The Visual Computer

Exact antialiasing of textured terrain models

Daniel Cohen-Or

Computer Science Department, School of Mathematical Sciences, Tel-Aviv University, Ramat-Aviv 69978, Israel
e-mail: daniel@math.tau.ac.il

We introduce a fast area-sampling antialiasing technique for textured terrain models. We scan the model in image order, averaging the pixel footprint values. The technique samples all, but only, the visible parts. It improves previous texture mapping methods that ignore self-occluded footprints. It is superior to super-sampling. The image quality is assessed in both spatial and temporal domains. The good precision of the sampling process in the spatial domain provides an alias-free temporal domain. The low computational cost of the rendering technique and its high-quality filtering in the spatio-temporal domain offer a tool for real-time rendering of discrete terrain models.

Key words: Terrian visualization – Visual simulations – Voxel-based modelling – Ray casting – Antialiasing – Area sampling – Supersampling

1 Introduction

Images of continuous scenes that contain arbitrarily high spatial frequencies are sample and represented in a finite set of raster pixels. Due to the discrete nature of the image raster, the image is limited to display only a finite range of frequencies. *Aliasing* is a phenomenon caused by high frequencies sampled at a lower rate. Spatial domain aliasing produces artifacts such as Moire patterns or jagged edges. Spatial aliasing, which can be acceptable in a still image, can cause stronger visual artifacts in a real-time animated sequence of images. An eye-irritating effect caused by spatio-temporal domain aliasing is known as *flickering*. This is due to the display of high frequencies at falsely spatial locations that are not correlated in time, which causes sudden changes in pixel values.

Antialiasing techniques use subpixel accuracy to reduce or eliminate aliasing artifacts. Accurate spatial antialiasing reduces flickering due to the coherency exhibited among the images in the time domain (shown in Sect. 5). Thus, real-time animations of textured models must employ a very accurate antialiasing technique to avoid the disturbing flickering. However, embedding accurate antialiasing into the rendering process traditionally causes a serious performance degradation, which conflicts with the real-time requirement. The antialiasing technique presented in this paper is embedded in an algorithm that renders textured terrains.

The paper is structured as follows. In Sect. 2, a brief review of antialiasing methods of textured models is presented In Sect. 3, previous works on terrain rendering are surveyed and the scheme that is implemented in this work is introduced. The new antialiasing technique is introduced in Sect 4. In Sect. 5, it is shown that, by implementing exact spatial aliasing, the technique reduces temporal aliasing artifacts. The algorithm's cost is analyzed in Sect. 6, and in Sect. 7 we show results from a few test cases. We end with conclusions in Sect. 8.

2 Antialiasing techniques

Antialiasing methods can be regarded as an approximation of the convolution integral between the continuous image I and a filter kernel H at

The Visual Computer (1997) 13:184–198

every output point $S(i,j)$ of the image:

$$S(i,j) = \iint I(i+x, j+y) H(x,y)\, dx\, dy. \quad (1)$$

Supersampling is a common antialiasing technique that approximates this integral. It samples the continuous image at N times the output image resolution. The samples are then low-pass filtered at the Nyquist limit of one cycle every two pixels. In computer graphics images, the spectrum energy does not necessarily fall off with increasing spatial frequencies, and some aliasing may always be introduced. This is due to the fact that the filter is applied only after the image is sampled and not on the continuous space. However, if the samples are positioned stochastically rather than over a regular grid, then the patterned artifacts caused by frequencies above the Nyquist limit are traded for a noise that is less irritating to the human eye (Cook 1986).

Another antialiasing approach applies the filtering prior to sampling. The visible area seen from an output image pixel is convolved with the kernel filter in order to calculate Eq. 1 accurately. However, determining the visibility for each output pixel can be extremely costly (Catmull 1978). Although the results are convolved with a simple box filter whose kernel extent is limited to one pixel, such an exact *area sampling* produces high-quality images (see Sect. 4). Other works have proposed techniques that approximate the subpixel visibility by small discrete masks (Carpenter 1984; Schilling 1991). These techniques operate in object order, thus partial visible subpixel data are accumulated at an image space buffer. Due to the small size (e.g., 4×4) of the masks, the dynamic color resolution is limited.

The convolution operation, as defined by Eq. 1, is a space-invariant filter. Fort practical scenarios, various shapes and sizes of filter kernels have to be applied to each output image pixel. A simple example is a textured plane viewed by a perspective transformation. where the *pixel footprint* gets larger towards the horizon. The pixel footprint is defined as the object-precision visible area seen from the window pixel. For arbitrary complex projection transformations, the pixel footprint shape can have an arbitrary complexity. However, when a perspective viewing transformation is applied, the quadrilateral defined by the four projected points of the pixel extent provides a good approximation of the pixel footprint over texture space. Common space-variant techniques (Watt and Watt 1992), however, do not account for subpixel visibility and convolve the entire footprint including its occluded parts.

The antialiasing technique presented in this paper uses a space-variant filter that is exact in the sense that it computes area sampling rather than supersampling. The pixel footprint is scanned, and only its visible area is filtered (averaged). A multiresolution prefiltered pyramid is employed to reduce the complexity of large footprints.

3 Rendering terrain models

Textured terrain models represented by a polygonal mesh can be rendered by a standard graphics pipeline that supports mip-maps texture mapping in hardware with bit-mask methods for edge antialiasing (Akeley 1993). Mip-maps (Williams 1983) are easily implemented, but are prone to low-quality antialiasing, sine the filter shape is limited to certain squares. Better filtering, implemented in software, was proposed by Arganov and Gotsman (1995) for avisual flythrough.

Digital textured terrains can be rendered backwards by a ray-casting approach (Cohen and Shaked 1993; Coquillart and Gangnet 1984; Musgrave 1991; Paglieroni and Petersen 1994). The image is generated by casting a ray-of-sight, emanating from the viewpoint, through each of the image pixels towards the terrain. The ray traverses above the terrain until it "hits" the terrain surface. The terrain's texture is then sampled and mapped back to the source pixel. If the sampled value is just point sampled and not filtered, then the quality of the output image is very low. Filtering can be applied either before sampling (prefiltering, area sampling) or after (supersampling). Figure 1 shows two images of a terrain. The top image is point sampled, while the bottom image is supersampled at a rate of $N = 8$ (pixels). As can be seen, the supersampled image has a much better quality.

