Algorithm for Capturing a 3D Model from Multiple Camera Views

Jerry Tessendorf
Cinesite Visual Effects

March 16, 2000

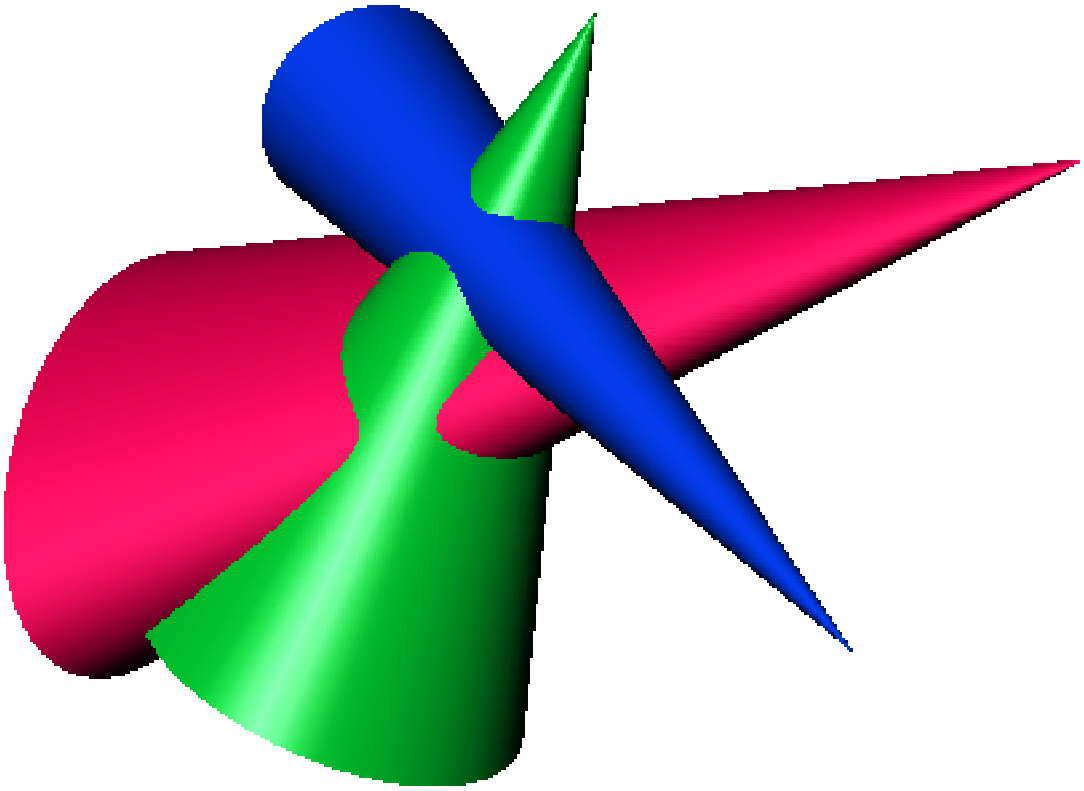

Figure 1: Depiction of three intersecting view cones.

1 Introduction

This document present a scheme for extracting a three-dimensional quad mesh surface geometry of an object whose shape and motion is captured with multiple ”ordinary” cameras (e.g. film or video cameras). The geometric problem is depicted in figure 1. In this approach, the outline of the object in the image plane of each camera is projected into space. The projected volumes from each of many cameras intersect with boundaries that may be quite complex. The red, green, and blue cones represent the projected volumes coming from the cameras at their apexes. The object that they view must be inside the volume common to all of the projected volumes. The surface that covers the intersection of the projections is a bounding hull of the actual object. With many cameras surrounding the object in many different views, the intersection surface approaches the actual shape of the object.

The algorithm is able to capture a 3D model even when just two cameras are used. With more cameras, the model more closely resembles the actual object shape. However, at all stages and with any number of cameras, the algorithm is ”robust” and ”resolution independent”.

