Buffer blends each pixel's fragments independently of the other pixels'. Although our current implementation does not take advantage of it, this theoretically allows for very efficient parallelization of the blending process.

blending routine. In this chapter, however, we shall restrict our scope to simple convex polygons, which serve as the basis of our rendering engine.

## 4.2 Fragments

Fragments are data structures that store a simplified representation of a polygon's projected image over a particular pixel, rasterized at "quasi-subpixel" resolution, as defined below. Each pixel is allocated zero or more fragments, one for each polygon whose interior contains any part of the pixel.[2] The fragments are stored in a list, ordered by the the values of their $\texttt{Z}_{\texttt{min}}$ fields which represent their minimum (or closest) $z$-values.

Storing depth information for each pixel touched by a rasterized primitive is an approach that the A-Buffer shares with its predecessor, the Z-Buffer. Complementing this similarity are two noteworthy differences: first, the A-Buffer's fragments contain a great deal more information than the simple depth values used in the Z-Buffer; and second, unlike the Z-Buffer, which keeps only the foremost depth value at each pixel, the A-Buffer stores fragments for *all* polygons, even those masked by closer ones. It is easy to see from these differences that the A-Buffer consumes considerably more memory than the Z-Buffer.

A minimal A-Buffer implementation such as ours represents the fragment's data structure with the following fields:

1. The coverage mask, `mask`
2. The colour, `colour`
3. The opacity, `opacity`
4. The minimum and maximum screen-space $z$-values, $\texttt{Z}_{\texttt{min}}$ and $\texttt{Z}_{\texttt{max}}$
5. The tag identifier, `tag`

Note that there is no information about surface normals stored in a fragment. That is because the local shading model is incorporated in the rasterization routine, so that a

[2]More precisely, any centre of a subpixel within the pixel. We have more to say about subpixels in §4.2.1.

fragment's `colour` field contains the *shaded* colour, as determined by the desired local illumination model.

We used the term "quasi-subpixel" to describe the resolution at which fragments are rasterized. That is because the fragment's supersampled image is simplified in the following ways: (1) all the subpixels share the same colour and opacity values, and (2) just the minimum and maximum $z$-values over the entire fragment are computed. Hence, only the pixel's *coverage* is stored at true subpixel resolution within the fragment.

### 4.2.1 The coverage mask field, `mask`

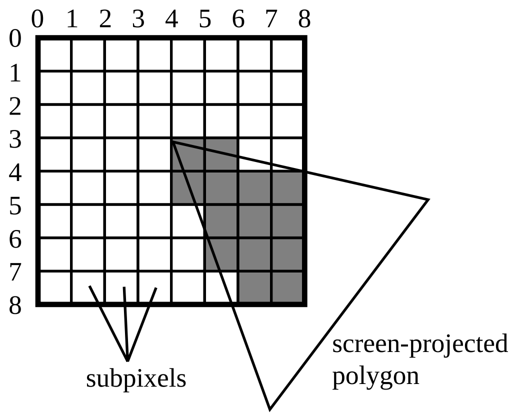


Figure 4.1: A coverage mask produced by rasterizing a small polygon, with a superimposed grid indicating subpixel positions.

The field `mask` is a two-dimensional bitmap that stores a screen-projected polygon's subpixel-resolution image over a pixel. Each bit in the coverage mask is associated with one subpixel, indicating whether the latter is covered. Subpixels are arranged in a uniform rectangular grid spanning the pixel. The shape of each subpixel may or may not match the shape of the pixel, depending on the subpixel resolution used. In [Car84], Carpenter uses an $8 \times 4$ resolution, making his subpixels rectangular (assuming square pixels). Our version employs a resolution of $8 \times 8$, which allows the bitmap to be stored in a single 64-bit integer variable, the rows being packed in consecutive 8-bit intervals. This permits an efficient implementation of set operations such as unions and intersections by means of bitwise operations on registers. These operations, as we shall see shortly, are essential to the fragment-blending portion of the algorithm. The amount of time spent generating fragments (and to a lesser extent, blending them) grows with the resolution of the coverage mask, but so does the quality of the antialiasing, due to the greater precision with which the coverage of a pixel can be approximated.

### 4.2.2 The tag identifier field, `tag`

This field is an optional identifier that may assigned to a fragment upon its creation. Although our current implementation does not make use of the `tag` field, its traditional role facilitates merging "compatible" fragments in order to free up memory, as described in [Car84]. To qualify as compatible, a pair of fragments must belong to the same pixel, be adjacent in depth, and come from polygons tiling a single surface. All the polygons tessellated from this surface share an identifier that is placed in the `tag` field of the fragments they generate, providing a simple and inexpensive test for merging compatibility. Another potential application for the `tag` field will be described in §5.1.3.

### 4.2.3 The colour field, `colour`

The colour of a fragment is stored as a set of integer $r, g, b \epsilon [0, 255]$ representing the quantized red, green, and blue components. Because the field `mask` does not store any colour information about subpixels[3], the image it contains is monochrome—all the filled subpixels share a single average colour value that is stored in the field `colour`. Allowing only one colour per fragment greatly simplifies the blending formulas while still permitting smooth colour variation across multiple pixels, since fragments are never bigger than a single pixel.

### 4.2.4 The opacity field, `opacity`

The opacity of a fragment is a floating-point value between zero and one, with zero corresponding to complete transparency (and therefore invisibility), and one to complete opacity. As with `colour`, the value of `opacity` applies to all the subpixels of a fragment.

[3]In more colloquial usage, a bitmap is often said to store binary colour values. This is not strictly true; it stores values that may represent two different colours, but the identity of those colours is not explicitly encoded in the bitmap.

### 4.2.5 The minimum and maximum $z$-value fields, `Zmin` and `Zmax`

The fields $\texttt{Z}_{\texttt{min}}$ and $\texttt{Z}_{\texttt{max}}$ contain the minimum and maximum screen-space $z$-values of a fragment, respectively. Unlike the Z-Buffer, which uses a single $z$-value, the A-Buffer attempts to antialias intersecting fragments. Given a pair of such fragments, it needs to determine what portion of each one is visible to the viewer. The $\texttt{Z}_{\texttt{min}}$ and $\texttt{Z}_{\texttt{max}}$ values of both fragments are used to estimate this, as described in §B.2.7.

## 4.3 Rasterization

Having examined the fragment's structure in some detail, we now turn to the task of generating fragments, which is performed by the rasterization algorithm. Our implementation rasterizes each polygon by linearly interpolating a vector of values over the interior and producing a fragment at each pixel. The vector, called the *interpolant vector*, consists of the following elements:

- screen-space position vector
- eye-space normal vector
- colour
- opacity
- $k_d$, the diffuse shading coefficient
- $k_s$, the specular shading coefficient[4]

We have developed two methods of interpolating this vector, bilinear interpolation and constant-increment interpolation, ultimately incorporating the latter into our rendering engine.

### 4.3.1 Bilinear vs. Constant-Increment Interpolation

Bilinear interpolation linearly scans the components of the interpolant vector along the polygon's left and right edges, starting at the top and proceeding downward. At each

[4] $k_s$ and $k_d$ are used in the shading model and will be explained in §4.3.6.

vertical step during the scanning, a horizontal interpolation is made between the interpolated points along the edges, scanning the breadth of the polygon. By thus composing the vertical interpolation with the horizontal, the full area of the polygon is scanned. An efficient way to perform such interpolations is to compute an increment for the interpolant vector corresponding to one horizontal or vertical step, and to apply the corresponding increment at each step during the scanning. Computing the increment requires subtracting the endpoint values and dividing by the (horizontal or vertical) distance between them. Although quite simple, this computation—particularly for the horizontal increment—can account for a significant portion of a polygon's rendering time, because it is performed at each subpixel row in the polygon.

Because the appropriate increment is re-computed for each row and for each pair of vertices along the edges, bilinear interpolation is suitable for approximating a nonlinear function from a set of samples. While this is a useful feature for many applications, a less expensive method exists for strictly linear functions: constant-increment interpolation. This technique is similar to bilinear interpolation, except that it applies *constant* horizontal and vertical increments when scanning anywhere within the polygon. Except for the $x$- and $y$- values (which are trivial to interpolate within the polygon), a plane equation is constructed for each scalar subcomponent of the interpolant vector, using its value at each vertex as a $z$-value for the plane, together with the vertex's screen-space $x$- and $y$-values. The plane equation is then used to compute the constant horizontal and vertical increments to be used during the scanning, as outlined in the following section. This represents a significant cost reduction over bilinear interpolation, which needs to recompute the horizontal increment many times for each polygon. Moreover, the eliminated recomputation is relatively expensive, involving one division and, for each component of the interpolant vector, one subtraction, multiplication, and assignment operation.

### 4.3.2 Computing the Plane Equations

To obtain the plane equation for a given quantity we wish to interpolate over a polygon, we construct a geometric analogue of the polygon by replacing the screen-space $z$-value at each vertex with the interpolated quantity at that vertex. The normal of this new polygon will provide the plane equation we seek. If the new vertices are coplanar, we could use any pair of edges to determine the normal. However, recognizing that the vertices may not all be coplanar, we compute an average normal direction by treating the polygon as a fan of triangles joined at one of the vertices, as shown in Figure 4.2. The choice of the common vertex is arbitrary, as will be evident from the ultimate formula we derive. We add the (non-normalized) normal vector of each triangle, obtained by the cross product of the two edges that share the common vertex, taken in a consistent cyclical order.[5] The sum of the normals represents an average normal direction of the set of triangles comprising the original polygon, with each triangle's contribution to the normal weighted by its area. This weighting is borne in the magnitude of each normal, and that is why we omit normalizing them.

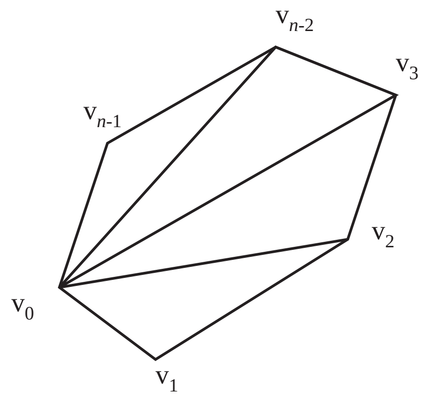


Figure 4.2: Fan of triangles used in computing the average normal.

Given $n$ 3-dimensional vertices, $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{n-1}$, we compute the sum of the cross products, $\mathbf{N}$, as follows:

$$\begin{aligned}
\mathbf{N} &= (\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0) && (4.1)\\
&\quad + (\mathbf{v}_2 - \mathbf{v}_0) \times (\mathbf{v}_3 - \mathbf{v}_0) && (4.2)\\
&\quad + \ldots && (4.3)\\
&\quad + (\mathbf{v}_{n-2} - \mathbf{v}_0) \times (\mathbf{v}_{n-1} - \mathbf{v}_0) && (4.4)\\
&= \mathbf{v}_1 \times \mathbf{v}_2 - \mathbf{v}_1 \times \mathbf{v}_0 - \boxed{\mathbf{v}_0 \times \mathbf{v}_2} + \mathbf{v}_0 \times \mathbf{v}_0 && (4.5)
\end{aligned}$$

[5]Note that any pair of edges in a triangle would yield the same cross product, provided their cyclical ordering was the same, so our choice is legitimately arbitrary.

$$\boxed{\text{Cancelled terms}} \quad +\mathbf{v}_2 \times \mathbf{v}_3 - \boxed{\mathbf{v}_2 \times \mathbf{v}_0} - \boxed{\mathbf{v}_0 \times \mathbf{v}_3} + \mathbf{v}_0 \times \mathbf{v}_0 \tag{4.6}$$

$$+\mathbf{v}_3 \times \mathbf{v}_4 - \boxed{\mathbf{v}_3 \times \mathbf{v}_0} - \boxed{\mathbf{v}_0 \times \mathbf{v}_4} + \mathbf{v}_0 \times \mathbf{v}_0 \tag{4.7}$$

$$+ \ldots \tag{4.8}$$

$$+\mathbf{v}_{n-2} \times \mathbf{v}_{n-1} - \boxed{\mathbf{v}_{n-2} \times \mathbf{v}_0} - \mathbf{v}_0 \times \mathbf{v}_{n-1} + \mathbf{v}_0 \times \mathbf{v}_0 \tag{4.9}$$

$$= \mathbf{v}_1 \times \mathbf{v}_2 - \mathbf{v}_1 \times \mathbf{v}_0 \tag{4.10}$$

$$+\mathbf{v}_2 \times \mathbf{v}_3 \tag{4.11}$$

$$+\mathbf{v}_3 \times \mathbf{v}_4 \tag{4.12}$$

$$+ \ldots \tag{4.13}$$

$$+\mathbf{v}_{n-2} \times \mathbf{v}_{n-1} - \mathbf{v}_0 \times \mathbf{v}_{n-1} \tag{4.14}$$

$$= \mathbf{v}_0 \times \mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2 + \ldots + \mathbf{v}_{n-2} \times \mathbf{v}_{n-1} + \mathbf{v}_{n-1} \times \mathbf{v}_0 \tag{4.15}$$

If we then represent each vector $\mathbf{v}_i$ using its components $x_i, y_i, z_i$, we can expand out the cross products to yield the ultimate formulas used in computing $\mathbf{N}$. Letting $\mathbf{v}_n$ equal $\mathbf{v}_0$[6], we can write:

$$\mathbf{N}_x = \sum_{i=0}^{n-1} (y_i z_{i+1} - z_i y_{i+1}) \tag{4.16}$$

$$\mathbf{N}_y = \sum_{i=0}^{n-1} (z_i x_{i+1} - x_i z_{i+1}) \tag{4.17}$$

$$\mathbf{N}_z = \sum_{i=0}^{n-1} (x_i y_{i+1} - y_i x_{i+1}) \tag{4.18}$$

These three normal components respectively correspond to the coefficients $A, B, C$ of the plane equation $Ax + By + Cz + D = 0$. The value $|\mathbf{N}_z|$ also equals twice the polygon's area, a quantity we will make use of shortly. The coefficient $D$ is obtained by substituting the average $(x, y, z)$ value of the vertices into the equation. The constant increments for one unit (in our case, one subpixel) of horizontal or vertical movement are then given by the expressions $-A/C$ and $-B/C$, respectively.

[6]In allowing this, we observe that the subscripts form a ring of modulo $n$ arithmetic, establishing the arbitrariness of the starting vertex $\mathbf{v}_0$ as promised.

### 4.3.3 Problems with Nonplanar Polygons

Because it is assumes linearity in the function to be interpolated, constant-increment interpolation has a potential shortcoming when applied to polygon scan-conversion: A polygon that is nonplanar in any component of the interpolant vector will yield an interpolant that fails to span the exact range of values spanned by the vertices. This can produce a discontinuity along the shared edge of adjacent polygons, one of which is nonplanar. In contrast, bilinear interpolation fully interpolates the vertex values across even a nonplanar polygon, so that such discontinuities cannot occur. However, when applied to such polygons, the results of bilinear interpolation become dependent on the polygon's orientation with respect to the scanning direction, as shown in Figure 4.3. This presents a problem, because the direction of the horizontal scanning is fixed relative to the viewport, and therefore varies relative to the polygon's frame of reference. While orientation-dependency is undesirable effect, it is much less noticeable than the edge discontinuities arising from the constant-increment method.

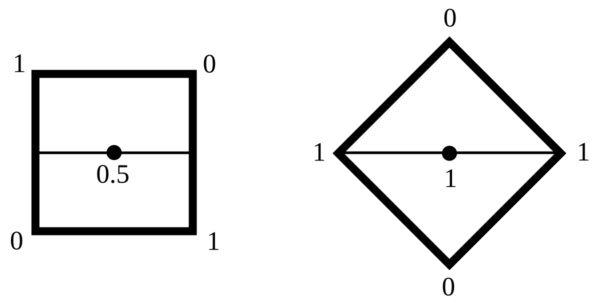


Figure 4.3: Orientation-dependency of bilinear interpolation for a nonplanar polygon.

The easiest and probably most common way of dealing with nonplanar polygons is to subdivide them into triangles, which are trivially planar in all interpolated quantities. While this method does indeed remove the discontinuities of constant-increment interpolation and also the orientation-dependency of bilinear interpolation, it would negate many of the savings afforded by tessellating paintstrokes into a small number of large polygons.

### 4.3.4 The Dynamic Triangulation Algorithm

Observing that the majority of polygons generated by our paintstroke tessellation scheme did not manifest the artifacts caused by nonplanarity, we opted to use the more efficient constant-increment method. Our experience has shown that for small polygons (under 8 pixels in area), discontinuities between adjacent polygons arising from nonplanarity are virtually undetectable—not because they cease to exist, but because the eye cannot discern them. To deal with larger polygons, we implement a dynamic triangulation mechanism, which uses a planarity test to determine whether subdivision is necessary. If it is, the polygon is subdivided into a fan of triangles, one per vertex. As Figure 4.4 illustrates, the triangles all share a new central vertex, which is obtained by averaging the vertices of the original polygon.[7]

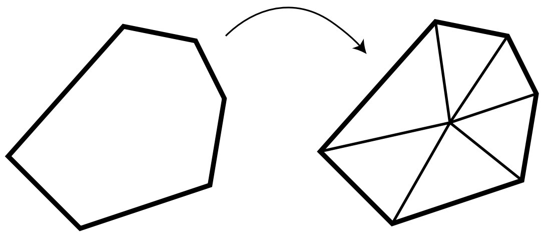


Figure 4.4: A polygon subdivided by the dynamic triangulation algorithm.

Because our constant-increment interpolation produces a least-squares linear solution, the deviations of the vertices from the interpolant plane are a good measure of the former's planarity. The maximum absolute deviation among the vertices is multiplied by an *ad hoc* linear function of the polygon's area (one function per component). If the product exceeds a given threshold, the polygon is triangulated.

The purpose of the area functions is to lower the triangulation threshold for larger polygons, where the effects of nonplanarity are particularly conspicuous. Each function is defined using a pair of values, $c_1$ and $c_2$, that specify the polygon areas for which the function reaches its minimum, 0, and its maximum, 1. An example appears in

[7]Contrary to intuition, we do not re-normalize the normal vector of the averaged vertex, as this would yield a noticeably different shading profile from the original polygon. Because we want the triangulated polygons to blend seamlessly with the ones that are not triangulated, we must ensure that the normals across both types behave similarly. Preserving the original (i.e. non-normalized) average normal at the centre of the former type maintains this similarity.

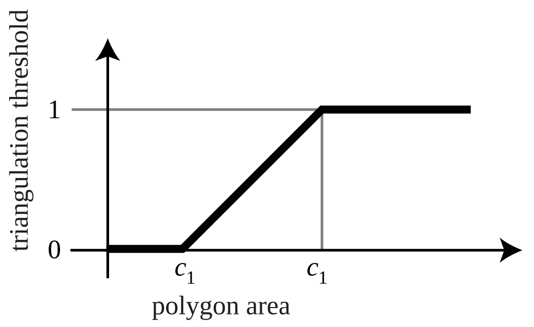
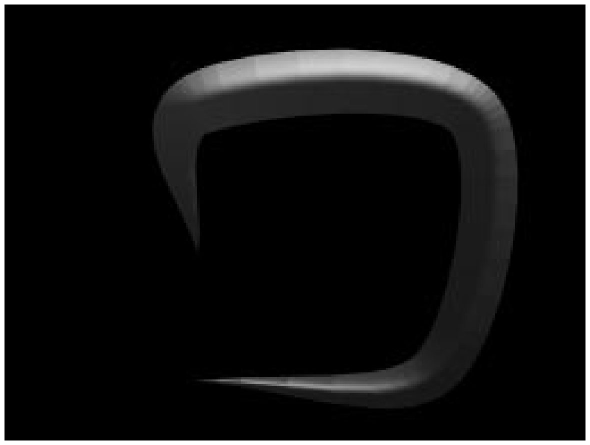


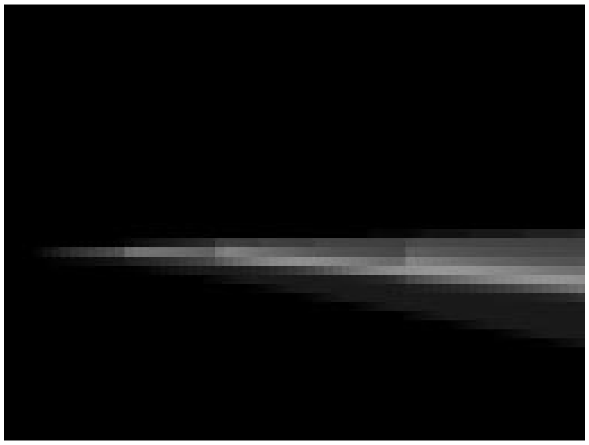
Figure 4.5: An area function, as defined by $c_1$ and $c_2$.