Musgrave (1991) used bilinear interpolation, which yields acceptable quality only when the pixel footprint is rather small. Point sampling of a multiresolution prefiltered texture (Cohen and Shaked 1993) can better deal with variant footprints, but is still not exact enough to avoid

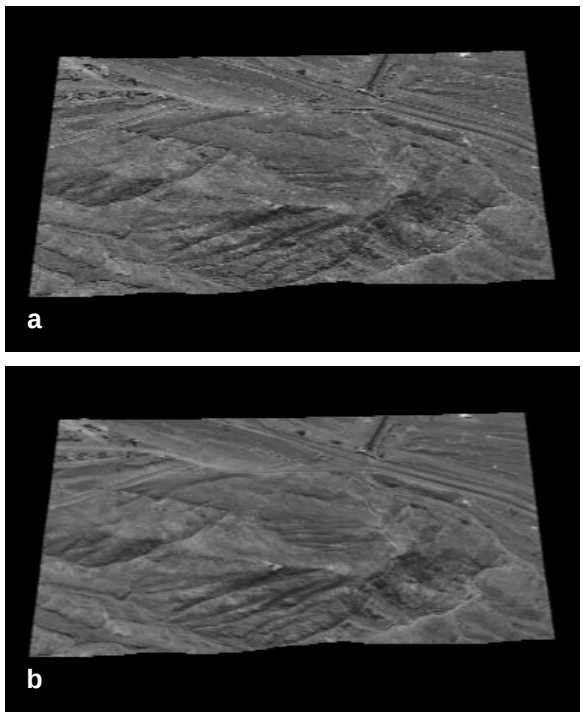


Fig. 1. The quality of an output image using: **a** point sampling; **b** supersampling

flickering. Better quality can be achieved by combining linear interpolation and supersampling (Cohen-Or 1996). In Sect. 7 we compare the quality of supersampling with exact area sampling and show the superiority of the latter.

We adopted the model in which the terrain data are represented by an array of heights. The rays can interpret these 1-D values as 3D parallelograms (known as voxels or sticks) or as thin vertical "walls" emanating from the grid lines. The basic rendering algorithm is a ray-casting algorithm that generates the image in column order (Fig. 2). Since the model is discrete, there is no explicit intersection calculation, but a sequential search for a "hit" between the ray and voxel.

In the technique we employ the method of Cohen-Or (1996); the steps along the ray are performed on the projection of the ray on the plane rather than in the 3D space. The heights along the ray are incrementally and uniformly sampled and compared to the height of the terrain below it until a hit occurs, and the color of the terrain at the hit point is mapped back to the source pixel. If no hit occurs, then the background color of the sky is mapped. Since the terrain is an elevation map, we can assume that the terrain model has no vertical cavities (i.e., a vertical line has only one intersection with the terrain), and thus the traversal can be accelerated using coherence between rays. The basic idea is that as long as the camera does not roll, a ray cast from a vertically adjacent pixel always hits the terrain at a greater distance from the viewpoint than that of the ray below it. When the pitch angle is not equal to zero, the image lane is not perpendicular to the xy-plane, and thus the ray emanating from the upper pixel is a bit slanted, making the implementation of the vertical coherence inexact (Lee and Shin 1995, Wright and Hsieh 1992).

The image pixels are generated column by column, from bottom to top. A ray R_i emanating above the ray R_{i-1} will always traverse distance not shorter than the distance of the ray R_{i-1}. Thus, the ray R_i can start its traversal from a distance equal to the range of the previous hit point of the ray R_{i-1} (Fig. 3). This feature shortens the ray's traversal considerably.

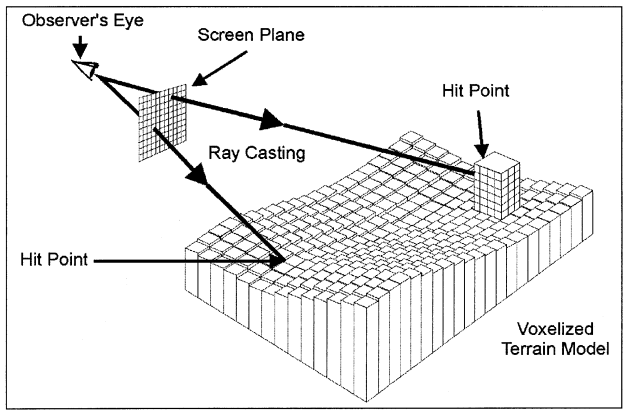
In Fig. 4 we introduce a pseudo-C code for the generation of one image column. The following notation is introduced and illustrated in Fig. 3. Denote the camera position by COP. Let $i-1$ denote the current screen pixel, and P_{i-1}, its position in world coordinates. Assumed that the ray R_{i-1} is x-major, and hits a voxel V_k located at the point x_k on the xy-plane. Denote by $s(R_i)$ and $H(R_i)$, the slope and height of R_i (above the current voxel V_k), respectively. Let $Sample(V_k)$ denote the interpolated color of voxel V_k and its adjacent voxel, and let $H(V_k)$ denote its height. To continue the traversal of the ray R_i from point x_k, the ray's slope is computed by

$$s(R_i) = (P_i.h - COP.h)/(P_i.x - COP.x) \qquad (2)$$

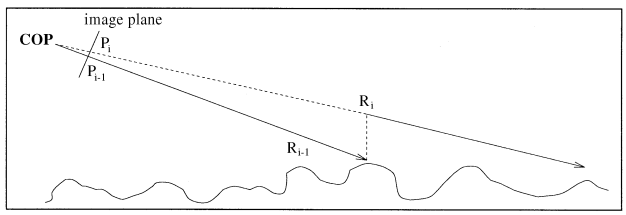
and the height of the ray R_i at point x_k is

$$H(R_i) = P_i.h + (x_k - P_i.x) * s(R_i). \qquad (3)$$

An analysis of the rendering cost has to take in account both the operations performed upon hitting the terrain and the cost of the basic ray traversal.



2



3

Fig. 2. Discrete ray casting of a voxel-based terrain

Fig. 3. Climbing from the hit point of ray R_{i-1} to ray R_i

```
PointSampleOneColumn( )
{
   k = i = 1 ;
   V_k = (H_1, x_1) ; /* set initial voxel */
   R_i = R_1 ; /* current ray emanates from bottom pixel */
   loop { /* visit current voxel */
      if (H(V_k) > H(R_i)) { /* The ray hits voxel V_k */
            Pixel(i) = Sample(V_k) ; /* compute the pixel's color */
            if (i++ > COLUMN_HEIGHT) return ;
            /* compute slope of new ray R_i. */
            s(R_i) = (P_i.h − COP.h)/(P_i.x − COP.x) ;
            /* compute height of ray R_i at current voxel */
            H(R_i) = P_i.h + (x_k − P_i.x) * s(R_i) ;
      }
      else { /* advance to next voxel along the line.*/
         V_k+1 = next(V_k) ;
         H(R_i)+ = s(R_i) ;
         if (++k > MAX_NUM_VOXELS) {
            while (i <= COLUMN_HEIGHT)
               Pixel(i++) = SKY_COLOR ;
            return ;
         }
      }
   }
}
```

4

Fig. 4. The basic point-sampling algorithm for generating one column of pixels

Upon hitting the terrain, the terrain texture is sampled, and the slope and height of the new ray are computed (Eqs. 2 and 3). For each image pixel through which the terrain (and not the sky) is visible, this operation is performed only once.