What follows in this document is arranged in this way:

- The basic camera imaging equations for many cameras are presented.
- We introduce the silhouette of the object and the projected volume that it implies. A systematic parametric description of the surface of this volume is created.
- The intersection of two projected silhouettes is a curve in space. The construction of that curve is worked out in detail, for the intersection of two cameras.
- A procedure is generated for creating the surface of intersection for two cameras. This special limiting case allows a lot of mathematical grinding to be explained in the simplest case possible.
- The surface generation process for two cameras is generalized to N cameras, with some looseness in the explanation. The process is an iterative scheme based on multiple two camera intersections. However, the algorithm does not single out any camera as a control or special one in any way. The outcome is independent of the order in which cameras are used, although the intermediate steps have some order dependence.
- We back up and define in more detail the intersection operation and the data that is produced.

2 Imaging Equations

The basic equation for imaging maps points $\vec{\mathbf{r}}$ in 3D space to points $\vec{\mathbf{x}}$ on the image plane. Using the camera position $\vec{\mathbf{r}}_C$ and the camera pointing direction $\hat{\mathbf{n}}$, the imaging equation for the perfect camera is

$$\vec{\mathbf{x}} = \frac{\vec{\mathbf{r}} - \vec{\mathbf{r}}_C}{\hat{\mathbf{n}} \cdot (\vec{\mathbf{r}} - \vec{\mathbf{r}}_C)} - \hat{\mathbf{n}} \tag{1}$$

It is easy to verify that, although $\vec{\mathbf{x}}$ is written as a point in 3D space, the locus of such points as $\vec{\mathbf{r}}$ varies all lie on a plane that has $\hat{\mathbf{n}}$ as its normal. Also, the units of $\vec{\mathbf{x}}$ are dimensionless "tangent" units.

For multiple cameras, it is necessary to add some notational baggage which may seem a little awkward, but it is chosen in anticipation of even *more* notational baggage to come. We will designate the position and pointing direction for camera k as $\vec{\mathbf{r}}_C[k]$ and $\hat{\mathbf{n}}[k]$. Points on the image plane for camera k are $\vec{\mathbf{x}}[k]$. A point $\vec{\mathbf{r}}$ in space maps to the image plane of camera k as

$$\vec{\mathbf{x}}[k] = \frac{\vec{\mathbf{r}} - \vec{\mathbf{r}}_C[k]}{\hat{\mathbf{n}}[k] \cdot (\vec{\mathbf{r}} - \vec{\mathbf{r}}_C[k])} - \hat{\mathbf{n}}[k] \; . \tag{2}$$

3 The Image Plane Silhouette of a 3D Object in Several Cameras

We imagine a scene containing one or more 3D objects and two or more cameras. The cameras are placed so that any objects of interest are within the FOV of each camera, but the cameras do not have to be pointing to a common point, or distributed in the scene in some special way (although there are some weak requirements on the camera distribution, presented in later sections).

The image of each object in each camera is bounded by a silhouette, defined for our purposes as the closed, contigous boundary of the image plane object data. Every silhouette has a closed boundary, even if the edges of the image plane is a part of that boundary. For most objects and scenes, the object will have a complex silhouette that is not convex. For out purposes, we will want to use only convex silhouettes, which can be constructed by divided up a complex one into a set of convex ones. The image plane for camera k has a set $\mathcal{S}[k] = \{S_i | i = 1, \ldots, N_k\}$ of convex silhouettes $S_i[k]$, including independent objects and complex objects that have been divided up.

The boundary of each silhouette can be parameterized by a closed convex curve dependent on a single parameter. For the purposes of this algorithm development, we are not particulary concerned with whether the parameter is continous or discrete, but we do require that it be contiguous. If the parameter is t, with $t_{min} \leq t \leq t_{max}$, we require that as t increments from t_{min} to t_{max}, the bounding curve is traversed exactly and completely one time. We must also be free to define the points on the bounding curve for parameter values outside t_{min} and t_{max}. This is accomplished using branching, with the curve given as a function of $\text{modulo}(t - t_{min}, t_{max} - t_{min}) + t_{min}$.

For camera k, convex silhouette i, the bounding curve will be denoted $\vec{\mathbf{x}}^i[k](t)$.

