

Real -Time Rendering of Photo-Textured Terrain Height Fields

LUIZ CARLOS GUEDES¹, MARCELO GATTASS², PAULO CEZAR P. CARVALHO^{2,3}

¹ ADDLabs - Departamento de Ciência da Computação, Universidade Federal Fluminense
Praça Valonguinho s/n, Ed. Instituto de Matemática 4o andar, Centro, 24210-130 Niterói, RJ, Brasil
guedes@dcc.uff.br

² TeCGraf - Grupo de Tecnologia em Computação Gráfica, Departamento de Informática, PUC-Rio
Rua Marquês de São Vicente 255, 22453-900 Rio de Janeiro, RJ, Brasil
gattass@tecgraf.puc-rio.br

³ IMPA- Instituto de Matemática Pura e Aplicada
Estrada Dona Castorina, 110, 22460-320 Rio de Janeiro, RJ, Brasil
pcezar@impa.br

Abstract. Interactive tridimensional visualization of terrain models can be found in Geographical Information Systems and computer games. Both applications share the need for high performance algorithms that are tailored to produce textured images in interactive time, say at least 5 frames per second. In this paper we propose and analyze the extension of a well known ray casting algorithm to the case where the view plane is not vertical. Three efficient algorithms are presented and compared. Experimental results are shown and conclusions are made.

1. Introduction

The evolution of personal computers and video game arcades has turned photo-realistic virtual worlds a common requirement in applications such as flight simulators, action games or geographic information systems. Particularly, geographic information systems deal with large scale virtual worlds and require the integration of many sources of information such as digital elevation maps and aerial imagery textures.

Although current graphics architectures suggest that polygonal meshes should be used, spatial requirements of real data is not appropriate for conventional rendering pipelines. Huge texture photographs must be loaded during the rendering process turning the small texture cache of graphics accelerators useless. A high resolution polygonal model would also be necessary in order to fit the required level of detail causing the triangles to be so tiny that most of them would be projected on a single pixel.

Reducing the resolution of the models tend to simplify the terrain in ways that impact the fidelity of the simulation. Most digital elevation models are compiled on a regular space grid, where geographic position is implicit from the position in the array. The enlargement of the step between sampled points would cause undersampling effects producing temporal aliasing and image “pixelization” .

The ray casting approach, on the other hand, is better suited for the task of rendering huge scale environments because texture is placed into the model prior to the exhibition and image driven algorithms avoid the overload of processing different indistinguishable information for each pixel.

The ray casting approach for terrain rendering assumes that the terrain is modeled by a Digital Elevation Map (DEM) and Digital Color Map (DCM). The DEM associates an elevation to each position (x,y) in the terrain and the DCM associates a color value to each position in the terrain. These maps are sampling of a height and color field in a uniform grid. A column of the terrain raised with a height and color taken from the DEM and DCM, respectively, is called a *voxel*. Fig. 1 illustrates many voxels in a terrain model.

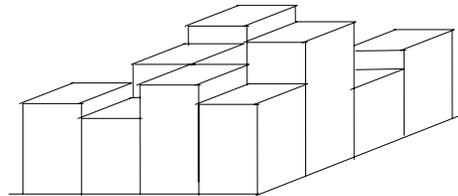


Fig. 1 - Height field model.

In this paper we propose and analyze the extension of a well known ray casting algorithm to the case where the view plane is not vertical. In all cases, we assume that the terrain is represented by a regularly spaced grid of column voxels. Each voxel has a height value, taken from the height field, and a color value, taken from the digital photograph.

Section 2 presents related works. In section 3 the initial basic algorithm for vertical view planes is fully analyzed. Section 4 discusses the difficulties involved in generalizing this algorithm for inclined view planes. In Section 5 the efficient algorithms for this case are

presented and compared. In Section 6 experimental results are shown and conclusions are made.

2. Related Works

Ray Casting algorithms may perform either forward casting or backward casting. Backward casting algorithms are very intuitive in the sense that they cast the rays that reach farther distances before the ones that reach closer distances. Rays are emitted downwards through each column. At each intersection with the ground, the height and the color of that point are taken and a column of pixels (voxel) of that height and with that color is painted on the current column of the screen. Each painted voxel overlaps the previous one and only the visible portion remains on the screen. Fig.2 illustrates the backward approach, darker columns represent the painted voxels.