Advancing along the ray requires a few basic instructions. If standard incremental traversal is implemented, the number of executions of this operation is equal to the number of voxels in the image footprint. This number is huge when the image footprint extends to the horizon. Instead, we implement a multiresolution traversal of the terrain, in which the size of one step along the ray is proportional to the pixel's footprint size. Thus, the rendering cost becomes independent of the size of the image footprint, and the number of steps over the terrain becomes proportional to the image size rather than to the terrain size (Cohen-Or 1996).

4 Exact antialiasing

A prominent concern in rendering an image of a terrain is to reduce aliasing artifacts caused by the inherent point-sampling properties of the ray-casting process. At the same time, we want to perform the antialiasing operation at a reasonable cost, without a significant degradation in the performance of the naive algorithm just shown.

The terrain image usually contains high spatial frequencies that are sampled and represented in the finite set of raster pixels. If the image is not filtered correctly, aliasing artifacts are inevitable. There are two types of aliasing artifacts, caused by the sampling nature of the ray casting: spatial aliasing the temporal aliasing. Temporal aliasing is evident when a sequence of images is generated and displayed at high speed from a flying camera,

so that each image is rendered from a slightly different viewpoint.
To overcome these aliasing problems, we have to improve the sampling of the terrain. In the following, we present three antialiasing techniques and evaluate their effectiveness and cost. We first consider the point sampling and linear interpolation techniques. Then, we introduce an efficient implementation of supersampling and area sampling and show the superiority of area sampling.
We use three tools to evaluate the quality of the sampling process. The first two tools are used to evaluate subpixel image quality, and the third is used to evaluate aliasing in the temporal domain.

1. A "zoom-in" image is an image of the terrain in which magnified voxels are displayed. Such an image is create by viewing the terrain from a very small window, which enables evaluation of the smoothness of an image (Sect. 4.1).
2. An enlarged subpixel image of one pixel, showing all the colors sampled from that pixel. Each color occupies an area proportional to the weight given to it when determining the pixel's color. Such an image serves as a tool for comparing the quality of sampling with supersampling with that of area sampling (Sects. 4.1 and 5).
3. An image of one column of an image over time. It enables one to evaluate the smoothness of the transition between consecutive frames and thus to visualize temporal aliasing in a single image (Sect. 5).

4.1 Point sampling and linear interpolation

Point sampling is implemented by sampling only one voxel per pixel. Linear interpolation can be implemented by sampling the two adjacent voxels hit by the ray and computing a weighted average of the voxel's colors. Let us assume that the ray steps along integer coordinates on the x-axis, and hits the terrain point (x, y). With point sampling, the pixel's color is given by

$$Sample(x, y) = C(x, \lfloor y \rfloor),$$

while with linear interpolation, the weighted average of the adjacent voxel's colors is computed with the formula:

$$Sample(x, y) = C(x, \lfloor y \rfloor) * (\lceil y \rceil - y) + C(x, \lceil y \rceil) * (y - \lfloor y \rfloor). \quad (4)$$

Since multiresolution traversal is implemented, the size of the voxels is determined according to the voxel's distance from the screen. The width of the voxel is equal to that of pixel footprint. Thus, by sampling two adjacent voxels for the pixel, the entire footprint's width is covered. As a consequence, the sampling quality improves considerably. This simple filter reduces noises caused by high spatial frequencies, and the created image has a smooth transition of color along its horizontal lines. This improvement is emphasized in a "zoom-in" image of terrain, in which the voxels are magnified. Figure 5 shows two pairs of such "zoom-in" images of the terrain. The left images are point sampled, while the right ones are filtered by linear interpolation. The filtered images have a smoother transition of colors along their horizontal lines.
Improving the quality of the sampling also helps to reduce temporal aliasing artifacts. The linear interpolation reduces flickering by reducing the number of visible voxels that are not sampled in the naive point-sampling process. This feature is elaborated in Sect. 5.

4.2 Supersampling

Linear interpolation filters with a wider support than a single pixel; however, linear interpolation alone fails to reduce aliasing when the pixel footprint is elongated at low-pitch angles. As a common remedy to the footprint undersampling, the pixel can be supersampled. Vertical supersampling can be implemented by ray casting N equally spaced rays from each pixel, from bottom to top. The pixel's color is the average of the N samples. Denote by R_i the ray emanating from pixel i and by R_{i+1}, the ray emanating from pixel $i + 1$. To simplify the supersampling process, we can assume that rays R_i and R_{i+1} are parallel. Thus, the rays in between them have the same slope (Fig. 6). This assumption enables an efficient implementation of supersampling, without any significant loss of quality.

