A Heuristic Method for Generating 2D CSG Trees from Bitmaps

Sarah F. Gibson
MERL
201 Broadway
Cambridge, MA 02139
U.S.A.
617-621-7525
gibson@merl.com

Joe Marks
MERL
201 Broadway
Cambridge, MA 02139
U.S.A.
617-621-7534
marks@merl.com

Danielle Feinberg
Pixar Animation Studios
1001 W. Cutting Blvd.
Richmond, CA 94804
U.S.A.
510-215-3532
danielle@pixar.com

Manuel Sosa
Dept. of Mech. Eng.
MIT
Cambridge, MA 02139
U.S.A.
617-225-6704
msosa@mit.edu

Abstract

This paper presents a method for generating compact and effective constructive-solid-geometry (CSG) object representations from 2D bitmap representations. The method consists of two parts: a stochastic search procedure for finding candidate CSG trees and a local optimization procedure that modifies the primitives of a given CSG tree to effect a better match. Results for several sample input bitmaps are presented and an application of this method to automatic geometric morphing between pairs of bitmaps is shown.

Key Words: Constructive solid geometry, conversion, optimization, geometric morphing.

Introduction

Motivation

The generation of object models for graphics and animation applications typically requires a major investment of resources. While CAD tools for building geometric models exist, developing models of even simple objects often requires sophisticated design skills and a significant amount of time. On the other hand, new scanning technologies, including laser range scanners and industrial CT scanners, can produce "point-cloud" and volumetric models of existing objects reasonably quickly and inexpensively. Methods that convert such scanned data into structured representations that can be used by standard design and manufacturing tools are needed in order to fully exploit these new scanning technologies.

While methods for converting point-cloud or volumetric data into compact or structured surface models exist (*e.g.*, [4, 5, 9]), surface models do not support all modeling and editing tasks equally well. Some tasks are more appropriately handled by true solid modeling, as embodied in constructive-solid-geometry (CSG). However, there are no existing practical techniques that convert scanned data to CSG models.

Shapiro and Vossler [13, 14, 15] have addressed a related problem. They present a practical system that converts limited boundary representations into CSG models.[1] Their method uses a greedy heuristic to construct a compact CSG representation from a given set of half-spaces, those bounded by lines and circular arcs. However, their system does not take volumetric or scanned data as input: it requires that the candidate half-spaces be provided by the user. Furthermore, it is not clear how their method can be extended to incorporate additional geometric primitives. Other related problems have been addressed in computer vision [2, 6, 16] but they are not readily applied to the general scanned-data-to-CSG conversion problem.

New contributions

This paper demonstrates the feasibility of the direct conversion from scanned data to an effective CSG model by solving the 2D version of the problem: given a scanned 2D target bitmap, generate CSG trees that are compact, easily modifiable, and model the target shape well.

Our algorithm consists of two cooperating processes: 1) A global search procedure that looks in the space of all possible CSG trees for structures that match the target bitmap well; and 2) A local optimization procedure that modifies the primitives of a given CSG tree to provide a better match to the bitmap.

The major contributions of this work are:

- *A complete solution.* Our system takes raw

[1]Less practical, but theoretically interesting, is the algorithm of Dobkin et al. [3].

scanned data — a bitmap — as input, and produces a compact CSG model using rectangular and circular primitives.

- *A flexible measure of model goodness.* Our evaluation metric for candidate CSG trees provides a quantitative measure of how well a tree models a target bitmap. Furthermore, our use of a stochastic search strategy permits this measure to be flexible; in principle it can be modified easily to reflect many factors including model accuracy, compactness, locality (proximity of primitives in the CSG tree should reflect proximity in the object), and appropriateness for design purposes.

- *Variety of solutions.* Unlike constructive heuristics, the proposed stochastic search strategy is capable of finding a variety of good solutions, some of which may be more useful than others for particular editing tasks.

- *Addition of new primitives.* Any shape that can be efficiently scan-converted can be added to the set of primitives elements at the cost of a larger search space.

In addition, as an example application of this work we show how bitmap-to-CSG conversion can be used to morph automatically between pairs of bitmaps.