Each image plane convex silhouette can be projected into the 3D space as a cone-like volume with a cross-section similar to the bounding curve. The surface of the silhouette volume is obtained just by raytracing every point on the bounding curve into space. Thus, the projected silhouette has the surface points

$$\vec{\mathbf{r}}[k](t, s) = \vec{\mathbf{r}}_C[k] + s\hat{\mathbf{r}}[k](t) \; , \tag{3}$$

with the ray direction

$$\hat{\mathbf{r}}[k](t) = \frac{\vec{\mathbf{x}}^i[k](t) + \hat{\mathbf{n}}[k]}{|\vec{\mathbf{x}}^i[k](t) + \hat{\mathbf{n}}[k]|} \tag{4}$$

Obviously for the projected volume, $s \geq 0$ and t is bounded by the min and max of that convex silhouette.

From this point on, the silhouette index label i will be suppressed, understanding that reference to any silhouette implies such a label.

4 Projected Silhouettes: Intersection Constraints

If two projected silhouettes, from two separate cameras, intersect, the intersection is characterized by a curve in space that is the locus of points that are on the surfaces of

both projections. This curve, for two cameras c and d, is defined by the equation

$$\vec{\mathbf{r}}[c](t_c, s_c) = \vec{\mathbf{r}}[d](t_d, s_d) \tag{5}$$

for suitable choices of the parameters t_c, t_d, s_c, and s_d. This vector equation provides three constraints on the four variables, leaving effectively one degree of freedom for the curve.

In this section we will examine the constraints and properties of the intersection curve that arise. The outcome of this examination is a reduction of equation 5 into expressions for s_c and s_d, along with a procedure for computing t_c and t_d from a single parameter which we label ℓ. This single parameter is the one degree of freedom describing the final curve. In the next section, these results are applied to build a concrete set of data structures and procedures for building a self-consistent description of the 3D curve of intersection, including the impact of limited resolution in the silhouette data in each camera.

After a little vector manipulation, we can solve for s_c and s_d directly in terms of $\hat{\mathbf{r}}[c]$, $\hat{\mathbf{r}}[d]$, and a vector $\Delta\vec{\mathbf{r}}[cd] \equiv \vec{\mathbf{r}}_C[c] - \vec{\mathbf{r}}_C[d]$ as

$$s_c = \frac{\hat{\mathbf{r}}[c] \cdot (1 - \hat{\mathbf{r}}[d]\hat{\mathbf{r}}[d]) \cdot \Delta\vec{\mathbf{r}}[dc]}{\hat{\mathbf{r}}[c] \cdot (1 - \hat{\mathbf{r}}[d]\hat{\mathbf{r}}[d]) \cdot \hat{\mathbf{r}}[c]} \tag{6}$$

$$s_d = \frac{\hat{\mathbf{r}}[d] \cdot (1 - \hat{\mathbf{r}}[c]\hat{\mathbf{r}}[c]) \cdot \Delta\vec{\mathbf{r}}[cd]}{\hat{\mathbf{r}}[d] \cdot (1 - \hat{\mathbf{r}}[c]\hat{\mathbf{r}}[c]) \cdot \hat{\mathbf{r}}[d]} \tag{7}$$

These two equations are symmetric, since one can be obtained from the other simply by interchanging the c and d indices. Another way to look at these solutions is to define the two unit vectors $\hat{\mathbf{e}}[c|d]$ and $\hat{\mathbf{e}}[d|c]$ by

$$\hat{\mathbf{e}}[c|d] = \frac{(1 - \hat{\mathbf{r}}[d]\hat{\mathbf{r}}[d]) \cdot \hat{\mathbf{r}}[c]}{\hat{\mathbf{r}}[c] \cdot (1 - \hat{\mathbf{r}}[d]\hat{\mathbf{r}}[d]) \cdot \hat{\mathbf{r}}[c]} \tag{8}$$

(switch c and d in this expression to get $\hat{\mathbf{e}}[d|c]$). The solutions are then

$$s_c = \hat{\mathbf{e}}[c|d] \cdot \Delta\vec{\mathbf{r}}[dc] \tag{9}$$

$$s_d = \hat{\mathbf{e}}[d|c] \cdot \Delta\vec{\mathbf{r}}[cd] \tag{10}$$

The solutions for s_c and s_d eat up two of the constraints in equation 5. The final one can be used to create a relationship between points on the image plane of camera d and the points on the image plane in camera c. As a step in that direction, note that one of the outcomes of equation 5 is the condition

$$(\hat{\mathbf{r}}[c] \times \hat{\mathbf{r}}[d]) \cdot \Delta\vec{\mathbf{r}}[cd] = 0 \; . \tag{11}$$