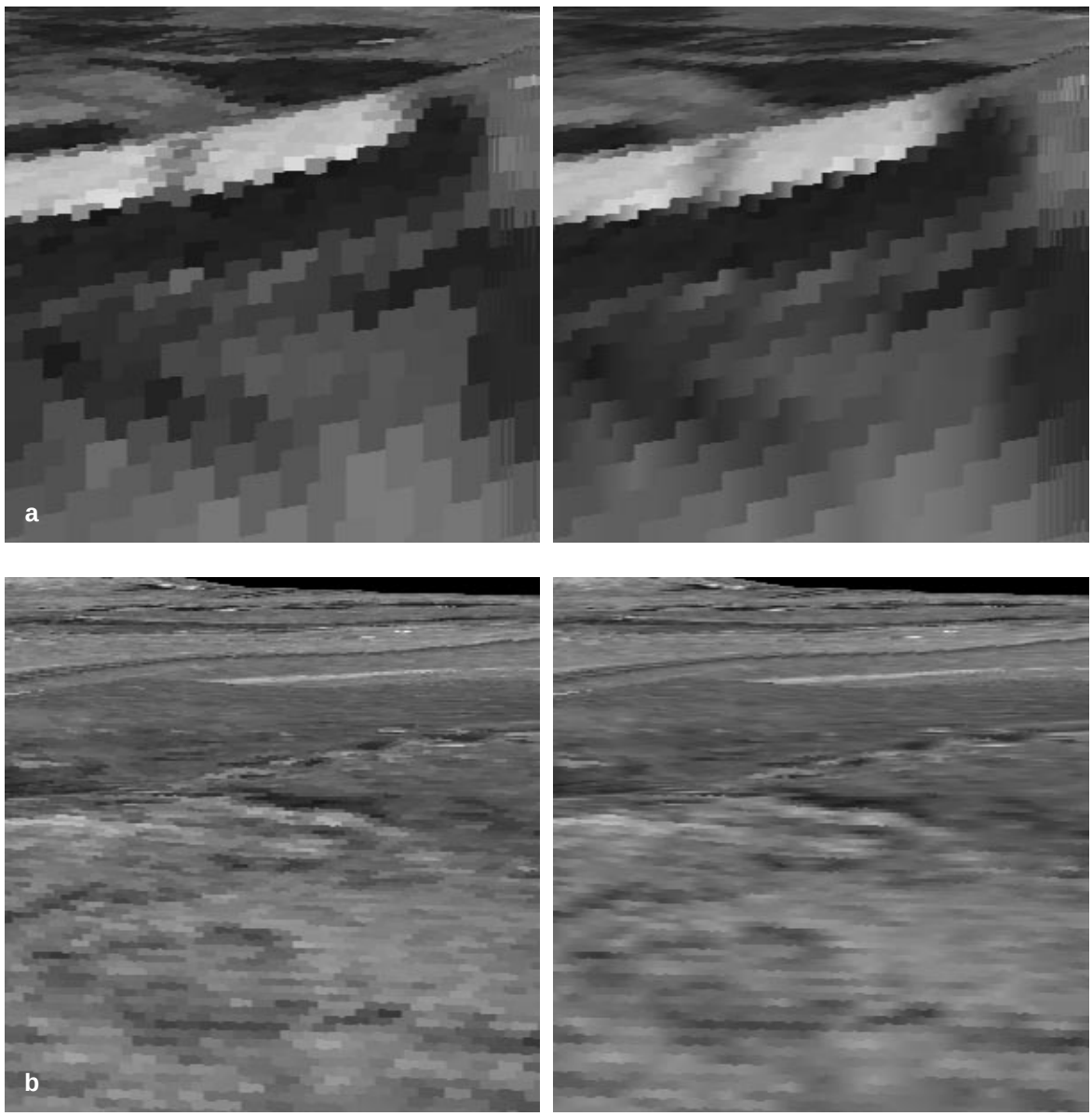
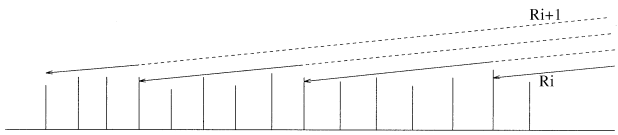
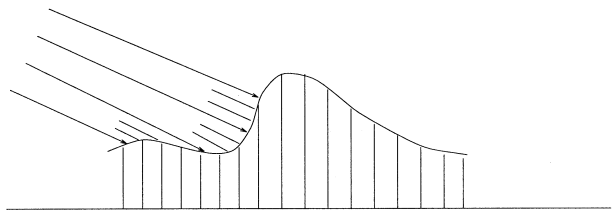


Fig. 5. Linear interpolation versus point sampling. Images generated by point sampling are on the left side, while images created by linear interpolation are on the right side: **a** the camera is very close to the voxels, emphasizing the smoothness of the filtered image; **b** the camera is a little farther away, but still close enough to emphasize the effect of the filtering

The implementation of supersampling is quite similar to the previously described point-sampling algorithm. However, whenever the ray hits the terrain (and it is not the Nth sample of the current pixel), the ray's height is updated by an increment of a constant value that is predetermined for the pixel. The next ray starts its traversal from the new height without recalculation of the ray's slope. In the Nth sample, the ray's height and slope are updated, and the rendering of the next pixel starts. The cost of supersampling remains relatively reasonable, since the algorithm traverses the same set of voxels as in the naive implementation with an addition of only a small



6



7

```
if (H(V_k) > H(R_i)) {
        Σ+ = Sample(V_k) ; /* accumulate the voxel's color */
        if (++j == N) { /* sampling current pixel is done */
                Pixel(i) = Σ >> log(N) ; /* compute the pixel's color */
                if ( ++i > COLUMN_HEIGHT ) return ;
                Σ = j = 0 ;
                /* pre-compute slope of upper ray R_{i+1} */
                s(R_{i+1}) = (P_{i+1}.h − COP.h)/(P_{i+1}.x − COP.x)
                /* pre-compute height of ray R_{i+1} at current voxel */
                H(R_{i+1}) = P_{i+1}.h + (x_k − P_{i+1}.x) * s(R_{i+1}) ;
                /* compute the vertical increment */
                Δ_i = (H(R_{i+1}) − H(R_i)) >> log(N) ;
        }
        else /* current pixel's sampling is not finished*/
                H(R_i)+ = Δ_i ; /* a vertical increment */
}
```

8

Fig. 6. During supersampling we assume that the rays R_i and R_{i+1} are parallel. When the pitch angle is small, the pixel's footprint is very large and undersampled

Fig. 7. When the pitch angle is small, a pixel's footprint can be small at some part, causing redundant voxel oversampling, and undersampling at some other parts

Fig. 8. The implementation on a voxel hit in supersampling

number of operations. A detailed analysis of the rendering cost can be found in Sect. 6.

In Fig. 8, we introduce a pseudo-C code for supersampling one column of pixels. We modify only the code for the operations that are performed upon a hit. The code uses the following notation, in addition to that introduced in the Sect. 3. Denote the supersampling rate by N per, and the index of the samples, by j. Denote the sum of colors for pixel i by Σ, and denote R as the current ray, set to R_i at the beginning of rendering pixel i. In the implementation, the slope and height of ray R_{i+1} are precomputed in the beginning of sampling pixel i. This computation is essential for determining the vertical increment made by hitting a voxel. Suppose the current voxel is $V_k = (H_k, x_k)$ and the sampling of pixel i begins. The difference between the height of ray R_{i+1} and ray R_i above voxel V_k is precalculated to set the vertical increment Δ_i to be the constant $(H(R_{i+1}) - H(R_i))/N$.

As already mentioned, the supersampling technique has its drawbacks. Suppose that we supersample at a predetermined ratio N. When the pitch angle is small, one pixel's footprint can become very large, and more than N samples are needed to determine the pixel's color, as seen in Fig. 6. This yields both spatial and temporal aliasing artifacts. When the terrain is steep, the pixel's footprint may contain only one voxel, making the supersampling of N rays redundant. Figure 7 illustrates occurrences of both oversampling and undersampling of the terrain.

A good implementation of supersampling must be adaptive to the true size of the pixel footprint. However, since the size of a pixel footprint

depends not only on the viewing angle, but also on the terrain topography, it is difficult to implement adaptive supersampling efficiently.
The area sampling implementation introduced in the next section improves the quality of sampling with only a marginal increase in the cost of the algorithm.

