Stochastic Pruning

Robert L. Cook John Halstead
Pixar Animation Studios

Abstract

Many renderers perform poorly on scenes that contain a lot of detailed geometry. Level-of-detail techniques can alleviate the load on the renderer by creating simplified representations of primitives that are small on the screen. Current methods work well when the detail is due to the complexity of the individual elements, but not when it is due to the large number of elements. In this paper, we introduce a technique for automatically simplifying this latter type of geometric complexity. Some elements are pruned (i.e., eliminated), and the remaining elements are altered to preserve the statistical properties of the scene.

CR Categories: I.3.3 [Picture/Image generation]: Antialiasing—[I.3.7]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

Keywords: Level of detail, procedural models, Monte Carlo.

1 Introduction

Geometry can overwhelm a renderer. Highly detailed scenes can increase memory requirements and degrade performance, even to the point of becoming unrenderable. Fortunately, the amount of detail involved is typically too great to be represented at the resolution of the rendered image. In such cases, renderer performance can be greatly improved by substituting a less detailed version of the scene without perceptibly changing the image. Varying the level of detail processed by the renderer is crucial for efficient rendering of highly complex scenes.

Creating these simplified representations manually [Crow 1982] can be prohibitively labor-intensive, so good automatic simplification techniques are important. Much of the research in this area has focused on simplifying complex surfaces (e.g., [Cohen et al. 1996] and [Lounsbery et al. 1997]), but these methods do not address one of the most important sources of geometric detail: complexity due to large numbers of simple, disconnected elements. For example, we recently had a landscape filled with plants like the one in Figure 1 stretching from the near distance to the far horizon. The scene contained over a hundred million leaves and was unrenderable with RenderMan. Surface simplification techniques won't help in this situation since the elements are already very simple.

Some existing methods can simplify this type of complexity, but they have serious limitations. For some procedural models, the number of elements generated can be adjusted based on screen size (e.g., [Reeves 1983] and [Smith 1984]). This close coupling of level of detail with model generation tends to further complicate already intricate algorithms and does not help with expensive models that are pre-computed because they are expensive to generate (e.g., [Prusinkiewicz et al. 1994]). Image-based, 2-D approximations such as impostors and textured depth meshes are view-dependent (see [Wilson and Manocha 2003] for a good survey and analysis), and using 3-D volumetric representations (e.g., [Andujar et al. 2002]) can significantly degrade the appearance of the object.

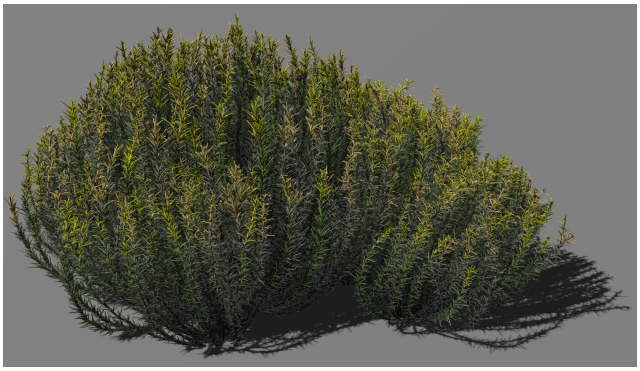
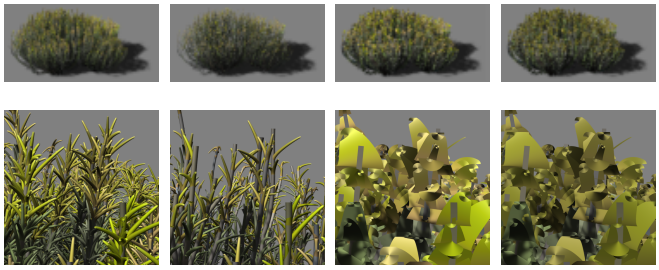


Figure 1: *A plant with 320,000 leaves.*



(a) (b) (c) (d)

Figure 2: *Distant views of the plant from Figure 1 with close-ups below: (a) unpruned, (b) with 90% of its leaves pruned, (c) with area correction, (d) with area and contrast correction.*

2 The Algorithm

In this paper, we introduce *stochastic pruning*, a Monte Carlo technique for automatically simplifying objects made of a large number of geometric elements. When there are a large number of elements in a pixel, we estimate the color of the pixel from a subset of the elements. The unused elements are *pruned*, and the rest are altered to preserve the appearance of the object. The method is easy to implement and fit into a rendering pipeline, and it lets us render scenes of very high geometric complexity without sacrificing image quality.