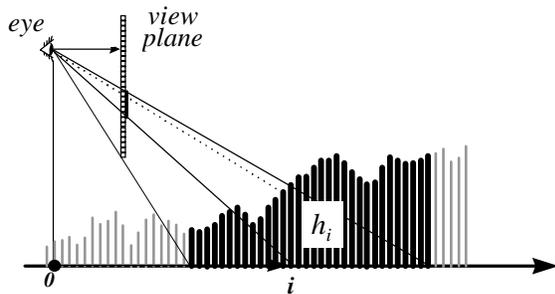


Fig. 2 - Backward ray casting.

In [LaMothe95], the backward approach is presented in a very introductory fashion, trying to give to game programmers a very first insight of real time terrain rendering. His model casts rays only through pixel located at the low quarter of the screen. Those rays intersect the ground and raise columns of voxels that may fill the entire screen.

In the forward casting approach, rays are emitted beginning at the ray that joins the observer to its projection on the ground and moving the destination of the ray farther away on the ground until it intersects the surface of the terrain. When an intersection occurs, the corresponding voxel is climbed up and the pixels of the screen are painted with its color. After the voxel is processed, the next position on the ground is inspected to find whether its corresponding voxel intersects the ray. This approach has the advantage of painting only the visible portion of each voxel. Fig.3 illustrates the forward approach for ray casting.

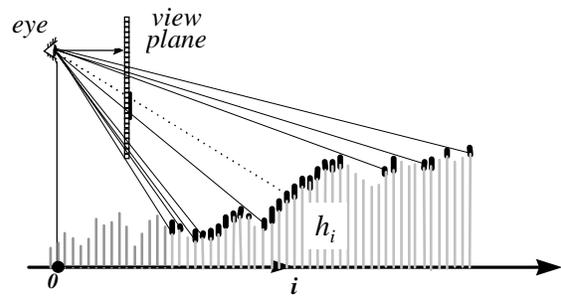


Fig. 3 - Forward ray casting.

In [Freese95] the forward approach is presented in a very optimized version for the case when the view plane is vertical. Many interesting properties based on the coherence among consecutive rays are detected and used in linear incremental procedures that optimize the rendering process. The view plane can only rotate about the Y axis (yaw). Rotation about the X axis (pitch) is not performed; instead, it is faked by moving the observer out of the center of the view plane across the central column. Rotation about the Z axis (roll) is performed by a rotation of the regular image prior to painting the image on the screen.

In [Cohen+95], a parallel version of the forward approach is proposed. His algorithm deals with the inclination of the view plane in a very naïve way. In the next sections we will briefly discuss his approach and present various solution to the deformations caused on the projected image by the inclination of the view plane.

In [Graf+94], a geographic information visualization is proposed that merges three kinds of data (terrain, sky and objects). They work with huge terrain data (~1.5Gb) and its visualization is performed by backward ray casting that is integrated with a 3D object ray-tracing software. No real time navigation is proposed, although predefined tours may be generated for simulation purposes,.

3. The standard algorithm

Height and texture maps are, generally, stored as image files where the value of each pixel at image position (i,j) represents, respectively, the height and the color of a point at coordinates $(i*s_{xy}, j*s_{xy})$ in the horizontal plane. The horizontal scale factor, s_{xy} , is usually implicit in the representation. The value of the height stored in the image is also affected by a scale factor, s_z . The color value is normally represented by a color index for a lookup table which represents the color of that portion of the terrain, including shading effects due to an implicitly assumed illumination.

In the standard algorithm the view plane is always a vertical plane. The eye can move in any direction but can

only rotate around the vertical axis. That is, the *up vector* is always vertical. The view plane window may move with respect to the eye provided its edges remain horizontal or vertical. Actually, the case when the window is rotated about the viewing direction can be solved by applying a suitable rotation in the image produced by the standard algorithm ([Freese95]).

All rays emanating from the eye to each column of pixels in the view plane define a plane, here called sampling plane, which is also vertical, as illustrated by Fig. 4. This plane holds an important property: only the terrain heights that are intersected by the sampling plane are necessary to define the colors of the corresponding column of pixels. That is, a ray casting algorithm does not need to test all column heights to determine the visibility of a given pixel. Only those heights which are intersected by the sampling plane are candidates to the intersection test. The 3D problem is reduced to 2D and the complexity of the intersection test is reduced from $O(m^2)$ to $O(m)$.