This says that the ray directions $\hat{\mathbf{r}}[c]$ and $\hat{\mathbf{r}}[d]$ form a plane that contains $\Delta\vec{\mathbf{r}}[cd]$. This is only true when the rays intersect. This equation also provides the final constraint equation that we are left with. Expanding the $\hat{\mathbf{r}}$ vectors using equation 4, we arrive at the following constraint:

$$0 = \{(\vec{\mathbf{x}}[c](t_c) + \hat{\mathbf{n}}[c]) \times (\vec{\mathbf{x}}[d](t_d) + \hat{\mathbf{n}}[d])\} \cdot \Delta\vec{\mathbf{r}}[cd] \equiv \lambda(t_c, t_d) \tag{12}$$

When this equation is satisfied, points on the image plane of one camera are directly mapped to points on the other image plane.

Note that the equation is pseudo-quadratic in image plane coordinates. This means that points of intersection that satisfy equation 12 come in pairs, i.e. the mapping is 1-to-$2n$, where $n \geq 0$ is some integer. This pairwise mapping property applies whether mapping from camera c to camera d, or from d to c. Geometrically, the reason for this behavior is that one of the points of a pair is the "entry wound" of a ray from a camera into the commmon surface region, and the other member of the pair is the "exit wound". We will find in the next section that the best approach to an implemented code is to use mappings in *both* directions between cameras.

There is one important situation in which there is only one intersection point however. This case is a degenerate situation, in which only an isolated point on the silhouettes has only one point of intersection in space. Geometrically, the effect occurs when the object in 3D space has a "corner" that is sufficiently sharp that two camera intersect exactly once at that cornder point. A simple example of this is a cube, then the cameras are looking straight onto adjacent faces. The corner(s) of the cube when both cameras see are degenerate points. There is a simple course of action to take in this case however: Continue to describe the situation in terms of entry and exit points, but now those points are colocated. The validity of this approach can be seen by moving to a point on the silhouette just away from the corner, and observe that as the silhouette point is moved closer to the corner, the entry and exit points come together as well.

An important property of all of the solutions to equation 12 is that they all lie on the ray $\vec{\mathbf{r}}_C[c] + s\hat{\mathbf{r}}[c](t_c)$, for suitable values of s. The particular values of s are obtained by converting the solution positons on the d camera image plane to direction vectors $\hat{\mathbf{r}}[d]$, and using equation 9 to compute the values of s.

Since we have restricted ourselves to convex silhouettes, there is at most one pair of points on camera c that map to a point on camera d, and vice-versa. Had we chosen to not restrict the silhouettes to convexity, there would in general be more that one pair in each mapping, and the integer n corresponds to the minimum number of convex pieces a silhouette can be divided up into.

5 Projected Silhouettes: Curve of Intersection

In this section we want to take the constraint equations, expressed in equations 9, 10, and 12 and algorithmically build data structures containing the curve of intersection. The data structures will describe this curve in several ways, convenient for building the final 3D geometry for the intersection of the projected silhouettes.

Suppose we begin with cameras c and d. We can construct the curve of intersection in the following steps:

1. Begin with the value $t_c = t_{c\ min}$. Set the curve parameter ℓ to its initial value ℓ_0. We now tag all quantities with the this parameter, as in: $t_c(\ell)$.
2. Compute the image plane point $\vec{\mathbf{x}}[c](t_c)$ and ray direction $\hat{\mathbf{r}}[c](t_c)$.
3. Search for the pair of values of t_d that make $\lambda(t_c, t_d) = 0$. This is likely to require that the data describing the silhouette boundary curve will be interpolated,

since the value of t_d will frequently lie between control vertices. If no values of t_d satisfies $\lambda(t_c, t_d) = 0$, skip to the last step[1].

4. Compute the value of s_c for each member of the pair. Identify the member that produces the smaller value of s_c as $s_c^{IN}(\ell)$, and the other member as $s_c^{OUT}(\ell)$.