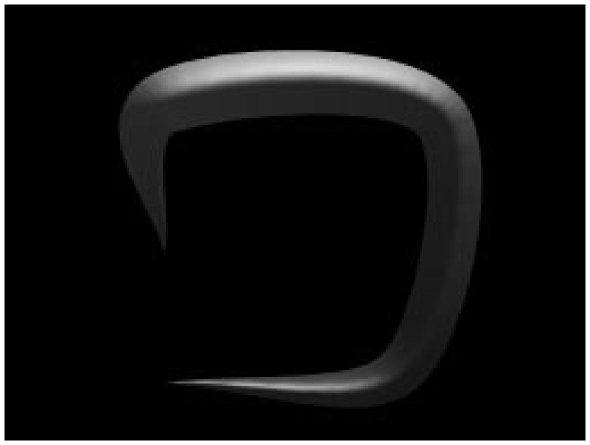
Figure 4.5. Using one such function per component allows us to vary the strictness of the triangulation criterion for different components.

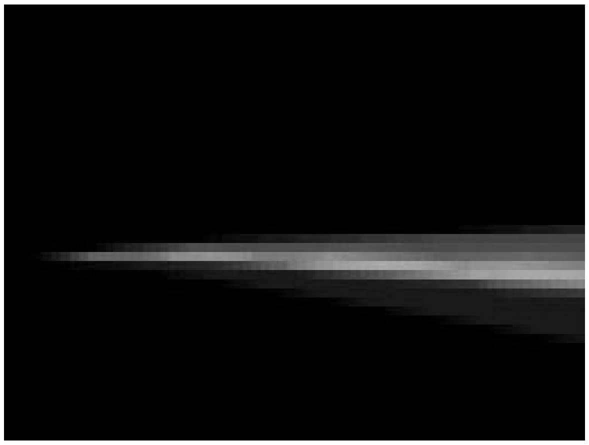
The advantage of our triangulation criterion is that it permits small polygons to be rendered very quickly, unencumbered by needless triangulation. Assuming that paintstrokes are used at a reasonably small scale, the vast majority of the polygons they generate will pass below the triangulation threshold. As the size of a paintstroke increases, more and more of its nonplanar polygons will become triangulated, gradually degrading performance but maintaining the image quality. When the amount of triangulation becomes high (e.g. 50% or more) it can be reduced by switching to a higher quality level paintstroke, which reduces both the size and the nonplanarity of the paintstroke polygons. In such a case, the switch provides a higher quality image, typically at only a slightly higher polygon count (after triangulation). While this switch is normally done automatically, it can be disabled by the user to eliminate potential popping artifacts that are possible when a paintstroke changes quality levels.

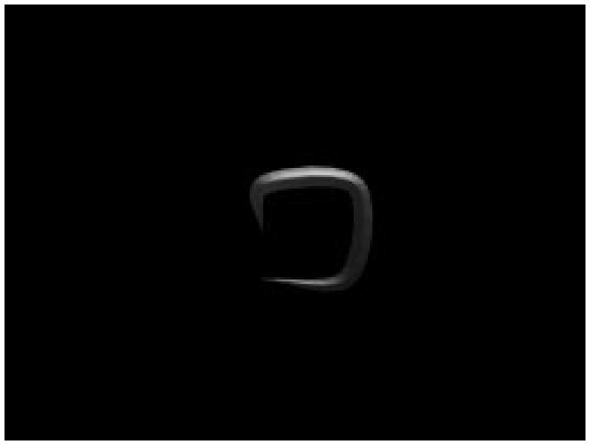
Figure 4.6 shows small and large level-zero paintstrokes, rendered with and without dynamic triangulation. In Figure 4.6(a), a total of 50 paintstroke polygons were generated. When dynamic triangulation was activated, 37 of them were triangulated (each into 4 triangles), yielding a total of 161 polygons for Figure 4.6(c). The paintstroke in Figure 4.6(e) consists of 20 polygons, which did not require triangulation, thus yielding identical images with and without dynamic triangulation.


(a) Large paintstroke without dynamic triangulation


(b) Close-up of the tail


(c) Large paintstroke with dynamic triangulation


(d) Close-up of the tail


(e) Small paintstroke with or without dynamic triangulation

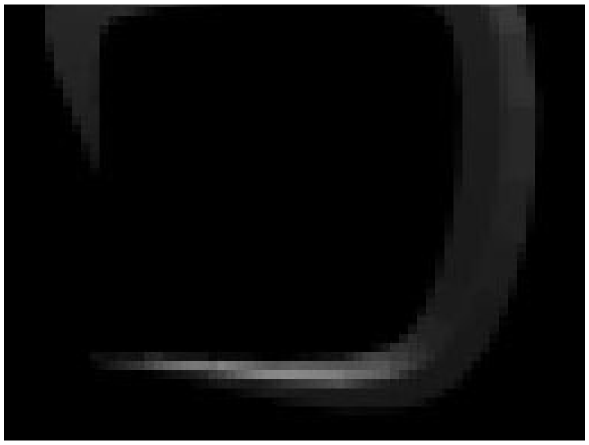

(f) Close-up of the tail

Figure 4.6: Paintstrokes rendered with and without dynamic triangulation.

### 4.3.5 The Rasterization Algorithm

Operating at an $8 \times 8$ subpixel resolution, a naive algorithm would take about 64 times as long to rasterize an image as it would at full pixel resolution. In such a case, the A-Buffer would offer no speed advantage over rendering a scene at the higher resolution with a Z-Buffer algorithm, and then filtering it down. Traditional A-Buffer implementations do much better than this: they operate at *pixel* resolution, while still achieving subpixel accuracy. The way this is done, as described in [FFR83, Car84], is by constructing a table of pre-computed coverage masks, based on (and indexed by) all possible horizontal and vertical edge intercepts within a pixel, expressed at subpixel resolution.[8] Whereas fully covered pixels in the interior of the polygon are trivially dealt with, those along the edges derive their coverage masks by appropriately combining the table entries corresponding to the edge intercepts.

Even though a coverage mask constructed at pixel resolution, as described above, retains subpixel accuracy, the same is not true of a fragment's shaded colour. Applying a single shading sample per pixel can produce severe aliasing artifacts at the small scales paintstrokes are intended for. Since our goal is to provide high overall image quality, and not just silhouette antialiasing, it is often necessary to compute the shading model at finer than pixel resolution. Because of this, the rendering cost tends to be dominated by the shading, with relatively little time devoted to constructing the coverage mask. Consequently, we opted for a simpler solution in creating coverage masks than the one described above. Although not as efficient, our approach is still far superior to a simple high-resolution Z-Buffer with filtering, and also provides more accurate average values for the other fragment components (such as colour and depth values) than the pixel-resolution algorithm.

Our algorithm optimizes the rasterization of fragments that are either completely covered or have entire rows of subpixels covered. As the example in Figure 4.3 suggests, the vast majority of fragments will typically fit at least the latter profile, and a significant number may also fit the former. As for those that fit neither, the rasterization is still

[8]Some table entries can be eliminated due to symmetry.

much more efficient than scanning each individual subpixel.[9] More specifically, fragments that are fully covered are sampled only once. For partially covered fragments, one sample is applied per fully covered row, and two per partially covered row. Further details of our rasterization algorithm can be found in Appendix A.

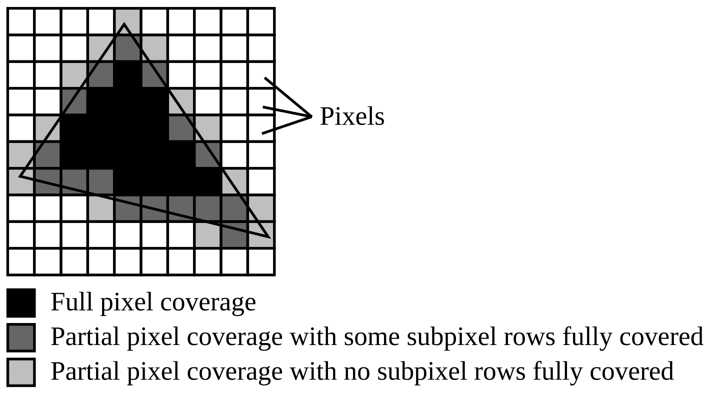

Pixels
■ Full pixel coverage
▩ Partial pixel coverage with some subpixel rows fully covered
▨ Partial pixel coverage with no subpixel rows fully covered

Figure 4.7: A typical polygon.

### 4.3.6 The Local Shading Algorithm

Computing the shaded colour of each fragment, based on a local shading model, is an important but time-consuming part of rasterizing a polygon. Using the familiar Phong shading model, described below, we sample an interpolated normal between 1 and 64 times per pixel in order to determine diffuse and specular reflection intensities. Despite a number of optimizations we have incorporated into this algorithm, it remains the most expensive element of rasterizing polygons.

For practical reasons, our current shading algorithm provides only a rudimentary set of parameters: a single directional light source of variable colour and direction. Its design is extensible, however, and could easily be adapted to accommodate multiple light sources, including point sources and spotlights. As a further simplification, we have fixed the colour of the specular component to the colour of the light source, and colours of the other two components to that of the polygon, as illuminated by the light source. These settings capture the behaviour of Lambertian and specular reflectance for many

[9]A possible exception to this arises in the shading model, which may require sampling the normal at each subpixel; none of the other interpolated quantities requires this.

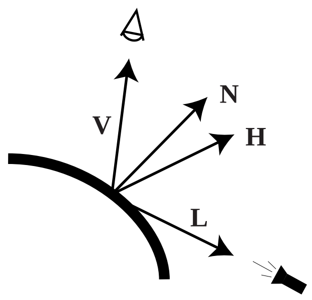


Figure 4.8: The elements of the Phong shading model.

common materials. Again, with only minor modification, our implementation could permit separate specular, diffuse, and ambient colours, as do many common rendering packages.

### The Phong Shading Model

At each sampling point, the diffuse and specular intensities, $I_d$ and $I_s$ are computed using the equations below, where the symbols $k_d$, $k_s$, $e_s$, $\mathbf{V}$, $\mathbf{N}$, $\mathbf{L}$, and $\mathbf{H}$ respectively denote the diffuse and specular reflectance coefficients, the specular exponent, the view vector, the surface normal, the light direction vector, and the halfway vector. The view vector extends from a point on the polygon's surface to the viewer. The halfway vector, as its name suggests, points halfway between $\mathbf{L}$ and $\mathbf{V}$. The relationship among the different vectors is shown Figure 4.8. In the equations that follow, these vectors are assumed to be normalized. A more thorough discussion of the Phong model can be found in [FV83, HB94].

Our implementation uses a constant view vector per polygon, which is based on the polygon's centre as derived by averaging the vertices. For reasonably small screen-projected polygons, this approximation is perfectly adequate and saves a great deal of work in eliminating per-sample renormalization of $\mathbf{V}$, which would otherwise be required in recomputing $\mathbf{H}$ for each sample.

$$I_d = k_d \mathbf{N} \cdot \mathbf{L} \tag{4.19}$$

$$I_s = k_s (\mathbf{N} \cdot \mathbf{H})^{e_s} \tag{4.20}$$

$$\texttt{colour} := k_a \mathbf{C}_L \mathbf{C}_M \; + \; (1 - k_a)(I_s \mathbf{C}_L + I_d \mathbf{C}_M) \tag{4.21}$$

As shown in Equation 4.21, the `colour` field of a fragment is assigned a value based on the material colour and the intensity values $I_d$ and $I_s$. The symbols $\mathbf{C}_M$ and $\mathbf{C}_L$ respectively denote the material colour and the light colour, and $k_a$ represents the ambient light coefficient. We treat the colours as 3-dimensional $[r, g, b]$ vectors with $r, g, b \,\epsilon\, [0, 1]$, and define the product $\mathbf{C}_L \mathbf{C}_M$ to be the vector $[\mathbf{C}_{Lr}\mathbf{C}_{Mr}, \mathbf{C}_{Lg}\mathbf{C}_{Mg}, \mathbf{C}_{Lb}\mathbf{C}_{Mb}]$.

Renormalization of the interpolated normal is optimized by storing the inverse reciprocal function $f(x) = 1/\sqrt{x}$ in a 192-element table for $x \,\epsilon\, (0, 1.5]$. The squared norm of the interpolated normal is converted to a table index, and the normal is then scaled by the table value at that index. Although it seems surprising that the length of the interpolated normal could exceed one, this can happen near the edges of a polygon that is nonplanar in one or more of its normal components.

Because the greatest expense in applying the Phong model lies in renormalizing the interpolated normal (despite the optimized table-lookup of the $1/\sqrt{x}$ function), we compute the diffuse intensity in addition to the specular at each shading sample. Although diffuse lighting did not appear to contribute significantly to aliasing, it is relatively inexpensive to sample, given that the surface normal needs to be normalized anyway. Moreover, the sign of the value obtained for $I_d$ can serve as an indication of whether the specular intensity needs to be computed at all (for opaque surfaces, if $\mathbf{N} \cdot \mathbf{L} < 0$, then $I_d := 0$ and $I_s := 0$, because of self-shadowing).

### Aliasing in the Shading Model

While the A-Buffer does a good job of eliminating spatial aliasing along an object's silhouette, it does not address potential aliasing artifacts in the shading. These can arise from a rapid variation in the shaded colours of fragments, which becomes difficult to faithfully capture with a standard per-pixel sampling approach. To achieve reasonable image quality at smaller scales, this type of aliasing also needs to be dealt with. Paintstrokes presented a particular challenge in this regard, due to the rapid variation in their surface normals over possibly very short distances. When used to model thin objects, such as hairs, they can have a screen-projected thickness of a couple of pixels or less, yet the surface normals across this width always span nearly 180 degrees.

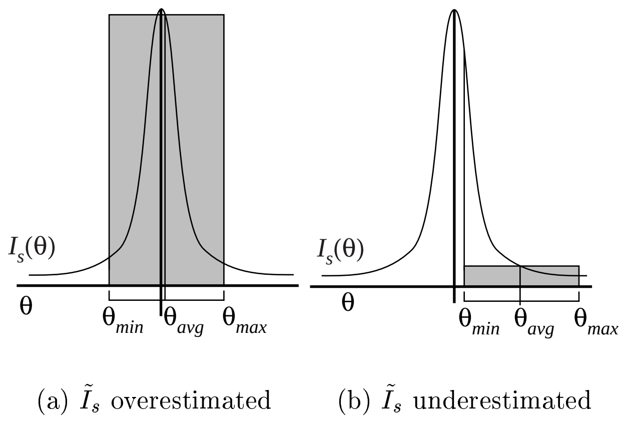
In its early stages of development, our shading algorithm applied a Phong sample at a single fixed position within each pixel, as is the norm with traditional scan-conversion methods. This approach met with generally poor results. Although it was fast, the image quality was substandard: there was considerable spatial aliasing in the specular highlights, especially noticeable during animation in the form of flickering and crawling artifacts. Materials such as water or hair, having high specular exponents, were particularly prone to this aliasing. An image rendered with only one sample per fragment (which resulted in approximately two samples per pixel of coverage, due to the large number of partially covered fragments) is shown in Figure 4.11 for reference. Note that much of the aliasing in the full-size image is disguised by the quality of the laser printing. The aliasing is more evident in the zoomed image.

We subsequently attempted to reduce the aliasing by averaging out the normal components over each fragment, as is done with the colour, opacity, and reflectance values. While this improved the results somewhat, it still left much to be desired for specular exponents higher than 4. The problem with using a fragment-averaged normal is that this is still just a once-per-fragment sampling approach (with a variable position), so it cannot faithfully capture very rapid variations in specular intensity. The general problem of filtering normal distributions has also been noted in earlier work by Alain Fournier [Fou92].

The problem with using an averaged normal is the following: When the specular exponent $e_s$ is high, the specular intensity function $k_s(\mathbf{N} \cdot \mathbf{H})^{e_s}$ decays very sharply from its maximum as $\mathbf{N}$ deviates from $\mathbf{H}$, so if the averaged normal misses $\mathbf{H}$ by only a small amount, it causes a gross underestimate of the pixel's intensity. By the same token, if the average happens to be very close to $\mathbf{H}$, it causes a large overestimate. Figure 4.9 helps to explain this phenomenon, showing two cases where an average normal yields a poor estimate of average shading intensity. In this figure, we assume that all the normals and the halfway vector are coplanar, so that each can be expressed as a single angle, $\theta$, with $\mathbf{H}$ represented by $\theta = 0$. Note that the angle $\theta_{avg}$ corresponding to the normalized average vector of the normals represented by $\theta_{min}$ and $\theta_{max}$ is in fact the average of the angles $\theta_{min}$ and $\theta_{max}$, since the average of two unit vectors bisects the angle between them. The intensity function $I_s(\theta)$ and its exact average $\tilde{I}_s(\theta)$ over the range $[\theta_{min}, \theta_{max}]$ are given by the equations

$$I_s(\theta) = k_s \cos^{e_s} \theta \tag{4.22}$$

$$\tilde{I}_s(\theta) = \frac{1}{\theta_{max} - \theta_{min}} \int_{\theta_{min}}^{\theta_{max}} I_s(\theta)\, d\theta \tag{4.23}$$



(a) $\tilde{I}_s$ overestimated (b) $\tilde{I}_s$ underestimated

Figure 4.9: Problems with using an average normal for local shading.

In an animated scene, this manifests itself through flicker, produced by alternately underestimating and overestimating the specular intensity. Moreover, the offending pixels tend to occur in conspicuous stairstep patterns, an artifact visible even in still images and especially so in animated ones where the "stairs" crawl disturbingly across the screen.

## Antialiasing the Shading

Our solution to the aliasing problem was to obtain a more precise sampling of the specular intensity function by applying multiple Phong samples per fragment, at a rate determined by four features of the underlying polygon: size, distribution of normals, the $k_s$ coefficient, and the exponent $e_s$. These attributes serve to identify polygons that are susceptible to specular aliasing. When rasterizing such polygons, the shading model is evaluated, as necessary, at up to subpixel resolution. Polygons that are not prone to aliasing are sampled at lower rates, to a minimum of one sample per pixel. More precisely, the

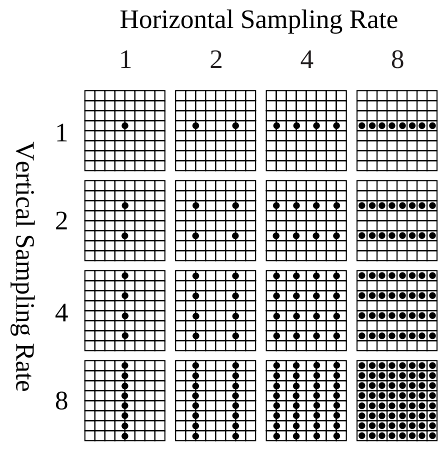


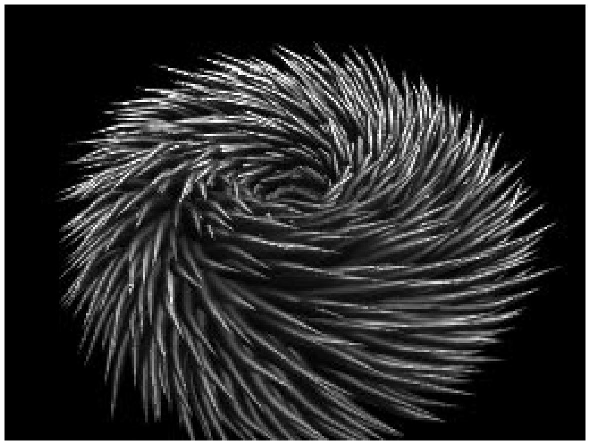
Figure 4.10: The 16 arrangements of sampling grids used in our model.

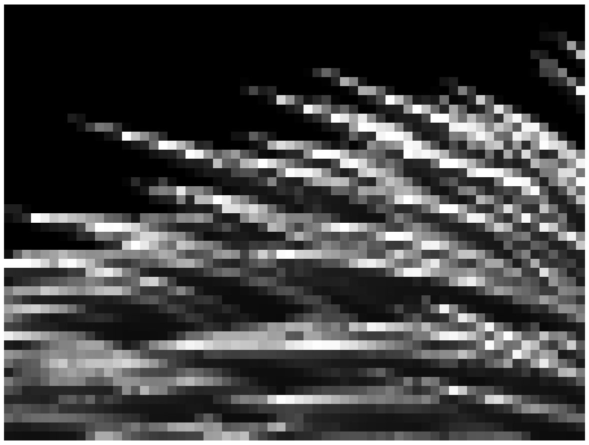
shading samples occur on a variable grid within the pixel, having horizontal and vertical densities of 1, 2, 4, or 8 subpixels. The 16 possible sampling grids for fully covered fragments are shown in Figure 4.10. The way in which the appropriate sampling grid is selected is detailed in Appendix A.

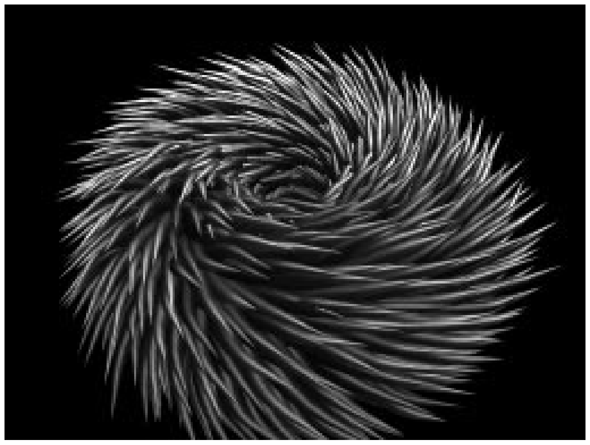
The rendered images in Figure 4.11 attest to the efficacy of this dynamic sampling approach. Note that Figure 4.11(c) achieves the same image quality as 4.11(e), using only half as many samples. Although our supersampling approach is quite effective, some aliasing may still be evident in scenes with rapid normal variation, even at the maximum sampling rate of 64 Phong samples per pixel. Using a higher subpixel resolution would certainly alleviate this, although other techniques such as jittered sampling or accurate table-based preintegration of the shading model's light intensity function may provide a more efficient solution. The latter has been implemented in polyline methods for rendering hair, as discussed in [LTT91, RCI91].