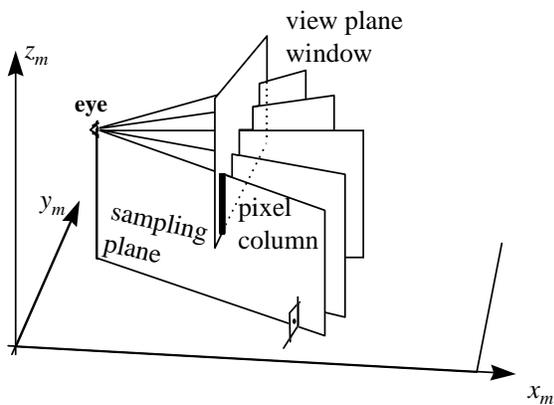


Fig. 4 - Vertical planes for each screen column.

Fig. 5 illustrates the sampling process that occurs for a pixel column k with the observer at the *eye* position looking at the direction *view* also shown in the figure. The intersection of the sampling plane with the column heights results in a non-uniform step function. If the heights are sampled at uniform spaces, as the standard algorithm assumes, the usual aliasing problem arises. When the observer is flying at low altitude, as he moves from one position to another the terrain color boundaries seem to move giving an unpleasant sensation. If the altitude is high, however, the ratio between the size of terrain cells and pixels in the view plane is reduced due to the perspective effect and this problem becomes less noticeable.

The basic idea of the standard forward algorithm is to explore the geometric coherence of the terrain in order to avoid the processing of each ray individually. If the sampling along the *sray* shown in Fig. 5 is uniform, the

rendering problem reduces to forward ray casting as shown in Fig. 3.

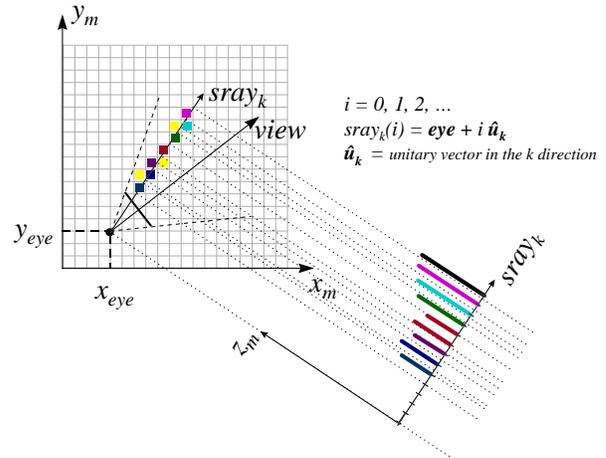


Fig. 5 - Sampling process along vertical planes.

Let us consider in Fig. 6 the first visual ray, of slope m_0 , corresponding to the lower pixel in the column. In order to determine the color to be assigned to this pixel it is necessary to verify, for successive abscissas $i=0, 1, \dots$, if the terrain height, h_i , is greater than the height of the first visual ray, $z_0 = h_{eye} + m_0 i$, at the abscissa i . One should note that if the terrain heights h_i do not intersect the visual ray corresponding to a given pixel j , they cannot intersect the visual ray corresponding to pixels $j+1, j+2, \dots$; thus, it's not necessary to consider again this terrain height for the rest of column k .

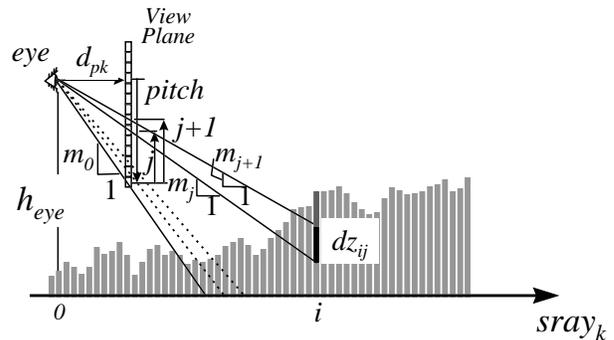


Fig. 6 - Floating horizon (visual rays).

