

# Animating a Camera for Viewing a Planar Polygon

Daniel Brunstein\*    Gill Barequet†    Craig Gotsman‡

Center for Graphics and Geometric Computing  
Department of Computer Science  
Technion – Israel Institute of Technology

## Abstract

Many applications, ranging from visualization applications such as architectural walkthroughs to robotic applications such as surveillance, could benefit from an automatic camera trajectory planner. This paper deals with that problem. We have automated the process of inspecting the outside of a simple two-dimensional polygon, given a few user parameters. Our algorithm preprocesses the polygon using Visibility Graph like concepts, and creates a visibility data structure for each polygon edge. From these structures, "good" camera zones are computed. Natural cubic splines are then used to create a closed camera trajectory that passes solely inside the zones. An iterative process refines the trajectory by minimizing a cost function until it converges to the optimal result.

**CR Categories and Subject Descriptors:** I.3.3 [Computer Graphics]: Picture/Image Generation—viewing algorithms; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation;

**Additional Keywords:** camera animation, cinematography, visibility graph, natural cubic spline, planar maps.

## 1. INTRODUCTION

Consider a planar shape that we want to visualize, or whose outside we want to inspect. In real life, this could be the floor plan of a 3D architectural model, an art gallery, a prison's exterior, or even a toy within a complex scene of a motion picture film. In all these cases the shape may be represented as a simple polygon in the plane. Our aim is to find a closed path for the camera's position and direction, so that the shape is filmed in a visually-pleasing way. Creating such a trajectory manually is a time-consuming task for an unexperienced animator, architect or 3D graphics artist. Many keyframes for the camera location and its target must be placed and iteratively revised. Hence a utility which automatically creates such a trajectory will release the animator from a lot of grunt work, allowing him to concentrate on the more important cinematographic aspects.

A trivial solution to our problem, namely placing control points next to edges or along the angular bisectors of vertices and connecting them to form a path, is far from adequate. Various problems such as path parameterization, camera look-at directions, occlusions and quality of the result immediately arise.

---

{\*danikoz,†barequet,‡gotsman}@cs.technion.ac.il

We would like to construct a camera trajectory that results in a visually pleasing animation of the polygon's exterior. By "visually pleasing", we mean that the edges are sufficiently visible throughout the animation, the camera is not too close to the polygon, and the trajectory is sufficiently smooth and not unnecessarily long. In Section 3 we elaborate on the first two criteria, which define regions where certain discrete viewpoints of the camera should be placed. The visibility criterion is the more difficult one, and most of our efforts focus on this criterion. In Section 4 we elaborate on the last two requirements which relate to the nature of the entire curve.

In a nutshell, our algorithm proceeds as follows: In the first stage we preprocess the given polygon and create a data structure that contains visibility information related to the polygon. In the second stage we define regions for each of the polygon edges, based on the data structures. The regions, called *zones*, comply with certain user-defined requirements (constraints). A camera trajectory through these zones is created in the last stage by minimizing a user-defined cost of the curve.

## 2. RELATED WORK

The focus of this work is cinematographic, since we are trying to find a trajectory for a camera which produces a given desired effect. This type of inverse problems have attracted the attention of computer graphics practitioners since the seminal paper of He et al. *The virtual cinematographer* [3]. The focus of their work is primarily on capturing the interaction of virtual actors. Our intention is not to follow a specific animated character, but to act as an observer. In another context, for a moving camera filming a two-dimensional landscape where historical data is visualized, a different approach was taken by Stoev et al. [8]. They position a virtual camera such that the projected area and the depth of the scene are maximized.

Cameras play an important role in robotic applications, and visual servoing (also known as image-based control) is widely used to control robots [4]. The paper most related to ours in the robotic arena [5] treats the control of a camera in a virtual environment, with the addition of constraints in order to react automatically to changes in the environment. A similar work by Halper et al. [2] arises from the field of computer games, where an automatic camera tracks the movement of a player. Halper et al. emphasize the trade-off between constraint satisfaction (guiding the camera controls) and frame coherence (smooth transitions with appropriate cuts). In both these works, which operate in near real-time scenarios, there is little emphasis on the perfection of the visual quality and the global continuity of the camera path.

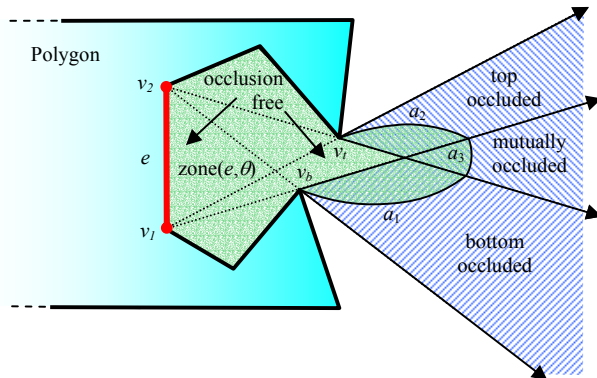
Another robotic approach to a similar problem is discussed in [10]. Here, near-optimal viewing positions for a robot-mounted camera are heuristically found. A robot's path between a given

sequence of singular visual tasks is generated. However, there is little emphasis on the viewing in between the visual tasks.