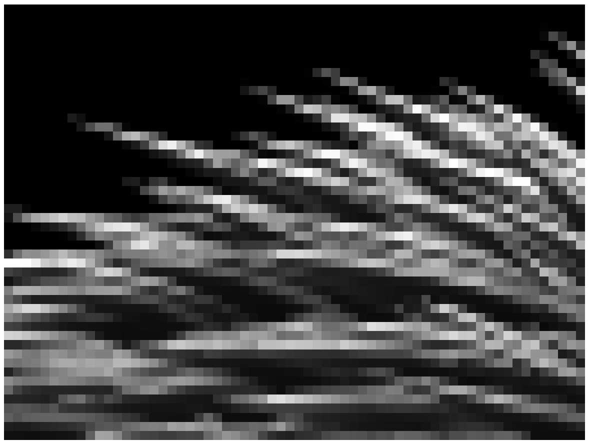
## 4.4 Blending the A-Buffer Fragments

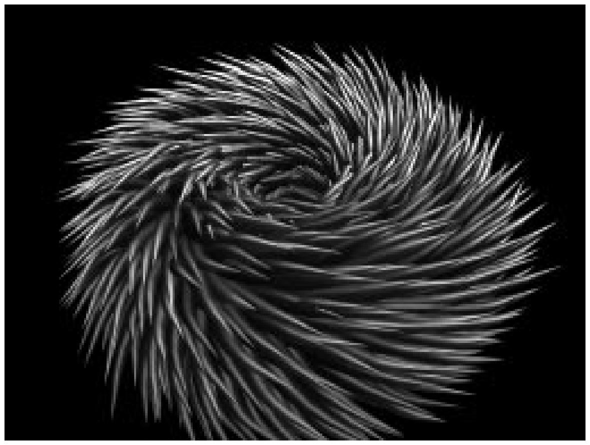
Once all the polygons have been rasterized, the A-Buffer consists of a multitude of fragments, each associated with a particular pixel and ordered by $\mathtt{Z_{min}}$. At this stage we are ready to blend the fragments over each pixel to determine the pixel's colour. The purpose of the blending algorithm we are about to describe is to express a set of contiguous (in


(a) Minimum sampling rate (2 samples/pixel on average)


(b) Zoomed image


(c) Adaptive Sampling Rate (33 samples/pixel on average)


(d) Zoomed image


(e) Maximum Sampling Rate (64 samples/pixel on average)

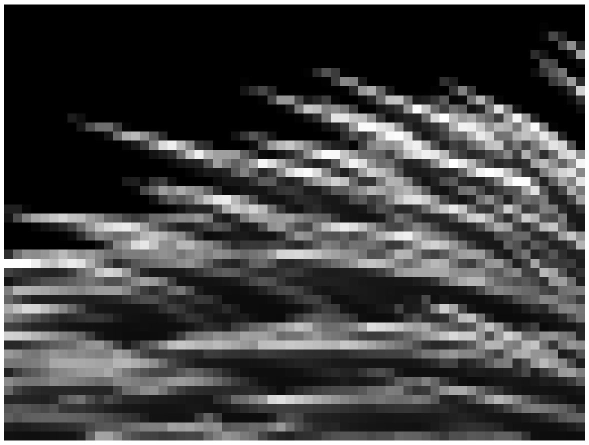

(f) Zoomed image

Figure 4.11: Images generated with minimum, maximum, and adaptive sampling rates.

terms of $Z_{min}$) fragments over a given pixel as a single visually equivalent fragment. We will explain what is meant by this shortly.

In simple terms, the A-Buffer blends the fragments over a pixel by computing a weighted average of the their colours, where the weighting factor for each fragment is its coverage.[10] However, we can only apply this simple rule to fragments with disjoint coverage masks. If any masks overlap, we need to compute the effective colour and coverage of the overlapping regions, and then blend these regions together, along with any non-overlapping ones, using the coverage-weighted averaging. The way in which overlapping fragments are combined can be quite complex, and this makes blending an interesting problem.

Our blending algorithm is embodied in the function `BlendFragment` which is based on the `Blend_under_mask` routine outlined in [Car84]. `BlendFragment` blends a list of fragments ordered by $Z_{min}$, yielding a composite colour and coverage value. Its parameters are the first fragment in the list, and a *search mask* which defines the region over which the blending is to occur. All the fragments' coverage masks are temporarily clipped to the search mask to restrict blending to only that region. The purpose of this is to break a region where fragments overlap in an arbitrary way into several simpler subregions over which their masks are either disjoint, or the front fragment's mask completely encloses the rear fragment's. These subregions are individually blended in back-to-front order using the formulas described in §B.2, and then merged together.

`BlendFragment` is a recursive function. Its initial call uses a search mask that covers the entire pixel. Each recursive entry into the function subdivides the search mask into a submask that overlaps the current fragment and a submask that does not (both possibly empty on later recursion), and then recursively blends all the remaining fragments using these submasks as search masks. The blended results over each submask are finally combined to yield the blended result over the original search mask, the entire pixel. In the worst case, each recursion involves splitting the submask into two regions, one overlapping the fragment and the other not, and both being constrained to the search

[10]For opaque fragments, the coverage equals the area, so the filtering scheme becomes one of area-averaging colours. This is probably why the A-Buffer's full name is, misleadingly, "the antialiased *area*-averaged accumulation buffer".

mask.

The pseudocode for the `BlendFragment` function, as well as a detailed derivation of all the formulas that it uses, are presented in Appendix B. We conclude this section with a brief evaluation of the A-Buffer's fragment-blending routine. Its strengths are fast, accurate compositing of overlapping transparent surfaces, as well as highly accurate edge antialiasing. Its weaknesses are high memory consumption, and an imprecise representation—and therefore occasionally poor antialiasing—of interpenetrating surfaces.

## 4.5 Summary

The polygon renderer we have discussed in this chapter offers two features that are critical for rendering complex geometry at small scales: good spatial antialiasing, and reasonable speed. The former is achieved by using the A-Buffer, which provides excellent silhouette antialiasing, and by supersampling our Phong shading, which mitigates aliasing in the shading model. The latter is due to the efficient fragment blending and Phong sampling algorithms that these techniques respectively use.

# Chapter 5

# Results

In this chapter, we complete our discussion of the paintstroke primitive by summarizing its strengths and weaknesses, and compare it with the alternative rendering methods discussed in Chapter 2. We begin with a general evaluation of the paintstroke, and then proceed to the comparisons.

## 5.1 Evaluating Paintstrokes

### 5.1.1 Performance

To evaluate the performance of the paintstroke primitive, we shall focus on the two main elements that determine its rendering speed, given the polygon-based projective rendering framework in which it operates. The first of these is the paintstroke's *tessellation quality*, as defined below. The second is the expense of obtaining the tessellation.

#### Quality of Tessellation

We measure a paintstroke's tessellation quality by the ratio of its polygon count to the (somehow quantified[1]) quality of the rendered image it produces. Given that a smooth silhouette, alias-free Phong shading, and consistency in animation (i.e. absence of popping) are the main criteria of image quality, the tessellation quality of paintstrokes is very high within their intended range of small to medium scales, albeit with a few

[1]This type of quantification is difficult, since our perception of image quality is highly complex. For example, if the thickness of a rendered paintstroke is slightly off (i.e. deviates from the model), this has little impact on perceived image quality. But if the thickness fluctuates during an animation (say between this wrong value and the right value), this degrades the image quality considerably, even though, on average, the fluctuating thickness is more accurate than if it were consistently wrong.

exceptions. By directly tessellating a generalized cylinder's screen projection rather than its true surface, paintstrokes obviate the large number of breadthwise polygons required by the latter approach to yield a smooth, round-looking silhouette.

Although a paintstroke's breadthwise distribution of normals differs from that of a true generalized cylinder (particularly at quality level zero), the discrepancy is consistent from frame to frame and is sufficiently small that it does not appreciably degrade the image quality, provided that the paintstroke's quality level is suited to its scale. However, if a paintstroke's shading needs to be particularly accurate, or if it needs to be precisely rendered when nearly collinear with the view vector, higher quality levels would need to be used even at small scales, increasing the polygon count accordingly. But unlike other methods which must *always* use large numbers of polygons just to ensure a smooth and consistent silhouette (as we shall see in §5.2), paintstrokes require higher quality levels only in these particular cases, in which they still tend to use fewer polygons than general methods that polygonize the eye-space surface. While there are cases where paintstrokes fail to produce good images (discussed in §5.1.3), in general, they achieve a high degree of image quality using few polygons.

### Cost of Tessellation

The tessellation cost of paintstrokes can be approximated by considering the key operations involved, which are summarized in the list below. The accompanying cost approximations are based on paintstrokes with no global shading or view-dependent opacity variation, since these are not strictly part of the tessellation. Nevertheless, there is still some overhead in supporting these features whether or not they are used. The cost of each step is expressed using an approximate number of FLOPs of the form $[+, \times, \div, \sqrt{}]$:

1. **Transform control points into eye-space.** [12,16,0,0] per control point.

2. **Generate piecewise interpolants based on control points and check for inflection points.** [30,40,0,0] per control point (no inflection points); [35,45,1,1] per control point (with inflection points).

3. **Lengthwise subdivision.** [50,45,5,3] per segment.

4. **Compute colour, opacity, and reflectance, based on interpolants.** [15,15,0,0] per segment.

5. **Generate ring of vertices and their normals; project vertices into screen-space.** {[35,35,6,2] for quality 0, [45,45,7,2] for quality 1, [60,60,8,2] for quality 2} per segment. This assumes only front polygons are drawn. Back polygons would add [10,10,1,0] or [15,15,1,0] per quality-one or quality-two segment, respectively.

6. **Check for degenerate polygons in tessellation and correct them.** [12,8,0,0] per simple polygon (typically $> 80\%$); [27,38,1,0] per complex polygon (typically $< 20\%$);

Overall, the tessellation uses a large number of additions and multiplications, but a modest number of the more expensive division and square root operations. Profiling tests on a 200 MHz PowerPC 604e system indicate that paintstroke tessellation consumes between 5 and 10 percent of the total rendering time, depending on the amount of screen coverage. This statistic is, of course, based on the speed of our software-based polygon renderer, which is more than an order of magnitude slower than the hardware-based systems found in graphics workstations. Nevertheless, even if speed improvements in the polygon rendering were to increase the proportion of tessellation time to as much as 50%, a multiprocessor pipeline architecture could effectively reduce the cost to zero by parallelizing the tessellation of each frame with the rendering of the previous one.

One way to improve the efficiency of tessellating simple paintstrokes would be to create a separate version of the algorithm that does not permit view-dependent opacity variation or reflectance interpolation. This would eliminate many of the additions and multiplications involved in interpolating these quantities. A more significant improvement would be a table-based vector normalization function. Normalization accounts for all but one of the square-root operations, and the same number of divisions. Because the range of vector norms encountered by the algorithm is not fixed, a simple reciprocal square root table, as was used in the Phong shading algorithm of Chapter 4, would not do. However, if the norm is decomposed into its mantissa and exponent, the former could be table-indexed and the latter divided by the constant -2, using a combination of right shifting and subtraction. In this case, the table index would incorporate the parity of the original exponent as an additional digit. This simple algorithm could, moreover, easily be incorporated into hardware.

### 5.1.2 Other Features

In addition to the efficiency of their tessellation, paintstrokes offer further modelling and rendering advantages. The special rendering effects described in §3.3 simulate global shading and volumetric opacity, which usually require the use of more expensive rendering techniques (e.g. shadow buffering and ray-tracing). Although these effects are only approximated with paintstrokes and are limited in scope, they nevertheless prove useful in modelling certain classes of objects.

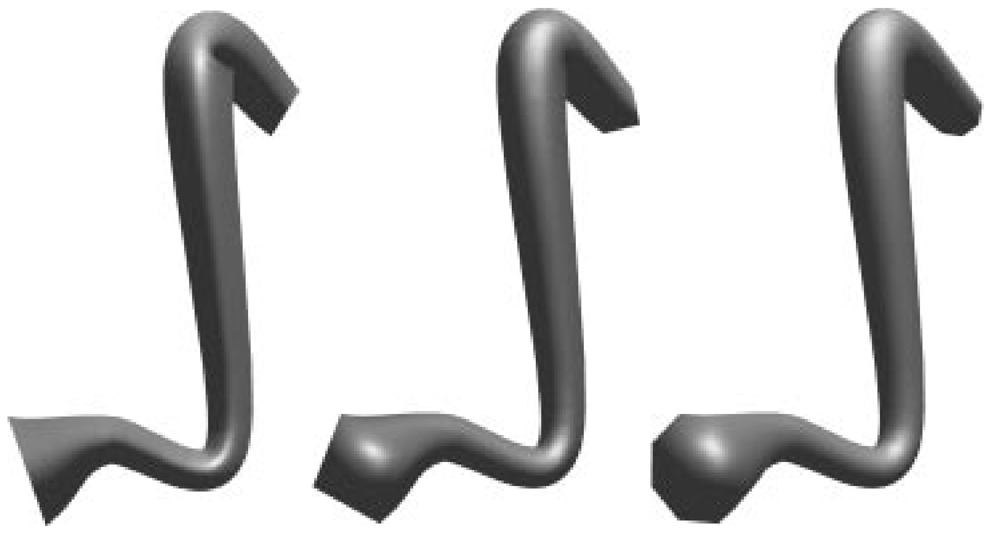
The paintstroke's representation of a tube, using a spline-based path with variable attributes at the control points, is efficient and intuitive. Whereas more general surfaces are specified with a mesh of control points, this would be difficult to manually generate for a generalized cylinder, probably requiring intermediate translation from a simpler representation similar to ours. It is also worth pointing out that physically-based simulators sometimes model tubular objects (e.g. hairs) with simple curves or even point masses. By outputting points on the curves or the point masses themselves, such programs can with minimal effort produce generalized cylinder descriptions that can be rendered with paintstrokes.

### 5.1.3 Limitations and Proposed Improvements

Although the paintstroke primitive as presented accomplishes its basic goal of efficiently rendering generalized cylinders at small to medium scales, it leaves plenty of room for improvement. The following are some of its current limitations, along with ideas on possible solutions that may find their way into future implementations. We begin with fairly specific problems, and then move on to general limitations.

### Quality Level Limitations

Many limitations of the paintstroke primitive are tied to its approximative nature. The breadthwise distribution of normals and the self-occlusion accompanying sharp screen-space curvature, both visible in Figure 5.1, exemplify approximations whose fidelity varies with the paintstroke's quality level. At quality level two, they produce very accurate results, but at quality zero, the approximations are clearly inaccurate. By regulating a paintstroke's quality level according to the estimated thickness of its screen projection—something our algorithm can do automatically—this inaccuracy can be prevented from degrading image quality at larger scales, while still allowing for lower quality paintstrokes to be used at smaller ones.



(a) Quality 0 (b) Quality 1 (c) Quality 2

Figure 5.1: A paintstroke rendered at the three quality levels.

### Endcaps

As is easily seen in Figure 5.1, the endcaps used with quality-one and quality-two paintstrokes are crude, consisting of simple squares or octagons. Although this is acceptable at smaller scales (which is where paintstrokes are most useful) it greatly degrades the image quality of larger paintstrokes, especially those that are beyond 10 pixels of thickness at the endcaps. This problem can easily be solved by using higher-degree polygons as the paintstroke's size increases. Tables could provide the vertex positions (relative to some fixed coordinate frame) for a number of such polygons, to be used at various scales. This improvement will likely be incorporated into future versions of the paintstroke primitive.

### Quality Level Transitions

Transitions between quality levels pose another challenge for paintstrokes, because the resulting abrupt changes in normal distribution tend to produce equally abrupt changes in shading. At present, the only way to completely eliminate this popping artifact is to hold a paintstroke's quality level constant. This still provides some degree of level-of-detail adjustment through the paintstroke's adaptive lengthwise tessellation, but it is clearly not an efficient solution.

A possible alternative representation of normal vectors that will be outlined in Chapter 6 would greatly improve the accuracy of normal distributions for low-quality paintstrokes, and virtually eliminate popping artifacts for all changes in quality level. Another solution may be a morphing approach similar to that of Hoppe's geomorph [Hop97].

### Viewing Direction Problems

Paintstrokes that are fairly straight and nearly collinear with the view vector can present further difficulties. At quality level zero, they become shortened and ultimately disappear, while at the higher quality levels they reveal a polygonal (square or octagonal) cross-section.[2] The problem arises because the paintstroke's central path begins to degenerate into a point. Since our method relies on this path to reproduce the geometry of a generalized cylinder, it cannot properly render paintstrokes in this situation. Moreover, the problem cannot be solved by using smoother endcaps, since the endcaps will not necessarily cover the paintstroke: consider a paintstroke that tapers at the ends, as shown in Figure 5.2. By judiciously adjusting the paintstroke's quality level to its rendering scale, the visual impact of this artifact can be minimized.

[2]As Figure 5.1 demonstrates, this does not happen with curved paintstrokes, even if they do approach collinearity with the view vector at some point.

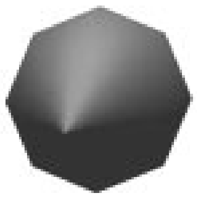


Figure 5.2: A tapering quality-two paintstroke viewed head-on.

### Transparency

Transparent paintstrokes of quality one or two exhibit regions of erroneously high opacity when viewed under high screen curvature, as shown in Figure 5.3. The problem arises from an overlap among neighbouring paintstroke polygons. Whereas the overlap has no visual effect on opaque paintstrokes, it is easily seen in transparent ones because of its higher opacity than the surrounding non-overlapping polygons.

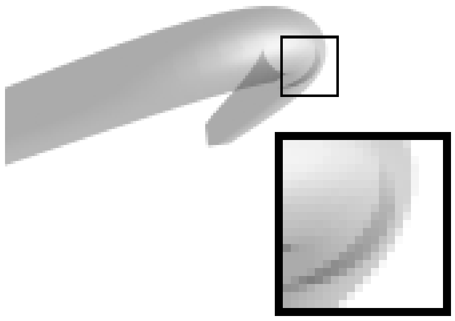


Figure 5.3: Transparent paintstrokes under sharp screen curvature.

The ideal solution to this problem would be to find a way to eliminate the overlap that accompanies sharp screen curvature. This is something we accomplished for quality-zero paintstrokes, but have been unable to do at the higher quality levels. Eliminating the overlap at all quality levels would not only solve the image quality problems with transparent paintstrokes, but also improve the efficiency of rendering opaque ones[3] by eliminating unnecessary rasterization and blending operations.

A simpler but less elegant alternative would be to keep the overlap, but to ignore its effect on opacity. For this purpose, a fragment's `tag` identifier, introduced in §4.2, proves

[3]Note, however, that the regions of overlap are quite small and have only a minor impact on overall efficiency.

useful. Fragments from each polygon could be assigned a unique tag number that is incremented for each successive polygon along the paintstroke's path. The fragment-blending routine could then inexpensively determine whether a pair of overlapping fragments originate from neighbouring polygons, simply by comparing their tag numbers. If they do, the rear fragment in the pair would be skipped, eliminating its effect on the pixel's blended opacity. This approach would also correctly handle the legitimate overlaps of paintstroke polygons, since such overlaps would never occur between neighbouring polygons.

### Shadows

A more general limitation is the absence of shadows in paintstroke rendering. Although our global shading algorithm is cheap and effective when applicable, its scope is quite narrow, being restricted to a fairly homogeneous layer of paintstrokes covering a convex shape. A more general global shading technique, like the depth buffering algorithm proposed by Williams in [Wil78], would make a useful addition to our rendering engine. Because this algorithm is applied at the polygon level, it would require no modification to work with opaque paintstrokes. Dealing with transparency, on the other hand, would require much further work.

### Fast Phong Shading

Another issue is the technique of fast Phong shading, which is common in hardware implementations. Fast Phong shading, as described by Bishop [BW86] and Kuijk [KB89], approximates the Phong intensity function using a Taylor polynomial, which can be efficiently evaluated using forward differences. The main limitation of this method is that it cannot be applied to polygons that contain a large amount of normal variation (over $60^0$, according to Bishop), which are precisely the sort of polygons that are generated by paintstrokes. In fact, even with standard Phong shading, the sizable normal variation of a quality-zero paintstroke segment will cause the interpolated normal to become very short at some point (as shown in Figure 3.25), thus changing direction very rapidly during the interpolation and causing aliasing if not appropriately supersampled.