Technical Approach

Global search

The global search strategy is outlined in Figure 1. The search begins with a set of random candidate solutions; hill climbing is performed on the candidate solutions in parallel; and at regular intervals the search is refocused on the more promising solutions by throwing away less promising ones. The CSG primitives consist of rectangles and circles, whose parameters — $\langle (x, y), (w, h), \theta \rangle$, indicating a rectangle's position, size, and orientation, and $\langle (x, y), r \rangle$ indicating a circle's position and radius — are chosen randomly from the following uniform distributions: $x \in [\frac{W}{4}, \frac{3W}{4}], y \in [\frac{H}{4}, \frac{3H}{4}], r \in [1, \frac{\texttt{min}(W,H)}{2}], w \in [1, \frac{W}{2}], h \in [1, \frac{H}{2}], \theta \in [0, 360]$, where W and H are the width and height of the bitmap, respectively. The type of each primitive, *i.e.*, rectangle or circle, is chosen with equal probability. Likewise, CSG operators are chosen with equal probability from the set: difference, intersection, and union. The number of primitives in a tree is determined by the function

```
Initialize $S$ to be a set of random CSG trees,
  each comprising $\texttt{norm}(a, s)$ primitive objects;
Locally optimize and evaluate each tree in $S$;
$i \leftarrow 0$;
$j \leftarrow 0$;
while $i < N$ and $j < B$ do
{
  Perturb, locally optimize, and evaluate
    a randomly selected tree in $S$;
  if perturbed tree is better then
  {
    Replace original tree with perturbed tree;
    $j \leftarrow 0$;
  }
  else $j \leftarrow j + 1$;
  if $i \bmod n = 0$ then
  {
    Sort $S$ by tree score;
    Replace bottom half of $S$ with top half;
  }
  $i \leftarrow i + 1$;
}
Return best tree in $S$;
```

Figure 1: Global search by parallel hill climbing.

$\texttt{norm}(a, s)$, which returns normally distributed random integers with average a and standard deviation s. Values for these and other global search parameters are listed in Table 2.

The key elements of the global search — those that most affect performance — are the perturbation operators used to generate new candidate solutions from existing ones. The operators and their probability of application are summarized in Table 1. The REP operator replaces an existing subtree containing e primitive objects with a new random subtree containing $\texttt{norm}(e, s')$ primitives. The CUT operator deletes a randomly chosen subtree, and the MOD operator changes the type of a randomly chosen primitive from a rectangle to a circle, or vice versa.

ADD is the most complicated operator: half the time it adds a newly generated subtree by unioning it with an existing subtree; otherwise it adds a new subtree by differencing it from an existing subtree. In both cases the new subtree is small, containing $\texttt{norm}(a', s')$ primitives. In contrast to the behavior of the initialization procedure and the REP operator, the ADD operator locates primitives at specific po-

Perturbation	*Probability*
REP: Replace a random subtree with a newly initialized one.	0.3
ADD: Add a targeted subtree at a randomly chosen location.	0.3
CUT: Delete a randomly chosen subtree.	0.3
MOD: Change the type of a randomly selected primitive.	0.1

Table 1: Perturbation operators for global search.

sitions of the target bitmap. These positions are the local maxima or minima (depending on whether they make a net addition or subtraction to the resultant CSG object) of a Euclidean distance map [7] derived from the bitmap. The primitive size parameters — width and height for rectangles, radius for circles — are given by the expression $\mathtt{max}(1, \mathtt{norm}(d, d))$, where d is the value of the Euclidean distance map at the relevant local maximum or minimum and represents the distance to the nearest edge in the bitmap.

Local optimization

Once a CSG tree has been generated, it is passed to the local optimizer, which adjusts the parameters of each primitive to improve the match between the CSG tree and the target bitmap.[2] In order to measure the goodness of this match, primitive elements of the CSG tree are scan-converted and combined using the node operators to generate a bitmap representation of the CSG tree. As illustrated in Figure 2, the mismatch between the target bitmap and the CSG bitmap is represented by the sum of the UNCOVERED region — pixels in the target bitmap that are not covered by the CSG tree — and the EXCESS region — pixels in the scan-converted CSG tree that do not lie in the target bitmap.

[2] This task is similar to the photogrammetric-modeling task of Debevec et al. [1] in which the parameters of a hierarchy of 3D rectangular blocks are adjusted to match photographs of an architectural scene. In their approach the hierarchical blocks model is created by hand; block edges are also associated manually with edges in the photographs. The modeling of architecture from photographs is a good example of an application that could make use of better methods for the automatic construction of hierarchical solid models that match scanned data.

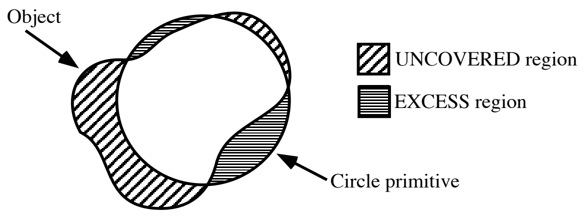

Figure 2: Excess and uncovered regions.

Figure 3 outlines the general approach taken in the local optimizer. Adjustments are made to each primitive in the CSG tree depending on the amount and distribution of the EXCESS and UNCOVERED regions local to the primitive. For example, the position of the primitive is shifted until it lies roughly centered between the local EXCESS and UNCOVERED regions. This is done by shifting the primitive in small steps towards the center of the local UNCOVERED region and away from the center of the local EXCESS region as illustrated in Figure 4a.