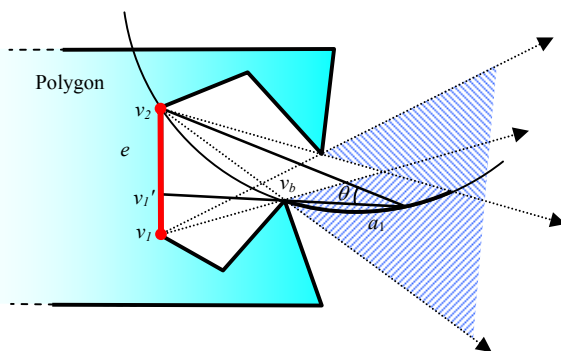
Another aspect of our work is related to visibility computation. The visibility of a polygon's edge, possibly occluded by other edges, is analogous to the case in which the edge acts as a uniform light emitter. This linear light source may cast shadows creating different regions (umbra, penumbra, antiumbra, and antipenumbra), as discussed in [9]. Stark et al. [7] deal with the resultant illumination of a radiant polygon with a polygonal occluder, and its shadow spread on a hyperplane (a plane of lesser degree). An extension of the well-known Visibility Graph data structure, described in detail in [1], is used in our algorithm.

### 3. CAMERA ZONES

The first stage of our algorithm is based on determining optimal camera location and view direction per polygon edge. These locations are such that the visible part of the edge extends an view angle of at least  $\theta$  to the camera, and the camera is no closer than distance  $d$  from the edge. In this section we show how to compute regions of the plane per edge which are the locus of the viewpoints satisfying this requirement. These regions will be called *camera zones*.



**Figure 1:** Occlusion Free, Top, Bottom, and Mutual occluded regions. The resulting visibility zone is overlaid in green.



**Figure 2:** Circular arc  $a_1$  in the bottom occluded region.

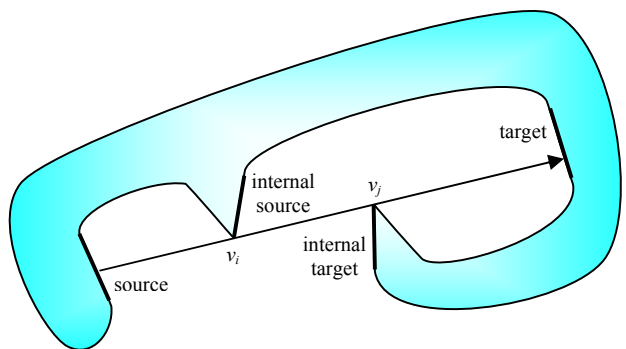
### 3.1 View Angle Criterion

The view angle criterion, which is the most potent one, is based on the desire that the view angle extended by the visible part of the edge is at least  $\theta$ . For a single edge  $e$ , the set of points satisfying this is a bounded region, which we call a *visibility zone*. The zone is created by preprocessing the polygon and creating the Angular Visibility Regions (AVR) data structure, which characterizes the visibility of each polygon edge in the plane. In the trivial case where there are no occluders in front of  $e$ , the visibility zone is the interior of a circular arc whose chord is  $e$  and arc length (in radians) is  $2\pi - \theta$ .

In the second case,  $e$  is partially occluded by the polygon itself (we call this *self-occlusion*). Hence the region in front of the edge is partially occupied by a fragment of the same polygon. Assume, without loss of generality, that  $e$  is vertical, and denote its endpoints by  $v_1$  and  $v_2$  (see Fig. 1). A convex vertex  $v_b$  reaching over (spiraling) from the top side of the edge, intrudes on the space in front of  $e$  and creates a region that is occluded from  $v_2$  (we call this a "top occluded" region). Another convex vertex  $v_b$  creates the symmetric region in which  $v_1$  is occluded ("bottom occluded" region). These occlusions may coexist and result in a region that is mutually occluded, namely neither  $v_1$  nor  $v_2$  are visible ("mutually occluded" region).

In these four different regions (occlusion free, top occluded, bottom occluded, and mutually occluded) the visibility zone is computed differently. However, it is still bounded by circular arcs. The occlusion-free region is identical to the trivial case discussed earlier. In the bottom occluded region, excluding the mutually-occluded region, the set of points seeing  $e$  with view angle  $\theta$ , is a circular arc  $a_1$  whose chord is  $v_2v_b$ . This is because all points of that circular locus view  $v_2v_b$ , hence  $v_2v_1'$  (where  $v_1'$  is the extension of  $v_b$  on  $e$ ), at the same angle  $\theta$  (Fig. 2). Similarly, the top-occluded region creates a circular arc  $a_2$  whose chord is  $v_1v_t$ . In the mutually-occluding region, this locus is a circular arc  $a_3$  whose chord is  $v_bv_t$ , since all the locus points can see the edge  $e$  at the same angle  $\theta$  allowed by the opening of  $v_b$  and  $v_t$ .