Suppose that a visual ray finally intersects a terrain height at abscissa i , that is, h_i is greater or equal to z_i . The corresponding pixel must be painted and the next visual ray must be considered. This implies in the need to update slope m_j and to repeat the comparison between h_i and z_i . Thus, one or more pixels related to this terrain height are painted until h_i becomes smaller than z_i . When this occurs it is time to consider the terrain height at the next abscissa, $i+1$, and to compare its value with the height of the visual ray point. The process ends when all

visual rays or all column heights are considered. For a screen with $n \times n$ pixels and the terrain with an $m \times m$ grid the complexity of the resultant algorithm is $O(n+m)$ per column or $O(n(n+m))$ for the entire image. Note that a naïve ray cast algorithm would test all terrain grids for each pixel in the screen with a complexity of $O(n^2 m^2)$ (or $O(n^2 m)$ if voxel coherence is exploited for each individual ray).

The slope for the j^{th} visual ray, shown in Fig. 6, can be computed by:

$$m_j = (\text{pitch} - j) / d_{pk} \quad (1)$$

where *pitch* is the distance shown between the horizon at eye level and the bottom pixel. Note that the word *pitch* in an aeronautical context refers to an angle and not a distance. We are using it here to denote a distance because this is the usual notation in game programming algorithms [Freese95]. Another important consideration is that j and *pitch* must be expressed in the same vertical scale, s_z , that affects the terrain heights.

The relationship between two consecutive angular coefficients can be obtained from equation (1) by:

$$m_{j+1} = (\text{pitch} - (j+1)) / d_{pk} = m_j - 1 / d_{pk} \quad (2)$$

The increment in z_i , dz_i , due to a change in angular coefficient can be computed from Fig. 5 by:

$$dz_{ij} = i(m_j - m_{j+1}) \quad (3)$$

If we replace equation (2) in equation (3) we get:

$$dz_{ij} = i / d_{pk} \quad (4)$$

Alg. 1 summarizes all above considerations for screen column k . The increments dx , dy are, respectively, the x and y components of the unit vector in the direction of the sampling direction $sray_k$.

```

z = heye; // init. visual ray height
x = xeye; y = yeye; // init. voxel position
m = pitch / dpk; // init. visual ray slope
dz = 0; // init. height increment
j = 0; // initial pixel
for i = 1 to MaximumRayDepth {
    z = z - m; // update ray height
    x = x + dx; y = y + dy; // update voxel position
    dz = dz + 1/dpk; // dz = i/dpk (see eq. 4)
    while (hi > z) { // ray intersects current voxel
        color = GetColor(x,y); // get voxel color
        SetPixel(k, j, color); // paint pixel
        m = m - (1/dpk); // see eq. 2
        z = z + dz; // update height
        j = j + 1; // next pixel
    }
}

```

Alg. 1 - Floating horizon algorithm.

It is important to note that game programming introduces some optimizations and a basic simplification in Alg.1. The optimizations are obtained with implementation strategies such as the use of assembly codes and the use of fixed point integer representations and are not treated here. The simplification, however, deserves to be discussed.

Note that Alg. 1 is very much dependent on the inverse of the distance d_{pk} from the eye to the k^{th} the pixel column,. For a planar projection this distance is not constant. It varies as the inverse of the cosine of the angle between directions *view* and *sray_k* shown in Fig. 5. In the game program literature this distance is sometimes treated as constant for all *srays*. That is, instead of dealing with d_{pk} , we would only use d_p . This corresponds to projecting the scene on the surface of a cylinder instead of projecting it on a plane. The *eye* would be at the axis of the cylinder, which would also be vertical. The problem with this approximation is that it deforms straight objects such as airport lanes. In the cylindrical projections straight lines are projected in ellipses. For this reason we cannot find straight-edged objects in such games. The advantage of this cylindrical projection, however, is that the sky horizon, which in plane projection is a straight line, appears as an ellipse segment giving the impression of a round surface such as the earth. It should be noted that the deformations introduced by cylindrical projections can be corrected with very little impact in the algorithm simplicity and performance. It suffices to adjust the step sizes along each column to compensate for the varying distances d_{pk} (see [Szenberg+97] for more details.)

4. Non-vertical view plane

We now consider the problem of generalizing the techniques in the previous section to the situation where the view plane is allowed to rotate about one of its horizontal lines. This implies that the view plane may no longer be vertical. Casting individual rays behaves as before; however, the optimization techniques based on the synchronization of image and terrain traversals are not applicable (at least in the same way). In the case of a vertical view plane, the rays corresponding to all pixels in a given image column are contained in a vertical plane. This does not hold when the view plane is not vertical: a plane containing the eye and a column of pixels in the image is no longer vertical (Fig. 7); on the other hand, vertical planes passing through the viewpoint intersect the view plane along lines which are not parallel to the edges of the visualization window.