Note that this problem affects *all* polygonal models with high per-polygon curvature, especially the triangular prisms that are frequently used to approximate thin tubes in hair rendering [Mil88, WS92]. Two solutions, neither of them particularly appealing, are to use higher quality levels for the paintstrokes, or to increase their nudge factors, as discussed in §3.2.4, so as to reduce the breadthwise range of normals. The former maintains image quality at the expense of rendering speed, while the latter does the opposite by avoiding extra breadthwise subdivisions and giving the paintstroke a non-circular (but consistent) normal distribution.

#### Level-Of-Detail Limitations

The level-of-detail adjustment of paintstrokes is unable to simplify a paintstroke-based object down to very coarse levels: at a minimum, a paintstrokes must have one polygon per control point (excluding the first one). This prevents radical global simplifications possible with many other schemes, as described in [CVM$^+$96, Hop96, LE97]. For example, given a dense spherical cluster of tubes (or a single tube tightly rolled up in a ball), the latter algorithms would at some point re-polygonize the model into a simple roughly spherical surface, whereas paintstrokes would never do this. We do not view this as a serious shortcoming, however, because paintstrokes are intended for use at scales large enough that individual tubes (and their parallax and blocking effects) are still discernible. Smaller scales than this are left to other techniques, which may well involve vast global simplifications of the kind just mentioned.

#### General Limitations

The paintstroke's circular cross-section and absence of texture-mapping are two very general limitations that present opportunities for future research. We will examine these possible extensions to the primitive in Chapter 6.

## 5.2 Comparison with Static Polygonal Models

In order to permit a thorough comparison of paintstrokes with efficient static models of generalized cylinders, we have implemented an algorithm that translates a paintstroke description into an equivalent statically tessellated polygonal model. For obvious reasons, the latter excludes the paintstroke's orientation-based global shading and opacity variation effects, but it captures all of the other features. The algorithm is similar to one used by Jules Bloomenthal in [Blo85] for polygonizing tree branches, although ours is adaptive to the lengthwise curvature of the tube.

### 5.2.1 Polygon Extrusion Algorithm

The polygonal representation produced by our algorithm is the orthogonal extrusion[4] of an $n$-sided regular polygon along a given path, where the polygon's size may vary, but it may not rotate about the path.[5] By this we mean that the polygon is fixed in the Frenet frame that travels along the path. A few examples are shown in Figure 5.4. Whereas the degree (i.e. number of sides) of the polygon is directly set by the modeller, the number of lengthwise segments is minimized by the algorithm, subject to the tube's world-space curvature and a user-specified tolerance value.

The polygon vertices generated by the algorithm occupy a number of circular rings, each centred about some point on the path of extrusion, and lying in a plane normal to the path's tangent at the point. The resulting model essentially has two dimensions of complexity, which determine its level of detail: the lengthwise granularity, defined by the number of rings of vertices used; and the breadthwise subdivision granularity, the number of vertices per ring. The first of these is determined jointly by a tolerance value chosen by the modeller, and by the magnitude of curvature in the path. The second is directly specified by the modeller.

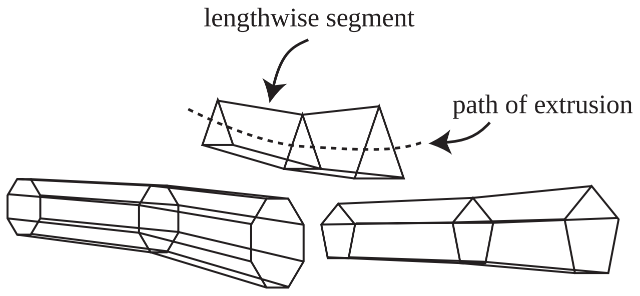


Figure 5.4: Extrusions of several regular polygons to produce tube tessellations.

The lengthwise subdivision algorithm is similar to that used in paintstrokes, recur-

[4]By this we mean that the polygon moves orthogonally to the plane spanned by its vertices.
[5]Allowing the rotation would result in a bumpy silhouette and a spiral-like quality to the shading.

sively splitting a segment in half until the eye-space analogues of the radius and positional constraints (as described in 3.12) are satisfied. Specifically, the algorithm compares each of the $x, y, z$ components of the tube's spline path over a given segment with the linear interpolant to the spline over the same segment. This is virtually identical to Position Constraint II of the paintstroke, except that no perspective adjustment is made. The radius component is compared with its linear interpolant in exactly the same way as the Radius Constraint of paintstrokes, but again with no perspective adjustment.

### 5.2.2 Properties of Polygonal Extrusions

As is the case with all statically tessellated models, in order to determine an appropriate level of tessellation granularity, the approximate range of scales at which the model will be rendered must be known. As is demonstrated in the work of Gavin Miller and Watanabe & Suenaga, [Mil88, WS92], very small-scale tubes, such as hairs viewed from a moderate distance, can typically be modelled with only triangular extrusions and fairly coarse lengthwise subdivisions. As the scale increases, finer and finer granularities become necessary to maintain image quality.

Experience has shown that low-degree polygonal extrusions can make very good approximations to a thin circular tube. For example, a triangular extrusion can in some cases produce acceptable results for tubes up to about 5 pixels in diameter of projection, despite inaccuracies in the thickness and shading of the image. The projected thickness of a triangular tube varies between $3/4$ and $\sqrt{3}/2$ of the true thickness of the equivalent circular tube, while the range of surface normals visible to the viewer spans between $120^0$ and $240^0$ (i.e. $2/3$ to $4/3$ of the true range). As shown in Figure 5.5, similar fluctuations are present with higher-degree regular polygons, though their ranges decrease as the degree increases. The decrease in the thickness ranges is not monotonic: extruding a polygon with an odd number of vertices produces a smaller range of thickness variation than extruding the polygon with one more vertex. As an example, consider the square, whose extrusion thickness lies within the range of $[1/\sqrt{2}, 1]$ times the diameter. The magnitude of this variation is 29.3% of the diameter, compared to the triangle's 11.6%. As a result, polygons with an even number of vertices (and the square in particular) are generally poor choices for extruding into a tube—it is better to either add or remove a vertex to make the polygon's degree odd.

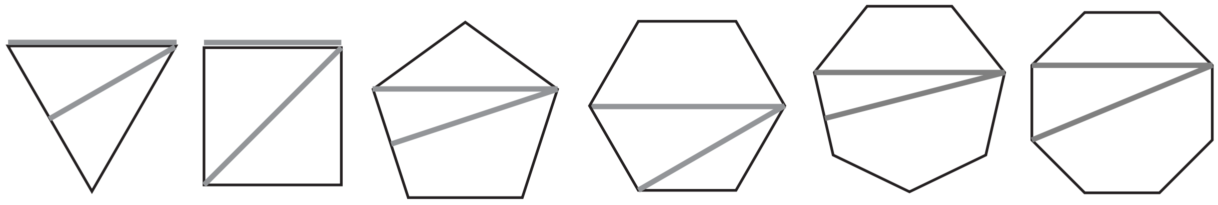


Figure 5.5: View-dependent thickness ranges of polygon extrusions.

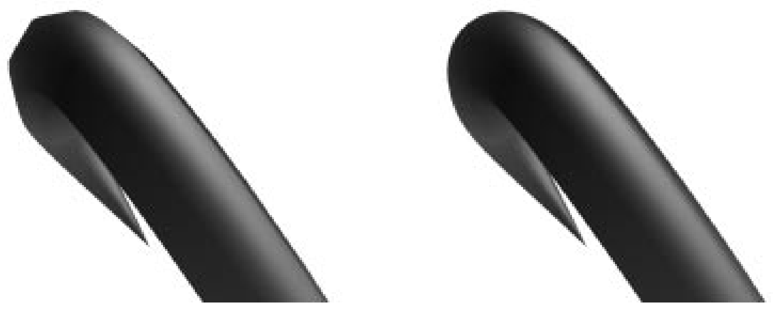
It can be shown that all polygonal extrusions whose vertices lie on the circular cross-section of a tube underestimate its diameter from some viewing angles.[6] Hence, a useful correction is to increase the radius of the ring of vertices to compensate for this and make the extrusion's average thickness over all possible viewing directions (or some desirable subset thereof) equal to the tube's true thickness. While this solution improves the accuracy of an extrusion's thickness, it does nothing to eliminate the fluctuations both in the thickness and the normals. Fortunately, however, these fluctuations are not highly conspicuous, because they are gradual. If the (extruded) tube's orientation changes smoothly with respect to the view vector, the thickness and highlights also change smoothly. That can be difficult to notice when the tube itself is moving across the screen, and especially so when there are a number of tubes in differing orientations moving in different directions. The thickness fluctuations can also be camouflaged by a lack of contrast with the background (which may consist of other similarly coloured tubes). These reasons help to explain why simple triangular extrusions worked so well for modelling hair and fur in [Mil88, WS92].

There are, however, a number of cases where low-degree polygonal extrusions fail to produce a reasonable image even at small scales. The worst-case scenario is one where a tube spins about its tangent vector at some point, which itself does not move. This has two effects: it maximizes the fluctuations while keeping the tube's image stationary at the point. The combination of these produces visible fluctuations even at subpixel

[6]They never overestimate it. If the number of vertices is even, they attain it exactly; otherwise, their maximum thickness always remains lower than the true diameter.

thicknesses.[7] Another situation in which the thickness inaccuracies can be troublesome is one where fairly straight tubes are placed in an orderly arrangement, such as an evenly spaced row or grid of parallel tubes. As the viewing direction changes, gaps between adjacent tubes will grow and shrink noticeably. Problems can also arise from the fluctuation of the normals, when light rays strike the tube from the side, as seen by the viewer. As the range of normals fluctuates, the illuminated side of the tube will perceptibly vary between higher (when the range is greater) and lower (when the range is smaller) intensity. Even at subpixel thicknesses, this artifact is easily visible as a variation in the tube's overall brightness.

At larger scales, all of the above problems persist, but are compounded by silhouette inaccuracies, which are no longer hidden by the small image size. When a section of a tube becomes nearly collinear with the viewing direction, the the polygonal cross-section can be easily seen, spoiling the illusion of a circular tube.[8] An example of this phenomenon is shown in Figure 5.6. Unlike the thickness and shading fluctuations, which are visible only in animation, this jagged silhouette must be avoided in still images as well.



(a) Octagonal extrusion (b) Quality 2 Paintstroke

Figure 5.6: A tube that becomes nearly collinear with the view vector.

[7]In fact, it would take a 16-sided polygon to attenuate the fluctuations to a reasonable level in this case.

[8]With paintstrokes, this only happens if the *entire* tube is nearly collinear with the view vector, which is impossible if the tube is moderately curved.

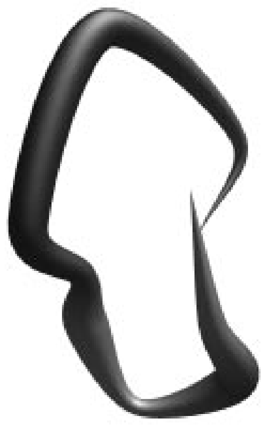
### 5.2.3 Benchmark Comparison

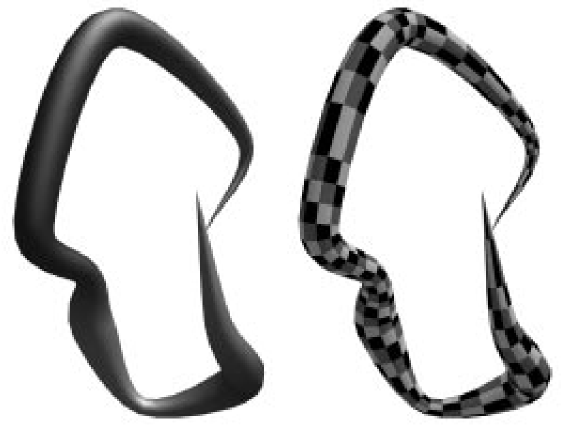
Our comparison of the paintstroke's tessellation with the static tessellation of polygonal extrusions is based on a benchmark that renders all the models of the tube shown in Figure 5.7 at various scales and viewing angles. Because our current algorithm for endcap generation in paintstrokes is crudely implemented, we tapered the tube at the ends to avoid using endcaps. This does not reflect a conceptual limitation of the paintstroke's tessellation, but only one of its implementation.

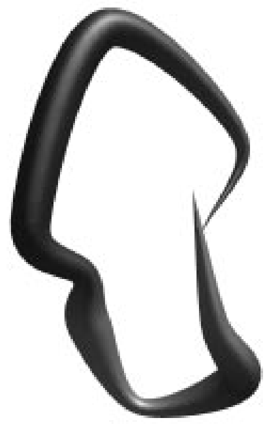
The methodology of the benchmark is as follows. We constructed a dense $3 \times 3 \times 3$ matrix of paintstrokes, and rendered a set of animations of it, each consisting of a single rotation about a diagonal axis (relative to the matrix), comprising 50 frames in total. The animations were carried out at three different constant distances from the (centre of the) matrix, so as to simulate rendering at a large, medium, and small scale. A single frame from the three scales is shown in Figure 5.8. At each scale, two animations were made, one using a conservative quality level that produced optimal image quality, and one using an aggressive one that improved the speed at the slight expense of image quality.

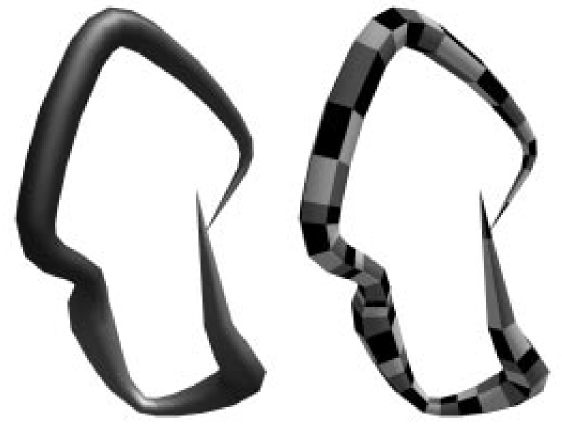
Next, we converted this paintstroke-based scene description into three static polygonal models, identical except in their tessellation granularities. The first one was coarse, the second medium, and the third fine. Each of these was optimized for the large, medium, or small scale of the animation, respectively, by using the minimum number of polygons required to ensure high image quality (as explained below) at its corresponding scale. Each polygonal model was then rendered in the same animations used with the paintstrokes, one at each scale. That yielded a total of nine animation runs, in addition to the paintstrokes' six. As Figure 5.8 shows, the matrix used was filled very densely with the tubes, so the screen-size variation of each tube's image due to perspective was negligible, even at the closest distance. This put the static models at virtually no disadvantage (because of their fixed level of detail) when rendered at their respective ideal scales.

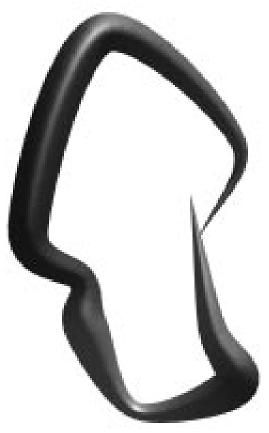
Although the notion of a high quality image is a complex one, in calibrating the static tessellation algorithm, we were primarily looking to eliminate silhouette discontinuities and abrupt transitions in the shading, which could both arise either from an overly


(a) Quality-Two Paintstroke


(b) Fine Static Model (558 polygons)


(c) Quality-One Paintstroke


(d) Medium Static Model (200 polygons)


(e) Quality-Zero Paintstroke

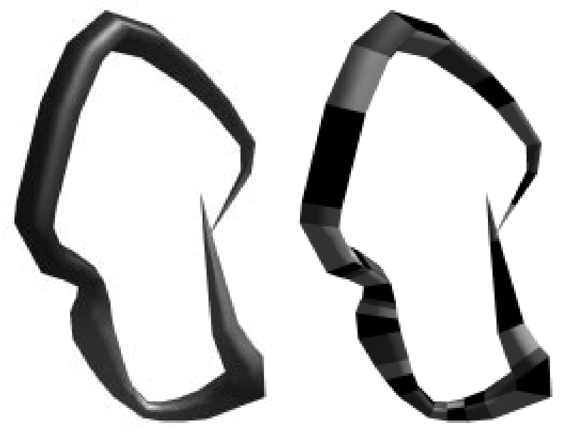
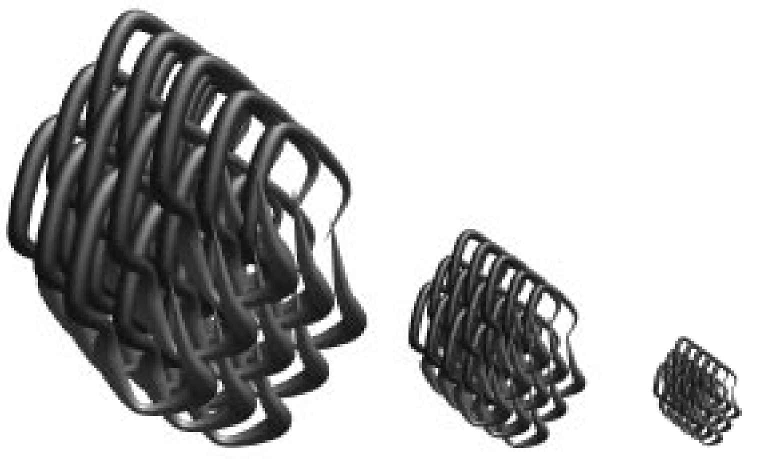

(f) Coarse Static Model (96 polygons)

Figure 5.7: Paintstroke-based and static models of the benchmarked tube.



(a) Large (b) Medium (c) Small

Figure 5.8: Models of the tube used in our comparison.

permissive lengthwise tolerance, or an excessively coarse breadthwise granularity.[9] In short, we were seeking the same level of image quality as was achieved in the aggressive paintstroke animations. Although the latter tended to have less accurate breadthwise shading profiles than the former, this is generally less important to overall image quality than a smooth silhouette and smooth highlights, as can be easily seen in Figure 5.7. On the other hand, if accurate shading is considered essential, then the conservative paintstrokes allow for a completely fair comparison, offering the same level of shading accuracy as the comparable static models.

The results of these benchmarks are summarized in Table 5.1. Statistics were gathered for all 1350 tubes rendered ($3 \times 3 \times 3$ tubes/frame $\times$ 50 frames) and then divided by 1350 to provide a per-tube average. For the paintstroke-rendered animations, the only difference between the conservative and aggressive primitives was in their quality levels; both used identical lengthwise subdivision tolerances. The differences in image quality between the two were very subtle, being evident only in the shading profiles the tubes.

At each scale, the most interesting comparisons are between the paintstrokes and the polygonal model that is best suited to the scale. Figures describing the latter form a

[9]This was an empirical process that involved repeated trials and errors in reducing the polygon count, while maintaining the quality of the entire animation sequence.

| **Scale** | **Avg. per Tube** | **Paintstroke** | | **Static Polygonal Model** | | |
|---|---|---|---|---|---|---|
| | | **Conservative** | **Aggressive** | **Fine** | **Medium** | **Coarse** |
| Large | Breadthwise Quality | 2 | 1 | **9-gon** | 5-gon | triangle |
| | Total Polygons | 268.4 | 135.0 | **558** | 200 | 96 |
| | Polygons Rendered | 238.3 | 118.2 | **275.9** | 98.6 | 46.5 |
| | Pixel Area | 1186.6 | 1188.0 | **1190.6** | 1182.0 | 1155.1 |
| | Rendering Time (s/60) | 11.01 | 7.48 | **12.19** | 6.78* | 5.91* |
| Medium | Breadthwise Quality | 1 | 0 | 9-gon | **5-gon** | triangle |
| | Total Polygons | 100.6 | 46.5 | 558 | **200** | 96 |
| | Polygons Rendered | 87.6 | 43.1 | 277.5 | **99.3** | 47.4 |
| | Pixel Area | 292.7 | 292.6 | 295.1 | **293.0** | 286.2 |
| | Rendering Time (s/60) | 4.00 | 2.59 | 9.61 | **4.78** | 2.71* |
| Small | Breadthwise Quality | 0 | 0 | 9-gon | 5-gon | **triangle** |
| | Total Polygons | 36.4 | 36.4 | 558 | 200 | **96** |
| | Polygons Rendered | 33.6 | 33.6 | 278.3 | 99.7 | **47.8** |
| | Pixel Area | 72.4 | 72.4 | 73.6 | 73.1 | **71.4** |
| | Rendering Time (s/60) | 1.48 | 1.48 | 8.52 | 3.45 | **1.88** |

Table 5.1: Comparison of paintstrokes with statically tessellated polygonal models.

diagonal of boldfaced entries in Table 5.1. Using a finer static model than the intended one produces the same image quality but at greater expense. The figures for this appear below the diagonal, and demonstrate the behaviour of static tessellation under non-ideal scales. Entries above the diagonal represent lower tessellation granularity than is required for the image scale. While these figures indicate the lowest rendering cost, they are not directly comparable to the paintstrokes' (whether conservative or aggressive), because of the substandard quality of the image that is produced. We now turn to some explanations and remarks about the statistics in the table.