There are 4 guiding principles needed to make this approach work; we discuss each of these below.

1. **Pruning order.** Deciding which elements to prune.
2. **Area preservation.** Altering the size of the remaining elements so the total area of the object does not change.
3. **Contrast preservation.** Altering the shading of the remaining elements so the contrast of image does not change.
4. **Smooth animation.** Making pruned elements fade out instead of pop off.

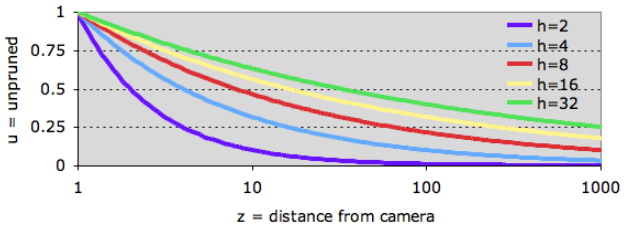


Figure 3: *u as a function of z at different pruning rates.*

2.1 Pruning Order

The farther away an object is, the smaller it is on the screen, the more elements there are per pixel, and the more we can prune. We need to determine u, the fraction of the elements that are *unpruned*, as a function of z, the distance from the camera. There are many ways this could be done. Since the number of elements per pixel is proportional to z^{-2}, we chose to use a similar formula for u:

$$u = z^{-log_h 2} \tag{1}$$

where h is the distance at which half the elements are pruned; this controls how aggressively elements are pruned as they get smaller (see Figure 3). Note that for simplicity we have scaled z so that $z = 1$ where pruning begins; this should be where the shapes of individual elements are no longer discernible, usually when they are about the size of a pixel. As a result, this z scaling will depend on the image resolution.

For animation, the elements must be pruned in a consistent order. This *pruning order* should not be correlated with geometric position, size, orientation, color, or other characteristics (e.g., pruning elements from left to right would be objectionable). Some objects are constructed in such a way that the pruning order can be determined procedurally, but we have found it more general and useful to determine the pruning order stochastically. A simple technique is to assign a random number to each element, then sort the elements by their random numbers. This is usually sufficient in practice, but using stratified sampling ([Cook 1986], [Mitchell 1987]) assures that elements close to each other geometrically are not close to each other in the pruning order. This spreads out the visual effects of pruning during animation and allows more aggressive pruning.

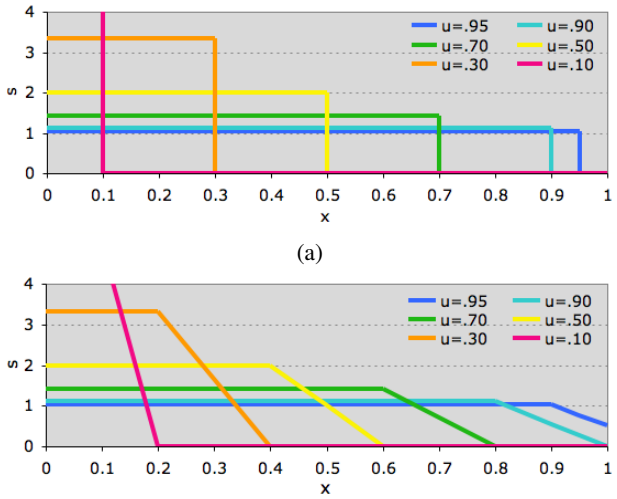
When n, the number of elements in the object, is large, the time spent doing even trivial rejects can be significant, so we need for the pruned elements to not even be considered. This can be done by storing the elements in a file in *reverse pruning order* so that only the first nu elements in the file need to be read at rendering time. This file can be created as a post-process with any model. It works especially well in a film production environment, where the common case is to create sets from randomly scaled and rotated versions of a small number of pre-built plant shapes.

