Aspects of computational geometry in gamut mapping implementations

Dennis Küpper, Klaus Simon

Swiss Federal Laboratories for Materials Testing and Research (EMPA), Dübendorf,
Switzerland

ABSTRACT

Even though gamut mapping (GM) is a three-dimensional polyhedron problem, algorithmically it is usually not treated as such. This is due to tight runtime constraints in practice, which suggest simple algorithms. We will show that basic geometric operations, like the intersection of a ray with the gamut surface, can be implemented very efficiently for typical GM applications that work with device gamuts. This opens up new possibilities to design algorithms without reducing their practicability.

Keywords: Gamut boundary descriptor, gamut mapping, triangle mesh, segment maxima method, ray intersection, point location

1. INTRODUCTION

Image reproduction nowadays is dominated by desktop publishing. The WYSIWYG principle necessitates a device independent color specification as a universal data standard. Imaging devices are limited by technical constraints, however. In this paper, we are focusing on their limited capability to reproduce arbitrary colors. We call the set G of reproducible colors the *gamut* of the device. In general, an image to be reproduced using a certain device will have some colors $p \notin G$. Such out-of-gamut (OOG) colors need to be replaced with colors $p' \in G$ in a process called *gamut mapping* (GM).

The currently dominant *color management system* (CMS) was proposed in 1993 by the *International Color Consortium* (ICC). It uses lookup tables and device profiles to store empirical device characterizations. Because it was devised as a color space transformation, the ICC-CMS is optimized for color correctness and encourages the implementation of GM algorithms as static point-to-point operations.

Apart from the ICC standard, gamut mapping has evolved into an active research area.¹ The gamut mapping algorithms presented in recent years showed significant advances in visual performance. Since many of them are based on geometrically and algorithmically complex operations, they are often too slow to be used in practice. Only a few seconds per image are acceptable in professional imaging systems. It is fair to say that a practicable gamut mapping solution requires both good visual performance and run-time efficiency.

GM is typically performed in a perceptually uniform color space like CIELAB² or IPT,³ which have one axis that is correlated with lightness and two axes that correspond to opponent colors and can be considered roughly orthogonal to lightness perception. In such color spaces, a print gamut is remotely bipyramid-shaped. The black and white point would be located at the lower and upper apex, respectively, and the primary and secondary colors would be located on the rim where the two pyramid-bases touch. Popular GM algorithms map OOG colors along rays into the target gamut, towards a so-called *focal point*. It can either be chosen to be fixed on the gray axis, but may also vary for every OOG color. For example, the position might depend on the lightness of the pixel to be mapped.

A fixed focal point on the gray axis may not be optimal in terms of visual performance, but it has algorithmic advantages if it is chosen as the center of a spherical coordinate system. The radial component of a surface color

Further author information: (Send correspondence to Dennis Küpper)

Dennis Küpper: dennis.kuepper@empa.ch

Klaus Simon: klaus.simon@empa.ch

then conveniently captures the distance to the gamut boundary, while the azimuth angle of any color coincides with its hue angle. The distance to the gray axis is a correlate of chroma.

The gamut surface can be described by partitioning space into angular segments, centered at the fixed focal point, and storing the color sample with the largest radial component within the segment. This well-known *gamut boundary descriptor* (GBD) is called the *segment maxima method*, which we will describe later in more detail. It is easy to compute and can be efficiently queried.

As an alternative GBD to the segment maxima method, we will investigate triangles meshes. This representation is often used in computer graphics. In order to be interesting for GM applications, it must support the efficient intersection with a ray. Related to this search process is the so-called *point location problem*⁴, which has a worst case lower run-time bound of $O(\log(n))$, where n is the number of edges.

For common, reasonably-sampled gamut surfaces of imaging devices, we will show how to use the segment maxima structure to implement the above-mentioned operations such that they terminate with the correct solution after the first few steps. Our solution is inspired by the interpolation search strategy.

2. DATA STRUCTURES

2.1 The segment maxima data structure

Even though we describe the surface itself as a triangle mesh, we rely on ideas taken from the segment maxima method,⁵⁻⁷ which we will introduce first, using the CIELAB color space as an example. The method describes the gamut in terms of a center point inside the gamut, typically lying on the L^* axis. Conceptually, this defines the origin of a spherical coordinate system. Space is then discretized into *segments* with respect to zenith angle, measured from the positive L^* axis, and azimuth angle, measured inside the a^*-b^* plane. The azimuth angle corresponds to the hue angle. From all the points that fall into a segment, the one that is farthest away from the origin is assumed to be a representative for the gamut boundary.