5. Define the structure $P[c|d](\ell)$ with

$$P[c|d](\cdot) \equiv \{t_c, s_c^{IN}, s_c^{OUT}\} \tag{13}$$

and set the values to

$$P[c|d](\ell) = \{t_c(\ell), s_c^{IN}(\ell), s_c^{OUT}(\ell)\} \tag{14}$$

6. Increment t_c and ℓ. If $t_c \leq t_{c\ max}$ return to step 2.

The outcome of this process is an array $P[c|d]$ with elements $P[c|d](\ell)$, for appropriate values of ℓ.

Of course, there is nothing special about choosing to start with camera c and looping over its silhouette. We could have started with d and generated the quantity $P[d|c]$, which typically can be very different from $P[c|d]$. In fact, we need to construct both. The reason why is that, at least for a two-camera problem, when we reconstruct the intersection surface, we will want to use $P[c|d]$ when retrieving points that lie on the projected surface from camera c, and $P[d|c]$ when retrieving points that lie on the projected surface from camera d. For many cameras, a similar need exists but is generalized by all of the possible two-camera comparisons. Those details are presented after the two-camera case is handled in detail.

Finally, to reconstruct the 3D surface of intersection, we need the set $\mathcal{P}(d, c) = \{P[d|c], P[c|d]\}$. This is insensitive to the ordering of d and c indices, which is a nice situation because it prevents undue importance being placed on a particular camera. We can express this insensitivity as $\mathcal{P}(d, c) = \mathcal{P}(c, d)$.

6 Surface of Intersection for Two Cameras

The set $\mathcal{P}(d, c)$ provides sufficient data to reconstruct the 3D surface of intersection for two cameras. It is important to explicitly go through the process of surface reconstruction for two cameras, because just two cameras gives lots of insight into what is going on and why, while the many camera case hides lots of the insight in the details of combinatorics.

The essential fact needed to reconstruct the intersection surface is to realize that the pieces of the surface correspond to sections of the individual projected surfaces,

[1] When no values exist, two conditions are possible: (1) the two silhouettes do not overlap, in which case none of the points on either silhouette satisfy $\lambda = 0$; or (2) the intersection curve may be in two unconnected pieces, in which case a little extra information must be tracked in order to flag them as separate and treat them in a consistent way through the rest of the process. We will assume that the tracking of a multiply connected set is accomplished via the parameter ℓ, i.e. the value of ℓ signals which disconnected segment of a point is part of.

with restricted ranges of s and t for each one. Identifying these ranges for each projected surface produces the intersection surface. In fact, $\mathcal{P}(d,c)$ contains precisely that information.

Lets see how the information in $\mathcal{P}(d,c)$ can be used to generate a wireframe of the surface consisting of quadrilaterals. Focusing on camera c, $P[c|d](\ell)$, for any valid choice of ℓ, tells us that a line segment from the point $\vec{\mathbf{r}}[c](t_c(\ell), s_c^{IN}(\ell))$ to the point $\vec{\mathbf{r}}[c](t_c(\ell), s_c^{OUT}(\ell))$ is a part of the intersection surface lying on the projected surface from camera c. Taken together, the ordered set of points $\{\vec{\mathbf{r}}[c](t_c(\ell), s_c^{IN}(\ell))\}$ forms a curve corresponding to the family of points of the "entry wound" of projected surface c into d. Similarly, $\{\vec{\mathbf{r}}[c](t_c(\ell), s_c^{OUT}(\ell))\}$ is the "exit wound" curve.

We construct quads out of this data as follows:

1. Assume a resolution step size ds of interest.

2. For curve parameter ℓ and adjacent value ℓ', we get the four points using values from $P[c|d](\ell)$ and $P[c|d](\ell')$

$$\vec{\mathbf{r}}_1 = \vec{\mathbf{r}}[c](t_c(\ell), s) \quad (15)$$
$$\vec{\mathbf{r}}_2 = \vec{\mathbf{r}}[c](t_c(\ell'), s) \quad (16)$$
$$\vec{\mathbf{r}}_3 = \vec{\mathbf{r}}[c](t_c(\ell'), s + ds) \quad (17)$$
$$\vec{\mathbf{r}}_4 = \vec{\mathbf{r}}[c](t_c(\ell), s + ds) \quad (18)$$

   for values of s and $s + ds$ within the range $[s_c^{IN}(\ell), s_c^{OUT}(\ell)]$ for points 1 and 4, and in the range $[s_c^{IN}(\ell'), s_c^{OUT}(\ell')]$ for points 2 and 3.