The above discussion suggests two different possible approaches for the case of a inclined view plane. The first approach, which is discussed below, is to

continue to cast rays for a column of pixels in the image at a time and deal with the difficulties posed by the fact that the plane containing these rays is not vertical. The other approach is to insist in treating together rays that are contained in a same vertical plane. Textures associated to such rays produce an intermediary image, which must be warped in order to yield the final image. This is not considered in detail here and is subject of part of our current research.

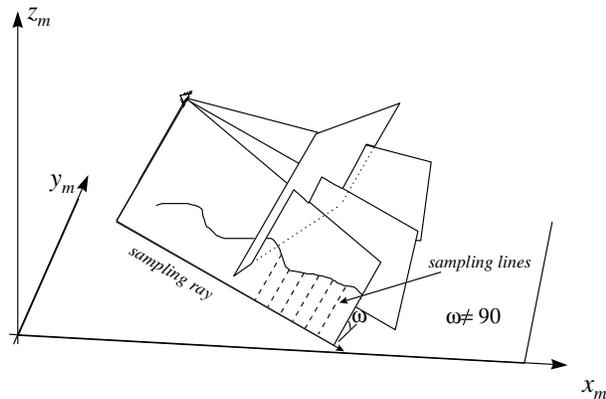


Fig. 7 - Sampling planes are no longer vertical.

Our goal, therefore, is to adapt the algorithm discussed in section 3 to the case of a inclined view plane. Pixels in the image will be treated a column at time, as before. As in the original algorithm, we sample the terrain, along the plane containing the rays, as shown in Fig. 7. However, since the sampling plane is not vertical, the sampling direction is no longer vertical. The best we can do is to choose a sampling direction contained in the sampling plane which is as close to the vertical direction as possible.

Several difficulties arise in this process:

- The terrain may not be a height field in the sampling direction. That is, a sampling line may intersect the terrain at several points. The problem becomes worse at the extreme columns of the image.
- Since the sampling lines are not vertical, deciding whether a given ray intersects the terrain along a given sample line is more complicated.
- If the terrain becomes sufficiently inclined, portions of the terrain which are behind the observer become visible. The algorithm has to be modified to cast rays both forward and backward.

The fact that the terrain may no longer be a height field is the most serious of the above difficulties. The optimized algorithm in section 3 relies heavily in the fact that if a given ray does not intersect the terrain at a certain sampling line, then the same is true for all subsequent rays; thus, after a voxel has been processed, we can continue to march along the terrain. Consider

however, the situation of Fig. 8, where the darker portions of the sampling lines indicate the portions that intersect the terrain. Ray 1 only hits the terrain when it reaches sampling line s . If sampling lines that precede s are ignored when subsequent rays are considered, the algorithm will erroneously report that ray 2 intersects the terrain at B and not at A.

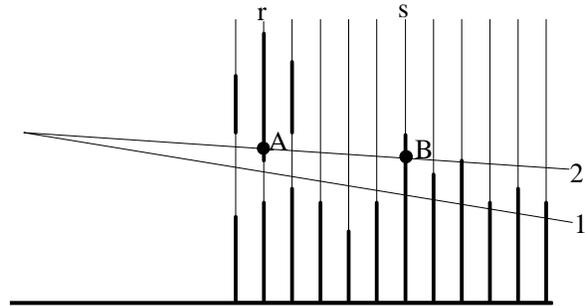


Fig. 8 - Ray coherence failure.

Therefore, if we decide to keep exploring ray and terrain coherence then we must be aware that errors such as the one described above may occur. These errors will be worse when the plane containing the rays is far from the vertical position and when the terrain is very irregular.

One way of avoiding these errors altogether would be to start over each time a new ray is considered. However, this increases algorithm complexity from $O(n(n+m))$ to $O(n^2m)$, which may be unacceptable for interactivity. This justifies studying approximate algorithms in which we retain the efficiency of the basic algorithm and, at the same time, obtain a rendered image which will not depart too much from the correct one.