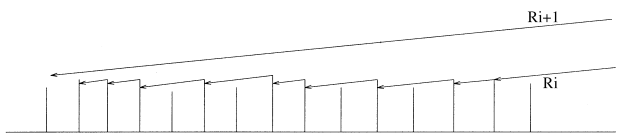
4.3 Area sampling

Area sampling is an antialiasing technique that takes into account all visible areas of the model for each image pixel and filters the colors of these areas. Exact area sampling determines the relative contribution of each part of the model to the pixel color.
Area sampling can be implemented by making a slight modification to the implementation of supersampling. Whenever the ray hits a voxel, the voxel's color is sampled and is given a weight proportional to the area visible from the pixel. Then, the next ray's traversal starts from the top of the voxel (Fig. 10), unlike the supersampling implementation in which the next ray starts from a predetermined value, independent of the voxel context. In this way, all the visible voxels are traced, and an exact weighted average of their colors is computed.
Efficient implementation of supersampling requires predetermining of a supersampling rate N at a reasonable value. As we have seen, this may result in understanding as well as oversampling. The major advantage of area sampling implementation is that all the visible area seen from a pixel is traced and filtered, regardless of the viewing parameters.
Figure 9 shows the footprint of an image rendered by supersampling at a ratio $N = 4$, from a 32×32 window. The figure illustrates the high-quality sampling of area sampling as compared to that of supersampling. Each sampled voxel is colored in blue, while each unsampled voxel is colored in red. The voxels hidden from the viewpoint are colored in green. It should be observed that there are many terrain voxels that are visible from the viewpoint, but are not sampled, while area sampling always covers all the visible voxels in the image footprint.
In Fig. 12, we introduce a pseudo-C code for the operations performed by the area-sampling implementation upon hitting a voxel. The notation used in this code is similar to that used in the code for implementing supersampling. As in the implementation of supersampling, we precompute the slope and height of ray R_{i+1} at the beginning of sampling pixel i. However, this time the height of ray R_{i+1} serves just as the "upper limit" when rendering pixel i, since rays R_{i+1} and R_i bound its footprint. The variables Σ_c and Σ_w hold the sum of colors and the sum of weights, respectively.
Whenever the ray hits a voxel V_k, the visible portion of the voxel is computed by setting its weight W_k to $H' - H(R)$, where H' is the minimum of the voxel's height $H(V_k)$ and the height of ray R_{i+1} at the hit point x_k. The case where $H' = H_k$ is illustrated in Fig. 11. The value of W_k is multiplied by $Sample(V_k)$, the interpolated color of voxel V_k.
In Sect. 7 we show that area-sampling implementation causes only a marginal increase in the cost of the algorithm over supersampling, while the main advantage of the technique is that it samples *all* the visible voxels, and *only* those voxels that are visible.

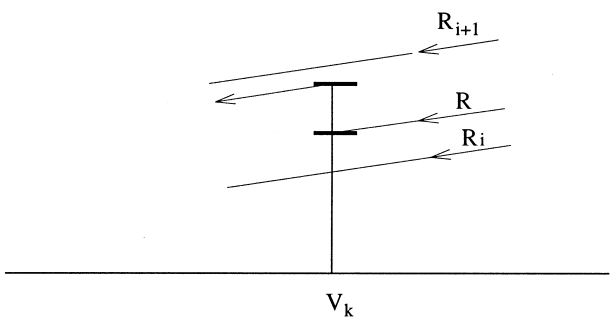
5 Spatiotemporal aliasing

We have seen that when the viewing angle is small, the pixel's footprint may become very long and only a small portion of the visible voxels are actually sampled. This is a typical spatial aliasing scenario, where the high frequencies of the pixel's footprint are undersampled. Some degree of spatial aliasing can be acceptable in a still image. However, the same aliasing can cause stronger visual artifacts in a real-time animated sequence of images. Flickering is a typical temporal aliasing effect. It exhibits itself by sudden changes in the pixel colors that, irritate the eye.
When the footprint is correctly sampled, the pixel color should not change sharply, since consecutive footprints (in time) have a large portion of overlapping. However, if the footprint is undersampled, some visible voxels may contribute to the footprint in one frame and not in the next. Thus, some colors appear and disappear alternatively, which causes the irritating flickering.
In order to investigate the flickering phenomenon closely, an image of an enlarged pixel is produced, visualizing the color components that contribute to the final pixel color (Fig. 13). Each color

9



10



11

```
if (H(V_k) > H(R_i)) {
    H' = MIN(H(V_k), H(R_{i+1})) ;
    W_k = H' − H(R) ; /* compute the weight of voxel V_k */
    Σ_c+ = W_k * Sample(V_k) ;
    Σ_w+ = W_k ;
    if ( H(V_k) >= H(R_{i+1})) { /* current pixel done */
        Pixel(i) = Σ_c/Σ_w ; /* compute pixel's color */
        if ( ++i > COLUMN_HEIGHT ) return ;
        Σ_c = Σ_w = 0 ;
        /* pre-compute slope of upper ray R_{i+1}.
        s(R_{i+1}) = (P_{i+1}.h − COP.h)/(P_{i+1}.x − COP.x) ;
        /* pre-compute height of ray R_{i+1} at current voxel */
        H(R_{i+1}) = P_{i+1}.h + (x_k − P_{i+1}.x) * s(R_{i+1}) ;
    }
    else /*current pixel sampling is not finished*/
    H(R_i) = H(V_k) ; /*start traversing from the voxel's top*/
}
```

12

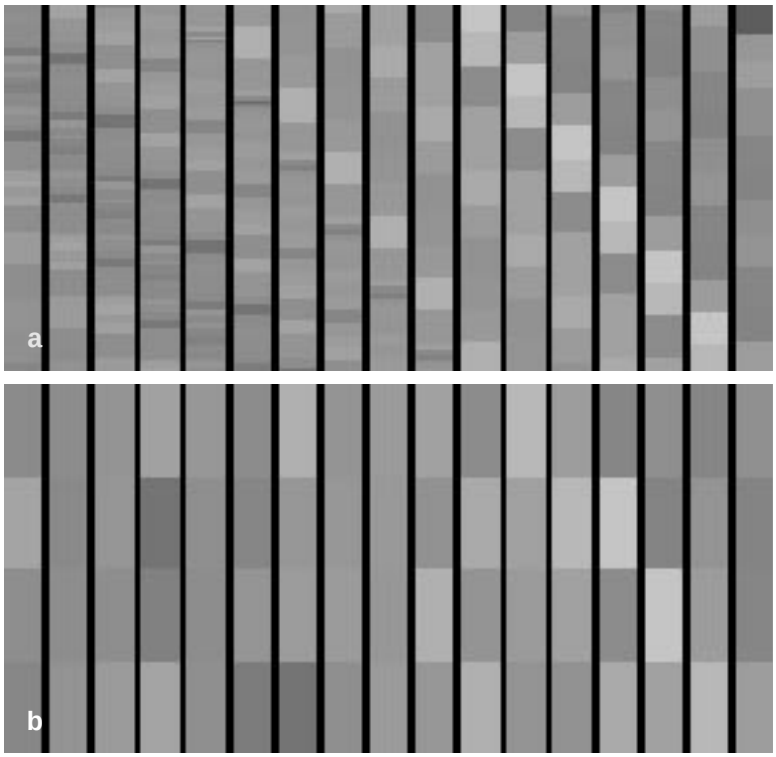
Fig. 9. The footprint of an image rendered from a 32×32 window and supersampled at a ratio of $N = 4$. Sampled voxels are colored in blue, unsampled voxels are colored in red, and hidden voxels are colored in green