2.2 Area preservation

The total area of the object is na, where a is the average surface area of the individual elements. Pruning decreases the total area to nua; to compensate for this, the area of the unpruned elements should be scaled by an amount $s_{unpruned}$ so that:

$$(nu)(as_{unpruned}) = na \tag{2}$$

Therefore $s_{unpruned} = 1/u$. Figure 4a shows the area scaling factor s as a function of x, the position of the element in the reverse pruning order. s is $1/u$ for unpruned elements ($x < u$) and 0 for pruned elements ($x > u$).



(a)

(b)

Figure 4: *For smaller values of u, more elements are pruned (have 0 area), and the remaining elements are enlarged more. In (a), elements are pruned abruptly; in (b) the pruning is gradual.*

For example, the unpruned plant in Figure 2a is noticeably less dense when 90% of its leaves have been pruned, as shown in Figure 2b. In Figure 2c, we correct for this by making the remaining leaves 10 times larger so that the total area of the plant remains the same. Depending on the type of element, this can be done by scaling in one or two dimensions; in this example, the leaf widths are scaled, as shown in close-up view in Figure 2c. The widening visible in this magnified view is not noticeable in practice because the elements are so small their shapes are not discernible.

2.3 Contrast preservation

From the central limit theorem, we know that sampling more elements per pixel decreases the pixel variance. As a result, pruning an object (i.e., sampling fewer elements) increases its contrast (i.e., higher variance). For example, notice how the pruned plant in Figure 2c has a higher contrast than the unpruned plant in Figure 2a. We now analyze this effect and show how to compensate for it.

The variance of the color of the elements is

$$\sigma^2_{elem} = \sum_{i=1}^{n} (c_i - \bar{c})^2 \tag{3}$$

where c_i is the color of the i^{th} element and $\bar{c}$ is the mean color. When k elements are sampled per pixel, the expected variance of

α	amount of color variance reduction
σ^2	variance
a	element surface area
c	color
h	z at which half the elements are pruned
k	number of elements per pixel
n	number of elements in the object
s	area scaling correction factor
t	size of transition region for fading out pruned elements
u	fraction of elements unpruned
x	position of an element in the reverse pruning order
z	distance from the camera

Figure 5: *Some symbols used in this paper*

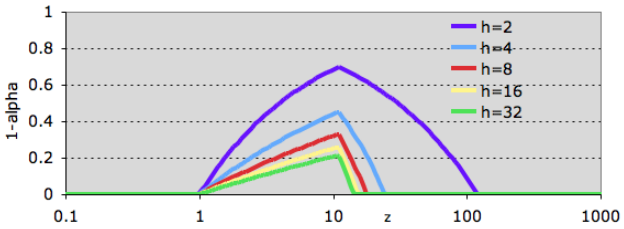


Figure 6: *Contrast correction is more important for aggressive pruning (small h). Parameters:* $k_1 = 1$, $k_{max} = 121$

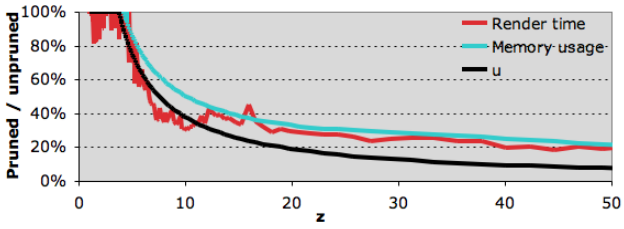


Figure 7: *Ratio of rendering time and memory use with and without pruning as a function of distance for the animation in the supplementary material of the plant in Figure 1 receding into the distance.*

the pixels is related to the variance of the elements by:

$$\sigma^2_{pixel} = \sum_{i=1}^{k} (w_i)^2 \sigma^2_{elem} \tag{4}$$

where the weight w_i is the amount the i^{th} element contributes to the pixel. For this analysis, we can assume that each element contributes equally to the pixel with weight $1/k$:

$$\sigma^2_{pixel} = \sum_{i=1}^{k} (\frac{1}{k})^2 \sigma^2_{elem} = k(\frac{1}{k})^2 \sigma^2_{elem} = \frac{1}{k}\sigma^2_{elem} \tag{5}$$