5. Efficient approximate algorithms

When designing approximate algorithms, there is often a trade-off between processing time and degree of correctness. The algorithms described below were adapted from the basic algorithm in section 3, and show progressive geometrical correctness, at a cost of increasing processing times.

a) The naïve approach

The most naïve approach consists in ignoring the fact that the sampling planes are not vertical, as done in [Cohen+95]. Actually, the sampling plane corresponding to the middle pixel column is vertical. If the view plane is not too far from the vertical position, the remaining view planes can be treated as vertical without distorting the image too much. As the plane becomes more inclined, however, the distortions produced by the algorithm become severe.

Let j_0 denote the row of pixels which, together with the view point, defines a plane perpendicular to the view plane. Each column of pixels is considered to be produced by taking the middle pixel column and sliding it along row j_0 , in such a way that it stays in a vertical plane containing the view point and its angle with the vertical direction is kept constant. It is easy to see that the pixel columns thus generated will not stay in the view plane; instead, they will generate a curved surface, which accounts for distortions in the image.

The basic algorithm can be adapted to deal with this new situation. The main required change is in the updating of the slope m of each consecutive ray. When the column of pixels is vertical, m increases by a constant amount $1/d_{pk}$ for each ray, which provides an efficient incremental procedure for computing ray slope. This does not hold for an inclined column, forcing the separate computation of each slope.

Let us denote by α the angle between the pixel columns and the vertical direction (Fig. 9). To be consistent to the terminology in section 3, we will call *pitch* the distance from the bottom of the screen to row j_0 . The horizontal and vertical components of vector r_j , which joins the eye to the j^{th} pixel in the column, are given by:

$$r_{jh} = d_{pk} \cos \alpha + (j - \text{pitch}) \sin \alpha$$

and

$$r_{jv} = -d_{pk} \sin \alpha + (j - \text{pitch}) \cos \alpha$$

Therefore, the slope of the j^{th} ray is given by:

$$m_j = \frac{-d_{pk} \sin \alpha + (j - \text{pitch}) \cos \alpha}{d_{pk} \cos \alpha + (j - \text{pitch}) \sin \alpha}$$

Both the numerator and the denominator of the above expression can be computed incrementally, but a division is required for each ray, causing inefficiency. The cost of slope computing can be reduced, however, by using pre-computed slope values. In fact, we may compute slopes only for the middle column and either use the same slopes for the other columns (which results in straight horizontal lines having curved projections) or correct the size of the horizontal step to account for the varying values of d_{pk} , as explained in section 3.

b) Non-vertical sampling

We now consider the actual geometry of the sampling planes. Each column (except the middle one) determines with the eye a non-vertical plane. Since the vertical direction is not contained in such a plane, we have to settle for a sampling direction which is not too far from the vertical position. The optimal sampling direction for each sampling plane can be obtained by projecting the vertical vector $(0, 0, 1)$ onto the plane. However, the

basic algorithm is more easily modified if, instead, we sample each plane along the lines obtained by intersecting them with a family of regularly spaced vertical planes parallel to the pixel rows. This is equivalent to consider the terrain as being composed by regularly spaced vertical planar slices. By using this approach, we can project orthogonally each sampling plane onto the vertical sampling plane corresponding to the central column of pixels and take advantage of the same pre-computed slopes used in the previous algorithm.

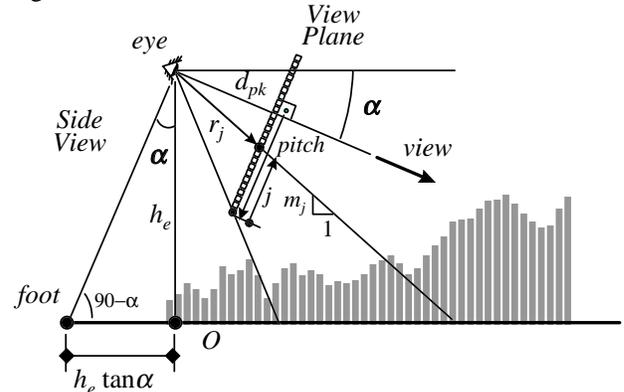


Fig. 9 - Ray slopes for a non-vertical view plane.

As in the basic algorithm, for each pixel column k , we march on the terrain along the line of intersection $sray_k$ of the corresponding sampling plane with the horizontal plane $z = 0$. The march starts at point P_k of intersection of $sray_k$ with the vertical slice through the eye. The steps along $sray_k$ are given by vector s_k . Let us consider the local coordinate system of Fig. 10 to derive expressions for P_k and s_k .