The segments are chosen in a very regular way. The azimuth angles are equally spaced in $[0, 2\pi]$ and the zenith angles are equally spaced in $[0, \pi]$. This makes it possible to store the representatives for the segments in a $m \times n$ matrix in a natural way by mapping the zenith angles to the m dimension and the azimuth angles to the n dimension, such that each column stores all samples along a certain hue plane.

The standard method⁶ to compute *all* intersections of the gamut by a ray lying in a single hue plane works as follows: interpolate the surface points of the desired hue plane from the samples of the left and right discretized hue planes. Connect the samples by line segments and test every one of them for an intersection. The inside/outside test is also simple: determine the segment of the point to be tested and construct the local surface using the neighboring segments, then see whether the point is above or below this interpolated surface.

This method does not make many assumptions about the shape of the gamut. Especially, it is neither tied to a specific characterization model such as Kubelka-Munk equations, nor does it assume that the gamut is convex. This makes it well-applicable for the computation of an *image gamut*, i.e. the range of colors contained in an image. It does, however, assume that the gamut is *star-convex*^{*} with respect to the central point, which is not satisfied by image gamuts in general. Even though we store the actual surface as a triangle mesh, we will also make this assumption to be able to efficiently determine whether a point is inside or outside of the gamut.

In our implementation, we do not store a sample for each segment, but rather a reference to the surface triangle that covers it. If the segment is covered by a combination of several surface triangles, we just pick one that covers the center of the segment.

2.2 Triangle mesh as gamut boundary descriptor

The use of a triangle mesh as GDB has the advantage that it is a very general way of representing the surface of a three-dimensional object, for which a wealth of algorithms have already been developed. The input is given in the form of an array of point-samples, together with a list of triangles in the form of three indices into the point array. The point array stores the *geometry* of the surface, i.e. where it is located in the color-space. The

^{*}A set S is star-convex w.r.t. a point $x_0 \in S$ if for all $x \in S$: $\{tx_0 + (1-t)x \mid t \in [0, 1]\} \subseteq S$

triangle array stores the *topology* of the surface, i.e. how the different parts of the surface are connected to each other. This can be thought of as a *graph* whose vertices are assigned positions in the color space.

We compute the complete surface by interpolating linearly from the point-samples, such that the surface is made up of flat pieces. Apart from that, however, the surface trivially captures all the samples given as input. This is important if one wants to capture the primary and secondary colors exactly, and is not guaranteed if one uses a GBD that resamples the surface along equidistant hue planes, since a primary will in general not lie exactly on any of the sampled hue planes. The quality of the approximation can simply be increased by locally increasing the number of provided sample-points and triangles. In this paper, we are not concerned with the problem of determining the sample points and the corresponding triangles, which is a research area of its own.

Storing the surface in such a general way was required for our experiments with free intersections, i.e. ones where the ray does not have its origin on the gray axis, or ones where the intersection is not restricted to a certain hue plane.

2.3 Basic operators

A triangle mesh⁸ comprises vertices, edges, and triangles. A triangle is a closed sequence of three distinct edges, with an edge connecting two distinct vertices. This is a very general way to describe a surface, and we can make further assumptions for the special case of a gamut surface. It must be a *2-manifold*, i.e. every point on it must have an arbitrary small neighborhood of points that can be considered topologically equivalent to a planar disk. This means that every edge must be shared by *exactly* two faces. Additionally, the surface must be *simple*, meaning it can be deformed into a sphere. This excludes surfaces such as tori, which have holes. We also assume that faces do not interpenetrate.

There are many data structures to describe such a surface, each with benefits and drawbacks with respect to factors such as storage size, cost of different types of traversal, cost of altering the surface, and ease of implementation. In our implementation, we have a *static* gamut surface where we want to be able to efficiently determine for each triangle (a) its three vertices, and (b) its neighboring triangles across the three edges. We chose to implement these operations using a *Corner Table*,⁹ which can handle these requirements efficiently and is also easily implemented.