Fig. 10. After the ray hits the terrain, the next ray starts its traversal from the top of the voxel, thus all the area visible from the pixel is sampled

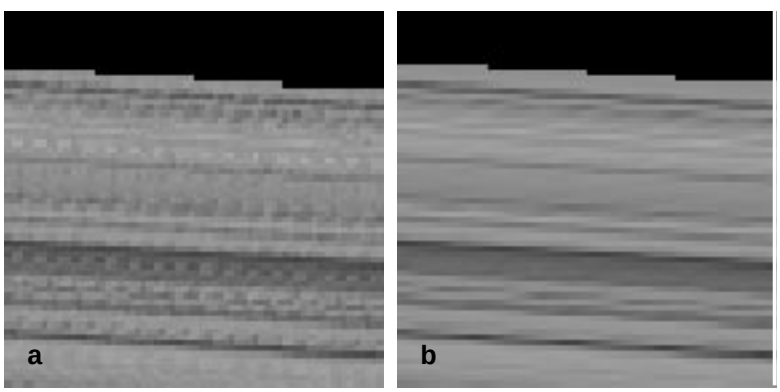
Fig. 11. Computing the weight W_k of voxel V_k by the difference between the voxel's height $H(V_k)$ and the height of the ray at the hit point $H(R)$

Fig. 12. The implementation on a voxel hit in area sampling

component occupies an area proportional to its weight. Note that supersampling uses an unweighted average. In fact, the image of the enlarged pixel reflects the view seen from the pixel window. A sequence of such enlarged pixels shows the changes of the pixel's color components in the time domain. Figure 13 shows such sequences for area sampling and supersampling. Note the smooth changes in the area sampling and compare them to the rough changes in the supersampling image. Obviously, the average of these colors (the pixel color) is much more



13



14

Fig. 13. The view "seen" from one pixel over time, where the horizontal axis is time and the vertical axis is the subpixel vertical coordinate: **a** area sampling; **b** supersampling at a ratio of $N = 4$

Fig. 14. One column over time. Notice the smoothness of transition in the area sampled image (**b**) versus the supersampled image (**a**)

stable, but this behavior affects the amount of flickering.

These sequences show how the footprint of a given pixel changes in time. We can see that, in area sampling, the amount of overlapping is large and whenever new voxels are introduced they do not contribute much, due to the weighted average computation. In supersampling, an unweighted average is computed, and new colors may cause a drastic change in the average color. Thus, good spatial sampling compensates for the simple point sampling in the temporal domain.

In order to quantize the problem, we have generated sequence of column images displayed overtime. In Fig. 14 we can see two images of one column cut in time (a cross section image), one in supersampling and the other in area sampling. The abrupt changes in a pixel color over time are reflected in the supersampling image, while the area sampling technique gives a smooth transition along the horizontal lines.
This shows the relation between the spatial domain and the flickering of the temporal domain. The area sampling technique reduces flickering by implementing an exact spatial antialiasing.

6 Cost analysis

All implementations of the algorithm traverse the same set of voxels in the generation of one image column. Thus, all antialiasing techniques have the same cost for the basic ray traversal. The main differences between the implementations of the algorithm is the number of sampled voxels, the number of ray-voxel hits, and the cost of the operations executed following a hit (note the distinction between a sampled voxel and visible voxel).
The cost for generating one image column in a simple linear interpolation implementation, without performing vertical supersampling, is given by

$$aL + bP + cM,$$

where L is the number of voxels traversed by the ray, and aL is the cost of the basic ray traversal steps. P is the number of voxels from which the terrain is visible, and bP is the cost of computing a ray's slope and the height of the upper ray (Eqs. 2 and 3). M is the number of voxels that are sampled in the generation of one image column, and cM is the cost of the operations performed for each sampled voxel, which includes sampling the two adjacent voxels' colors and performing a linear interpolation between those colors (Eq. 4). Note that in the simple linear interpolation implementation, the equality $M = P$ holds.
In the supersampling implementation, the operations represented by aL and bP have the same cost as in the simple implementation of the algorithm. However, as the supersampling rate N increases, the number of voxels that are sampled in the generation of one image column (M) usually increases as well.
The supersampling implementation introduces an extra overhead with the following operations. For each hit between a ray and a voxel, the voxel's color is added to the sum of colors, and a vertical increment within the current pixel is made for each of its first $N - 1$ samples. Denote the cost of one execution of these two simple operations by d. The number of ray-voxel hits is NP.
Thus, the cost of generating one image column is given by

$$aL + bP + cM + dNP.$$

The cost of supersampling goes up as the supersampling rate N increases, since the number of sampled voxels M usually increases as a result and the value of the term dNP increases linearly in the supersampling rate. However, note that $L > M$, and for small values of N, $L > NP$. Thus, the basic ray traversal still dominates the execution of the algorithm, and increasing the supersampling rate does not cause as severe a degradation in the performance of the algorithm as in common supersampling implementation.
The main difference between area sampling and supersampling implementations is the number of ray-voxel hits in the generation of one image column. In supersampling, this number is NP, while in area sampling there are $M' + P$ ray-voxel hits, where M' is the number voxels sampled in the area sampling algorithm. This is because, whenever the ray hits a voxel, the next ray traversal either starts from the voxel's top or from the ray emanating from the upper pixel. Thus, each voxel is hit only once, except for a transition between pixels. As can be seen in Figs. 8 and 12, the operations performed upon a ray-voxel hit are slightly more expensive in the area-sampling implementation. Denote the cost of these operations by d'.
The cost of generating one image column in area sampling is given by

$$aL + b'P + cM' + d'(M' + P),$$

where each pixel's rendering requires a division operation in area sampling, rather than a less

expensive shift operation in supersampling. Thus $b' > b$.

Since the area sampling implementation detects all the visible voxels, $M' > M$. However, the value of M increases with the supersampling rate N.

The most important factor in comparing the two algorithms is the number of ray-voxel hits. It is NP in supersampling and $M' + P$ in area sampling. These values depend on the viewing parameters and the supersampling rate N. The extra overhead of the area sampling implementation over the naive implementation of the algorithm is not great, since $L > M'$, and thus the basic ray traversal still dominates the algorithms performance. In Sect. 7 it is shown that, for images created with small pitch angle, the supersampling rate N should be increased considerably to avoid temporal aliasing artifacts. In such cases, supersampling is more expensive than area sampling, but still not as accurate.

7 Results

In order to test and master the algorithm's performance, we specify a few viewing parameters and render scenes according to these specifications. If the pitch angle is small, the image footprint can contain a huge number of voxels. Thus, to stimulate a real-life scenario, we create a mirror image of the original DTM by which we generate an infinite terrain. This enables us to specify viewing parameters, so that an image column footprint contains up to 5000 voxels.