The dimensions of the primitive are adjusted until the amount of local EXCESS is roughly equal to the amount of local UNCOVERED. Each dimension of the object (radius for circles, height and width for rectangles) is adjusted independently. As illustrated in Figure 4b, if the amount of local UNCOVERED exceeds the amount of local EXCESS, the primitive dimension is increased. Otherwise if the local EXCESS exceeds the local UNCOVERED, the primitive dimension is reduced.

Adjusting the rotation of the primitive using the local EXCESS and UNCOVERED regions is somewhat more complicated. The general approach involves rotating the primitive until its principal axis is aligned with the principal axis of local pixels of the target bitmap. This could be accomplished by rotating the primitive away from the principal axis of the local EXCESS region and towards the principal axis of the local UNCOVERED region. The principal axis of a region is determined by the principal eigenvector of the region's inertial tensor matrix. This method is valid for both 2D and 3D systems, however it is computationally expensive and can be subject to instability. Hence, in the 2D system, a simpler method is used: the primitive is rotated by small steps in a positive direction while the score improves; if the score does not initially improve, the primitive is rotated in a negative direction.

Once the parameters of each primitive have been

```
$P \leftarrow$ Number of primitives in CSG tree;
Generate a bitmap representation
  of the CSG tree;
Determine EXCESS and UNCOVERED regions;
Calculate the score;
$j \leftarrow 0$;
**while** $j < M$ **and** score is improving **do**
{
  **for** $i \leftarrow 0$ **to** $P - 1$ **do**
  {
    Adjust position of primitive $i$;
    Adjust dimensions of primitive $i$;
    Adjust orientation of primitive $i$;
  }
  Recalculate the score;
  $j \leftarrow j + 1$;
}
Prune the CSG tree;
Return the optimized tree and its score;
```

Figure 3: Local optimization of primitive parameters.

locally optimized, primitives that do not contribute to the final CSG model are pruned from the tree. The tree score is then computed as the sum of the number of EXCESS pixels, the number of UNCOVERED pixels, and a scalar weight, k, times the number of primitives in the CSG tree. (Including the number of primitives in the score promotes compact trees.) This score is then returned to the global search.

Results

Bitmaps and their CSG trees

Figures 5, 8, and 10 contain hand-drawn bitmaps that served as some of our test input. The algorithm described previously was run five times for each bitmap using different random-number seeds. The CSG trees shown below were selected from the five final trees generated for each bitmap. For each tree we report the total number of different CSG trees that were evaluated during its run before arriving at the result shown: this number always consists of 500 initial random trees, plus some variable number of trees produced by application of the perturbation operators.[3]

[3] An alternative measure of computational effort is the number of primitives that were scan-converted. A maximum of 400,000 primitive scan-conversions is performed in the local optimization and evaluation of a 10-primitive CSG tree; the

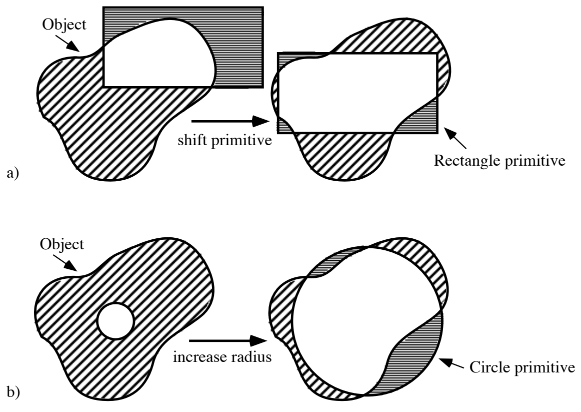

Figure 4: Position and dimension adjustments.

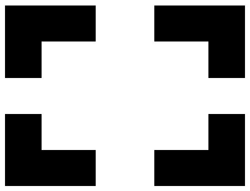

Figure 5: An example bitmap.

Parameter	*Value*
$\|S\|$: Candidate-solution set size.	500
N: Max. # of trees searched.	5000
B: Max. # of trees searched w/o improvement.	500
n: Refocus interval for global search.	50
a: Avg. size of initial tree.	10.0
s: Std. dev. of initial tree size.	2.0
a': Avg. size of added tree.	3.0
s': Std. dev. of added-tree size.	1.5
W, H: Width and height of bitmap.	128
M: Max. # of iterations in local optimizer.	80
k: Tree-size weighting in score.	25

Table 2: Parameter values used for examples shown.

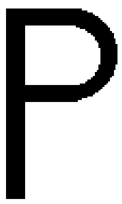

Figure 8: Bitmap for the letter 'P'.

As with many heuristic search methods, this approach requires setting many parameters. The parameter values that were used in the examples reported here were derived by experimentation and are listed in Tables 1 and 2.