**Total Polygons** The total number of polygons processed, prior to backfaceculling (no clipping was required during the animation). Notice how paintstrokes have much lower total polygon counts than the comparable static models. This is largely because very few backfacing polygons are generated by paintstrokes. On a related note, the storage and bandwidth requirements of the static models are directly proportional to their polygon counts. Consequently, their data files and memory requirements were considerable: approximately 1 MB, 370 K, and 180 K for the three granularities. That compares with only 18 K for the paintstroke file, of which about a quarter was devoted to unused global shading and opacity variation parameters.

**Polygons Rendered** The total number of polygons that were rendered after backfaceculling. While both conservative and aggressive paintstrokes have fewer rendered polygons per tube than the corresponding static model, the difference is much narrower than with the total polygon counts. In contrast to the static polygonal models, only a small portion of paintstroke-generated polygons are culled, even at the higher quality levels.

**Pixel Area** The total number of pixels rasterized, measured to within a subpixel, or 1/64th of a pixel. Regions that are overlapped by closer ones still contribute to the pixel area. The slight discrepancies in average pixel area between conservative and aggressive paintstrokes are mainly caused by their differing approximations of the screen-space folds that appear at high screen curvature (shown at a larger scale in Figure 5.1). The larger discrepancies among the static models arise from their view-dependent thickness variations. Although we adjusted the radii of the vertex rings used in the tessellations to yield an accurate *average* screen-space thickness over a uniform distribution of viewing directions around (and orthogonal to) the tube, the animation produces a different distribution. These area discrepancies have negligible impact on benchmark performance.

**Breadthwise Quality** For paintstrokes, this refers to the quality level used. For static models, it describes the degree of the regular polygon that was extruded. Notice how at each scale, the conservative paintstroke quality level is roughly commensurate with the degree of the corresponding extruded polygon: quality zero, using one breadthwise polygon, is matched with the triangle; quality one, using two breadthwise polygons, with the pentagon; and quality two, having four breadthwise polygons, with the nonagon. This produces shading profiles of nearly equal accuracy, though the paintstrokes' are always perfectly consistent whereas the static models' are not. The aggressive paintstrokes, on the other hand, have a smaller number of breadthwise polygons than the comparable static models, resulting in less accurate, though still acceptable, shading.

**Rendering Time** The time taken to render the complete animation on a 200 MHz PowerPC 604e system with 32 MB of RAM and 1 MB L2 cache, VM turned off. All polygons were rendered using the software-based A-Buffer engine with adaptively supersampled Phong shading, as described in Chapter 4. The rendering times shown include the screen update at each frame (which is negligible compared to the rendering time) and exclude the time to read in the data file (which was significant, taking up to 4 % of the rendering time for the static models). They are expressed in ticks, or sixtieths of a second. Times above the boldfaced diagonal in the 'static polygonal model' category are marked with an asterisk, indicating that they are inadmissible because the image generated was of noticeably lower quality than that produced with aggressive paintstrokes.

### 5.2.4 General Remarks

Our comparison shows that paintstrokes can provide a faster and more efficient means of rendering generalized cylinders than statically tessellated models, even at the latter's optimal scale. This is especially evident with the aggressive use of quality levels, as Table 5.1 clearly shows. This result has significance beyond simple static models, because it implies that even a dynamic polygonal model, which consistently maintains appropriate tessellation granularity, is unlikely to outperform paintstrokes in rendering generalized cylinders, unless it takes advantage of their symmetries and view-invariant properties as do the paintstrokes.

While these results are encouraging, they come with a few caveats. Hardware-based polygon renderers tend to work faster (on a per-polygon basis) with static tessellations than with dynamic ones. This is because such rendering engines rely on a pipeline architecture to parallelize and thus expedite the rendering process. While a static set of polygons can trivially keep the pipeline full, a dynamic tessellation scheme, as used by paintstrokes, may not be able to keep up. Moreover, static models can be combined with pre-computed modelling transformations into a display list, which further enhances rendering speed. On the other hand, using such a rendering engine would most likely require abandoning both the A-Buffer and the efficient antialiased Phong shading that we are using, since these are not widely available in hardware. If that is acceptable, then a good solution would be the pipeline approach, whereby each frame is dynamically tessellated into a (static) polygon list while the previous one is being rendered. Given a multiprocessor architecture, this is a fast and efficient solution, which may become a useful option if fast hardware A-Buffers, such as the one proposed by Schilling and Straßer in [SS93], become widespread.

## 5.3 Comparison with Dynamic Polygonal Models

### 5.3.1 Blinn's Optimal Tubes

Jim Blinn's optimal tubes [Bli89] are a simpler modelling primitive than paintstrokes. For drawing plain cylinders, their rendering speed is certain to be greater than that of equivalently shaped paintstrokes, because of the fast (usually hardware-based) Gouraud-shading polygon renderers they are suited to. However, optimal tubes lack many of the key features that make paintstrokes a useful and flexible primitive: radius variation, adaptive lengthwise subdivision, a specular shading model, normal interpolation (which provides more accurate Lambertian shading as well as specular), and the ability to accommodate multiple light sources.[10] While optimal tubes may be valuable when these features are not needed, paintstrokes are amenable to a much greater variety of modelling scenarios.

Because Gouraud-shading does support the Phong shading model, and can therefore capture specular reflectance, one may wonder why optimal tubes cannot do so. Although Gouraud shading can apply the Phong model at polygon vertices (interpolating the resulting colours rather than the normals themselves), this feature is inherently unsuitable for optimal tubes. Specular highlights usually involve a fairly abrupt intensity variation, which can only be accurately captured by a large number of Phong samples. Because these samples are only taken at polygon vertices, the purpose of optimal tubes, which is to reduce this number of vertices, is fundamentally incompatible with achieving alias-free Phong shading.

[10]Optimal tubes could in fact accommodate additional light sources by introducing extra breadthwise subdivisions along the shading boundaries of each light. Such a solution would, however, rapidly undercut their chief advantage of using fewer polygons than more general tessellation schemes.

## 5.3.2 General Methods

As we have remarked earlier, one of the main reasons behind the paintstroke's efficiency is its specificity in modelling the generalized cylinder. Unlike general-purpose tessellation algorithms that tessellate the true surface they are given, paintstrokes only tessellate the approximated *screen projection* of a generalized cylinder. This is what allows them to achieve accurate silhouette and shading approximations using a very small number of large polygons, which more general methods fail to do.

General dynamic tessellation schemes, as used with NURBS or Bézier patches, may vary the overall granularity of the polygon mesh according to curvature or screen size, but they still tessellate the entire surface—they make no attempt to replace it with a screen-projection (which is difficult to do with arbitrary surfaces). At any scale, their ideal tessellation will be similar to the static models described in §5.2. As we have seen, even without counting its tessellation cost, such a model renders more slowly than the equivalent paintstrokes. Moreover, to render a tube with general parametric surfaces, a mesh of control points would need to cover the tube's surface. At very small scales, the number of polygon vertices in an efficient tessellation may be less than the number of these control points, which all need to be transformed to eye-space prior to the tessellation. This problem does not arise with paintstrokes, which only specify the tube's path and thus use a much smaller number of control points.

Dynamic polygon simplification schemes, such as Hoppe's [Hop96, Hop97], achieve smooth silhouettes by increasing the granularity of their polygon mesh near the edges. This differs from the paintstroke's approach, which is to construct polygons that precisely conform to the silhouette. Consequently, paintstrokes are able to use fewer polygons than these simplification schemes. Although the paintstroke's tessellation might be more expensive, this should make very little difference given its 5–10% CPU utilization rate (based on profiling our implementation). Moreover, as mentioned in §5.1.1, a pipelining architecture could be used to reduce the tessellation cost to virtually zero, provided that it is at least as fast as the rendering phase.

Hoppe's progressive meshes do offer two techniques that paintstrokes lack: view-frustum-based simplification, and geomorphing. The first of these would be fairly easy to incorporate into paintstrokes, by (conservatively) suppressing tessellation over segments that extend beyond the view frustum. A conversion of our Catmull-Rom splines to the Bézier basis would help in this regard, by providing a convex hull for the paintstroke's path. A form of geomorphing could also be used with paintstrokes to eliminate possible popping artifacts when quality levels change, although it would not be needed for the re-tessellations within a single quality level, which are already free of popping.

As mentioned earlier, polygon simplification methods are able to work on a global scale, collapsing unrelated surfaces into single polygons. This is something that pure dynamic tessellation algorithms, such as the paintstroke's, are unable to accomplish. Although such global surface simplifications make valid shape approximations, the normal distribution of the simplified geometry can differ markedly from that of the original geometry, resulting in inaccurate shading. Volumetric textures, discussed in §5.6, provide an alternative that addresses this important issue.

## 5.4 Comparison with Particle Systems

### 5.4.1 Brush Extrusions

Though intuitive and useful for interactive drawing applications, brush extrusions [Whi83] make a poor general-purpose rendering primitive for particle systems. Their design bears the fundamental inefficiency of touching many more pixels than the number that actually appear in the brush stroke, a consequence of rendering multiple overlapping images of a particle. Although Whitted proposes an efficient cache-based implementation of the copy operations that composite the image of the brush tip into the stroke, this still involves processing a large number of covered pixels that never appear on the screen. Moreover, as mentioned in Chapter 1, in order to accurately render a generalized cylinder trail with nondirectional lighting, the spherical tip would need to be continually re-rendered as it moves. This would require computing the colours of many pixels that would be overwritten by later tip samples—clearly not an efficient solution.

Another problem is that brush extrusions do not correctly handle transparency in cases where their screen-projected image is self-intersecting. As with paintstrokes, this will occur with *any* curved path in eye-space from *some* viewing direction. Thus it is a common occurrence, by no means restricted to paths that self-intersect in eye-space. The problem is that, regardless of its opacity, a more distant particle is always overwritten by a closer one. If, as a solution to this, true transparency blending were applied to all the particles, that would make the pixel-overwriting nature of this approach even more inefficient than it already is.

### 5.4.2 Cone-Spheres

Cone-spheres are quite similar to quality-zero paintstroke sections, but with two significant differences: they use linear positional interpolants—whereas the latter use splines—and they render spheres in order to join conical sections, something paintstrokes do not (and need not) do.

Although the spheres ensure a smooth silhouette at the joint between successive cones, the cones themselves are straight. Consequently, a large number of cone-spheres are needed to adequately represent a tube with curves or nonlinear radius variation. This number is greater than the number of paintstroke segments required for the equivalent tube. One reason is the paintstroke's adaptive lengthwise subdivision, which is based on screen-space curvature—cone-spheres are static models that do not attempt to take advantage of the fact that from some viewing angles, fewer of them are needed to visually approximate a tube than from other angles. Another reason is that lengthwise normal variation is interpolated across paintstroke segments, producing properly curved specular highlights, whereas the cones of cone-spheres have straight highlights that are merely blended together at the spheres to smooth possible corners.

As a final problem, when many short cone-spheres are concatenated to improve the silhouette and highlights of the resulting tube, the spheres become increasingly enclosed within the cones, and therefore unexposed. Yet, they are still rendered in their entirety, only to be covered by the adjacent cones. As with the brush extrusion approach, this type of pixel overwriting is inefficient.

Despite these shortcomings, cone-spheres are useful for rendering fairly straight tubes, especially with texture-mapping or bump-mapping, which paintstrokes are currently unable to provide.

### 5.4.3 Polylines with Precomputed Shading

When used at a thickness of several subpixels to a pixel, polyline methods with precomputed shading prove to be an efficient and high-quality method for rendering generalized cylinders. Their shading model eliminates the aliasing artifacts that creep into paintstrokes at these small scales, and it does so without incurring the expense of massive oversampling, as the paintstrokes are forced to do. Incorporating this technique's pre-integrated shading model into quality-zero paintstrokes would extend the latter's scope to much smaller-scale geometry.

While polylines work extremely well within the above range of thicknesses, they perform poorly at larger or smaller scales. One reason for the former is that each line segment has a constant colour, and therefore cannot express the breadthwise shading variation one would expect to see in a real tube at close range. Another reason is that adjacent line segments cannot always be seamlessly joined together, given that they are drawn as skinny rectangles. From some angles, the joints will either contain cracks or will have corners jutting out, which, though inconspicuous at small scales, becomes increasingly noticeable as the tube's screen thickness increases. While the latter problem can be eliminated by using mitre joints, this actually means drawing trapezoidal polygons instead of line segments[11], which is more expensive and arguably not a true polyline solution. At smaller scales, polylines can become prone to aliasing due to their sub-subpixel thicknesses. Moreover, larger and larger numbers of these (however inexpensive) primitives are needed to fill a certain volume. At this point, constant-time methods such as texture mapping and volumetric rendering become appropriate alternatives.

Thus, polylines are a superior alternative to paintstrokes when used at a small (subpixel thickness) scale. By incorporating their pre-computed shading model into quality-zero paintstrokes, the polyline's advantages at small scales could be seamlessly integrated with the paintstroke's advantages at larger ones, all in a single hybrid primitive. Such an integration presents an intriguing opportunity for future work.

[11]This is similar in principle to quality-zero paintstrokes.

## 5.5 Comparison with Global Texture-Mapping Methods

Rendering geometrically rich models by texture-mapping is a viable alternative to using paintstrokes, but only at a very small scale. Because it does not capture the full three-dimensional geometry of a scene, a texture map quickly loses its image quality as the scale of the textured objects grows. At larger scales, the effects of occlusion and local illumination within a geometric model become increasingly view-dependent, and the failure of texture-mapping to account for this tends to result in unrealistic, flat-looking images, especially when viewed in motion. Moreover, objects incorporated into textures are difficult to animate without directly re-rendering the geometry. At small scales (e.g. fur viewed from far away) animation may not be necessary, as the motions tend to be inconspicuous. But at larger scales, the need for animation can pose a problem for global texture-mapping.

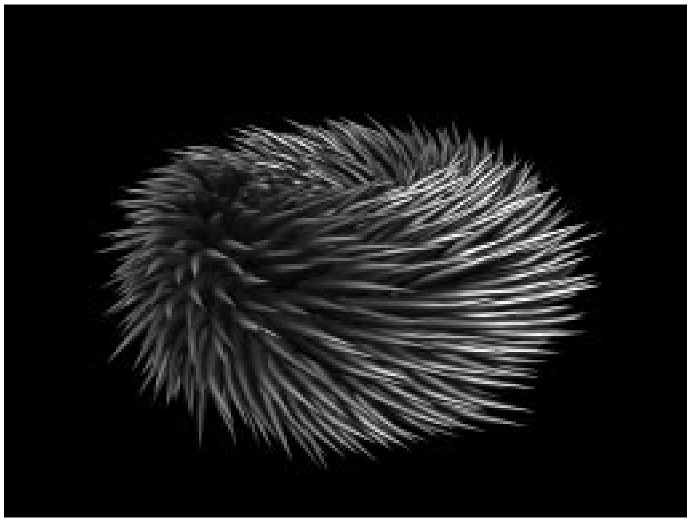


Figure 5.9: Example of an image not suitable for rendering with a texture map.

Although the hierarchical image caching approach by Shade *et al.* [SLS+96] helps to alleviate the problems with parallax, it essentially trades parallax error for rendering time. This works well at moderately small scales, but as objects approach the viewer, the lifespan of their cached images becomes smaller and smaller, requiring them to be frequently re-rendered. This technique is not really a competitor to the paintstroke, but a framework within which paintstrokes and other rendering methods could be incorporated.

## 5.6 Comparison with Volumetric Textures

Because they entail the significant per-pixel expense of using volumetric ray-tracing, volumetric textures represent an efficient alternative to paintstrokes only at a tiny scale, well below any reasonable size for paintstrokes. Their near-constant rendering time involves a high overhead, so texels need to contain a great deal of detail in order to make this method worthwhile. Used appropriately, however, volumetric rendering is unparalleled in its ability to generate images of high quality with very minor aliasing artifacts, and doing so at a low cost relative to the other methods discussed in this chapter (except global texture-mapping). These strengths make volumetric textures a vastly superior method to paintstrokes, polylines, and other methods that attempt to render a scene's geometry in its full detail at extremely small scales.

Whereas mesh simplification is effective for approximating shapes, it fails to accurately maintain the normal distribution of the underlying geometry. Because volumetric textures contain exact reflectance distributions, rather than approximating them from the geometry, they do not succumb to this problem; their approach is specifically geared to shading microgeometry, which is difficult and expensive to accomplish with polygonal representations.

Although considerably slower than the global texture-mapping methods of §5.5, volumetric rendering generates images of much higher quality: By storing density and reflectance distributions, texels accurately capture the small-scale appearance and anisotropic reflectance of the true three-dimensional geometry they represent. The two methods are not incompatible, however—volumetric rendering can serve to provide the cheaper global texture-mapping methods with high-quality pre-rendered images that are needed to construct their texture maps.

## 5.7 Summary

In this chapter we have argued that paintstrokes provide an efficient alternative to other methods in rendering generalized cylinders, albeit within a limited range of scales. Although much of our reasoning is based on a comparison with only a statically tessellated model, the superior performance of paintstrokes even at the latter's ideal level of detail attests to the advantage of their view-dependent tessellation over the other more general methods.

# Chapter 6

# Conclusions and Future Work

## 6.1 Conclusions

### 6.1.1 Summary

A wide variety of models used in computer graphics can be reasonably approximated by generalized cylinders. An efficient technique for rendering the latter is therefore of considerable value. While a number of traditional rendering methods have been applied to the task, they generally fail to achieve a good balance of speed and image quality at small to medium scales. The purpose of this thesis was to provide an efficient means of rendering generalized cylinders at precisely these scales. This was achieved through the paintstroke primitive and its supporting A-Buffer-based projective rendering architecture.

By applying a view-adaptive tessellation algorithm that exploits the simplicity and symmetry of the generalized cylinder's screen-space projection, paintstrokes are able to accurately approximate this surface using much fewer polygons than competing methods. Because of their view-dependent arrangement, a paintstroke's polygons capture the generalized cylinder's appearance from all viewing directions[1], despite the coarseness of their tessellation.

The polygon renderer we have developed incorporates variable-resolution Phong shading within an A-Buffer framework, thereby efficiently reducing spatial aliasing along silhouettes and near specular highlights. This allows paintstrokes to be used at the relatively small scales for which they are intended, scales at which the faster, non-antialiasing methods like the Z-Buffer fail to produce images of reasonable quality.

[1]With the few exceptions noted in §5.1.3.

### 6.1.2 Contributions

While the general elements of our solution—view-dependent tessellation, the A-Buffer, and adaptive Phong shading—are not new, their integration into a technique for rendering generalized cylinders is unique. The following are our major contributions toward this technique.

1. Tessellation of Generalized Cylinders

    - Representing a generalized cylinder using a parametric spline path with the radius specified at points along the path.
    - Adaptively subdividing the generalized cylinder along the path, based on the latter's screen-space curvature.
    - Using the geometry of the path to determine the positions and normals of the view-dependent edges and centre of the generalized cylinder at any point along the path.

2. View-Dependent Rendering Effects

    - Simulating global shading by estimating the penetration distance of a light ray to a point within a sphere.
    - Simulating volumetric opacity using the path tangent and normal vectors of a generalized cylinder.

3. Adaptive Phong Shading

    - Estimating the maximum rate of angular variation during the componentwise linear interpolation of a surface normal over a polygon.
    - Using the above, along with the polygon's size, orientation, and reflectance properties, to determine an appropriate horizontal and vertical Phong sampling rate over the polygon.

4. A-Buffer

- Presenting a rigorous derivation of Carpenter's formulas [Car84] for blending fragments.
- Improving upon Carpenter's blending formula for intersecting fragments.

## 6.2 Directions for Future Work

A number of potential enhancements to the paintstroke primitive have been discussed in the preceding chapters, particularly in Chapter 5. Some of these are straightforward to implement, while others represent major avenues for future work. In this section we briefly examine the latter.

### 6.2.1 Alternative Representation of Surface Normals

As we have seen in §3.2.4, the accuracy of a paintstroke's shading is highly dependent on its quality level. The lower the quality level, the smaller the number of polygons over which the paintstroke's (large) breadthwise normal variation is interpolated, resulting in a less accurate approximation of a generalized cylinder's circular profile. As we have established in §3.2.4, linearly interpolating the components of a normal vector across the breadth of a level-zero paintstroke yields a normal distribution that differs noticeably from that of a generalized cylinder. Moreover, the distribution becomes increasingly prone to aliasing as the range of interpolation approaches $180^0$. While quality-zero paintstrokes redress the latter problem by using using less than $180^0$ of breadthwise curvature, this does nothing to correct (and, indeed, exacerbates) the former.

There is an alternative way to represent surface normals, which would enhance the shading accuracy for all paintstrokes, particularly those of quality zero. Furthermore, it would make shading differences among the three quality levels very small, eliminating the potential popping that can arise when a paintstroke's quality level changes. The idea is to represent the normal as a pair of angles $\phi$ and $\theta$, corresponding to the spherical coordinates of a point on the unit sphere, expressed in rectangular coordinates as $[\cos\theta\sin\phi, \sin\theta\sin\phi, \cos\phi]$. While the rectangular representation would be used in computing the basic elements of the Phong model, namely, $\mathbf{N}\cdot\mathbf{L}$ and $\mathbf{N}\cdot\mathbf{H}$, the normal interpolation would be linear in $\phi$ and $\theta$, using a single table of sine values to provide quick approximations to $\sin\phi$, $\cos\phi$, $\sin\theta$, and $\cos\theta$. The four table indices corresponding to the sines and cosines of the initial values for $\phi$ and $\theta$ would be computed, and then simply incremented by constant amounts to locate the sines and cosines for subsequent angular values.