We will intersect the gamut surface with *rays* which have a starting point $\mathbf{o} \in \mathbb{R}^3$ that lies *inside* the gamut, and a normalized direction vector $\mathbf{d} \in \mathbb{R}^3$. The rays only expand in one direction and can be written as $r(t) = \mathbf{o} + t\mathbf{d}, t \in \mathbb{R}_{\geq 0}$. We assume that intersecting the surface with such a ray results in a *unique* point. Should the intersection be non-unique, we find *one* of the intersections, but not necessarily the one closest to $\mathbf{o}$. In practice, we have found that this did not restrict us when designing GM algorithms. If they led to non-unique intersections, they would produce artifacts whether or not the intersections were disambiguated according to some rule, so we avoided them altogether.

3. ALGORITHMS

Two of the most prevalent operations in gamut mapping algorithms are (a) the determination of whether a given point is in- or outside the gamut, and (b) the intersection of a ray with the gamut surface. The simple clipping algorithm, for example, exclusively uses these two operations: for all colors p that are outside the gamut, the algorithm determines the intersection q of the gamut surface with the line defined by p and a focal point c inside the gamut. The color q is then the result of gamut mapping color p .

We describe how we efficiently implement the two operations mentioned above in terms of our mesh data structure.

3.1 Triangle intersections

Since our gamut surface is described by a series of triangles, the problem of intersecting it with a ray can be reduced to finding the one triangle that is intersected. If run-time efficiency was not a concern, a simple algorithm to determine the intersection would consist of simply enumerating all the triangles and testing them one by one.

The rest of this paper will be concerned with finding a strategy that tests as few triangles as possible before it finds the desired one.

The problem of finding the intersection of a ray and a surface is related to the so-called *point location problem*. The general version can be stated as follows: for a partitioned space, find the partition that contains a given query point p . For the intersection problem, the query point to be found is not explicitly given but rather defined implicitly by the ray and the surface. Nevertheless, we can employ search strategies developed for the point location problem.

There are several known solutions to the point location problem in the planar case, i.e. the plane is segmented into polygonal faces and the task is to locate the one that contains the query point p . This can be optimally solved in $O(\log(n))$ time and with $O(n)$ storage space, where n is the number of edges. These are typically based on precomputed search structures that allow to quickly reduce the number of faces to be tested in each step, but cannot be easily generalized to our case, where the actual query point is unknown.

However, there are strategies for the planar point location problem that do not rely on any precomputations and have average runtimes in $O(\sqrt{n})$. They *walk*¹⁰¹¹ towards the query point by comparing the coordinates of the current face with the ones of the query point and then going to the neighboring face that seems closest to the goal. We will use this for our surface intersections. When the ray misses the current triangle, we can compute the approximate direction in which we should walk to find the actual intersection.

3.2 Graph traversal

Finding the intersection of the ray can be thought of as a graph search problem. The vertices of the graph are the triangles, which are connected to their neighboring triangles via edges. This is called the *dual* of the graph that describes the surface topology. The goal is to find the vertex that has the property of being intersected by the ray. The naive search can be expressed as a recursive *depth-first traversal*:

Algorithm 1: DFT

```

Input: A starting triangle  $t$ 
Output: The intersected triangle if found, else NIL
1 begin
2   if HasBeenVisited( $t$ ) then
3     return NIL
4   else if IsIntersected( $t$ ) then
5     return  $t$ 
6   else
7     for  $w \in \text{Neighbors}(t)$  do
8        $q \leftarrow \text{DFT}(w)$ 
9       if  $q \neq \text{NIL}$  then
10        return  $q$ 
11      end
12    end
13    return NIL
14  end
15 end

```

Note that it is not obvious that line 2 can be executed in $O(1)$. If we stored this information in a simple array and reset it for every triangle intersection, the run-time would immediately become linear in the number of triangles. One solution to this problem is described by Mehlhorn.¹²

Algorithm 1 is a so-called *uninformed* search, since it does not use any knowledge of the problem to decide which triangle is examined in line 7. It is expected to search half the mesh before it finds the solution, depending on the starting triangle. In the following, we will describe (a) how to make the search *informed* and (b) how to find a good starting triangle for the search.