The union of these four regions yields the desired visibility zone for this case (green in Fig. 1).



**Figure 3:** A general 'LR' VVV, with source, internal source, internal target and target edges. This VVV plays a role in determining changes in the visibility of each of these edges.

### 3.1.1 Vertex-Vertex Visibility Segments

To construct the AVR data structure for edge  $e$ , we need to detect the combinatorial changes in the visibility of  $e$  that occur due to occluders. We define a *VVV* (Vertex-Vertex Visibility) to be a (possibly unbounded) directed line segment, passing through vertices  $v_i$  and  $v_j$ , if they are mutually visible (i.e. the line segment between them lies completely outside the polygon), and at least one of which is convex. We associate with each VVV up to four different edges (see below). Crossing a VVV may change the visibility of some of these edges. We have extended the classical Visibility Graph [11] by adding to each VVV this information (Fig. 3).

A VVV where the two vertices are nonconvex is not interesting since crossing it does not affect the visibility of any of the four adjacent edges. The VVV direction is arbitrary, but once fixed, the vertices and edges associated with the VVV are classified according to that direction. Each of the two vertices is classified as one of the type 'R', 'L', and 'B', for a total of six combinations (see Fig. 4). The type specifies whether the designated vertex touches the VVV from its right side, from its left side, or from both sides (in case the vertex is nonconvex). Traversing the VVV along its defined direction, the source edge (if exists) intersects it immediately before  $v_i$ , and the target edge (if exists) intersects it immediately after  $v_j$ . The internal source edge is the edge incident to  $v_i$ , that lies in the direction of the VVV. Symmetrically, the internal target edge is the edge incident to  $v_j$ , that lies in the opposite direction of the VVV. By definition, there may be at most one source edge and/or one target edge per VVV. In the case where vertex  $v_i$  is of type 'B', the VVV has two internal source edges (and naturally no source edge at all). The same holds for VVVs with vertex  $v_j$  of type 'B', having two internal target edges and no target edges.

The algorithm to find these VVVs is:

#### Procedure CalcVVVs(P)

```

Input: A simple polygon P of n vertices
Output: An array of valid VVVs
for each vvv of all possible vertex pairs of polygon P
  set vvv type (depicted in Fig. 4)
  if vvv vertices start by entering the polygon P or
    vvv of type 'BB'
  then skip this vvv and continue for next vvv
  endif
  set vvv internal edges
  for each edge e of the polygon P
    find intersection between edge e and the vvv line
    if no intersection
    then skip this edge and continue for next edge
    endif
    if intersection lies between vvv vertices
    then exit for each edge loop and
      skip this vvv and continue to next vvv
    endif
    keep the edge
  end for
  set the nearest intersection edge as source and/or target edge
  keep vvv
end for

```

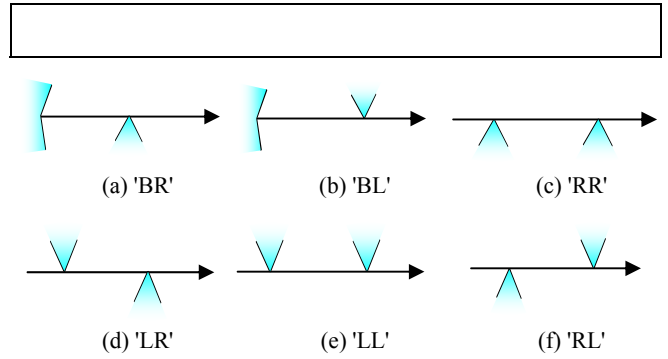


Figure 4: All possible type combinations for a VVV.

### 3.1.2 Computing AVRs

The AVR is a data structure that maintains, per each edge, the different visibility regions created in the self-occluding case (as in Fig. 1). As such, it is a planar map. The AVRs are constructed based on the VVVs. All the four different regions in the self-occluding case are simple polygons whose visibility relationship with the edge are different. Their visibilities are different since each region is created due to a different occluding vertex, and sees a different endpoint. This means storing for each region the occluding vertex and the relevant visible vertex. These two vertices are actually the endpoints of the chord of the circular arc for that region (see Fig. 2). The mutual occlusion zone can be derived from the top and bottom occlusion regions, thus is not explicitly present in the AVR.