Figures 6 and 7 show two CSG trees for the bitmap depicted in Figure 5. The former was found after evaluating 2,096 trees; the latter after evaluating 2,787 trees. The tree in Figure 6 contains a minimal number of primitives and is preferable for editing operations that will preserve the object's symmetry. The tree in Figure 7 contains one more primitive, but is preferable for asymmetric editing. These examples illustrate the utility of an algorithm that is capable of producing a variety of near-optimal results for the scan-and-edit applications we have in mind.

The bitmap in Figure 8 is one of those used for the morphing tests described below. A corresponding CSG tree, the product of 2,824 tree evaluations, is shown in Figure 9. After 400 iterations of the search procedure, the likelihood that each perturbation operator (Table 1) would yield an improvement was fairly high: MOD, 0.49; REP, 0.33; ADD, 0.48; and CUT, 0.26. However, by the time the best tree was found, these numbers had declined significantly: MOD, 0.15; REP, 0.12; ADD, 0.10; and CUT, 0.09. This behavior was typical of the other tests too.

Finally, the bitmap shown in Figure 10 was inspired by the familiar icon on the back of the U.S. $1 bill. The results for this bitmap are shown in Figures 11 and 12. The former required 3,074 CSG-tree evaluations; the latter, 4,485. The two candidate trees further illustrate the diversity of solutions that can be generated by the stochastic search method.

average number is somewhat less. Multiplying the maximum number by the number of trees evaluated gives an upper bound on the number of primitive scan-conversions performed.

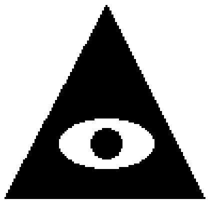

Figure 10: Bitmap for the eye-in-diamond icon.

Application to geometric morphing

To demonstrate the capability and flexibility of the approach described here, the algorithm was modified to perform geometric morphing between pairs of bitmaps. The modifications are straightforward:

- Two target bitmaps are used.
- Each candidate solution consists of two CSG trees that are structurally identical, but that can have different primitive objects.
- Each tree is locally optimized with respect to its corresponding target bitmap, and the score given to a candidate solution is the average score for its two trees.
- Structural identity between the two trees is maintained by having the initial trees be identical copies, and by applying the same REP, ADD, and CUT operations to both trees. However, MOD operations and local optimization are applied independently to each tree, and no pruning is done by the local optimization procedure.

The solution returned by the modified algorithm consists of two trees that are structurally identical, but each of which is a good match for one of the two target bitmaps by virtue of possible differences between the primitive objects in each tree. Morphing is accomplished by straightforward interpolation of the corresponding primitives in both trees.[4]

Sample morphs are illustrated in Figures 13 and 14. Animated versions of these and other sample morphs can be downloaded from our Web site at URL `http://www.merl.com/people/marks-/gi97.html`. The algorithm parameters used were identical to those reported in Table 2, with one exception: in the morphing runs, for which simpler

[4]Interpolating between two orientations (which is relevant for rectangles only) can be done either clockwise or counterclockwise. We chose the directions of rotation for effect.

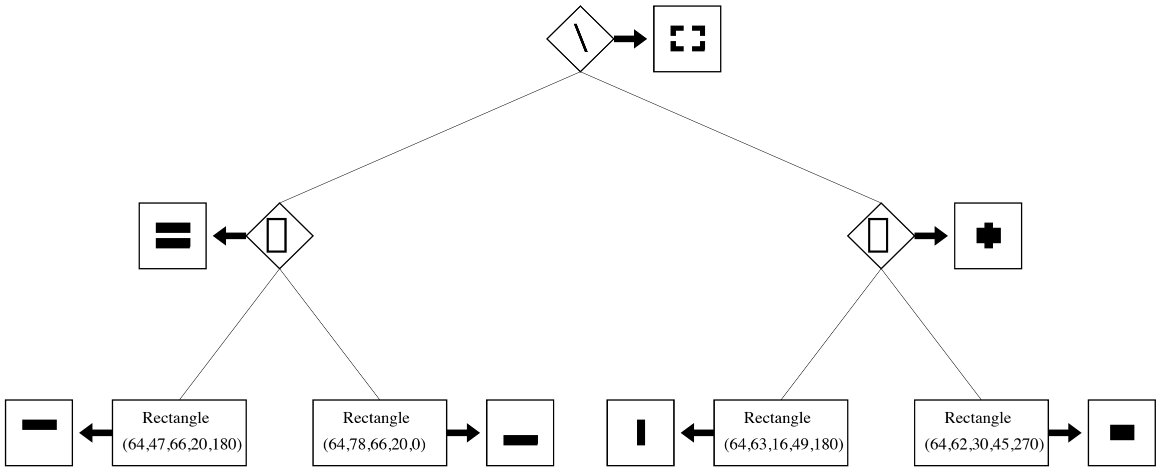
\
Rectangle (64,47,66,20,180)
Rectangle (64,78,66,20,0)
Rectangle (64,63,16,49,180)
Rectangle (64,62,30,45,270)

Figure 6: Automatically generated CSG tree for bitmap from Figure 5.