We give detailed results of three test cases, using different viewing parameters for each test. Each test case is characterized by the height of the view point and the pitch angle. Table 1 contains attributes of the three tests. The height of the viewing point is denoted by H and the pitch angle is denoted by α. P denotes the average number of pixels in one column through which the terrain is visible. L denotes the average number of voxels in a column footprint (ignoring the voxels traversed until the first ray-terrain hit), and V denotes the average number of visible voxels in one column footprint.

Each of the viewing specifications uses a field of view of 10 degrees and a 400×400 screen resolution. The height of each viewing specification was chosen to fit the width of the pixel footprint at the first terrain hit point to one voxel. Thus, the view-point gets higher as the pitch angle gets larger.

The three test cases are shown in Figs. 15–17. Tables 2–4 summarize values of the parameters needed to compare the different implementations of the algorithm. The column AS denotes area sampling, PS denotes point sampling and SS_i denotes supersampling at a ratio i. The values considered are M, NP, and M/V, which is the ratio between the number of voxels that are sampled and the number of visible voxels. This coverage values serves as an indication of the algorithm's accuracy and of the image quality. Note that the area-sampling technique always samples all the visible voxels (a full coverage), that is, $M = V$.

As already explained in Sect. 6, the dominating factor in determining the algorithm's efficiency is the number of ray-voxel hits. Thus, the value of M at the area sampling column should be added to P and then compared with the value NP in a supersampling column. If the the values are approximately equal, then both implementations have about the same cost.

It can be seen from Table 4 that for an image rendered with a pitch angle of 5 degrees, even a very high supersampling rate $N = 32$, achieves only a relatively small coverage ratio (0.67), while supersampling at the rate $N = 16$ is already as expensive as the area-sampling implementation, but achieves only a small coverage ratio 0.61.

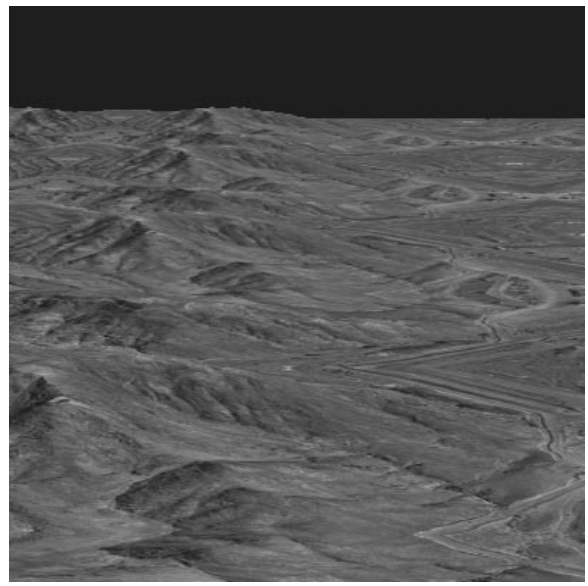
Table 5 summarizes the relation between the supersampling rate and the coverage ratio. It shows the supersampling rate needed for achieving a given coverage ratio.

It can be seen from these examples that, as the pitch angle gets smaller, the area sampling implementation gets more expensive, since it samples all the visible voxels. Supersampling at a low rate

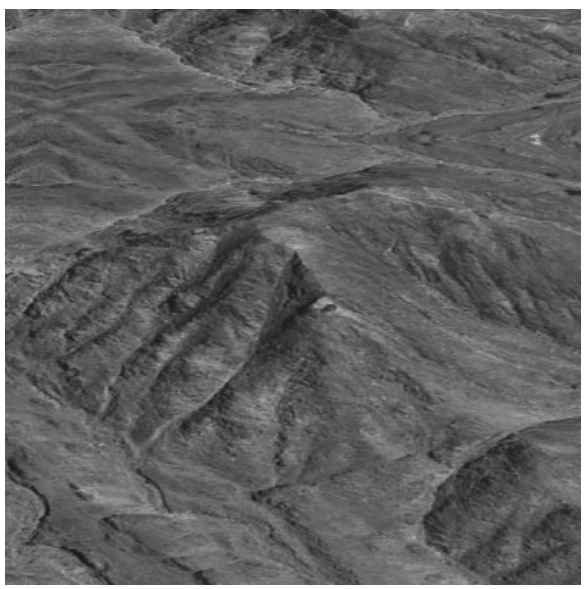
Table 1. The viewing parameters and the associated attributes of the three tests cases

Figure	Height (H)	Pitch (α)	Pixels (P)	Voxels (L)	Visible (V)
15	2800	40	400	847	847
16	1300	20	400	1350	1347
17	1000	5	324	4555	4217

15



17



16

Fig. 15. An image created with a pitch angle of 40 degrees

Fig. 16. An image created with a pitch angle of 20 degrees

Fig. 17. An image created with a pitch angle of 5 degrees

can be more efficient than area sampling, but since the basic ray traversal still dominates the algorithm's cost, the effective cost difference is quite small. As the pitch angle gets smaller, images created with supersampling at a low rate have a low quality due to the low coverage ratio. Increasing the supersampling rate slowly increases the coverage ratio, as seen in Table 4. Thus, the area-sampling implementation becomes more attractive as the pitch angle gets smaller.

In Sect. 5, we claim that, although we sample a simple point in the time domain, an accurate sampling in the spatial domain improves the quality of an animated sequence. Thus, another

Table 2. An analysis of the algorithm's operations when generating an image with a pitch angle of 40 degrees

Parameter	*AS*	*PS*	SS_2	SS_4	SS_3	SS_{16}	SS_{32}
M	847	400	751	844	847	847	847
NP	–	–	800	1600	3200	6400	12 800
M/V	1	0.47	0.88	0.99	1	1	1

Table 3. An analysis of the algorithm's operations when generating an image with a pitch angle of 20 degrees

Parameter	*AS*	*PS*	SS_2	SS_4	SS_3	SS_{16}	SS_{32}
M	1347	400	799	1151	1244	1272	1283
NP	–	–	800	1600	3200	6400	12 800
M/V	1	0.3	0.59	0.85	0.92	0.94	0.94

Table 4. An analysis of the algorithm's operations when generating an image with a pitch angle of 5 degrees

Parameter	*AS*	*PS*	SS_2	SS_4	SS_8	SS_{16}	SS_{32}
M	4271	328	654	1298	2110	2630	2900
NP	–	–	648	1296	2592	5184	10 368
M/V	1	0.07	0.15	0.3	0.49	0.61	0.67

way to analyze the accuracy of the sampling is by rendering a sequence of images from a flying viewpoint, and considering a cut-in-time image of one column. The image smoothness corresponds to the accuracy of the rendering algorithm.