We now describe the structure of a single AVR of an edge. The AVR consists of three fields: (a) A single polygon defining the "occlusion free" region which is linked to the two edge vertices  $v_i$  and  $v_{i+1}$ ; (b) A list of "top occluded" regions, which is a simple linked list of regions that are occluded from the edge by a top convex vertex. Each of these regions is stored as a polygon and linked to the appropriate chord, e.g.  $v_i$  and  $v_i$ ; and (c) A list of "bottom occluded" regions that are occluded from the bottom, with their corresponding chords, e.g.,  $v_{i+1}$  and  $v_b$ .

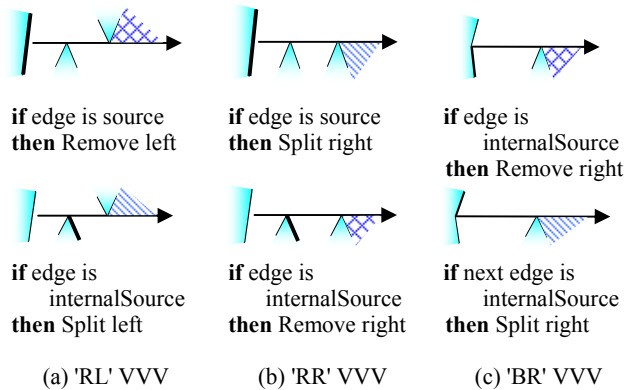
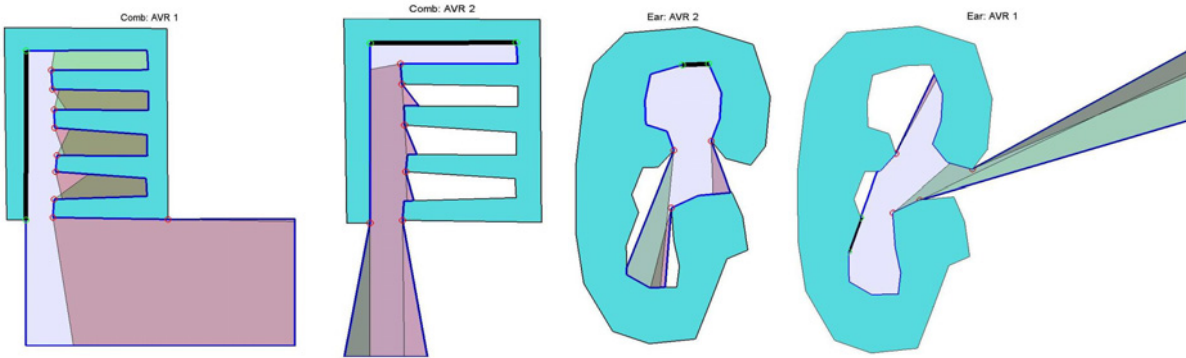


Figure 5: All possible cases for updating the AVR. The edge in question is highlighted.

When all VVVs have been computed, we collect for each edge all its corresponding VVVs and process them to create the AVR for that edge. Since each VVV causes a change in the region in front of the edge, we process each VVV and update the AVR. These



**Figure 6:** Four different AVRs, two per polygon. The polygon is in blue, and the relevant edge is highlighted. The occlusion-free region is colored in transparent blue. The top-occluding region is colored in transparent purple, while the bottom-occluded region is colored in transparent green. Note the color mixing in the mutually occluded region.

changes can affect the full, top, or bottom occluding region list, by either removing or splitting part of the region (see Fig. 5).

Since the VVV is directed, we consider only the source and internal source edges. Thus the visibility effect occurs at the second VVV vertex (the second vertex along it). Target and internal target edges are handled similarly. The two main actions are 'Remove' and 'Split'. Their input is either 'left' or 'right', indicating the direction in which the action should be performed. 'Remove left' is activated when the region to the left of the VVV, after the second vertex, can be removed since it is completely obscured from the edge (see Fig. 5(a), top case). The removal of the left region takes place in the three AVR data fields. In contrast, 'Split left' is activated when the region to the left of the VVV, after the second vertex, is about to become a "top occluded" region, since the second VVV vertex occludes the top vertex of the edge (see Fig. 5(a), bottom case). In a 'Split' case we modify only the top or bottom occluded regions, depending on the specific situation, by splitting an existing region and creating two new subregions. In the 'left' case, the new left subregion will have the second VVV vertex as its occluder, while the right subregion will maintain its old occluder.

The AVR induces an arrangement of line segments (possibly rays), defining a set of cells in a planar map. The visibility of the edge is the same for all points in a given cell, and different from cell to cell. With each cell we associate the two relevant vertices that are the endpoints of the chord of the circle whose visibility angle of the edge is  $\theta$ . Computing the cells explicitly is done using polygon boolean operations on the arrangements. Fig. 6 shows some examples of AVR's.