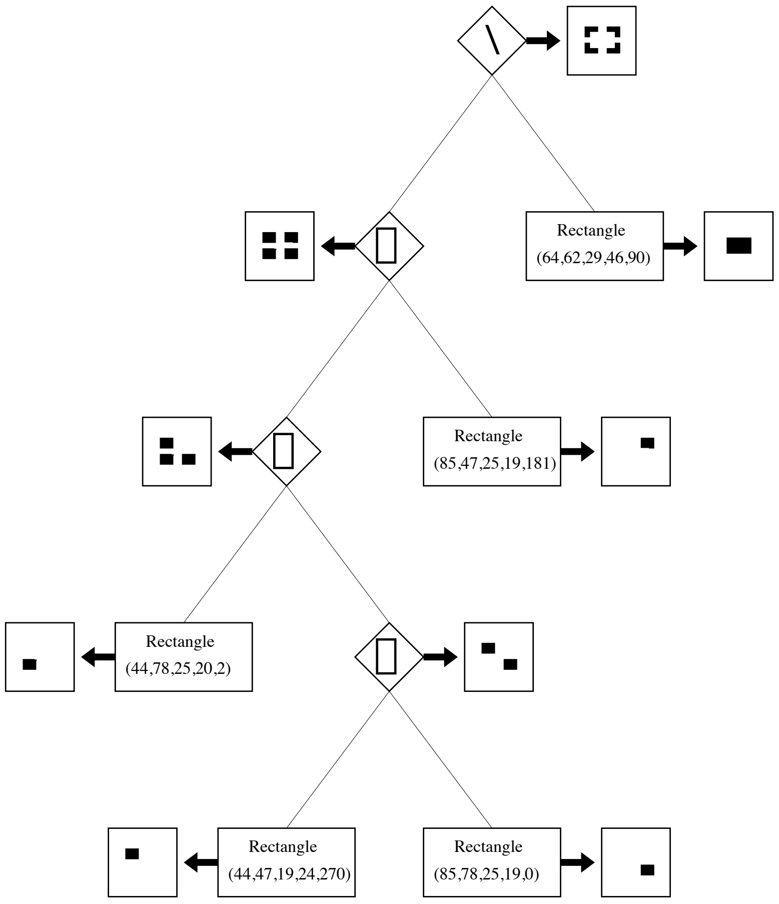
\
Rectangle (64,62,29,46,90)
Rectangle (85,47,25,19,181)
Rectangle (44,78,25,20,2)
Rectangle (44,47,19,24,270)
Rectangle (85,78,25,19,0)

Figure 7: A second CSG tree for bitmap from Figure 5.

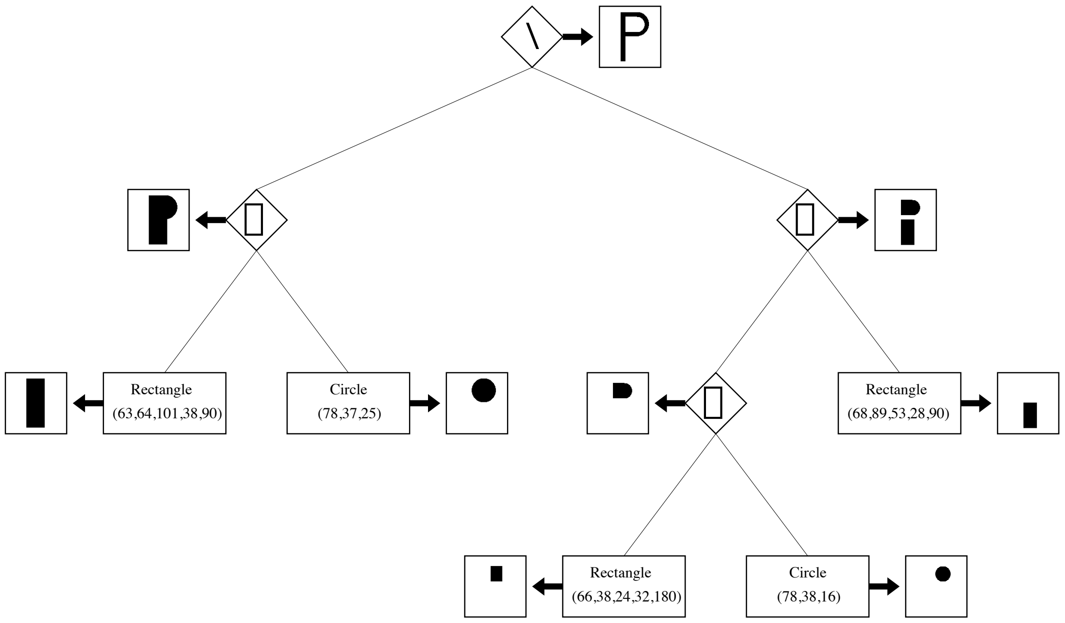

Figure 9: CSG tree for the letter 'P'.