3. The four points $\vec{\mathbf{r}}_i$ form a quadrilateral approximating a patch of the c camera projected surface. Loop over values of ℓ and s to produce the full set of quads in the intersection surface from camera c.

Having built a set of quads for projected surface c, denote the set as $Q[c|d]$.

The remaining part of the intersection surface is a patch from the camera d projected surface. We can build that one by following the same process as just outlined, switching the c and d labels everywhere, producing the set $Q[d|c]$. After doing so, the outcome is two quad-based patches which connect on seams at the intersection curve. Each patch was converted to quads at independent spatial resolutions suitable for the camera qualities and desired artistic use. The final combined surface of intersection, $Q(c,d)$ is the union of the two pieces:

$$Q(c,d) = Q[c|d] \cup Q[d|c] \; . \quad (19)$$

7 Surface of Intersection for N Cameras

Having built the surface of intersection for just two cameras, we want now to move on to a process for building the surface from any number of camera views. with just a little more definition work, it is relatively straight forward to extend the process to any number of cameras, keeping the overhead relatively small.

Recalling that the quantity $P[c|d]$ comes from comparing camera d to c, we want to define a quantity $P[c|]$, which compares camera c with iteself, i.e., is just the set of parametric points that define the entire surface:

$$P[c|](\ell) = \{t_c(\ell), 0, s_{max}\} \,, \tag{20}$$

In this case, the ℓ parameter runs over every value of t_c on the silhouette, and s_{max} is an arbitrary large and safe cut-off value. With this definition, we can build an intersection operator by the relationship

$$P[c|d] = P[c|] \cap P[d|] \,. \tag{21}$$

This intersection operator must identify disconnected regions of intersection, and limits its scope to only ranges of t_c which are within the set defined in the left-hand of the operator. Also, in addition to the steps performed in section 6, the comparison test $\lambda = 0$ is supplemented with a test that the s_c^{IN} and s_c^{OUT} values are within the range allowed by $P[c|]$ (i.e. the left-hand argument of the $\cap$ operator).

Now suppose there are N cameras of interest. For the moment, pick one of them and label it c, and the others are labelled d_1, d_2, ..., d_{N-1}. The portion of the intersection surface coming from the c surface is obtained by intersecting all of the others with c. For one camera, the intersection surface is $P[c|]$. For two cameras, c and d_1, it is $P[c|d_1]$. For three cameras c, d_1, and d_2, the intersection surface piece is

$$P[c|d_1 d_2] = P[c|d_1] \cap P[d_2|] \,. \tag{22}$$

We can continue this iteractive approach for N cameras. With the iterative expression

$$P[c|d_1 d_2 \ldots d_{m+1}] = P[c|d_1 d_2 \ldots d_m] \cap P[d_{m+1}|] \,, \tag{23}$$

the final intersection surface coming from camera c is $P[c|d_1 d_2 \ldots d_{N-1}]$.

As with the two camera case, we can build a set of quads from $P[c|d_1 d_2 \ldots d_{N-1}]$, and label it $Q[c|d_1 d_2 \ldots d_{N-1}]$. The full surface of intersection is defined as the union of all of the intersection pieces as the surfaces are cycled through:

$$Q(1, 2, \ldots, N) = \cup_{c=1}^{N} Q[c|d_1 d_2 \ldots d_{N-1}] \,. \tag{24}$$

This can be built iteratively as well, by setting $Q(0) = \{\}$, the empty set, and iterating as

$$Q(1, \ldots, m+1) = Q(1, \ldots, m) \cup Q[m+1|d_1 d_2 \ldots d_{N-1}] \,. \tag{25}$$

Several comments are in order at this point.

1. The combined process of iteratively building $P[c|d_1 d_2 \ldots d_{N-1}]$ and $Q[c|d_1 d_2 \ldots d_{N-1}]$ is relatively efficient in cpu resources. The data for any particular camera is needed in a very localized portion of the algorithm, so that the full data set from *all* of the cameras need not be resident simultaneously. But delaying input of camera data until the moment it is needed does not increase cpu time, so there is no cost to the delay.