### 3.1.3 Creating the Visibility Zone

The last step in the creation of the visibility zone is using the cells and the given view angle  $\theta$  to find the zone per cell. In each cell, the angle  $\theta$  defines a circle whose chord is delimited by the relevant vertices (shared with the cell). The zone per cell is the intersection of the cell and its circle. This gives us a zone contained in the cell, such that each point within complies with the desired angle requirement.

By uniting all the zones obtained from the cells into one large region, we create the visibility zone for the given edge. All the interior points of the zone have a view angle to the edge larger than  $\theta$ , and all the boundary points "see" the edge at an angle of

exactly  $\theta$ . The zone is a simple nonconvex polygon shape consisting of straight edges and circular arcs. In our implementation we approximated the arcs by polylines to facilitate boolean operations on the zones.

## 3.2 Distance Criterion

Another important criterion for edge viewing is based on distance. We would like to prevent the camera from approaching an edge closer than a minimum given offset distance  $d$ . This value may be fixed for all edges, or a function of the edge size. Each such offset defines a halfplane, the so-called offset zone, in which the camera can be placed.

## 3.3 The Camera Zone

Each criterion defines a zone and the intersection of the visibility zone and the offset zone is the *camera zone*. Using these criteria with varying parameters allows us to control the quality and functionality of the camera trajectory.

A variety of different camera zones result from different combinations of the parameters  $\theta$  and  $d$  governing the criteria. For example, by using a small  $\theta$  and a large  $d$ , we produce camera zones that are far away from the edges, which are suitable for scenic footage. A large  $\theta$  and very small  $d$  would yield simpler camera zones that are extremely close to the edges.

## 3.4 Common Zones

Consider the situation in which the camera zones of two consecutive edges do not intersect. This may easily happen at convex vertices if  $\theta$  and  $d$  are large enough. More precisely, if  $\theta > 2\alpha$ , where  $\alpha$  is the angle at the vertex (e.g. consider  $v_{i+2}$  in Fig. 7 and increase the offset and angle). If the camera zones are disjoint, any path will pass through some points outside the zones. Since our requirement is that all points satisfy the criteria, we cannot allow this. Thus a region, called the *common zone*, that is the intersection of two consecutive camera zones, is defined in front of each common vertex. In the next section we use these regions to place the initial camera trajectory control points.

If the parameters specified by the user create a discontinuity of the camera zones, our algorithm solves this locally. When an empty intersection is detected, the criteria for the two corresponding

edges are relaxed till an intersection of the zones occurs. Namely,  $\theta$  and  $d$  are reduced, thereby increasing overall zone area.

#### 4. PATH GENERATION

In this section we show how to generate, given a polygon, an optimal camera trajectory. Having already processed the polygon, we have for each edge the camera zones—the locus of points from which the edge is seen at the desired qualitative level.

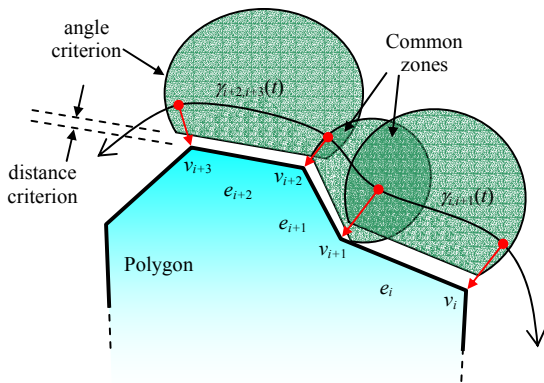
We now define the path generation problem. Given an ordered list of simple polygons (visibility zones), each two consecutive polygons with a nonempty intersection, find a path lying entirely in the union of these polygons, passing through each in turn, and minimizing a given cost function.

We restrict ourselves to paths that are cubic spline curves, interpolating *one control point per common zone*. In our setting a *curve* is a cubic curve connecting two control points, and a *path* is the closed  $C^2$  curve connecting all pairs of consecutive points.

A trivial uniform parameterization  $t \in [0,1]$  of each curve is inadequate in our case. We are interested in a smooth transition from one curve to another, and a uniform parameterization would cause the transition from a very long curve to a very short curve to be abrupt. The "velocity" in one curve should be similar to that in the neighboring curve. Hence an approximation of an arclength parameterization was used:

$$t \in [0, L_{i,j+1}], \text{ where } L_{i,j+1} = |P_{i+1} - P_i|.$$

The most important requirement from the path – that it resides completely inside the camera zones – is encouraged by ensuring that each curve passes within its two corresponding common zones, hence we place the control points in these zones. See Fig. 7.



**Figure 7:** Camera zones created from intersections of visibility zones and offset zones. Common zones are intersections of two consecutive camera zones. The curve control points (in red) are located inside the common zones.

#### 4.1 Camera and Target Parameterization

A very important aspect, which has not yet been addressed, concerns the camera target. In other words, what is the camera looking at?

