Real-Time Vo el Te ai Ge e a io

For the past few years, game programmers have been steadily adding terms to their 3D vocabulary — terms such as binary space partition, rasterization, perspective-correct textures, MIP-mapping, and so on. One of these terms has become a catch-all describing a general means to an end rather than the specific structure that it was originally conceived to describe. That word is "voxel," and the context is voxel engines. Voxel engines have come to mean anything that renders terrain-like imagery; whether or not the engine is based on real voxels is immaterial as long as it looks good.

Figure 1 is a screen shot of the voxel engine created in this article compared to a screen shot from the quintessential voxel-based game COMANCHE 3 from NovaLogic. I don't know exactly what technique they're using, but as long as ours looks the same, why ask why? Anyway, in this article, we're going to hold fast with the tradition of using the term voxel loosely to describe a technique used to render data that has volumetric information in it rather than polygonal data. However, before we take off into mathematical hell, let's take a look at real voxel graphics and their data representations.

Real Voxels

Voxel graphics originated in medical imaging. Voxel means "volumetric pixel" and is essentially a cube in 3D space. Each cube has a volume dependent on the length of one of its sides. However, most representations use voxels that are 1 × 1 × 1, so each voxel has a volume of 1 × 1 × 1 = 1.0 voxels. At this point, you may be wondering, "How do I represent voxel data in a computer?" There is no correct answer, but most of the time, voxels are represented by either a 3D array or a more abstract data structure such as a tree. Figure 2 illustrates a voxel geometrically, along with the two most common representations. Voxel data is usually obtained from medical imaging systems such as MRI and CT scanners.

Voxel rendering can be accomplished in a number of ways, such as ray tracing or even

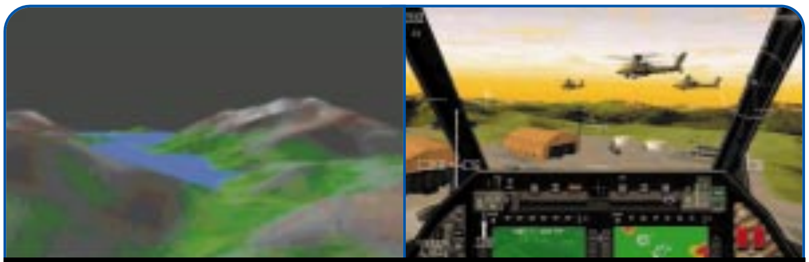


FIGURE 1. *The author's voxel engine is on the left; COMANCHE 3 is on the right.*

with polygon graphics. In the case of ray tracing, imagine that each voxel is a tiny little cube, and a ray tracer traces out the voxel data by computing ray intersections from the viewpoint through the view plane into the voxel data. Of course, this approach would be extremely slow because there are zillions of voxels in even the simplest voxel data set. For example, say that you're scanning your head and you have a scan resolution of 100 dots per inch; that means that there are 100 × 100 × 100 voxels in each cubic inch or 1,000,000 voxels! Now image that your head is about 6 × 6 × 10 (I don't have a ruler) inches for a total of 360 cubic inches — 360 * 1,000,000 = 360,000,000 voxels! At eight bits per voxel, you've got 360 megabytes. Call it half a gigabyte for people with large foreheads, and you've got a real problem. There must be a better way to store and display voxel data. And, of course, there is.

Voxel data can be compressed, take advantage of spatial partitioning techniques, and so on. However, the real problem isn't in the storage, but in the rendering. Processing hundreds of megabytes of data per frame would be a bit slow, to say the least. Therefore, algorithms have been devised that take advantage of the cubic and regular structure of a voxel data set to render the data quickly and in almost photorealistic quality. One such algorithm is the "Marching Cubes" technique, which is based on rendering slices of voxel data. Fortunately, we're not going to learn the marching cubes algorithm, because we don't need to render full voxel data. Rather, we're concerned with data that is partially volumetric, such as height data.

SOME OF THE MOST REALISTIC TERRAINS IN TODAY'S GAMES USE VOXEL TECHNOLOGY. THIS TUTORIAL EXPLAINS VOXELS IN THE CONTEXT OF BUILDING A RUDIMENTARY FLIGHT SIMULATOR by André LaMothe

Unreal Voxels

Since we're interested in simply drawing 3D terrain (in real time of course), we don't need full 3D voxel data. A mountain range is basically a 2D grid with a height (and color) at each position, or in another words, a height map (Figure 3). The trick is to figure out a way to render height map data in such a way that it looks 3D and has some resemblance to the data set being rendered. Because we don't want to write a polygon engine that renders little cubes, or more precisely, vertical parallelepipeds, we're stuck with using some form of ray tracing to create our display.