2. Because the final geometry $Q(1, 2, \ldots, N)$ is a union, it can be built up as a file that is appended as each $Q[c|d_1 d_2 \ldots d_{N-1}]$ is constructed. So the memory footprint of the code and data is independent of the complexity or resolution of the final geometry.

3. This algorithm is inherently independent of the order in which cameras are processed.

4. The iterative implementation of the construction of $P[c|d_1 d_2 \ldots d_{N-1}]$ provides an opportunity for quickly eliminating silhouette combinations that do not produce geometry. At some stage in the iteration, such a situation would produce an empty set at the result, immediately terminating the iteration without producing any geometry.

8 General Expression of the Intersection Operator and Process

The intersection operator $\cap$ used in section 7 is the fundamental activity in this algorithm. Because of its importance, this section is devoted to restating its operation in detail, for the most general conditions that might be encountered. Specifically, the intersection operator gives meaning to the relationship

$$C = A \cap B \tag{26}$$

with A, B, and C defined as silhouette sections.

The results of this section applies to generally-shaped silhouettes in any camera image planes. The silhouettes need not be convex.

8.1 Silhouette Sections

Before explicitly defining the operation of $\cap$, we first more carefully define the structures that it operates on. Silhouette sections have the general form

$$SS = \left\{P^1, P^2, \ldots, P^n, M\right\} \tag{27}$$

where each P^i is an ordered set with the form

$$P = \left\{t(\ell), s^{IN}(\ell), s^{OUT}(\ell) | \ell = \text{index}\right\} \tag{28}$$

and M is the camera data/functions needed to compute points on the image plane from a value of t and points in space from values of t and s.

The index ℓ may be continous or discrete, depending on the circumstances defining the silhouette. However, in all cases, the elements of P are ordered, in the sense that $t(\ell)$ is a monotonic function of ℓ (with the exception of branching), and adjacent values of $t(\ell)$, $s^{IN}(\ell)$ and $s^{OUT}(\ell)$ may be interpolated in some way to produce intermediate values. There can be any number of P^i sets in SS, but each one represents a contiguous piece of the surface generated by the silhouette, and the P^i do not overlap.

8.2 Intersection Operation

To begin fleshing out the meaning of equation 26, we express A, B, and C in terms of their contents as

$$R = \left\{P_R^1, P_R^2, \ldots, P_R^{n_R}, M_R\right\} \quad (29)$$

$$P_R^i = \left\{t_R(\ell_R^i), s_R^{IN}(\ell_R^i), s_R^{OUT}(\ell_R^i)\right\} \quad (30)$$

where R = A, B, or C.

Before defining intersection, we first need to understand an operation called a *split*. Imagine a Silhouette section SS, containing a member P consisting of

$$P = \left\{t(\ell), s^{IN}(\ell), s^{OUT}(\ell)\right\} \quad (31)$$

for some range of ℓ. A split at ℓ_s is the creation of two separate sets from P:

$$P_- = \left\{t(\ell), s^{IN}(\ell), s^{OUT}(\ell) | \ell < \ell_s\right\} \quad (32)$$

$$P_+ = \left\{t(\ell), s^{IN}(\ell), s^{OUT}(\ell) | \ell \geq \ell_s\right\} \quad (33)$$

When a split is performed, P is replaced by P_- and P_+.

The intersection operator $\cap$ expresses $C = A \cap B$ via the following set of steps:

1. Initialize $C = A$.
2. Begin with the first triplet, $(t_A, s_A^{IN}, s_A^{OUT})$ from the sets of triplets in A. Because of the initialization in the previous step, there is a corresponding triplet in $(t_C, s_C^{IN}, s_C^{OUT})$ in C.
3. Compute the image plane point $\vec{\mathbf{x}}[A](t_A)$ and ray direction $\hat{\mathbf{r}}[A](t_A)$.
4. Search for the values of t_B that make $\lambda(t_A, t_B) = 0$. The search should be conducted systematically through the P_B^i until all of them are found. There should be exactly two or none of them, or possibly one in the case of degenerate intersections.. If no values of t_B satisfies $\lambda(t_A, t_B) = 0$, perform a split on the current P_C, and remove this element from the split set.
5. Compute the two values of s_A for the two values of t_B. Label the two values $s_A(1)$ and $s_A(2)$, with $s_A(1) \leq s_A(2)$. If $s_A(1) > s_C^{OUT}$ or $s_A(2) < s_C^{IN}$, there is no overlap. When there is no overlap, perform a split on the current P_C, and remove the present element from the split set. If there is overlap, set