The correct way to view the problem is to treat it in one higher dimension. We view the path as a curve in the 3D space of  $x$ ,  $y$ , and  $\alpha$ . Each control point consists of its coordinates and the angle at which it views the corresponding polygon vertex. The parameterization in this space will make the transition from one control point to another smooth both in the spatial and the angular domains. Moreover, using this approach allows us to incorporate in the cost function factors that are solely dependent on the viewing direction.

#### 4.2 The Path Cost Function

The key to a successful path is a well-defined cost function, one that incorporates the features that we want, with the appropriate weights indicating the importance of each feature.

Our cost function incorporates both local and global features. We want the weights to be geometrically unitless, so features like curvature and length are normalized by dividing the respective term by the term of a "normal" path. The normal path that we use is the unconstrained path whose control points are the polygon's vertices. This is a path that albeit partially being inside the polygon, its curvature and length are proportional to the features of the polygon. Thus it is a good entity to normalize with, and makes the minimum invariant to similarity transformations of the input polygon.

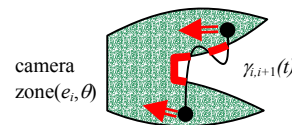
The cost function that we use is

$$C = W_{\text{OutOfZone}} \cdot C_{\text{totalOut}} + W_{\text{LocalCurvature}} \cdot \frac{C_{\text{LocalCurvatureSum}}}{C_{\text{DefaultLocalCurvatureSum}}} + W_{\text{MaxCurvature}} \cdot \frac{C_{\text{MaxCurvature}}}{C_{\text{DefaultMaxCurvature}}} + W_{\text{Length}} \cdot \frac{C_{\text{TotalLength}}}{C_{\text{DefaultTotalLength}}}$$

We next describe the different terms in this function.

##### 4.2.1 Keeping the Curve Within the Zone

The most important property of the path we want to enforce is that each curve resides entirely within its corresponding camera zone. The two control points lie inside the zone, but nothing guarantees that the entire curve does so too. We charge a penalty for each time the curve leaves the camera zone, by summing the lengths of the zone boundaries that correspond to the portions of the curve where it passes outside the polygon (see Fig. 8). We call this zone boundary the *external boundary*.



**Figure 8:** The external boundary of the camera zone (in bold red), consisting of two zone boundaries that correspond to external portions of the curve.

By minimizing the external boundary, we "encourage" the iterative process to move the curve control points to a configuration in which the curve lies completely inside the zone (see Fig. 8).

To find the external boundary we process the edges of the zone in order, and for each edge compute the possible intersection with the curve. This amounts to solving a cubic equation. Note that the

crossing points of the curve in and out of the zone partition its perimeter into two complementary boundaries. We choose the shorter one to be the external boundary. Though this is not theoretically guaranteed, we found experimentally that this choice was always correct. The cost is the sum of lengths of these segments.

#### 4.2.2 Curvature Control

Another important constraint we impose on our path is that it has small curvature everywhere, as much as possible.

The cubic splines we use,  $\gamma(t)$ , have a non-arc length parameterization (this is true for most polynomial curves). Its curvature  $\kappa$  is given by

$$\kappa(t) = \frac{\|\dot{\gamma}(t) \times \ddot{\gamma}(t)\|}{\|\dot{\gamma}(t)\|^3}$$

The maximum of this function cannot be expressed analytically, so we compute it numerically by sampling the curve. The higher the curvature is (for example, at the vicinity of a small peak), the denser the sampling should be to capture it. Conversely, a straight line (having  $\kappa = 0$ ) naturally needs only two sampling points. We track the maximum curvature of each curve and the cost is the sum of the curvature maxima of all the curves of the path.

#### 4.2.3 Curve Length

Another important feature which we wish to consider is the total length of the path. We want the path to be as short as possible, so that the camera will not need to travel far, nor to rotate much. In order to do that, we sum the lengths of all the curves along the path.

The integral representing the curve length is approximated numerically using a recursive-adaptive Simpson quadrature. The total sum over all curves of the path is then accumulated.

### 4.3 Minimizing the Cost Function

We use a steepest descent procedure to find a path that minimizes the cost function. In each iteration we locally perturb each control point, one at a time. The order of perturbing points is random. A set of candidate control points are checked in a circular neighborhood of the current point  $p_i$ , excluding those lying outside the common zones. The path and cost function is evaluated for all candidates. If the cost of the best path is better (lower) than that of the original path, then the respective point replaces the point  $p_i$ .

Updating the control points in a sequential order may cause the procedure to get stuck in a local minimum. This is evident when it generates an asymmetric solution for a symmetric input polygons. A random permutation of point updates results in a better result, converging to a symmetric path.