3.3 Informed search

Three distinct points $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ in 3-space not only define a triangle, but also the plane in which the triangle is located. A simple ray-triangle intersection algorithm is to compute the intersection $\mathbf{p}$ of the ray with aforementioned plane, and then to test whether the intersection is located inside the triangle. This last step can be performed by computing the *barycentric coordinates* of $\mathbf{p}$ with respect to the points $(\mathbf{a}, \mathbf{b}, \mathbf{c})$.

Any point p in the plane of said triangle can be written as $p = w_a a + w_b b + w_c c$, such that $w_a + w_b + w_c = 1$. The barycentric coordinates of p are then (w_a, w_b, w_c) . For a point inside the triangle, all components are in $[0, 1]$. A point outside has at least one component that is negative. Assume w_a was zero. Then p lies somewhere on the line spanned by b and c . Making w_a either positive or negative moves p either to the left or right of that line. This means that by looking at the sign of component, we can see on which side of an edge a point lies.

If the current triangle is missed by the ray, we can inspect the barycentric coordinates to guess where the actual intersection is located. If only one coordinate is negative, we can step over the corresponding edge. If two coordinates are negative, we have an ambiguity and simply step over one of the corresponding edges. All three coordinates cannot be negative, since they have to sum to 1.

With this technique, we can change line 7 in algorithm 1 to make it significantly faster. Specific numbers for typical gamuts are given in the last section.

3.4 Guessing a good starting triangle for *hue preserving* intersections

Figure 1. The picture shows a 2D cut through a gamut surface and its approximation. The point O is the center of a segment maxima data structure. The points K and W are the black and white point, respectively. The goal is to compute the intersection of the real surface and the ray that is not tied to O . The point O' is the intersection of the gamut approximation. The point O'' is the point at which we will start the informed search procedure.

Figure (1) illustrates the general idea that we will use to guess a starting point to search for a hue preserving intersection. We compute the intersection of the ray with a gamut approximation. Using the segment maxima

data structure, which stores a triangle reference for every segment, we can guess a starting triangle to walk towards the real intersection. The more similar the real surface is to the approximation, the closer the starting point will be to the real intersection.

More precisely, figure (1) shows a cut of a gamut along two complementary *hue-planes*. What we call a *hue-plane* is actually a half-plane which is assigned an azimuth angle in $[0, 2\pi]$. It extends from the *gray-axis*, where the colors are unsaturated, outwards toward colors with increasing saturation. All colors in a hue-plane have the same hue. The hue-plane with angle α is complementary to the hue-plane with angle $(\alpha + \pi) \bmod 2\pi$, such that two complementary hue-planes together create a single plane that divides space into two halves.

Our triangle mesh is an object in three dimensions. The goal is to intersect the two-dimensional hue plane (living in Euclidean three-space) with a ray $r(t) = \mathbf{o} + t \cdot \mathbf{d}$, where $\mathbf{o} \in \mathbb{R}^3$, $\mathbf{d} \in \mathbb{R}^3$, $t \in \mathbb{R}_{\geq 0}$, living in the same hue plane. Since we restrict $r(t)$ to lie in a single hue-plane, the intersection problem is essentially two-dimensional. However, since we allow the ray to lie in an *arbitrary* hue-plane, the problem has to be solved with a three dimensional data-structure. Our goal will be to efficiently solve the 2D problem in an *approximate* way, such that it will allow us to quickly find the *exact* solution in 3D. In order to do this, we first intersect a number h of hue-planes with the gamut-surface, the angles equally spaced in $[0, 2\pi]$. This is done as a precomputation. Figure (1) shows two such intersections. The left and the right (complementary) hue planes join at the vertical line in the middle of the picture.

Figure (1) indicates that the gamut surface along a hue plane is usually curved, which we approximate with four line segments between the black-point $\mathbf{K}$, the white-point $\mathbf{W}$, and two cusps $\mathbf{C}_{\text{Left}}$ and $\mathbf{C}_{\text{Right}}$, forming a quadrangle. The cusp is the outermost color in a hue-plane, i.e. the most saturated color with a given hue. We avoid intersecting the real gamut shown in figure (1) in 2D and postpone this step to 3D. We instead compute the intersection with the approximation, which requires some care.

We are assuming that (a) $\mathbf{o}$ and $\mathbf{d}$ are lying in the same hue plane, (b) that $\mathbf{o}$ is inside the gamut, (c) that the intersection is unique, and finally (d) that the surface meets a certain regularity condition, which is needed for a special case that we will describe below. We now fetch the two precomputed cusps from the closest complementary hue planes that have been sampled and construct the approximation using the black and white point. The computations will all take place in a 2D setting.