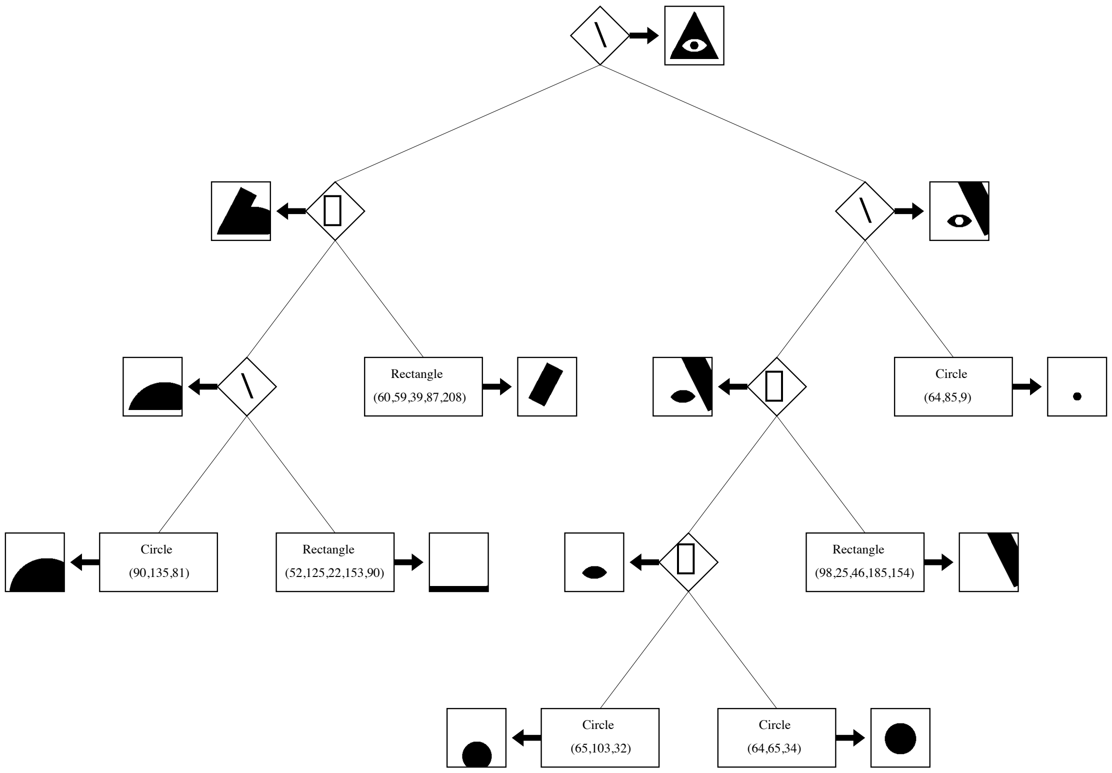

Figure 11: CSG tree for the eye-in-diamond icon.