While this technique may seem expensive compared to the traditional interpolation in rectangular coordinates, notice that it requires no normalization of the interpolated normal. This is a big advantage, since even table-assisted normalization requires computing a vector's norm, and then multiplying it by an appropriate scalar. More importantly, this approach allows a single polygon to cover a full $180^0$ of curvature and still closely approximate the generalized cylinder's "circular" normal distribution. Whereas (one-dimensional) normal interpolation in rectangular coordinates yields the normal profile of a parabola, this new method produces a distribution based on the curve $-\ln(\cos\theta)$, an antiderivative of $\tan\theta$. This can be derived by an approach similar to the one presented in §3.2.4. As Figure 6.1 illustrates, this curve's normal profile closely approximates that of a circle. Note also that the derivative of the normal vector with respect to $\theta$ (the interpolation distance) approaches infinity along the circle's edges, whereas for the angular interpolant, this derivative has a bounded, constant norm. This means the interpolated distribution is less susceptible to aliasing than the true distribution of a generalized cylinder, which is ill-behaved near the edges.

To apply this method to paintstrokes, an angular representation of their normals needs to be derived. While this can be trivially accomplished by computing the normals in rectangular coordinates (as we currently do) and then converting them to the spherical representation, it may be more efficient to directly compute the normals in the spherical coordinates. Also, if an inexpensive angular-coordinate transformation can be used to rotate world-space normals into eye-space, this new representation would be feasible for general polygonal models as well. Clearly, this approach warrants further examination.

### 6.2.2 Non-Circular Cross-Sections for Paintstrokes

Although a paintstroke's shape is quite general, there are a variety of elongated objects that the primitive cannot capture. These include blades of grass, feathers, leaves, cer-

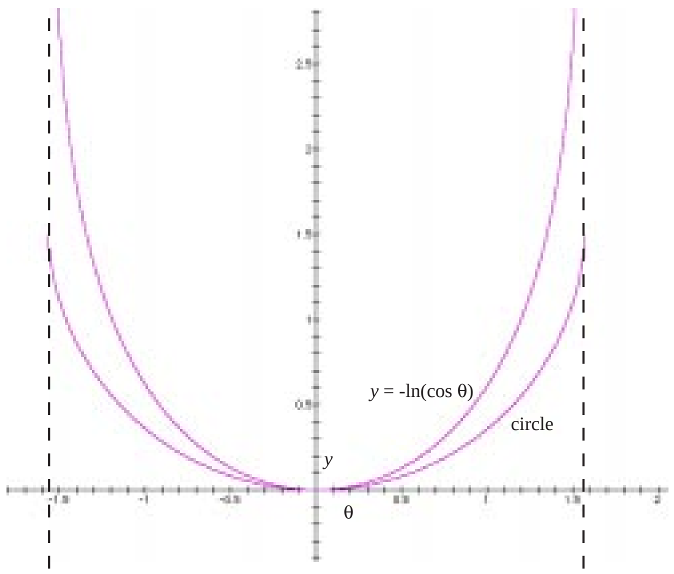


Figure 6.1: Circular vs. angle-interpolated normal profiles.

tain types of fish, and similar long, flat objects. Extending our primitive's scope to include tubes of variable cross-section would permit a much wider variety of objects to be modelled using paintstrokes.

The challenge in implementing this extension is the following: the more a paintstroke's shape is generalized, the more its symmetry is reduced and the complexity of its screen projection increased. Yet, most of the paintstroke's efficiency derives from the symmetry of the generalized cylinder and the simplicity of its screen-space projection. One solution that may provide greater modelling flexibility without greatly complicating the paintstroke's tessellation is to use an elliptical cross-section. As a start, the silhouette of this primitive could be approximated from most angles with a standard paintstroke whose radius varies around its girth, according to the viewing direction. This could be implemented by incorporating a binormal vector into each control point, which, together with the path's tangent, would specify a Frenet frame. However, providing accurate shading for such a paintstroke, as well as ensuring a correct head-on view clearly requires further work.

### 6.2.3 Texture-Mapping

A general limitation of paintstrokes is their inability to bear textures. This is a shortcoming we hope to redress in a future version, although, as mentioned in Chapter 3, the ability to texture-map small-scale objects like paintstrokes is usually not of paramount importance. At the small end of their useful range of scales, paintstrokes are too thin to allow a two-dimensional texture on them to be discerned; in this case, simple lengthwise colour variation, as we have provided, is all that is required. On the other hand, larger paintstrokes would probably benefit from texture-mapping.

A simpler but also highly useful feature would be the application of one-dimensional lengthwise textures. Although this can currently be simulated by using control points to specify colour variation along the paintstroke's path, the large amount of geometric overhead involved makes this approach inefficient. Thus, an efficient implementation of one-dimensional textures, and possibly traditional two-dimensional ones, would make a useful addition to our primitive.

# Appendix A

# Details on the Polygon Renderer

This chapter provides additional implementation details on our polygon rendering engine. It discusses the specifics of the rasterization algorithm, and the way in which the variable sampling rate for the shading model is determined.

## A.1 Rasterization Algorithm

The rasterization proceeds as follows. The plane equation and unit increments for each component of the interpolant vector (as defined in §4.3.2) are computed, and then the polygon is scanned vertically, and at each vertical step, horizontally. The vertical scanning proceeds one subpixel row at a time. The $x$-position component is interpolated from top to bottom along the left and right edges of the polygon, while the other elements of the interpolant vector are scanned only along the left edge. For them we do not require a right endpoint, since their horizontal increments are already known from the plane equations, so all we need is a starting point.

The interpolant vectors at the left and right edges begin at the same height, that of the highest vertex (or vertices). Prior to commencing the actual scanning, they are shifted vertically a fraction of a subpixel, so as to position them exactly in the vertical middle of the subpixel row they occupy. This initial nudge serves to align the interpolant vectors with the vertical component of our sampling grid. A similar nudge occurs prior to scanning each row, to achieve horizontal alignment. As shown in Figure A.1, a proper alignment of the sampling positions is important because it provides a fast, accurate, and

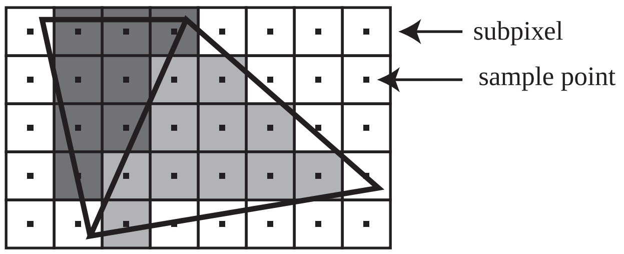


Figure A.1: Sampling positions for rasterization.

consistent way of determining which subpixels lie within a polygon and which don't.[1] This avoids the problem of doubly covering subpixels along a boundary shared by two polygons, which, aside from being inefficient, produces higher-opacity seams along the edges when the polygons are transparent.

Once the interpolant vectors are vertically aligned, they move down in unison, marking the endpoints of the long rows of subpixels between them. We call these rows *subpixel scanlines*, and the interpolant vectors at their endpoints *endpoint vectors*. At each subpixel scanline, the endpoint vectors are stored in an array. When all the subpixel scanlines in a row of fragments have been traversed, the fragment row is scanned horizontally, as described in the following paragraph. After that, the vertical scanning resumes, capturing the endpoint vectors for the subpixel scanlines of the following fragment row, and so on. The left endpoint vector contains all the values to be interpolated, while the right one has only the $x$-position.

Horizontal scanning interpolates components of the left endpoint vector of each subpixel scanline, moving one pixel to the right at each step and possibly skipping over any fully covered fragments along the way, depending on the scanline in question. Each horizontal step involves adding the constant horizontal pixel increment, which equals 8 times the subpixel increment obtained from the plane equations.

At this point, the initial horizontal nudge is applied to ensure horizontal alignment with the subpixel grid. For reasons that will be explained shortly, only the first and fourth subpixel scanlines, which run approximately through the top and middle of the fragment

[1]Sample points that straddle an edge are (arbitrarily) considered interior for the left edge and exterior for the right. If the edge is horizontal, an analogous rule is applied based whether it is the top or bottom edge.

row, respectively, are scanned across fully covered fragments. The remainder skip across them to any partially covered fragments on the right side of the polygon. As Figures 4.7 and A.2 illustrate, the full fragments can only occur in a contiguous block flanked by partially filled ones. And similarly, full subpixel rows must occur in a contiguous block, possibly flanked by a partially covered row at either end. These facts follow from our assumption that all rendered polygons are convex.

The reader may be wondering why the endpoint vectors for all the subpixel scanlines in the fragment row were stored during the vertical scanning, only to be traversed again during horizontal scanning. Indeed, it seems simpler to horizontally interpolate each subpixel scanline as its endpoint vectors are computed, rather than storing them in an array and then using them later. The reason we compute all the endpoint rows before scanning them is because we need all eight of them to determine which fragments in the row are fully covered. Since six of the eight subpixel scanlines will skip over these fragments, we cannot scan them until we know the range of fully covered fragments in the row.

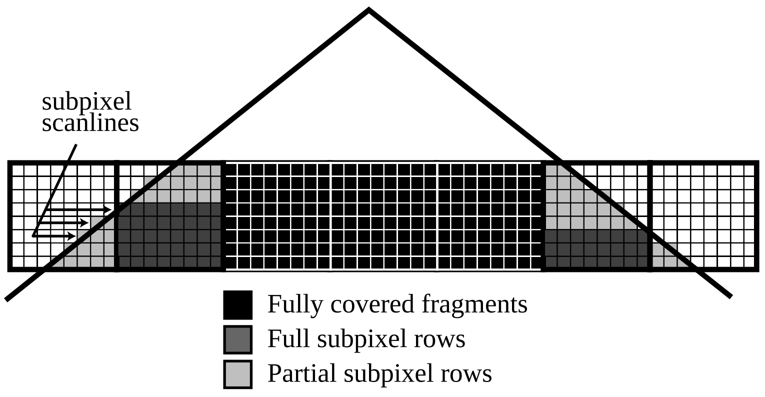


Figure A.2: Close-up of a fragment row.

Figure A.2 shows the three types of fragment rows our algorithm may encounter: partially covered rows, fully covered rows belonging to partially covered fragments, and (fully covered) rows belonging to fully covered fragments. Each type is rasterized in a specific way that best capitalizes on the coherence of its coverage. Before we look at the specific cases, we complete our general discussion of horizontal rasterization.

At each subpixel scanline, the $x$-components of the endpoint vectors are used to locate which fragment and subpixel within that fragment begins and ends the scanline.[2] These values will be used in constructing the coverage masks of the fragments, as well as determining the range of fully covered fragments in the row, if there are any.

The `colour`, `opacity`, $k_s$, and $k_d$ values associated with each fragment in the row are calculated as a weighted average of the respective interpolated values at each subpixel in the coverage mask. As will be shown below, we compute these quantities without actually sampling each subpixel by recognizing that for convex polygons the subpixels must always be covered in contiguous rows and columns. While the normal components could also be averaged over the fragment, they are not; their raw interpolated values are directly used by the shading model, as was explained in §4.3.6

The $x$- and $z$- values of the left endpoint vector and the plane equation of the polygon are used to derive the minimum and maximum $z$-values for each subpixel row of the leftmost pixel. These extrema are computed for subsequent fragments along the subpixel scanlines by adding the constant horizontal $z$-increment, based on the appropriate plane equation. Because the $z$-values are sampled at the vertical middle of a subpixel scanline, they do not yield exact extrema (which occur at subpixel corners). This sampling error, while significant with respect to the subpixel, is very small relative to the full fragment, since *any* sample within the closest and farthest subpixels is still a good approximation for $\mathtt{Z}_{\mathtt{min}}$ and $\mathtt{Z}_{\mathtt{max}}$. Moreover, the formula that uses these values is something of an approximation itself (see §B.2.7), so the small inaccuracy is of no consequence.

Partially covered rows are the worst-case scenario for our scan-conversion algorithm, requiring two samples per subpixel row. The bitmap for the subpixel row is produced using the formula $(2^{8-left} - 1)$ `xor` $(2^{7-right} - 1)$, where $left$ and $right$ denote the integer endpoints in subpixel coordinates. This value is then left-shifted by the appropriate number of bits to place it in the correct row within the coverage mask, leaving the others filled with zeros. The result is `or`ed with the coverage mask under construction, effectively inserting the new row into the final mask. The colour and opacity values are found by sampling at the horizontal and vertical middle of the row, the former given

[2]This involves dividing by the horizontal subpixel resolution and obtaining the remainder. Since the resolution is a power of two, the remainder can be quickly computed using a logical `and` operation.

by the expression $\frac{left+right}{2}$, and the latter taking a value of $0.5, 1.5, \ldots, 7.5$ in subpixel coordinates. This yields an accurate subpixel-area-weighted average, unlike with the faster method that samples a partially covered pixel only once, such as the one in [Car84]. Finally, the $z$-values at the endpoints of the row are compared with the with $z$-extrema of the other rows in the fragment, as candidates for $\texttt{Z}_{\texttt{min}}$ and $\texttt{Z}_{\texttt{max}}$. This routine is further optimized for the special cases where $right = 7$ or $left = 0$, corresponding to the leftmost and rightmost fragments in a multipixel row.

Any rows that are fully covered but belong to a partially covered fragment can be handled more expeditiously. There are eight possible coverage masks having exactly one full row of subpixels, and their numeric representations are obtained by shifting the integer $2^8 - 1$ by $0, 8, 16, \ldots, 56$ bits to the left. Depending on the row in question, the appropriate value is `or`ed with the coverage mask under construction. The colour, opacity, and reflectance values are sampled at the horizontal and vertical centre of the row, multiplied by a weighting factor of 8, and added to the fragment's `opacity` and `colour` fields. Once all the subpixel scanlines within the pixel row have been traversed, the values in these fields for each fragment are divided by its subpixel count to obtain the respective average values. The $\texttt{Z}_{\texttt{min}}$ and $\texttt{Z}_{\texttt{max}}$ are determined by sampling the first and last subpixels in the row. Once these values are computed for the leftmost fragment, they are incremented by 8 times their respective subpixel increments, effectively scanning the subsequent fragments one at a time. This continues until either a block of fully covered fragments is reached or a partially covered row marks the end of the subpixel scanline. In the former case, the block of fully covered fragments is either skipped, by incrementing the interpolant accordingly, or it continues, depending on which subpixel scanline it is.

The rasterization of fully covered fragments is the most highly optimized. The entire coverage mask can be generated in a single step by assigning the value of $2^{64} - 1$ (or, using two's complement, -1) to a 64-bit integer variable that stores the bitmap. A single sample positioned near the centre of the coverage mask grid[3] yields a very close approximation to the average value for the colour, opacity, and reflectance values of the fragment. For

[3]The subpixel position is (4,3.5). The vertical component is slightly off-centre because the samples occur along the fourth subpixel row, and each subpixel row is sampled at its vertical centre. The error of half a subpixel is negligible relative to 8 subpixel height of the fragment.

the $\texttt{Z}_{\texttt{min}}$ and $\texttt{Z}_{\texttt{max}}$ values, we first determine the $z$-value of the top corner subpixel where either $\texttt{Z}_{\texttt{min}}$ or $\texttt{Z}_{\texttt{max}}$ occurs, and then apply an offset to it, based on the polygon's plane equation, to obtain the other extremum. Once we have the $z$-extrema of the leftmost fully covered fragment, we apply fixed horizontal increments to them to obtain their values for subsequent fragments to the right. Finding these offsets and matching the top corner to an extremum are easily accomplished by using the appropriate plane equation of the polygon. Because the $z$-samples are taken along the top of the fragment row, the interpolation occurs only on the first scanline. So to summarize, the $\texttt{Z}_{\texttt{min}}$ and $\texttt{Z}_{\texttt{max}}$ are interpolated along the first scanline, and the other interpolated components along the fourth. All the other scanlines are skipped over. As usual, the normal component is an exception. Its interpolation is discussed in the following section.

No perspective correction is applied to the non-positional interpolated quantities, since the primary target of our rendering engine—the paintstroke polygon—exhibits only minor perspective effects at its intended scale, so the additional per-pixel division that is required would have little benefit to counterbalance its cost.

## A.2 Determining the Sampling Rate of the Shading Model

The sampling grid used at each pixel is based on the required horizontal and vertical sampling rates, which are independently computed once per polygon. Fully covered fragments are sampled exactly according to this grid. But for partially covered fragments, the grid's horizontal and vertical densities are increased, if necessary, to ensure that at least one sample per (horizontal and vertical) dimension occurs. For example, a fragment with only one subpixel of coverage will always be sampled at the maximum sampling rate, ensuring that the one sample is taken. For an arbitrary partially covered fragment, we determine the horizontal and vertical span of its contiguous subpixel coverage (these are computed while the coverage masks are being built), and set the sampling rates for that fragment to the lowest values such that the horizontal and vertical distances between sample points do not exceed the horizontal and vertical coverage spans. This prevents the coverage mask from "falling through the cracks" of the sampling grid.

Once the horizontal and vertical unit increments of the normals have been computed from the polygon's plane equations, the next step is to find the maximum rate of angular normal variation in the horizontal and vertical directions. These values are dependent on the amount of curvature the polygon interpolates, based on the normal values at its vertices. To obtain the values, we first need the shortest length attained by the interpolated normal over the entire polygon. That length, along with the lengths of the unit increments, will provide a measure of the maximum angular change per horizontal or vertical step. The shortest length, which we denote by the scalar $N_{min}$, can be calculated given the magnitude of the maximum range of normals over the polygon, $R$. We compute $R$ as the maximum norm of the differences in normals between any pair of vertices $i$ and $j$:

$$R = \max_{i,j} ||\mathbf{N_i} - \mathbf{N_j}|| \tag{A.1}$$

When the value of $R$ is found, it is easy to obtain $N_{min}$ using the equation below, as illustrated in Figure A.3. But before we do this, we check to see if the value of R is extremely small. If it is, we consider the polygon "flat" and simply set the horizontal and the vertical sampling rates to once per pixel (the lowest rate), and skip the remaining tests.

$$N_{min} = \frac{1}{\sqrt{1-(R/2)^2}} \tag{A.2}$$

Finally, the sines of the angular steps $\theta_h$ and $\theta_v$, corresponding to one subpixel of horizontal or vertical normal variation ($\Delta N_h$ or $\Delta N_v$) are given by

$$\sin\theta_h = \frac{\Delta N_h}{\sqrt{(\Delta N_h)^2 + N_{min}^2}} \tag{A.3}$$

$$\sin\theta_v = \frac{\Delta N_v}{\sqrt{(\Delta N_v)^2 + N_{min}^2}} \tag{A.4}$$

The horizontal and vertical sampling rates $s_h$ and $s_v$ are then chosen, each bearing a value of 1, 2, 4, or 8 samples per pixel. The value for $s_h$ is the lowest of these that keeps $\frac{8}{s_h}\sin\theta_h$ below a user-specified threshold $tol_s$. The equivalent criterion is also applied to $s_v$. To improve performance, the formulas are squared and multiplied out so as to eliminate the expensive square root and division operations.

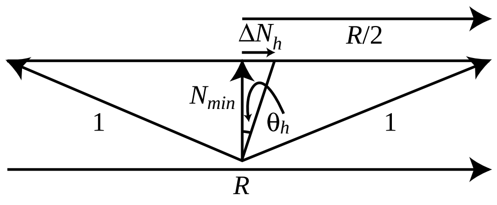


Figure A.3: $\theta_h$, the greatest angular increment for the horizontal scanning direction. ($\theta_v$ is analogous.)

Next, three tests are applied. If any component of the halfway vector lies outside the corresponding component's range of the polygon's normals, the maximum absolute difference between the two is used to adjust $tol_s$ to favour a lower sampling rate. Similarly, if the maximum value of $k_s$ over the entire polygon is low, that is used to relax the threshold further. Lastly, a small value of $e_s$ (a constant over the polygon), relaxes the threshold further still. The cumulative effect of the adjustments is captured through multiplication: given adjustment factors $a_1$, $a_2$, and $a_3$, based on the three tests, the original threshold value $tol_s$ is re-assigned as follows:

$$tol_s := a_1 a_2 a_3 \, tol_s \tag{A.5}$$

# Appendix B

# Blending the A-Buffer Fragments

In this chapter we present the pseudocode structure of the `BlendFragment` function, and derive the blending formulas it uses.