Our simplified gamut description, by design, correctly captures the two cusps and the black and white point. It will typically either over- or underestimate the rest of the shape. We shall call the under- and overestimated parts bumps and dents, respectively. Figure (2) illustrates the geometric meaning. A *bump* is a segment B_i of the true gamut surface that is underestimated by the approximation with a straight line-segment b_i . B_i and b_i coincide at the endpoints of the bump, where the approximation b_i captures the gamut exactly. Together, B_i and b_i form a closed loop. *Dents* D_i and their approximations d_i are defined analogously.

The ray will intersect the real surface in either a dent D_i or a bump B_i . The following method ensures that the starting point is located in the same dent or bump as the real intersection, even though it only has knowledge of the gamut *approximation*. We have to make an assumption on well-formedness of the gamut for this to be generally true, see the proof of *Case 2* below. We will refer to the gamut approximation as GA, and the segment maxima structure as SMS.

We distinguish between two possibilities: either $\mathbf{o}$ lies inside or outside the GA. For the sake of simplicity, we assume that the points and lines are *in general position*, which means that we do not treat degenerate cases such as a point lying exactly on a line.

Case 1 If $\mathbf{o}$ lies inside, then it is either in a region of type 1 or 2 in figure (2). We can exclude region 2, since it is outside the real gamut. So, if $\mathbf{o}$ is inside, do the following: intersect the GA with the ray, resulting in a unique point Q (because the GA is convex), then look up the associated triangle using the SMS. It will lie in the correct bump or dent.

Figure 2. This shows a gamut surface that has been exaggerated for clarity. The different regions in which the origin of the ray can lie are numbered from 1 to 4. The curved bumps and dents are drawn with solid lines, while the straight approximations are drawn with stippled lines.

Proof Assume the ray hits a bump. Segments B_i and b_i form a closed loop. The ray cannot have entered through B_i , since that would mean that it had left the gamut somewhere else, implying a non-unique intersection. So the ray has entered through b_i at the point Q . If we shoot a ray from the center c through Q , it will also enter at b_i and exit at a unique point on B_i , since we assume the gamut to be star-convex w.r.t. c . So we find a starting triangle on B_i . Analogously, if the ray hits a dent, then it must have entered through D_i and exited through d_i , at the point Q . A ray from c through Q will also enter through D_i , which gives us the starting triangle. $\square$

Case 2 If $\mathbf{o}$ lies *outside*, then it is either in a region of type 3 or 4 in figure (2). We can exclude region 4, since it is outside the real gamut. So, if $\mathbf{o}$ is outside the GA, do the following: if the ray does *not* intersect the GA, then determine the closest point on the GA to $\mathbf{o}$, resulting in a unique point Q , then look up the associated triangle using the SMS. It will lie in the correct bump. If the ray *does* intersect the GA, then it will enter it at a point P and leave it at a point Q . Using the SMS, look up the associated triangle of Q . It will lie in the correct bump or dent.

Proof Assume the ray intersects the GA. It enters it at some point P and leaves it at point Q through either a bump or a dent. We can apply the same reasoning as in case 1 to point Q ; point P is only an artifact of the under-estimated gamut. Assume the ray does not intersect the GA. This means it exits the bump through B_i . If the point Q , which is the closest point to $\mathbf{o}$ on the GA, is located on b_i , then shooting a ray from c through Q will find a starting point on B_i . This does not work if B_i leaned over to some neighboring b_j , resulting in a starting point in B_j .

We have not seen such a gamut yet, but we have to assume that some combination of device gamut and working color space will exhibit such behavior. $\square$

Even though we do not describe it here, this method could be generalized to compute true three-dimensional intersections with the gamut. Instead of the cusps, one could use the positions of the primary and secondary colors to arrive at a reasonable approximation of the gamut's shape.

3.5 Inside/outside test

Since we assume that the gamut is star-shaped with respect to the center, testing whether a point p is inside or outside the gamut can be done using the segment maxima structure. We shoot a ray from the center c through p to compute the intersection with the surface and determine whether p is below or above the respective triangle. This does not involve the guessing process we described before, since we already store triangle references for rays with origin c in the segment maxima structure. The search would then proceed using the geometric information, but typically the correct triangle is found at the first try.