The direction of $sray_k$ can be found by conducting a horizontal vector from the eye to point H_k on the corresponding pixel column. The coordinates of this vector are given by:

$$(k - k_m, \frac{d_{pk}}{\cos \alpha}, 0),$$

where k_m denotes the index of the middle pixel column.

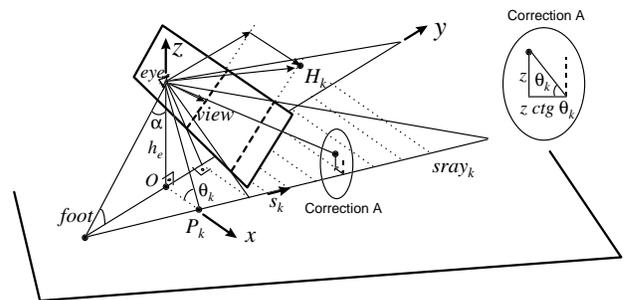


Fig. 10 - Non-vertical sampling planes.

The step vector s_k is then computed by applying the scaling factor $\frac{\cos \alpha}{d_{pk}}$, leading to a vector with unit y-component given by:

$$s_k = \left(\frac{(k - k_m)}{d_{pk}} \cos \alpha, 1, 0 \right) \quad (5)$$

Choosing s_k in such a way ensures that the slopes m_j computed for the middle pixel column are valid for all pixel columns, in the sense that they correspond to the change in height for each horizontal step.

A simple way to obtain the coordinates of P_k consists in observing that each line $sray_k$ goes through the point denoted by *foot*, obtained by intersecting plane $z = 0$ with the line through the eye that is parallel to the pixel columns. The coordinates $(0, -h_e \tan \alpha, 0)$ of *foot* are readily available (Fig. 10). Adding to *foot* the appropriate multiple of s_k yields

$$P_k = \left(\frac{h_e}{d_{pk}} (k - k_m) \sin \alpha, 0, 0 \right). \quad (6)$$

The above derivation is not valid when the view plane is horizontal. However, both s_k and P_k are easily computed in that case, and are given by $s_k = (0, 1, 0)$

$$\text{and } P_k = \left(\frac{h_e}{d_{pk}} (k - k_m), 0, 0 \right).$$

The last aspect to be considered is the actual sampling of the terrain. In the standard case, the height and texture of the terrain at a given position $P(x, y)$ along $sray_k$ is obtained from the corresponding point in the height and texture maps. If we do the same in the non-vertical case, we obtain an approximate solution which may be appropriate in many cases. The projection obtained by such a procedure is not a true parallel or perspective projection, but has the property that vertical lines always project vertically. A more accurate projection is obtained by correcting the position where to sample based on the height z of the current ray. As the detail in Fig. 10 shows, the slope of each sampling line is

given by $-\frac{d_{pk}}{(k - k_m) \sin \alpha}$. Therefore, the position where to sample the terrain can be obtained by adding the correcting vector zc_j to P , where c_j is the vector whose local coordinates are given by:

$$c_j = \left(-\frac{(k - k_m)}{d_{pk}} \sin \alpha, 0, 0 \right). \quad (7)$$

This correction should be applied only when $z > 0$. If $z < 0$, we may either sample the terrain at point P

(without applying any correction), or try to estimate the point where the ray intersects the terrain (using linear interpolation, for instance). Our experiments show that the resulting images are similar.

Algorithm 2 below summarizes the above discussion.

```

z = heye; // init. visual ray height
P = Pk; // init. voxel position (see eq. 6)
m = m0; // pre-computed slope
j = 0;
for i = 1 to MaximumRayDepth {
    P = P + sk; // update voxel position(see eq. 5)
    z = z - m;
    h = GetHeight(P+zcj); (see eq. 7)
    while (h > z) { // ray intersects current voxel
        color = GetColor(P+zcj); // get voxel color
        SetPixel(k, j, color); // paint pixel
        j = j + 1;
        m = mj; // pre-computed slope
        z = z + i * m; // update height
        h = GetHeight(P+zcj);
    }
}

```

Alg. 2 - Floating horizon for non-vertical view plane.

6. Experimental results and conclusions

For comparison purposes, the following algorithms were implemented:

- the standard algorithm (for vertical view planes);
- the naive algorithm of section 5a;
- the algorithm in section 5b in two versions, with and without the correction for the sampling position;
- the brute force algorithm that casts each ray individually;

Average running times for a Pentium 133 machine are given in Table 1.