### 4.4 Scaling

We encountered the problem of considering simultaneously the  $x$ ,  $y$ , and  $\alpha$  domains, arising from mixing two incompatible types of dimensions. The axes  $x$  and  $y$  represent coordinates in the spatial domain of the polygon, whereas  $\alpha$  represents angles. We need to calibrate the effects of the different dimensions in order to control them. For example, we may amplify the effect of the angular dimension with respect to that of the planar dimensions by applying a larger scaling factor on the  $\alpha$  axis.

### 4.5 Edge Affects Curve Length

An unattended issue so far is the problem that the polygon geometry does not directly affect the path generated. The curve between two consecutive control points in the  $xy\alpha$  space is parameterized regardless of the size, distance, and orientation of the edge that lies in between the two corresponding vertices. Here is a representative scenario: Consider two far-apart consecutive control points viewing an edge (for example the exterior of a room), and another two nearby consecutive control points seeing a similar sized edge (the interior of the room), and such that the angular difference between the two pairs is similar. Since we parameterize uniformly in  $xy\alpha$  space, the curve between the first

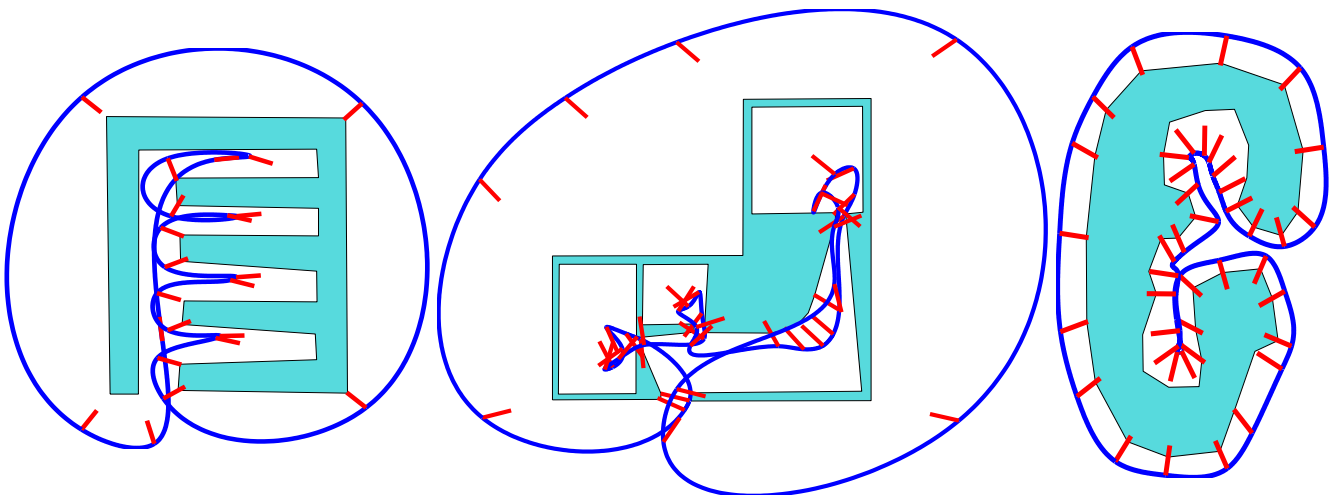


Figure 9: Some sample camera paths (in blue) generated by our algorithm. The red line segments are the camera view directions.

pair will be sampled much more times than the curve between the latter pair, making the angular velocity of the second pair much larger.

We solve this by computing the length of the curve to be a combination of the Euclidean distance between the control points, and the length of the edge. This solves the problem since, although the control points are not moved, their tangents are affected by the artificial length, which also affect the parametrization of the path.

## 5. EXPERIMENTAL RESULTS

The three-dimensional path that we obtain is smooth and the visualization of the polygon when traveling the path is visually pleasing. The user has several degrees of freedom. The first choice is the view angle  $\theta$  and offset  $d$ . Adjusting these parameters influences the path directly, as these define the regions where the path must pass. The second degree of freedom is the cost function with its different weights, and the relative  $xy\alpha$  axis scaling factors. In our implementation, we set  $W_{\text{outOfZone}}$  four times the magnitude of the other weights, to ensure that the path does not leave the zones. The other three weights were given varying values, and modifying them affected the path accordingly. The axis scaling factors used were (1,1,1.5), giving a slight higher importance to the angular axis. This gave us the desired effect of scanning the interior of a model slower.

Some static path results are depicted in Fig. 9, including the ones featuring in the video accompanying this paper.

Since the algorithm is nondeterministic, several runs of the algorithm are performed to obtain different paths with different costs (corresponding to different local minima of the cost function), and then the best is kept.

### 5.1 The Video

A 5 minute video accompanies this paper. It may be downloaded (in a variety of formats) from <ftp://ftp.cs.technion.ac.il/pub/misc/cggc/public/Dani/Vis2003/>.