3.6 Approximate hue preserving minimum ΔE_{ab}^* clipping

One goal of gamut mapping can be that one wants stay as close to the original colors as possible, without generating too many artifacts. The color difference can be approximated using the ΔE_{ab}^* distance formula. A simple algorithm with such a goal in mind is *hue preserving minimum ΔE_{ab}^* clipping*. To map an OOG color p , it determines the closest color q on the gamut boundary, with the restriction that it has to lie in the same hue plane as p .

This algorithm can be costly, since a large part of the surface has to be considered to be able to determine the closest color. We implemented this algorithm such that it finds an approximate solution quickly.

To find a close color on the surface to a query point p , our algorithm proceeds as follows: first, guess a starting point for an informed search procedure. This is done by determining the closest point on the GA and looking up the associated triangle in the SMS. Second, should that triangle happen to lie outside the hue plane of p , walk parallel to the a^*-b^* plane on the gamut surface towards the correct hue plane. Third, once a triangle in the correct hue plane has been found, walk up or down on the surface until a local minimum for the distance has been found.

This approximate algorithm typically terminates after a few steps. We are still experimenting with its visual and run-time performance, so we cannot yet present a detailed analysis.

4. PERFORMANCE MEASUREMENTS

We investigated the performance of our heuristic with several print gamuts. The computations were performed using CIELAB as a working color space. To test a specific gamut, we randomly generated one million point pairs $(\mathbf{a}, \mathbf{b})$. We required that point $\mathbf{a}$ was inside the gamut and point $\mathbf{b}$ was outside (but still inside the sRGB gamut). We also required that the two points define a unique intersection with the gamut.

We arrived at one million pairs by randomly generating pairs within the bounding boxes of the two gamuts and discarding the ones that did not meet the requirements. Using the segment maxima structure, space was discretized into 200 bins around the equator, each containing 100 bins from pole to pole. Each contained a reference to a face, which amounts to an average of about 15 to 40 samples per surface triangle, depending on the gamut.

Table (1) shows the results for different gamuts. We counted for each of the one million point pairs how many triangles needed to be intersected before the correct one was found, i.e. how many times was the correct triangle found at the first try, how many times at the second try, and so on. The results are given as percentages. The last column lists how many triangle intersections were performed on average.

To get a better picture of the algorithm, we list the relative costs of the two parts of the algorithm in table (2). We implemented our algorithm in C/C++ and the measurements were performed with fully serial code, i.e. we did not employ any parallelization techniques. As one would expect, the timings do not differ significantly for the various gamuts surfaces.

(Number of triangles)	1	2	3	4	5	6	7	8	9	≥ 10	μ
ISOnewspaper26v4 (768)	72.24	12.48	2.33	1.63	1.83	6.43	1.74	0.40	0.21	0.70	2.02
ISO Coated (768)	69.30	13.51	2.53	1.78	2.26	6.62	1.60	1.16	0.17	1.07	2.2
FOGRA Coated (1240)	68.48	11.33	3.67	2.39	2.19	2.74	2.64	1.94	1.33	3.29	2.89
Euroscale Coated (1240)	62.09	12.13	4.35	3.20	3.22	3.32	3.02	2.17	1.63	4.88	3.62
Japan Newspaper (550)	74.02	8.98	2.96	1.86	1.92	2.09	1.81	1.59	0.82	3.95	2.32
Japan Uncoated (702)	73.40	9.42	3.00	2.29	2.19	2.47	1.63	1.35	0.92	3.32	2.45
Japan Coated (1158)	67.08	10.70	3.79	2.54	2.44	2.61	3.18	1.74	1.11	4.81	3.67
Xerox Phaser 7750 (984)	82.38	7.83	1.16	0.84	1.29	1.67	1.30	1.15	0.62	1.77	2.11
US Web Coated SWOP (1120)	67.08	11.13	3.75	2.71	2.58	2.51	2.39	1.89	1.78	4.19	3.08

Table 1. Relative amount of inspected triangles in % for different gamuts. The last column shows the average amount of inspected triangles.