We consider a cut-in-time image created by rendering a sequence of 400 consecutive images. The initial viewing parameters are the same as in Fig. 17. Each consecutive image was created by moving the camera position 0.5 meter horizontally.

The image displayed in Fig. 17 is rendered with a pitch angle of 5 degrees, and the column footprint contains an average number of 4271 voxels. This image serves as a good example for showing the superiority of the area-sampling algorithm over supersampling, by pointing out that, in order to achieve good coverage ratio, the supersampling rate must be huge, as can be seen in Table 5. Supersampling with a rate of $N = 16$ is as costly as area sampling, while the coverage ratio is only 0.61. Such voxel undersampling yields flickering in the time domain.

Table 5. The supersampling rate needed in order to achieve a given coverage ratio (CR)

CR	Figure 15	Figure 16	Figure 17
0.3	1	1	4
0.5	1	2	8
0.7	2	4	32
0.9	2	8	256

Table 6. Changes in pixel colors

Change	*AS*	*PS*	SS_8	SS_{16}
0	32 183	28 903	23 827	23 283
1	14 933	11 287	14 070	18 346
2	1725	912	5614	4985
3	362	708	2842	1721
4	99	659	1368	630
5	46	599	726	232
6–60	52	5432	953	203

A cut-in-time image helps in visualizing significant changes in pixels colors between consecutive images. A slight change made in the camera position should cause a slight change in the pixel's color. In order to measure the accuracy of a temporal antialiasing technique, we measure how many pixels changed their color by a given value. Such a histogram analysis is shown in Table 6. Each entry in that table counts the number of pixels that change their value by the range specified in the left column, accumulating the pixels of one image column, over 400 consecutive frames. The superiority of the area-sampling technique over point sampling is evident from this table. Note that the significant changes that cause the flickering are those of the last row. As the supersampling rate increases, the changes in the pixels colors diminish. For $N = 16$, the changes in the pixels colors are almost as mild as in the area-sampling implementation, but still, the superiority of area sampling is evident, since the number of pixels with a relatively significant change (6–60) is four times larger than in area sampling.

8 Conclusions

In this paper we presented an area-sampling antialiasing technique for a terrain with dense textures. The technique is incorporated in a ray-casting algorithm, and performs an exact area sampling of the terrain model that considers all and only the visible area. All previous antialiasing algorithms for terrain rendering are either inaccurate or too expensive. Existing efficient algorithms for antialiasing of texture mapping do not perform visibility tests, thus causing temporal aliasing.

We introduced an implementation of supersampling and showed that the varying topography of the terrain can result in undersampling for large footprints and oversampling at steep hills, where footprints are short. Area sampling is implemented by making a slight change to the supersampling implementation. For each image pixel, its entire footprint is traced, and a weighted average of its visible voxels is computed. The area-sampling technique treats the voxels along one dimension, and a linear interpolation filters along the orthogonal dimension, together forming a wider support than a single pixel.

We have shown that good precision of sampling in the spatial domain eliminates temporal aliasing artifacts, despite the simple point sampling in the temporal domain. This feature is emphasized and can only be perceived in a real-time animated sequence of images.

We consider this antialiasing technique as exact in the sense that all, and only, the visible portion contributes to the area sampling. There is one reservation to be made, since along the "horizontal" axis the sampling is naive, while the accuracy of the technique is mainly along the rays. However, the assumption is that the rays emanating from a horizontal row of pixels are dense enough; that is, the field of view is small. Recall that we use a hierarchy of voxels so that at a distance the divergent rays sample larger voxels. All the visible area (including near the horizon) is guaranteed to be sampled.

Acknowledgements. The author thanks E. Rich, whose ideals led to many of the results in this paper. He also thanks Y. Nardi, O. Gabbay and G. Tal for implementing the algorithms and producing the images, Y. Shapira of Imatrix for providing the terrain data, A. Solomovici, A. Shaked and E. Carmel for reviewing early drafts of this paper. Figure 2 is by courtesy of Tiltan System Engineering.

References

1. Akeley A (1993) Reality engine graphics. Comput Graph 27:109–116
2. Arganov G, Gotsman C (1995) A parallel system for rendering realistic terrain image sequences. Vis Comput 11:455–464
3. Carpenter L (1984) The A-buffer, an antialiased hidden surface method. Comput Graph 18:103–108
4. Catmull (1978) A hidden-surface algorithm with anti-aliasing. Computer Graphics, (Proceeding of SIGGRAPH '78) 12(3):6–11
5. Cohen D, Shaked A (1993) Photo-realistic imaging of digital terrains. Comput Graph Forum 12:363–373
6. Cohen-Or D, Rich E, Lerner U, Shenker V (1996) A real-time photo-realistic visual flythrough. IEEE Trans Vis Comput Graph 2:255–265
7. Cook RL (1986) Stochastic sampling in computer graphics. ACM Trans Comput Graph 5:51–72
8. Coquillart S, Gangnet M (1984) Shaded display of digital maps. IEEE Comput Graph Appl 4:35–52
9. Lee C, Shin YG (1995) An efficient ray tracing method for terrain rendering. Proceedings of the Third Pacific Conference on Computer Graphics and Applications, Pacific Graphics '95, 180–193, Seoul, Korea, 21–24 August 1995. Shin SY, Kunii TL (eds), World Scientific
10. Musgrave KF (1991) Grid tracing: fast ray tracing for height fields. Technical Report RR-639, Department of Mathematics, Yale University, New Haven, USA
11. Paglieroni DW, Petersen SM (1994) Height distributional distance transform methods for height field ray tracing. ACM Transactions on Graphics 13(4):376–399
12. Schilling A (1991) A new simple and efficient antialiasing withy subpixel masks. Comput Graph 25:133–141
13. Watt A, Watt M (1992) Advanced animation and rendering techniques: theory and practice. Addison-Wesley, New York
14. Williams L (1983) Pyramidal parametrics. Comput Graph 17:1–11
15. Wright J, Hsieh J (1992) A voxel-based, forward projection algorithm for rendering surface and volumetric data. In: Kaufman AE, Nelson GM (eds) Proceedings of Visualization '92, IEEE Computer Society Press, Boston, pp 340–348

DANIEL COHEN-OR is a Senior Lecturer at the Department of Computer Science in Tel-Aviv University. His research interests include rendering techniques, virtual reality, morphing and blending techniques, architectures and algorithms for voxel-based graphics. He received a BSc Cum Laude in both Mathematics and Computer Science (1985), an MSc Cum Laude in Computer Science (1986) from Ben-Gurion University, and a PhD from the Department of Computer Science (1991) at State University of New York at Stony Brook. In 1992–1993, Dr. Cohen-Or designed a real-time flythrough at Tiltan System Engineerng. In 1994–1995, he worked on the development of a new parallel architecture at Terra Computer, and recently he has been working wih MedSim on the development of an ultrasound simulator.