The video demonstrates the various stages of our algorithm and shows the resulting animation on two examples: The exterior of a simple polygon, and that of a furnished house. These animations demonstrate the superior quality of the camera paths, which were generated automatically.

### 5.2 Implementation Details

We implemented the procedures described here in the Matlab [6] environment. This environment has a large variety of built-in tools, such as polygon boolean operations, numerical solvers, 2D and 3D graphics capabilities, which allowed a relatively easy implementation.

Our implementation has not yet been optimized for speed, and MATLAB is considerably slower than C++ due to its interpretative nature. For simple polygons of about 20 edges, on our 800MHz Pentium III, a typical path is found after 10 iterations in 12 minutes. There are numerous steps in which the performance of our algorithm could be boosted, such as using a faster interpolating scheme (rather than natural cubic splines), faster MinMax solvers (rather than naive steepest descent), and more efficient line-sweep algorithms to manipulate the planar maps.

The computational complexity of our method is rather high. A full analysis of the complexity of our algorithm is omitted from this version of the paper.

## 5.3 Applications

### Overall inspection

The basic application of our technique is for inspecting a two-dimensional model and understanding its overall appearance. This includes virtual situations, such as the examination of a futuristic building (exterior and interior), as well as real-life situations like navigating in a complex maze. In this scenario we are not interested so much in the exact details of the model's surface, but rather in the general idea of its geometry.

Since our goal is to step back from the model as much as possible to receive an overview of it, we specify a small view angle parameter (allowing for distant shots) and a large offset parameter. This results in big far-away zones, that allow distant picture shots, and the paths thus generated result in the desired viewing clips.

### Guaranteed resolution

Surveillance or close visual inspection of a wall is another natural application of our algorithm. Envision a prison guard who needs to check the integrity of a wired and fenced prison; a border patrol checking the border dirt path for foreign footprints; or an automatic robot probe scanning the integrity of a nuclear reactor wall. All these problems are basically the same: the problem of inspecting or scanning a 2D object, where it is most important to maintain a minimal degree of visual resolution. Our algorithm provides a lower bound on the resolution of the scanned edge.

## 6. SUMMARY AND FUTURE WORK

We have completely automated the task of filming the exterior of an arbitrarily planar shape. We show how to create a closed camera trajectory for filming the outside of a 2D polygon to a pleasing level. The user need only modify the two viewing parameters, the cost function weights, and/or the scaling factors.

We see several different future avenues for our work. In architecture-based applications and animations, more focus should be placed on visualizing the complete structure (less than of each edge) and create a proper walking or flying sensation, for example by looking in the direction of the camera movement, thus always moving forward rather than sideways panning. For robotic inspection, we should possibly focus on guaranteeing details and resolution on the scanned surface, maintaining a collision-free path.

This work addresses the problem of a single input polygon. In real-world applications, multiple polygons are present in the scene. This creates several nontrivial occlusion situations, and, among other complications, requires our data structure to support polygons with holes.

## References

- [1] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf. Computational Geometry: Algorithms and Applications (2nd Rev. Ed). Springer-Verlag Berlin Heidelberg 2000.
- [2] N. Halper, R. Helbing, T. Strothotte. A camera engine for computer games: Managing the trade-off between constraint satisfaction and frame coherence. Computer Graphics Forum: Proc. Of Eurographics 2001, 20(3):174-183, September 2001.
- [3] L. He, M. F. Cohen, D. H. Salesin. The Virtual Cinematographer: A paradigm for automatic real-time camera control and directing. Computer Graphics (Proc. of ACM SIGGRAPH 96), (30):217-224, August 1996.
- [4] S. Hutchinson, G. D. Hager, P. I. Corke. A tutorial on visual servo control. IEEE Trans. on Robotics and Automation, 12(5):651-670, October 1996.
- [5] É. Marchand, N. Courty. Controlling a camera in a virtual environment. The Visual Computer, 18(1):1-19, February 2002.
- [6] The MathWorks, Inc. MATLAB, The language of technical computing, Version 6.5.0.180913a (R13).
- [7] M. M. Stark, E. Cohen, T. Lyche, R. F. Riesenfeld. Computing exact shadow irradiance using splines. Computer Graphics (Proc. of ACM SIGGRAPH 99), pp. 155-164, July 1999.
- [8] S. L. Stoev, W. Strasser. A case study on automatic camera placement and motion for visualizing historical data. Proc. IEEE Visualization, 545-548, 2002.
- [9] S. J. Teller. Computing the antipenumbra of an area light source. Computer Graphics (Proc. of ACM SIGGRAPH 92), 26(2):139-148, July 1992.
- [10] B. Triggs, C. Laugier. Automatic task planning for robot vision. Proc. of the 7th Int. Symp. on Robotics Research, 428-439, October 1995.
- [11] S. K. Wismath. Computing the full visibility graph of a set of line segments. Information Proc. Letters, 42(5):257-261, 1992.