	Time per frame (ms)	Frames per second
Standard	107	9.3
Naive	123	8.1
No correction	134	7.5
With correction	277	3.6
Brute force	4697	0.21

Table 1 - Frame rate of proposed algorithms.

The data shows that both the naive and the first algorithm in section 5b have performances that are comparable to the standard algorithm, while the version

with the sampling correction (which must be done for each painted pixel) is roughly half as fast as the others. The running time for the brute force algorithm demonstrates the need for optimized versions.

We ran each algorithm with two types of data: terrain real data and artificial scenes of blocks on a checkered ground. The first type of data was used to assess the ability of each algorithm to generate images which help the user to understand the terrain being examined. The second type of data was used to see how precise was the projection generated in each method. Figures 11 e 12 shows some of the results. Both figures show, in the top row, images produced by the brute force and the naive algorithm and, in the bottom row, images produced by the algorithm in section 5b, without and with correction. In both figures, the view plane was inclined 30° with respect to vertical.

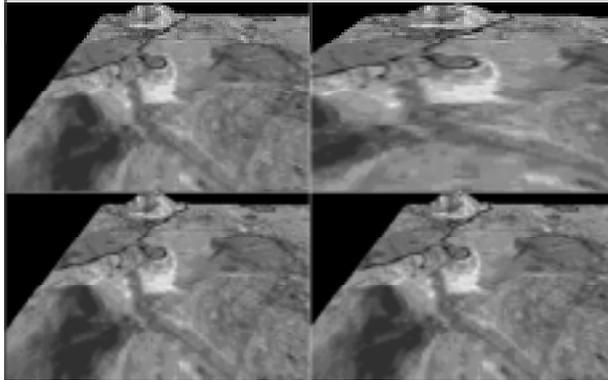


Fig. 11 - Visualization of real terrain data.

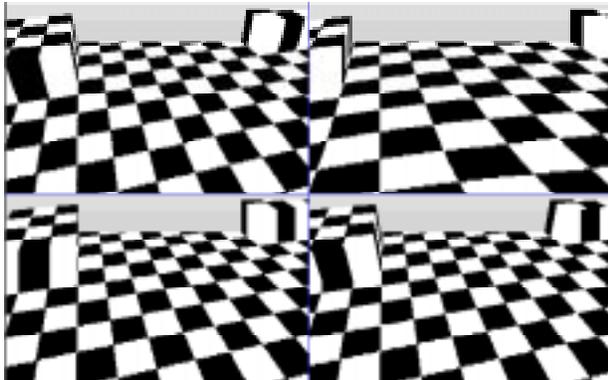


Fig. 12 - Visualization of artificial data.

The general conclusions are:

- For terrain data, all methods generate reasonable images when the view plane position is not far from the vertical. As the view plane becomes more horizontal, the images generated by the naive method become unacceptable, but both methods in section 5b give reasonable results.

- The artificial data shows that the correction in the algorithm of section 5b helps producing images that are very close to the one obtained without the approximations. However, images produced without the correction may be acceptable in many cases, especially because they have the property that vertical lines project vertically, which may result in some comfort for the user.

9 Acknowledgments

This work was mainly developed in TeCGraf/PUC-Rio and was partially funded by CNPq, by means of fellowships and the PROTEM/CC-GEOTEC project. TeCGraf is a Laboratory mainly funded by PETROBRAS. Part of the research was done at the VISGRAF/Impa laboratory. The authors want to thank Paula Frederick for her valuable contribution in the early stages of this work.

References

- La Mothe, *Black Art of 3D Game Programming*, Waite Group Press, 1995.
- Freese, *More Tricks of the Game Programming Gurus*, Chapter 7, SAMS Publishing, 1995.
- Williamson, H., "Algorithm 420 Hidden-Line Plotting Program", *CAC*, 15(2), February 1972, 100-1003.
- Cohen and A. Shaked, "Photo-realistic Imaging of Digital Terrains", *Computer Graphics Forum*, 12(3): 363-373, 1993.
- Ch. Graf, et alli, "Perspective Terrain Visualization - A Fusion of Remote Sensing, GIS and Computer Graphics", *Computer and Graphics*, 18(6): 795-802, 1994.
- Szenberg, et alli. *An algorithm for visualization of a terrain with objects*. Proceedings of SIBGRAPI, 1997.