## B.1 The `BlendFragment` Function

The `BlendFragment` function takes four parameters: the top fragment *top*, the search mask *smask*, and the blended colour and alpha values $C_{blend}$ and $\alpha_{blend}$. Whereas the first two parameters contain input, the last two are used to store the output.

```
BlendFragment(Fragment: $top$, Mask: $smask$, var Colour: $C_{blend}$, var Alpha: $\alpha_{blend}$)
    $M_{in} := smask \cap top_{\texttt{mask}}$
    $A_{in} :=$BitCount$(top_{\texttt{mask}})/64$
    $\alpha_{blend} := top_{\texttt{opacity}} \cdot A_{in}$
    if $top_{\texttt{next}}$ = nil then
        $\boxed{\text{This is the last fragment in the list. We return its colour and coverage.}}$
        $C_{blend} := top_{\texttt{colour}}$
        return
    end if
    $M_{out} := smask \cap \neg top_{\texttt{mask}}$
    $under := top_{\texttt{next}}$
    $M_{inter} := M_{in} \cap under_{\texttt{mask}}$
```

```
if $top_{\texttt{Zmax}} > under_{\texttt{Zmin}}$ and $M_{inter} \neq \emptyset$ then
    The top two fragments intersect. We blend them using
    Blending Formula 4(b) [see §B.2.8].

    $k := \frac{under_{\texttt{Zmax}} - top_{\texttt{Zmin}}}{top_{\texttt{Zmax}} - top_{\texttt{Zmin}} + under_{\texttt{Zmax}} - under_{\texttt{Zmin}}}$
    $\alpha_{top} := \alpha_{blend}$
    $A_{under} := \texttt{BitCount}(under_{\texttt{mask}})/64$
    $\alpha_{under} := under_{\texttt{opacity}} \cdot A_{under}$
    $\alpha_{blend} := \alpha_{top} + \alpha_{under} \cdot (1 - top_{\texttt{opacity}})$
    $C_{blend} := \frac{top_{\texttt{colour}} \cdot [\alpha_{top} - (1-k) \cdot \alpha_{under} \cdot top_{\texttt{opacity}}] + under_{\texttt{colour}} \cdot \alpha_{under} \cdot (1 - k \cdot top_{\texttt{opacity}})}{\alpha_{blend}}$
      if $under_{\texttt{next}} \neq$nil and $A_{in} > 0$ and $\alpha_{blend} < A_{in}$ then
        We now blend the (blended) intersecting fragments with
        all the fragments underneath them, using Blending For-
        mula 4(a) [see §B.2.8].

        BlendFragment($under_{\texttt{next}}, M_{in}, C_{under}, \alpha_{under}$)
        $\alpha_{inter} := \alpha_{blend}$
        $C_{inter} := C_{blend}$
        $o_{inter} := \alpha_{blend}/A_{in}$
        $\alpha_{blend} := \alpha_{inter} + (1 - o_{inter}) \cdot \alpha_{under}$
        $C_{blend} := \frac{C_{inter} \cdot \alpha_{inter} + C_{under} \cdot (1 - o_{inter}) \cdot \alpha_{under}}{\alpha_{blend}}$
    end if
    if $M_{out} \neq \emptyset$ then
        Lastly, we blend the above result with the surrounding
        region within the search mask. We use Blending Formula
        2 [see §B.2.6].

        BlendFragment($top_{\texttt{next}}, M_{out}, C_{out}, \alpha_{out}$)
        $\alpha_{in} := \alpha_{blend}$
        $C_{in} := C_{blend}$
        $\alpha_{blend} := \alpha_{in} + \alpha_{out}$
        $C_{blend} := \frac{\alpha_{in} \cdot C_{in} + \alpha_{out} \cdot C_{out}}{\alpha_{blend}}$
    end if

else if $top_{\texttt{opacity}} < 1$ then
```

```
        Top fragment is transparent and does not intersect the
        next one. We apply Blending Formula 4(a) [see §B.2.8].
        BlendFragment$(under, M_{in}, C_{under}, \alpha_{under})$
        $\alpha_{top} := \alpha_{blend}$
        $\alpha_{blend} := \alpha_{top} + (1 - top_{\texttt{opacity}}) \cdot \alpha_{under}$
        $C_{blend} := \frac{top_{\texttt{colour}} \cdot \alpha_{top} + C_{under} \cdot (1 - top_{\texttt{opacity}}) \cdot \alpha_{under}}{\alpha_{blend}}$
        if $M_{out} \neq \emptyset$ then
            We blend the above result with the surrounding region
            within the search, using Blending Formula 2 [see §B.2.6].
            BlendFragment$(under, M_{out}, C_{out}, \alpha_{out})$
            $\alpha_{in} := \alpha_{blend}$
            $C_{in} := C_{blend}$
            $\alpha_{blend} := \alpha_{in} + \alpha_{out}$
            $C_{blend} := \frac{\alpha_{in} \cdot C_{in} + \alpha_{out} \cdot C_{out}}{\alpha_{blend}}$
        end if


    else
        Top fragment is opaque and does not intersect the next
        one. (This is the most common case.) We apply Blending
        Formula 2 [see §B.2.6] to blend it with the surrounding
        region within the search mask.
        if $M_{out} = \emptyset$ then
            $C_{blend} := top_{\texttt{colour}}$
        else
            BlendFragment$(under, M_{out}, C_{out}, \alpha_{out})$
            $\alpha_{in} := \alpha_{blend}$
            $\alpha_{blend} := \alpha_{in} + \alpha_{out}$
            $C_{blend} := \frac{\alpha_{in} \cdot top_{\texttt{colour}} + \alpha_{out} \cdot C_{out}}{\alpha_{blend}}$
        end if
    end if
end
```

Before proceeding to the blending formulas, we present a few implementation details which aid in understanding the overall algorithm. The set operators $\cap$ and $\cup$, which are used to combine coverage masks, are implemented using bitwise `and` and `or` operators, respectively. The speed at which these operations can be performed in large measure accounts for the A-Buffer's efficiency. The `BitCount()` function counts the number of one bits in a coverage mask, stripping off eight bits at a time and using their numeric representation as an index into a table of precomputed bit counts. The formulas involving colour are essentially vector equations; the three colour components can be treated as independent scalars.

## B.2 The Blending Formulas

We begin by introducing some basic definitions and axioms, which will serve as a foundation for the derivations that lie ahead.

### B.2.1 Basic Definitions

Let $a$ be an arbitrary fragment. The following definitions of $a$'s attributes are device-independent analogues of the fields described in section 4.2. We have shortened their names in order to keep our equations at a reasonable length.

**Def'n I** $M_a$ is the *coverage mask* of $a$, represented as a function that maps a point from the unit square to a value of zero or one. Because it is convenient to use set notation when dealing with $M_a$, we define some useful set operators in terms of arbitrary fragments $a$ and $b$:

(a) $M_a : (x, y) \longrightarrow m$, where $(x, y) \in \mathbb{R}^2$ and $m \in \mathbb{Z}$, such that $0 \leq x, y, m \leq 1$

(b)

$$(M_a \cup M_b)(x, y) = \begin{cases} 1 & M_a(x, y) = 1 \quad \text{or} \quad M_b(x, y) = 1 \\ 0 & \text{otherwise} \end{cases}$$

(c)

$$(M_a \cap M_b)(x, y) = \begin{cases} 1 & M_a(x, y) = 1 \quad \text{and} \quad M_b(x, y) = 1 \\ 0 & \text{otherwise} \end{cases}$$

(d) $M_a = \emptyset$ if and only if $M_a(x,y) = 0, \forall\, x, y$

**Def'n II** $A_a$ is the *area* of the fragment $a$'s coverage mask. It is a real number obtained by integrating $M_a$ over the unit square.

$A_a = \int_0^1 \int_0^1 M_a(x,y)dy\,dx$

**Def'n III** $C_a$ is the *colour* of $a$, represented as an arbitrary vector.

**Def'n IV** $o_a$ is the *opacity* of $a$. It is a real number in the range $[0,1]$, where a value of zero represents complete transparency and a value of one complete opacity.

**Def'n V** $\alpha_a$ is the *coverage* of $a$. It is defined as area times opacity, and represents the degree to which the colour of $a$ influences the final blended colour of the pixel.

$\alpha_a = A_a o_a$

**Def'n VI** $Zmin_a$ and $Zmax_a$ are the respective minimum and maximum $z$-values attained by $a$, expressed as real numbers. We assume that the positive $z$-axis points away from the viewer.

**Def'n VII** $a \oplus b$ is a virtual fragment derived from blending $a$ with an arbitrary fragment $b$. The properties of $a \oplus b$ are defined by the axioms below.

**Def'n VIII** $b$ is said to be *behind* $a$ iff $Zmin_a \leq Zmin_b$. Note that this definition allows for $a$ and $b$ to intersect, as described in the following definition.

**Def'n IX** If $b$ is behind $a$ and $Zmax_a > Zmin_b$ and $M_a \cap M_b \neq \emptyset$, then $a$ and $b$ are said to *intersect.*[1]

**Def'n X** $a$ and $b$ are said to be *disjoint* iff $M_a \cap M_b = \emptyset$. Otherwise, they are considered *overlapping.* In the latter case, if $M_a = M_b$, they are *fully overlapping*, and otherwise, *partially overlapping.*

[1]Note that this meaning of intersection does not imply a geometric intersection between the polygons from which fragments $a$ and $b$ are derived, although it is intended to model this.

### B.2.2 Axioms

The following three axioms provide a fundamental set of rules for blending the coverage masks, opacities, and colours of two arbitrary fragments $a$ and $b$. Note that the final axiom describes how the colours may be blended, without giving an explicit formula. As we shall see, different relations between $a$ and $b$ call for different colour blending formulas; however, they all must abide by this axiom.

**Axiom I** $M_{a\oplus b} = M_a \cup M_b$

**Axiom II** (a) $o_{a\oplus b} = \frac{A_a o_a + A_b o_b}{A_a + A_b}$ for $M_a \cap M_b = \emptyset$

(b) $o_{a\oplus b} = o_a + o_b(1 - o_a)$ for $M_a = M_b$ and $Zmax_a \leq Zmin_b$

**Axiom III** $C_{a\oplus b} = \frac{sC_a + tC_b}{\alpha_{a\oplus b}}$, such that

(a) $s + t = \alpha_{a\oplus b}$

(b) $s \propto \alpha_a$ if $M_a \cap M_b = \emptyset$ or $M_a = M_b$

(c) $t \propto \alpha_b$ if $M_a \cap M_b = \emptyset$ or $M_a = M_b$

### B.2.3 Useful Derivations

The following derivations are based on the above definitions and axioms. They will be used in the recursive blending formulas described in the following section.

1. First, we establish that the blended area of a pair of disjoint fragments is the sum of their individual areas. We do this by considering Definitions I and II, and applying Axiom I.

**For** $M_a \cap M_b = \emptyset$**,**

$$\begin{aligned} A_{a\oplus b} &= \int_0^1 \int_0^1 M_{a\oplus b}(x,y)dy\,dx & \text{(B.1)} \\ &= \int_0^1 \int_0^1 (M_a \cup M_b)(x,y)dy\,dx & \text{(B.2)} \\ &= \int_0^1 \int_0^1 \left[M_a + M_b - (M_a \cap M_b)\right](x,y)dy\,dx & \text{(B.3)} \end{aligned}$$

$$= \int_0^1 \int_0^1 M_a(x,y)\,dy\,dx + \int_0^1 \int_0^1 M_b(x,y)\,dy\,dx \tag{B.4}$$
$$= A_a + A_b \tag{B.5}$$

2. We now use the above derivation along with Axiom II(a) to determine the blended coverage for a pair of fragments with *disjoint* coverage masks:

   **For $M_a \cap M_b = \emptyset$,**

$$\alpha_{a\oplus b} = A_{a\oplus b}o_{a\oplus b} \tag{B.6}$$
$$= A_{a\oplus b}\frac{A_a o_a + A_b o_b}{A_a + A_b} \tag{B.7}$$
$$= (A_a + A_b)\frac{A_a o_a + A_b o_b}{A_a + A_b} \tag{B.8}$$
$$= A_a o_a + A_b o_b \tag{B.9}$$
$$= \alpha_a + \alpha_b \tag{B.10}$$

3. Finally, we apply Axiom II(b) to derive the blended coverage for a pair of non-intersecting fragments with *identical* coverage masks:

   **For $M_a = M_b$ and $Zmax_a \leq Zmin_b$,**

$$\alpha_{a\oplus b} = A_{a\oplus b}o_{a\oplus b} \tag{B.11}$$
$$= A_{a\oplus b}\left[o_a + o_b(1 - o_a)\right] \tag{B.12}$$
$$= A_{a\oplus b}o_a + A_{a\oplus b}o_b(1 - o_a) \tag{B.13}$$
$$= A_a o_a + A_b o_b(1 - o_a) \tag{B.14}$$
$$= \alpha_a + \alpha_b(1 - o_a) \tag{B.15}$$

### B.2.4 Recursive Blending Formulas

We now have the necessary tools to derive the recursive blending formulas used in `BlendFragment`. We begin by introducing the recursive blending operator, $\tilde{}$ , which is our notational equivalent of the `BlendFragment` function. Given an arbitrary fragment $a$, the expression $\tilde{a}$ denotes a virtual fragment computed by recursively blending $a$ with all the fragments behind it. Our goal is to compute $M_{\tilde{a}}, \alpha_{\tilde{a}}$, and $C_{\tilde{a}}$.

**Base Case** If $a$ is the only fragment to be blended, then $\tilde{a}$ is trivially equal to $a$.

**Recursive Case** If there are two or more fragments to be blended, we label them $a$, $b$, $c$, $\ldots$, such that $Zmin_a \leq Zmin_b \leq Zmin_c \leq \ldots$. Except for the case where $a$ and $b$ intersect, $\tilde{a}$ is equivalent to $a \oplus \tilde{b}$. The intersecting case will be handled separately.

### B.2.5 Blending Formula 1: $M_{\tilde{a}}$

The formula for the blended coverage mask is trivial, since Axiom I applies to all fragments:

$$M_{\tilde{a}} = M_{a \oplus \tilde{b}} \quad \text{(B.16)}$$

$$= M_a \cup M_{\tilde{b}} \quad \text{(B.17)}$$

### B.2.6 Blending Formula 2: $\alpha_{\tilde{a}}$ and $C_{\tilde{a}}$ when $M_a \cap M_{\tilde{b}} = \emptyset$

In order to apply these formulas, one must ensure that the coverage masks of $a$ and $\tilde{b}$ are disjoint. The `BlendFragment` function does this without explicitly computing $M_{\tilde{b}}$, by restricting the blending of $\tilde{b}$ to the region outside of $M_a$, using the search mask.
The formulas follow directly from Derivation II and Axiom III.

**For $M_a \cap M_{\tilde{b}} = \emptyset$,**

$$\alpha_{\tilde{a}} = \alpha_{a \oplus \tilde{b}} \quad \text{(B.18)}$$

$$= \alpha_a + \alpha_{\tilde{b}} \quad \text{(B.19)}$$

$$C_{\tilde{a}} = C_{a \oplus \tilde{b}} \quad \text{(B.20)}$$

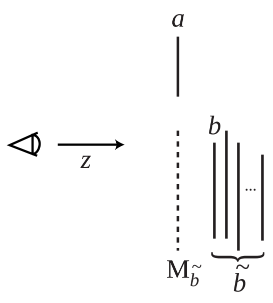


Figure B.1: Fragment arrangement suitable for Blending Formula 2.

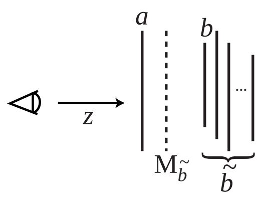


Figure B.2: Fragment arrangement suitable for Blending Formula 3(a).

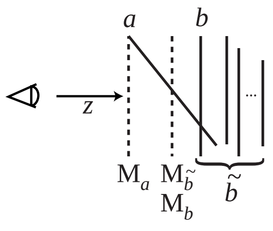


Figure B.3: Fragment arrangement suitable for Blending Formula 3(b).

$$= \frac{C_a\alpha_a + C_{\tilde{b}}\alpha_{\tilde{b}}}{\alpha_{\tilde{a}}} \tag{B.21}$$

### B.2.7 Blending Formula 3: $\alpha_{\tilde{a}}$ and $C_{\tilde{a}}$ when $M_a = M_{\tilde{b}}$

There are two possibilities here: the two foremost fragments, $a$ and $b$, may intersect (in the sense of Definition IX) or not, depending on their respective coverage masks and $Zmin$ and $Zmax$ values. If $a$ and $b$ do intersect, we add the further restriction that $M_b = M_{\tilde{b}}$, which can be satisfied by clipping all fragments behind $b$ to $M_b$. This restriction significantly simplifies the formulas.

**Blending Formula 3(a): Non-Intersecting Top Fragments ($M_a = M_{\tilde{b}}$ and $Zmax_a \leq Zmin_b$)**

These formulas follow trivially from Derivation III and Axiom III.

**For $M_a = M_{\tilde{b}}$ and $Zmax_a \leq Zmin_b$,**

$$\alpha_{\tilde{a}} = \alpha_{a \oplus \tilde{b}} \tag{B.22}$$

$$= \alpha_a + \alpha_{\tilde{b}}(1 - o_a) \tag{B.23}$$

$$C_{\tilde{a}} = C_{a \oplus \tilde{b}} \tag{B.24}$$

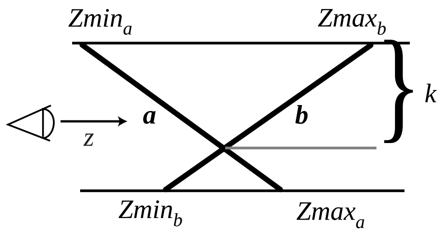
$$= \frac{C_a \alpha_a + C_{\tilde{b}} \alpha_{\tilde{b}}(1 - o_a)}{\alpha_{\tilde{a}}} \tag{B.25}$$

**Blending Formula 3(b): Intersecting Top Fragments ($M_a = M_{\tilde{b}} = M_b$ and $Zmax_a > Zmin_b$)**

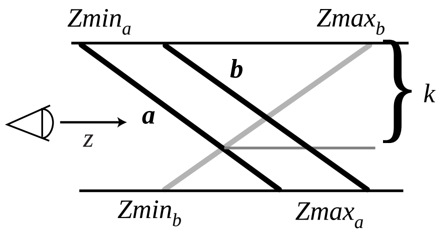
Dealing with intersecting fragments requires using a *front visibility factor*, denoted by $k \, \epsilon \, (0, 1)$. This factor estimates the proportion of fragment $a$ that is not obscured by fragment $b$. The portion of $b$ not obscured by $a$ is then weighted $1 - k$. If each fragment contained a plane equation representing the orientation of its originating polygon, it would be possible (though expensive) to compute an *exact* visibility factor for a pair of intersecting fragments by determining the line or plane of geometric intersection. However, the only geometric information stored in a fragment is the coverage mask and the $z$-value extremes. This is not enough to determine the true visibility factor, so we use Carpenter's approximation [Car84].

$$k = \frac{Zmax_b - Zmin_a}{Zmax_a - Zmin_a + Zmax_b - Zmin_b} \tag{B.26}$$

The geometric interpretation for this formula is shown in Figure B.4(a). It is important to remember that this is only an approximation and although it works well most of the time, it may produce inaccurate results, as illustrated in Figure B.4(b). As the figure shows, overlap in the coverage masks and $z$-extrema of two fragments does not even guarantee a true geometric intersection, much less specify the exact visibility of each one.



(a) Good approximation



(b) Poor approximation

Figure B.4: Geometric interpretation of $k$ for intersecting fragments $a$ and $b$.

The challenge in computing $\tilde{a}$ when the top two fragments intersect is that we cannot simply decompose it into $a \oplus \tilde{b}$, as we have done in the other cases. To explain why, we refer to Figure B.5. Note that portion $b_1$ of fragment $b$ should be blended with portion $a_2$ of fragment $a$, as well as the fragments behind $b$. The algorithm presented in [Car84] overlooks this, simply blending the front fragment with the combined result of the ones behind it. The problems of this are most evident in the following situation. Consider a transparent polygon intersecting an opaque one. Along the edge of intersection, the pixels will contain colour from items behind the opaque polygon, clearly an impossibility.

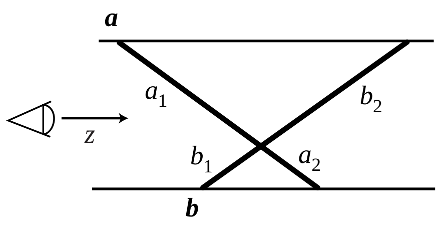


Figure B.5: Regions of Intersection.

Our solution to the problem is simple: we blend the two intersecting fragments at the top separately, and then blend that with the blended result of the rest of the fragments. This approach yields $\tilde{a} = (a \oplus b) \oplus \tilde{c}$, where $c$ is the fragment following $b$. This eliminates the above anomaly by preserving the opacity of an intersection of a transparent fragment with an opaque one, thereby preventing colour contamination from subsequent fragments. Since our algorithm assumes that intersections involve exactly two fragments, the blending of $a \oplus b$ with $\tilde{c}$ does not involve any further intersections, and is thus amenable to Blending Formula 3(a).[2]

In order to blend the intersecting fragments $a$ and $b$, we treat them as two disjoint pairs of overlapping fragments: portions $a_1, b_2$ and $b_1, a_2$ in Figure B.5. The first pair has $a$ in front followed by $b$, and has area $kA_a$ (since $A_a = A_b$, by our assumption of identical coverage masks). The second has $b$ in front and $a$ behind, with area $(1-k)A_a$.