$$s_C^{IN} = \max\left(s_C^{IN}, s_A(1)\right) \quad (34)$$

$$s_C^{OUT} = \min\left(s_C^{OUT}, s_A(2)\right) \quad (35)$$

6. Move to the next triplet $(t_A, s_A^{IN}, s_A^{OUT})$ in A with corresponding triplet in C.
7. Repeat beginning at step 3 until all triplets in A have been evaluated.

8.3 Properties of the Intersection Operator

The intersection operator has several properties which may or may not ever be useful. Some of them are listed here:

- Idempotence:

$$A \cap B = (A \cap B) \cap B \tag{36}$$
$$A \cap B = A \cap (A \cap B) \tag{37}$$

- Noncommutative

$$A \cap B \neq B \cap A \tag{38}$$

This completes the more rigorous formalized definition of the intersection operation used in equation 23.

8.4 Special Cases

There are two special situations in which the intersection process does not work as described above. These cases, though rare, are easily identified and handled by extension. In this section we present each special case and the extension to the intersection process.

8.4.1 Sharp Corner Intersections

If the object has "sharp" corners that two or more cameras preserve as sharp in their silhouettes, then for both cameras the entrance and exit points can be colocated. For example, a rectangular box, seen by two cameras looking down principle axes of the box, will contain a common corner of the box in their silhouettes. That corner will appear to only have one intersection point. In fact, it is a degenerate case in which the two intersection points on at the same 3D location. This case is handled by setting the IN and OUT points to the same location, but otherwise treating it the same as all other intersections. The criteria for identifying that this case is present are: (1) only one intersection point, and (2) neither camera is in the fov of the other.

8.4.2 Camera Embedded in FOV

In some situations the camera being used to build an ordered set P is inside the field of view of camera that is being intersected. In this case, only one intersection will be found, which is the OUT point of the intersection. The IN point is the previous value. Detecting this situation is relatively easy. If there is only one point of intersection, and the camera is inside the fov of the second camera, then this case applies.

9 Texture Coordinates

Certainly an important part of using the intersection geometry $Q(1, 2, \ldots, N)$ is putting textures on the surface. In particular, putting the image data from the N cameras onto

the surface. Since all of the points on the intersection surface are seen by all of the cameras, there is the possibility of having N different texture values assigned to each point of the surface. We do not at this stage want to "weed out" which texture(s) is (are) the best to use, since that may be very dependent on the application. For example, the texture value chosen may depend on the view angle of the rendering camera with respect to the data cameras. So we need a scheme to provide texture coordinates from multiple camera perspectives.

The fundamental thing to do here is to go back to the point at which the quad vertices $\vec{\mathbf{r}}_i$ were generated for each set $Q[c|d_1 d_2 \dots d_N]$. Each of these points has a corresponding position in each of the N camera planes, i.e. each quad vertex generates a set of texture coordinates:

$$\vec{\mathbf{r}}_i \rightarrow T_i \equiv \{\vec{\mathbf{x}}_i[1], \dots, \vec{\mathbf{x}}_i[N]\} \tag{39}$$

So, as part of the quad data, we store the texture coordinate set T_i.

Of course, this procedure does not define a *unique* texture coordinate system over the entire intersection surface. If the desired texture were some synthetically generated texture rather than the image data, it would not be clear which member of T_i is the one to use. There is no mathematically clean answer to this issue. However, one may try using the coordinates for the camera that is the most othogonally incident to a particular face of the surface.

10 Implementation

This algorithm is built as an iterative scheme, so that at any stage of the process data from only two cameras is needed. An implementation should take advantage of that efficiency. If a large number of cameras are employed in a practical case, it will be very difficult to keep all of the data present at any given time. Limiting it to just to two necessary cameras at any given stage will be important.