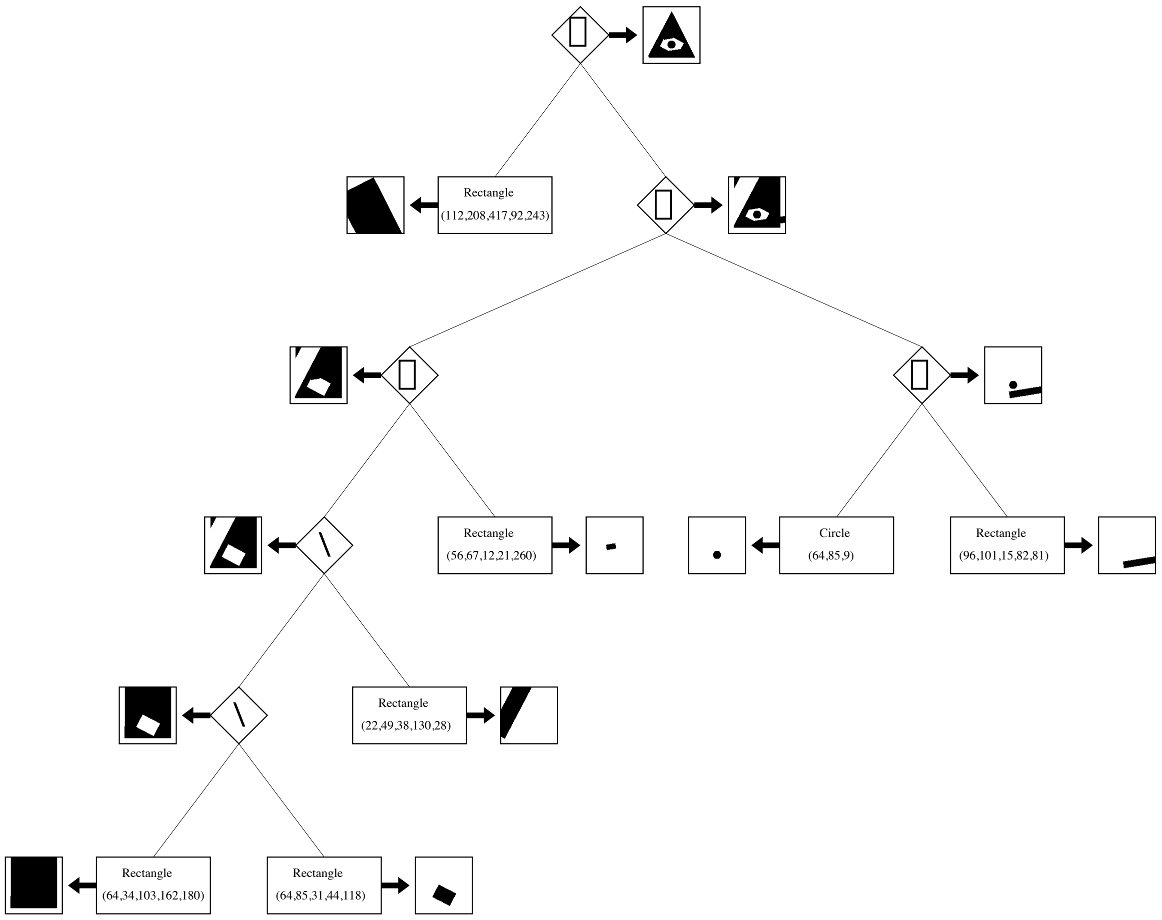

Figure 12: A second CSG tree for the eye-in-diamond icon.

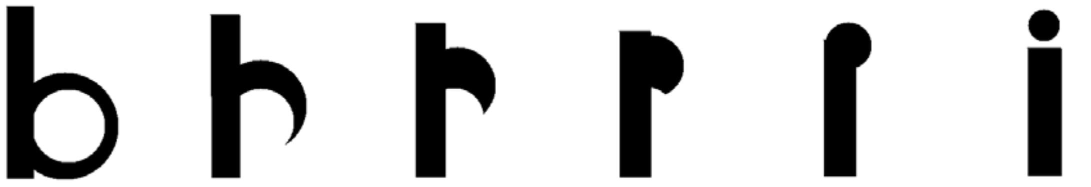

Figure 13: An automatically generated morph between 'b' and 'i'.

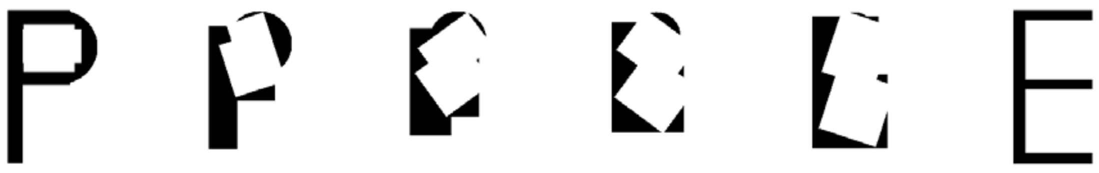

Figure 14: An automatically generated morph between 'P' and 'E'.

shapes with smaller expected tree sizes were used, the average initial tree size, a, was set to 5.0. Each pair of bitmaps shown depicts two topologically different letters. This approach is different from previous work on automatic 2D geometric morphing [10, 11, 12] in both its ability to morph between topologically different shapes and in the style of morphs that it produces.

Conclusions and Future Work

In this paper we have presented a complete system for 2D bitmap-to-CSG conversion. Before attempting to solve the general problem of converting 3D point data to a useful, equivalent CSG representation, a better grasp of the 2D problem is still required. The following extensions to our current approach may be worth investigating:

- *A greater variety of primitives.* One advantage of our system is its potential for accommodating different primitive objects. If a primitive can be scan-converted efficiently, it can be incorporated straightforwardly. For example, triangles and ellipses are obvious candidates for inclusion. However, the added benefit of having extra primitives will be offset somewhat by the increase in search-space dimensionality.

- *Faster scan-conversion.* The dominant computational cost in our current system is due to scan-conversion of primitive shapes. Taking advantage of scan-conversion hardware on high-end graphics workstations will increase the capacity for further experimentation and the ability to explore bigger search spaces.