The pixel variance when the unpruned object is rendered is:

$$\sigma^2_{unpruned} = \sigma^2_{elem}/k_{unpruned} \tag{6}$$

and the pixel variance when the pruned object is rendered is:

$$\sigma^2_{pruned} = \sigma^2_{elem}/k_{pruned} \tag{7}$$

We can make these the same by altering the colors of the pruned elements to bring them closer to the mean:

$$c'_i = \bar{c} + \alpha(c_i - \bar{c}) \tag{8}$$

which reduces the variance of the elements to:

$$\sigma'^2_{elem} = \sum_{i=1}^{n} (c'_i - \bar{c})^2 \tag{9}$$

$$= \sum_{i=1}^{n} (\bar{c} + \alpha(c_i - \bar{c}) - \bar{c})^2 \tag{10}$$

$$= \alpha^2 \sum_{i=1}^{n} (c_i - \bar{c})^2 \tag{11}$$

$$= \alpha^2 \sigma^2_{elem} \tag{12}$$

which in turn reduces the variance of the pixels to:

$$\sigma'^2_{pruned} = \sigma'^2_{elem}/k_{pruned} \tag{13}$$

$$= \sigma^2_{elem}\alpha^2/k_{pruned} \tag{14}$$

$$= \sigma^2_{unpruned}\alpha^2 k_{unpruned}/k_{pruned} \tag{15}$$

We want $\sigma'^2_{pruned} = \sigma^2_{unpruned}$, which is true when

$$\alpha^2 = k_{pruned}/k_{unpruned} \tag{16}$$

An image of a pruned object with these altered element colors will then have the same variance as an image of the unpruned object with the original element colors.

Because screen size varies as $1/z^2$, we would expect $k_{unpruned} = k_1 z^2$, where k_1 is the value of k at $z = 1$, which we can estimate by dividing n by the number of pixels covered by the object when $z = 1$. Similarly, we would expect $k_{pruned} = uk_{unpruned} = uk_1 z^2$, so that $\alpha^2 = u$. Many renderers, however, have a maximum number of visible objects per pixel k_{max} (e.g., 64 if the renderer point-samples 8x8 locations per pixel), and this complicates the formula for α:

$$\alpha^2 = \frac{min(uk_1 z^2, k_{max})}{min(k_1 z^2, k_{max})} \tag{17}$$

Figure 6 shows α as a function of z for values of h. Notice that the contrast only changes in the middle distance. When the object is close, the contrast is unchanged because there is no pruning. When the object is far away, the contrast is unchanged because there are more than k_{max} unpruned elements per pixel, so that k_{max} elements contribute to the pixel in both the pruned and unpruned cases. The maximum contrast difference occurs at $z = \sqrt{k_{max}/k_1}$. The smaller u is at this distance, the larger this maximum will be; contrast correction is thus more important with more aggressive pruning (Figure 6). Figure 2d shows the plant in Figure 2c with contrast correction.

If there are different types of elements in a scene (e.g. leaves and grass), each type needs its own independent contrast correction. $\bar{c}$ should be based on the final shaded colors of the elements, but it can be approximated by reducing the variance of the shader inputs.

2.4 Smooth Animation

As elements are pruned during an animation, they should gradually fade out instead of just abruptly popping off. This can be done by gradually making the elements either more transparent or smaller as they are pruned. The later is shown in Figure 4b, where the size t of the transition region is 0.1. The orange line shows that for a desired pruning level of 70% ($u = .3$), the first 20% of the elements in the reverse pruning order ($x <= u - t = .2$) are enlarged by $1/u = 10/3$ and the last 60% ($x > u + t = .4$) are completely pruned. From x=.2 to x=.4, the areas gradually decrease to 0. As we zoom in and u increases, the elements at $x = .4$ are gradually enlarged, reaching their fully-enlarged size when $u = .5$ (the yellow line). The area under each line is the total pruned surface area and is constant.

3 Results and Conclusions

Figure 7 shows memory usage and rendering time for the plant in Figure 1 as it recedes into the distance (see movie in supplementary material). Figure 8 shows a variety of plants rendered with this

Figure 8: *A gallery of different pruned objects.*

Figure 9: *A scene renderered with pruning that was unrenderable in RenderMan without pruning.*

technique. The scene in Figure 9 contains over one hundred million leaves and requires so much memory that without pruning it is unrenderable with RenderMan.