(Number of triangles)	Guess [μ s]	Intersection [μ s]
ISOnewspaper26v4 (768)	0.102	0.118
ISO Coated (768)	0.102	0.117
FOGRA Coated (1240)	0.102	0.120
Euroscale Coated (1240)	0.103	0.12
Japan Newspaper (550)	0.102	0.12
Japan Uncoated (702)	0.103	0.119
Japan Coated (1158)	0.102	0.119
Xerox Phaser 7750 Standard (984)	0.102	0.12
US Web Coated SWOP (1120)	0.102	0.119

Table 2. Absolute timings for the two main parts of the intersection algorithm.

Guessing Strategy To measure how long the guessing step takes, we applied it to the one million point pairs described before. The whole process is timed and the result divided by one million.

Triangle Intersection To measure the cost of a single triangle intersection, including the decision process for the next triangle to be tested, we again used the one million point pairs. To avoid noise in the measurements, we did not stop when the intersection was found, but rather iterated over the mesh until all faces had been tested. The result was then divided by the number of faces.

5. CONCLUSION

The point location problem is a part of our problem. In the worst case, it is equivalent to a search problem with a lower run-time bound in $O(\log n)$. This assumes that the distribution of the elements is not known. Better bounds can be established when one knows more about the data. In our case, we can make assumptions based on the shape of a typical print gamut, which allows us to guess a good starting point for the search. Even though we only perform one step and then resort to some sort of geometric linear search, we could also iterate the guessing procedure with a hierarchy of more and more refined surface descriptions. This would lead to a type of *interpolation search* for non-uniformly distributed data, which is also related to *hash tables*. Such algorithms can achieve runtimes of $O(\log \log n)$, but our empirical studies with only one step cannot make any predictions with respect to such runtimes.

We showed that easy to re-implement techniques borrowed from computational geometry allow one to describe a gamut boundary with a triangle mesh without sacrificing performance. Our GBD does not rely on resampling the provided surface, but rather operates on the original samples themselves. As a benefit, triangle meshes have been well-studied in the literature, leading to many readily available algorithms that can be used for prototyping and designing new gamut mapping algorithms.

REFERENCES

- [1] Morovič, J., [*Color Gamut Mapping*], Wiley, West Sussex, England (2008). Chapter 10: Survey of Gamut Mapping Algorithms.
- [2] Reinhard, E., Khan, E. A., Akyüz, A. O., and Johnson, G. M., [*Color Imaging: Fundamentals and Applications*], A. K. Peters, Ltd., Natick, MA, USA (2008). Chapter 8: Color Spaces.
- [3] Ebner, F. and Fairchild, M., “Development and testing of a color space (IPT) with improved hue uniformity,” in [*Proc. 6th IS&T/SID Color Imaging Conference*], 8–13 (1998).
- [4] de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O., [*Computational Geometry*], Springer (February 2000). Chapter 6: Point Location.
- [5] Morovič, J. and Luo, M. R., “Gamut mapping algorithms based on psychophysical experiment,” in [*Proceedings of the 5th IS&T/SID Color Imaging Conference*], 44–49 (1997).
- [6] Morovič, J. and Luo, M. R., “Calculating medium and image gamut boundaries for gamut mapping,” *Color Research & Application* **25**, 394–401 (2000).
- [7] Morovič, J., [*Color Gamut Mapping*], Wiley, West Sussex, England (2008). Chapter 7: Basic Computational Geometry for Gamut Mapping.
- [8] Foley, J. D., van Dam, A., Feiner, S. K., and Hughes, J. F., [*Computer Graphics: Principles and Practice in C*], Addison-Wesley Professional, second ed. (August 1995). Chapter 11: Representing Curves and Surfaces.
- [9] Johnson, C. and Hansen, C., [*Visualization Handbook*], Academic Press, Inc., Orlando, FL, USA (2004). 3D Mesh Compression, Jarek Rossignac.
- [10] Green, P. J. and Sibson, R., “Computing Dirichlet tessellations in the plane,” *The Computer J.* **21**, 168–173 (May 1978).
- [11] Wu, Y., He, Y., and Tian, H., “A spherical point location algorithm based on barycentric coordinates,” in [*ICCSA (3)*], 1099–1108 (2005).
- [12] Mehlhorn, K., [*Datenstrukturen und effiziente Algorithmen, Band 1: Sortieren und Suchen*], vol. 1 of *Leitfaden und Monographien der Informatik*, Teubner, Stuttgart, Germany, 2 ed. (1988). 270–271.