- *More intelligent searching.* The use of a Euclidean distance map to generate targeted subtrees in the ADD operator caused a significant improvement in performance. Similar improvements might be achieved from additional heuristics for proposing and positioning primitive objects. The analysis of Shapiro and Vossler [13, 14, 15] is a possible source of relevant ideas. However, care must be taken when developing such heuristics: the benefits of a more focused search can be outweighed by a decrease in the variety of solutions found.

- *Different objective criteria.* Compactness alone is not a sufficient criterion for identifying CSG models that are easy to modify and edit. Locality is doubtless an important consideration, and additional criteria may be derived from a study of typical model-editing tasks [8].

Acknowledgments

Thanks to Bill Freeman, Ted Gibson, and Stuart Shieber for helpful discussions, to Ben Bernard for initial exploratory work, and to Denny Bromley for audio-visual assistance.

References

[1] P. E. Debevec, C. J. Taylor, and J. Malik. Modeling and rendering algorithms from photographs: A hybrid geometry- and image-based approach. In *Proc. of SIGGRAPH 96*, pages 11–20, New Orleans, Louisiana, Aug. 1996. ACM SIGGRAPH, New York. In *Computer Graphics* Annual Conference Series, 1996.

[2] S. J. Dickinson, A. P. Pentland, and A. Rosenfeld. 3-D shape recovery using distributed aspect matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2), Feb. 1992.

[3] D. Dobkin, L. Guibas, J. Hershberger, and J. Snoeyink. An efficient algorithm for finding the CSG representation of a simple polygon. In *Proc. of SIGGRAPH 88*, pages 31–40, Atlanta, Georgia, Aug. 1988. ACM SIGGRAPH, New York. In *Computer Graphics* 22(4), Aug. 1988.

[4] H. Hoppe, T. DeRose, T. Duchamp, M. Halstead, H. Jin, J. McDonald, J. Schweitzer, and W. Stuetzle. Piecewise smooth surface reconstruction. In *Proc. of SIGGRAPH 94*, pages 295–302, Orlando, Florida, July 1994. ACM SIGGRAPH, New York. In *Computer Graphics* Annual Conference Series, 1994.

[5] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. In *Proc. of SIGGRAPH 93*, pages 19–26, Anaheim, California, Aug. 1993. ACM SIGGRAPH, New York. In *Computer Graphics* Annual Conference Series, 1993.

[6] Y. G. Leclerc. Constructing simple stable descriptions for image partitioning. *International Journal of Computer Vision*, 3(1):73–102, 1989.

[7] J. C. Russ. *The Image Processing Handbook.* CRC Press, Boca Raton, 1992.

[8] E. Saund and T. P. Moran. Perceptual organization in an interactive sketch editing application. In *Fifth International Conference on Computer Vision*, pages 597–604, Cambridge, Massachusetts, June 1995. IEEE Computer Society Press.

[9] W. J. Schroeder, J. A. Zarge, and W. E. Lorenson. Decimation of triangle meshes. In *Proc. of SIGGRAPH 92*, pages 65–70, Chicago, Illinois, July 1992. ACM SIGGRAPH, New York. In *Computer Graphics* 26(2), July 1992.

[10] T. W. Sederberg, P. Gao, G. Wang, and H. Mu. 2-D shape blending: An intrinsic solution to the vertex path problem. In *Proc. of SIGGRAPH 93*, pages 15–18, Anaheim, California, Aug. 1993. ACM SIGGRAPH, New York. In *Computer Graphics* Annual Conference Series, 1993.

[11] T. W. Sederberg and E. Greenwood. A physically based approach to 2-D shape blending. In *Proc. of SIGGRAPH 92*, pages 25–34, Chicago, Illinois, July 1992. ACM SIGGRAPH, New York. In *Computer Graphics* 26(2), July 1992.

[12] M. Shapira and A. Rappoport. Shape blending using the star-skeleton representation. *IEEE Computer Graphics and Applications*, pages 44–50, Mar. 1995.

[13] V. Shapiro and D. L. Vossler. Construction and optimization of CSG representations. *Computer-Aided Design*, 23(1):4–20, Jan./Feb. 1991.

[14] V. Shapiro and D. L. Vossler. Efficient CSG representation of two-dimensional solids. *Transactions of the ASME: Journal of Mechanical Design*, 113(3):292–305, Sept. 1991.

[15] V. Shapiro and D. L. Vossler. Separation for boundary to CSG conversion. *ACM Transactions on Graphics*, 12(1):35–55, Jan. 1993.

[16] L. R. Williams. Perceptual organization of occluding contours. In *Third International Conference on Computer Vision*, pages 133–137, Osaka, Japan, December 1990. IEEE Computer Society Press.