**For $M_a = M_b$ and $Zmax_a > Zmin_b$,**

$$\begin{aligned}
\alpha_{a\oplus b} &= k\left[\alpha_a + (1-o_a)\alpha_b\right] + (1-k)\left[\alpha_b + (1-o_b)\alpha_a\right] && \text{(B.27)}\\
&= k\left[\alpha_a + (1-o_a)\alpha_b\right] + \alpha_a + \alpha_b - \alpha_a o_b - k\left[\alpha_b + \alpha_a - \alpha_a o_b\right] && \text{(B.28)}\\
&= k\left[\alpha_a + \alpha_b - A_b o_b o_a\right] + \alpha_a + \alpha_b - A_a o_a o_b - k\left[\alpha_b + \alpha_a - \alpha_a o_b\right] && \text{(B.29)}\\
&= k\left[\alpha_a + \alpha_b - A_a o_b o_a\right] + \alpha_a + \alpha_b - A_b o_a o_b - k\left[\alpha_b + \alpha_a - \alpha_a o_b\right] && \text{(B.30)}\\
&= k\left[\alpha_a + \alpha_b - \alpha_a o_b\right] + \alpha_a + \alpha_b - \alpha_b o_a - k\left[\alpha_b + \alpha_a - \alpha_a o_b\right] && \text{(B.31)}\\
&= \alpha_a + \alpha_b(1-o_a) && \text{(B.32)}
\end{aligned}$$

Unsurprisingly, the value $k$ disappears from the formula; the blending order of fragments is irrelevant to the coverage. That is why splitting $a$ and $b$ at the intersection (where their order reverses) and merging the results yields the same coverage as ignoring the intersection and blending the entire fragments using Derivation 3.

To determine the colour of $a \oplus b$, we proceed in the same fashion. As one may expect, the $k$ factor is retained in this formula, since blending order does affect colour.

**For $M_a = M_b$ and $Zmax_a > Zmin_b$,**

$$\begin{aligned}
C_{a\oplus b} &= \frac{k\left[\alpha_a C_a + (1-o_a)\alpha_b C_b\right] + (1-k)\left[\alpha_b C_b + (1-o_b)\alpha_a C_a\right]}{\alpha_{a\oplus b}} && \text{(B.33)}\\
&= \frac{C_a \alpha_a\left[1-(1-k)o_b\right] + C_b\alpha_b(1-ko_a)}{\alpha_{a\oplus b}} && \text{(B.34)}
\end{aligned}$$

[2]Although there can be multiple intersections within a chain of fragments attached to a given pixel, a single intersection (in the sense of Definition IX) involving more than two fragments is undefined. If three or more surfaces do intersect at a single point, our formula will only capture the foremost intersection, treating the rest of the fragments as overlapping but not intersecting the front two. In practice, this rarely causes any discernible colour distortion.

For this formula, unlike with the others we have presented so far, it is not obvious that parts (a), (b), and (c) of Axiom III are satisfied, so we must verify that they are.

$$s = \alpha_a \left[1 - (1-k)o_b\right] \tag{B.35}$$

$$t = \alpha_b(1 - k o_a) \tag{B.36}$$

$$\begin{aligned}
s + t &= \alpha_a \left[1 - (1-k)o_b\right] + \alpha_b(1 - k o_a) && \text{(B.37)}\\
&= \alpha_a + \alpha_b + (1-k)\alpha_a o_b + k\alpha_b o_a && \text{(B.38)}\\
&= \alpha_a + \alpha_b + (1-k)A_a o_a o_b + k\alpha_b o_a && \text{(B.39)}\\
&= \alpha_a + \alpha_b + (1-k)A_b o_a o_b + k\alpha_b o_a && \text{(B.40)}\\
&= \alpha_a + \alpha_b + (1-k)\alpha_b o_a + k\alpha_b o_a && \text{(B.41)}\\
&= \alpha_a + \alpha_b(1 - o_a) && \text{(B.42)}\\
&= \alpha_{a\oplus b} && \text{(B.43)}
\end{aligned}$$

Since $[1 - (1-k)o_b] \geq 0$ and is not a function of $\alpha_a$, $s \propto \alpha_a$. Similarly, since $1 - k o_a \geq 0$ and is independent of $\alpha_b$, $t \propto \alpha_b$.

### B.2.8 Blending Formula 4: $\alpha_{\hat{a}}$ and $C_{\hat{a}}$ when $M_{\tilde{b}} \subseteq M_a$

In this final set of formulas, we ease the requirement that the coverage masks of $a$ and $\tilde{b}$ be identical, and allow the rear mask to be a subset of the front mask. Our motivation is to provide a single blending formula that handles this common case, rather than forcing the `BlendFragment` function to subdivide the fragments, apply Blending Formula 3 to the subfragments, and then merge the results. Again, we consider intersecting and non-intersecting fragments as separate cases. As with Formula 3, we assume that $M_{\tilde{b}} = M_b$ in the case of intersecting fragments. The `BlendFragment` function satisfies this assumption by clipping all the fragments beyond $b$ to $M_b$.

At this point we introduce some new syntax. Given fragments $a$ and $b$, such that $Zmin_a \leq Zmin_b$ and $M_b \subseteq M_a$, let $\hat{a}$ denote fragment $a$ clipped to the mask $M_b$, and let $\bar{a}$ denote the fragment $a$ clipped to the mask $M_a - M_b$. Hence, $M_{\hat{a}} = M_b$ and $M_{\bar{a}} \cap M_b = \emptyset$. Of course, $o_{\hat{a}} = o_{\bar{a}} = o_a$ and $C_{\hat{a}} = C_{\bar{a}} = C_a$.

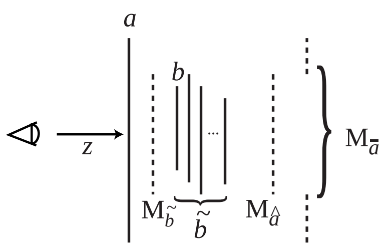


Figure B.6: Fragment arrangement suitable for Blending Formula 4(a).

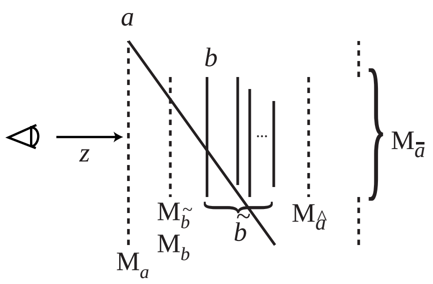


Figure B.7: Fragment arrangement suitable for Blending Formula 4(b).

**Blending Formula 4(a): Non-Intersecting Top Fragments ($M_{\tilde{b}} \subseteq M_a$ and $Zmax_a \leq Zmin_b$)**

Applying Derivation II, and then III, we reason as follows:

**For $M_{\tilde{b}} \subseteq M_a$ and $Zmax_a \leq Zmin_b$,**

$$\alpha_{\tilde{a}} = \alpha_{(\hat{a}\oplus\tilde{b})\oplus\bar{a}} \tag{B.44}$$

$$= \alpha_{\hat{a}} + \alpha_{\tilde{b}}(1-o_{\hat{a}}) + \alpha_{\bar{a}} \tag{B.45}$$

$$= A_{\tilde{b}}o_a + \alpha_{\tilde{b}}(1-o_a) + (A_a - A_{\tilde{b}})o_a \tag{B.46}$$

$$= \alpha_{\tilde{b}}(1-o_a) + A_a o_a \tag{B.47}$$

$$= \alpha_a + \alpha_{\tilde{b}}(1-o_a) \tag{B.48}$$

$$C_{\tilde{a}} = C_{(\hat{a}\oplus\tilde{b})\oplus\bar{a}} \tag{B.49}$$

$$= \frac{C_{\hat{a}\oplus\tilde{b}}\left[\alpha_{\hat{a}} + \alpha_{\tilde{b}}(1-o_{\hat{a}})\right] + C_{\bar{a}\alpha_{\bar{a}}}}{\alpha_{\tilde{a}}} \tag{B.50}$$

$$= \frac{\frac{C_{\hat{a}}\alpha_{\hat{a}} + C_{\tilde{b}}\alpha_{\tilde{b}}(1-o_{\hat{a}})}{\alpha_{\hat{a}} + \alpha_{\tilde{b}}(1-o_{\hat{a}})}\left[\alpha_{\hat{a}} + \alpha_{\tilde{b}}(1-o_{\hat{a}})\right] + C_{\bar{a}}\alpha_{\bar{a}}}{\alpha_{\tilde{a}}} \tag{B.51}$$

$$= \frac{C_{\hat{a}}\alpha_{\hat{a}} + C_{\tilde{b}}\alpha_{\tilde{b}}(1 - o_{\hat{a}}) + C_{\bar{a}}\alpha_{\bar{a}}}{\alpha_{\tilde{a}}} \tag{B.52}$$
$$= \frac{C_a A_{\tilde{b}} o_a + C_{\tilde{b}}\alpha_{\tilde{b}}(1 - o_a) + C_a(A_a - A_{\tilde{b}})o_a}{\alpha_{\tilde{a}}} \tag{B.53}$$
$$= \frac{C_a A_a o_a + C_{\tilde{b}}\alpha_{\tilde{b}}(1 - o_a)}{\alpha_{\tilde{a}}} \tag{B.54}$$
$$= \frac{C_a\alpha_a + C_{\tilde{b}}\alpha_{\tilde{b}}(1 - o_a)}{\alpha_{\tilde{a}}} \tag{B.55}$$

Interestingly, although Formula 4(a) applies to a more general blending scenario ($M_{\tilde{b}} \subseteq M_a$) than Formula 3(a) ($M_{\tilde{b}} = M_a$), their coverage and colour equations turn out to be identical. Equally interesting is the fact that if we consider the case where $M_a \subset M_{\tilde{b}}$, the colour equation becomes different from the other two cases, namely $C_{\tilde{a}} = \frac{C_a\alpha_a + C_{\tilde{b}}(\alpha_{\tilde{b}} - \alpha_{\tilde{a}} o_{\tilde{b}})}{\alpha_{\tilde{a}}}$. We have no occasion to use this equation in `BlendFragment`, so we omit a derivation for it.

### Blending Blending Formula 4(b): Intersecting Top Fragments($M_a \subseteq M_b$ and $Zmax_a > Zmin_b$)

As in Formula 3(a), the coverage and colour of $\tilde{a}$ are obtained by blending $(a \oplus b) \oplus \bar{c}$. We compute $a \oplus b$ with the formulas below, and then blend that result with $\bar{c}$ using Formula 4(a). Again, we assume intersections involve only two fragments, and $M_{\tilde{b}} = M_b$.

**For $M_{\tilde{b}} = M_b \subseteq M_a$ and $Zmax_a > Zmin_b$,**

$$\alpha_{a\oplus b} = \alpha_{(\hat{a}\oplus b)\oplus\bar{a}} \tag{B.56}$$
$$= \alpha_{\hat{a}} + \alpha_b(1 - o_{\hat{a}}) + \alpha_{\bar{a}} \tag{B.57}$$
$$= A_b o_a + \alpha_b(1 - o_a) + (A_a - A_b)o_a \tag{B.58}$$
$$= \alpha_b(1 - o_a) + A_a o_a \tag{B.59}$$
$$= \alpha_a + \alpha_b(1 - o_a) \tag{B.60}$$
$$C_{a\oplus b} = C_{(\hat{a}\oplus b)\oplus\bar{a}} \tag{B.61}$$
$$= \frac{C_{\hat{a}\oplus b}\left[\alpha_{\hat{a}} + \alpha_b(1 - o_{\hat{a}})\right] + C_{\bar{a}}\alpha_{\bar{a}}}{\alpha_{a\oplus b}} \tag{B.62}$$
$$= \frac{\frac{C_{\hat{a}}\alpha_{\hat{a}}[1-(1-k)o_b]+C_b\alpha_b(1-ko_{\hat{a}})}{\alpha_{\hat{a}}+\alpha_b(1-o_{\hat{a}})}\left[\alpha_{\hat{a}} + \alpha_b(1 - o_{\hat{a}})\right] + C_{\bar{a}}\alpha_{\bar{a}}}{\alpha_{a\oplus b}} \tag{B.63}$$

$$= \frac{C_a A_b o_a \left[1-(1-k)o_b\right] + C_b \alpha_b (1-ko_a) + C_a o_a (A_a - A_b)}{\alpha_{a\oplus b}} \quad (B.64)$$

$$= \frac{-C_a \alpha_b o_a (1-k) + C_a \alpha_a + C_b \alpha_b (1-ko_a)}{\alpha_{a\oplus b}} \quad (B.65)$$

$$= \frac{C_a \left[\alpha_a - (1-k)\alpha_b o_a\right] + C_b \alpha_b (1-ko_a)}{\alpha_{a\oplus b}} \quad (B.66)$$

Finally, we verify Axiom III(a) for the color formula. Note that parts (b) and (c) do not apply, since $M_a$ and $M_b$ are neither disjoint nor identical.[3]

$$s = \alpha_a - (1-k)\alpha_b o_a \quad (B.67)$$

$$t = \alpha_b (1-ko_a) \quad (B.68)$$

$$(B.69)$$

$$s+t = \alpha_a - (1-k)\alpha_b o_a + \alpha_b(1-ko_a) \quad (B.70)$$

$$= \alpha_a + k\alpha_b o_a (1-1) + \alpha_b (1-o_a) \quad (B.71)$$

$$= \alpha_a + \alpha_b (1-o_a) \quad (B.72)$$

[3]If we were to apply Axiom III(b) and III(c) nevertheless, we would see that the former does not hold. This is because part of fragment $b$ that lies in front of $a$ (due to the intersection) causes $a$'s colour contribution over that region to be constant with respect to $A_a$—since $M_b \subseteq M_a$, the contribution is just a function of $k$, $A_b$, and $o_a$. Hence, $a$'s *total* colour contribution $s$, as a function of $A_a$, is linear but contains a constant arising from the above region: $s = c_1 A_a + c_2$. This explains why the expression for $s$ in line B.67 has two terms that differ by a factor of $A_a$ and why we therefore cannot say that $s \propto \alpha_a$.

# Bibliography

[AES94] S. S. Abi-Ezzi and S. Subramaniam. Fast dynamic tessellation of trimmed NURBS surfaces. In *Computer Graphics Forum*, volume 13, pages 107–126. Eurographics, Basil Blackwell Ltd, 1994.

[Bea91] Robert C. Beach. *An Introduction to the Curves and Surfaces of Computer-Aided Design.* Van Nostrand Reinhold, New York, 1991.

[Bli89] James F. Blinn. Jim Blinn's corner: Optimal tubes. *IEEE Computer Graphics and Applications*, September 1989.

[Blo85] Jules Bloomenthal. Modeling the mighty maple. In *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19, pages 305–311. ACM SIGGRAPH, Addison Wesley, July 1985.

[BW86] Gary Bishop and David M. Weimer. Fast Phong shading. In *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 103–106. ACM SIGGRAPH, Addison Wesley, August 1986.

[Car84] Loren Carpenter. The A-buffer, an antialiased hidden surface method. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 103–108. ACM SIGGRAPH, Addison Wesley, July 1984.

[CPD$^+$96] Andrew Certain, Jovan Popović, Tony DeRose, Tom Duchamp, David H. Salesin, and Werner Stuetzle. Interactive multiresolution surface viewing. In *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 91–98. ACM SIGGRAPH, Addison Wesley, August 1996.

[CVM$^+$96] Jonathan Cohen, Amitabh Varshney, Dinesh Manocha, Greg Turk, Hans Weber, Pankaj Agarwal, Frederick P. Brooks, Jr., and William Wright. Simplification envelopes. In *SIGGRAPH 96 Conference Proceedings*, pages 119–128. ACM SIGGRAPH, Addison Wesley, August 1996.

[Far88] Gerald Farin. *Curves and Surfaces for Computer Aided Geometric Design*. Academic Press, Toronto, 1988.

[FFR83] E. Fiume, A. Fournier, and L. Rudolph. A parallel scan conversion algorithm with anti-aliasing for a general purpose ultracomputer. In *Computer Graphics (SIGGRAPH '83 Proceedings)*, volume 17, pages 141–150. ACM SIGGRAPH, Addison Wesley, July 1983.

[Fou92] Alain Fournier. Normal distribution functions and multiple surfaces. In *Graphics Interface '92 Workshop on Local Illumination*, pages 45–52, May 1992.

[FV83] James D. Foley and Andries Van Dam. *Fundamentals of Interactive Computer Graphics*. Addison–Wesley, Don Mills, Ontario, 1983.

[HB94] Donald Hearn and M. Pauline Baker. *Computer Graphics*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1994.

[HG94] Paul Heckbert and Michael Garland. Multiresolution modeling for fast rendering. In *Proceedings of Graphics Interface '94*, pages 43–50, Banff, Alberta, Canada, May 1994. Canadian Information Processing Society.

[HG97] Paul Heckbert and Michael Garland. Survey of polygonal surface simplification algorithms. Technical report, Carnegie Mellon University, 1997.

[Hop96] Hugues Hoppe. Progressive meshes. In *SIGGRAPH 96 Conference Proceedings*, pages 99–108. ACM SIGGRAPH, Addison Wesley, August 1996.

[Hop97] Hugues Hoppe. View-dependent refinement of progressive meshes. In *SIGGRAPH 97 Conference Proceedings*, pages 189–198. ACM SIGGRAPH, Addison Wesley, August 1997.

[KB89] A. A. M. Kuijk and E. H. Blake. Faster phong shading via angular interpolation. *Computer Graphics Forum*, 8(4):315–324, December 1989.

[KK89] James T. Kajiya and Timothy L. Kay. Rendering fur with three dimensional textures. In *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23, pages 271–280. ACM SIGGRAPH, Addison Wesley, July 1989.

[LE97] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. In *SIGGRAPH 97 Conference Proceedings*, pages 199–208. ACM SIGGRAPH, Addison Wesley, August 1997.

[LTT91] Andre M. LeBlanc, Russell Turner, and Daniel Thalmann. Rendering hair using pixel blending and shadow buffers. *The Journal of Visualization and Computer Animation*, 2:92–97, 1991.

[Max90] Nelson L. Max. Cone-spheres. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 59–62. ACM SIGGRAPH, Addison Wesley, August 1990.

[Mil88] Gavin S. P. Miller. From wire-frames to furry animals. In *Proceedings of Graphics Interface '88*, pages 138–145, June 1988.

[NDW93] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide.* Addison–Wesley Publishing Company, Don Mills, Ontario, 1993.

[Ney95a] Fabrice Neyret. Animated texels. In *Computer Animation and Simulation '95*, pages 97–103. Eurographics, Springer-Verlag, September 1995.

[Ney95b] Fabrice Neyret. A general multiscale model for volumetric textures. In *Proceedings of Graphics Interface '95*, pages 83–91, 1995.

[RB85] William T. Reeves and Ricki Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19(3), pages 313–322. ACM SIGGRAPH, Addison Wesley, July 1985.

[RCI91] Robert E. Rosenblum, Wayne E. Carlson, and Edwin Tripp III. Simulating the structure and dynamics of human hair: Modelling, rendering and animation. *The Journal of Visualization and Computer Animation*, 2:141–148, 1991.

[Ree83] W. T. Reeves. Particle systems – a technique for modeling a class of fuzzy objects. *ACM Trans. Graphics*, 2:91–108, April 1983.

[RSC87] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. In *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 283–291. ACM SIGGRAPH, Addison Wesley, July 1987.

[SC88] Michael Shantz and Sheue-Ling Chang. Rendering trimmed NURBS with adaptive forward differencing. In *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 189–198. ACM SIGGRAPH, Addison Wesley, August 1988.

[Sil90] M. J. Silbermann. High-speed implementation of nonuniform rational B-splines. In *Curves and Surfaces in Computer Vision and Graphics*, pages 338–345. The International Society for Optical Engineering, The International Society for Optical Engineering, August 1990.

[SLS$^+$96] Jonathan Shade, Dani Lischinski, David Salesin, Tony DeRose, and John Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. In *SIGGRAPH 96 Conference Proceedings*, pages 75–82. ACM SIGGRAPH, Addison Wesley, August 1996.

[SS93] Andreas Schilling and Wolfgang Straßer. EXACT: Algorithm and hardware architecture for an improved A-buffer. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 85–92. ACM SIGGRAPH, Addison Wesley, August 1993.

[Whi83] T. Whitted. Anti-aliased line drawing using brush extrusion. *Computer Graphics (SIGGRAPH '83 Proceedings)*, 17:151–156, July 1983.

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, volume 12, pages 270–274. ACM SIGGRAPH, Addison Wesley, August 1978.

[WP95] Jason Weber and Joseph Penn. Creation and rendering of realistic trees. In *SIGGRAPH 95 Conference Proceedings*, pages 119–128. ACM SIGGRAPH, Addison Wesley, August 1995.

[WS92] Yasuhiko Watanabe and Yasuhito Suenaga. A trigonal prism-based method for hair image generation. *IEEE Computer Graphics and Applications*, 12(1):47–53, January 1992.