Stochastic pruning is an automatic, straightforward level-of-detail method that can greatly reduce the geometric complexity of objects with large numbers of simple, disconnected elements. This type of complexity is not effectively addressed by previous methods. The technique fits well into a rendering pipeline, needing no knowledge of how the geometry was generated. It is also easy to implement: just randomly shuffle the elements into a file and use code like that in the Appendix to read just the unpruned elements. We have successfully used this technique in a production environment with a variety of models to render highly complex scenes that we found impossible to render otherwise.

4 Acknowledgments

Omitted for anonymous review.

References

ANDUJAR, C., BRUNET, P., AND AYALA, D. 2002. Topology-reducing surface simplifications using a discrete solid representation. *ACM Transaction on Graphics 21*, 2 (April), 88–105.

COHEN, J., VARSHNEY, A., MANOCHA, D., TURK, G., WEBER, H., AGARWAL, P., BROOKS, F., AND WRIGHT, W. 1996. Simplification envelopes. In *SIGGRAPH '96 Proceedings*, ACM Press, New York, NY, 119–128.

COOK, R. 1986. Stochastic sampling in computer graphics. *ACM Transaction on Graphics 5*, 1 (January), 51–72.

CROW, F. C. 1982. A more flexible image generation environment. In *SIGGRAPH '82 Proceedings*, 9–18.

LOUNSBERY, M., DEROSE, T., AND WARREN, J. 1997. Multiresolution analysis for surfaces of arbitrary topological type. *ACM Transaction on Graphics 16*, 1 (January), 34–73.

MITCHELL, D. 1987. Generating antialiased images at low sampling densities. In *SIGGRAPH '87 Proceedings*, 65–72.

PRUSINKIEWICZ, P., JAMES, M., AND MECH, R. 1994. Synthetic topiary. In *SIGGRAPH '94 Proceedings*, 351–358.

REEVES, B. 1983. Particle systems - a technique for modeling a class of fuzzy objects. *ACM Transaction on Graphics 2*, 2 (April), 91–108.

SMITH, A. 1984. Plants, fractals, and formal languages. In *SIGGRAPH '84 Proceedings*, 1–10.

WILSON, A., AND MANOCHA, D. 2003. Simplifying complex environments using incremental textured depth meshes. In *SIGGRAPH '03 Proceedings*, 678–688.

Appendix. Sample code

```
// This code was written as a compact example, not for speed or generality.
// For example, these are in-memory routines, but in practice the elements
// would be streamed in from a file.
// The "Prune" routine takes an array "eIn" of elements in reverse pruning order
// at distance "z", and returns a pruned array of elements "eOut" to be rendered.
// These graphs illustrate how s and t change near u=0 and u=1:
//
//    (when u-trans<0)                              (when u+trans>1)
//   sMax-> s            ...sss <- sMax              ...sss <- sMax
//          |s                    s                        . s
//          | s                    s                       .   s
//          |  s<t>      or         s<- t ->     or        .<-t->s <- sMin
//          |   s                   . s                     .     |
//          |    s                  .    s                  .     |
// sMin=0-> +-----s- x           ---+------sss...      -----+-----+ x
//          0     u                 u                     u     1
Prune( element *eIn, element *eOut, double z, double k1, int nIn, int *nOut ) {
    double h=3, kmax=121, k=k1*z*z;
    double u = (z<=1) ? 1 : pow(z,-log(2,h));
    double trans = 0.1;                                     // halfsize of transition region
    double t = (u-trans<0) ? u : (u+trans>1) 1-u : trans;   // t is trans adjusted for the ends
    double sMax = (u+trans<1) ? 1/u : 1/(1-t*t/trans);      // max area scaling for this u
    double sMin = ( u + trans < 1 ) ? 0 : sMax*(1.0-t/trans); // min area scaling for this u
    *nOut = (u+t) * nIn;                                    // # unpruned, including transition
    for (i=0; i<*nOut; i++) {
        double x = (i+0.5)/nIn;                             // position in pruning order
        double sLerp = (x<u-t) ? 1 : (x<u+t) ? (u+t-x)/(2*t) : 0;
        double s = sMin + (sMax-sMin) sLerp;                // area scaling for this element
        double alpha = sqrt(min(kmax,k*u)/min(kmax,k));     // contrast correction
        eOut[i] = eIn[i];
        eOut[i]->scaleAreaBy(s);                            // scales area of element
        eOut[i]->scaleContrastBy(alpha);                    // scales contrast of element
    }
}
```