This isn't as bad as it seems. Just as ray casting (the technique used for WOLFENSTEIN and RISE OF THE TRIAD) works for worlds made of rectangular solids, we can use ray casting to render worlds that are composed of rectangular solids that

FIGURE 2. *Three representations of a voxel.*

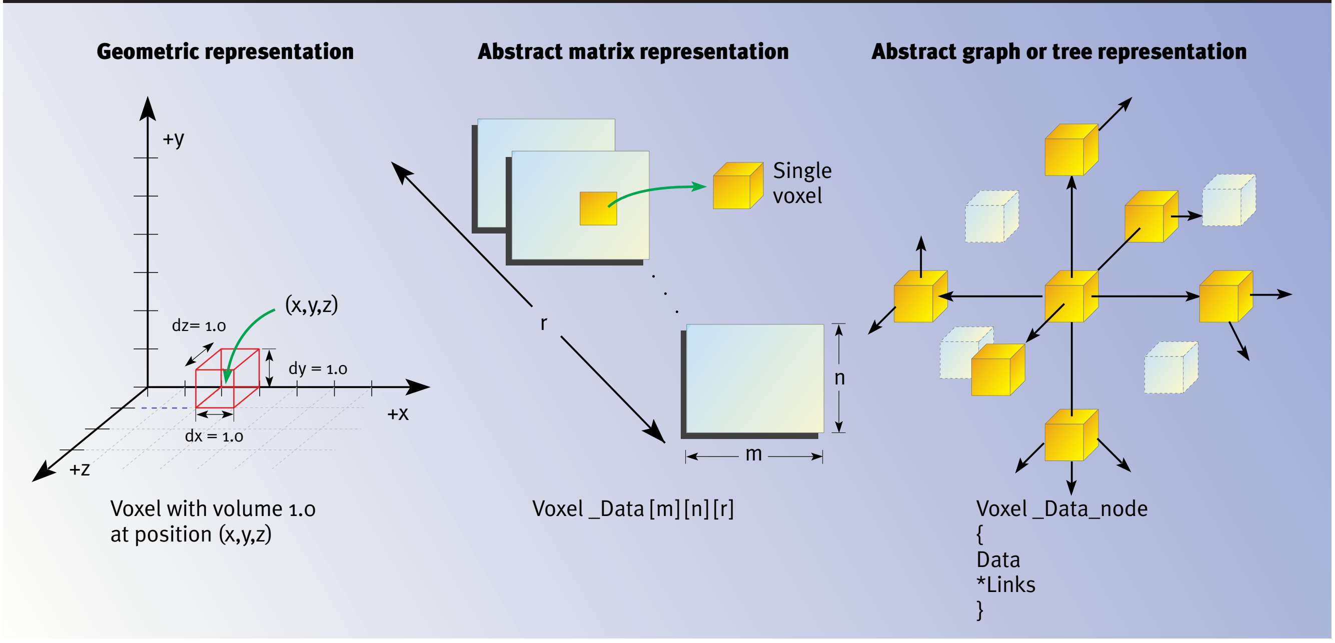
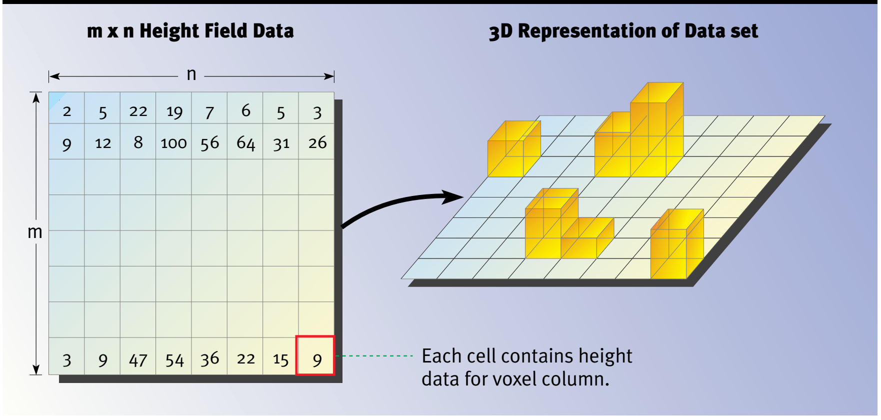


FIGURE 3. *Height map data.*



just happen to be about a pixel or two in thickness. All we need to do is change the scale of our thinking and use the same idea. In addition, because we're going to use height map data, we don't need a ton of 3D voxels to describe our data set. We simply need a 2D array that contains a height in each cell. Therefore, we've already killed at least one full dimension of storage — and that is a good thing.

There is once catch to our representation/compression technique; the minimum altitude of all heights is the same (that is, sea level), and each column must have the same color along its vertical extent. Of course, we could add another 2D array that contains the starting altitude, allowing us to render caves, but we'll keep it simple this time around. The real bummer is that each voxel column in the height map must be the same color. However, if you think about it, this doesn't matter, because when you're looking at a mountain range, you only see the top of each vertical column. The color of the material at various depths is irrelevant unless you have x-ray vision and can see through the mountain material. So the bottom line is that all we need is one 2D map to represent the height data, and maybe another 2D map, similar to a texture map, that represents the color for each voxel column. So if the world is 1,000 × 1,000, then we need a total of 1MB × 2 or 2MB of storage, assuming a byte for each height and color. That's a pretty good savings from our original 500 Mb of full 3D voxel data!

FIGURE 4. *The 3D viewing system.*

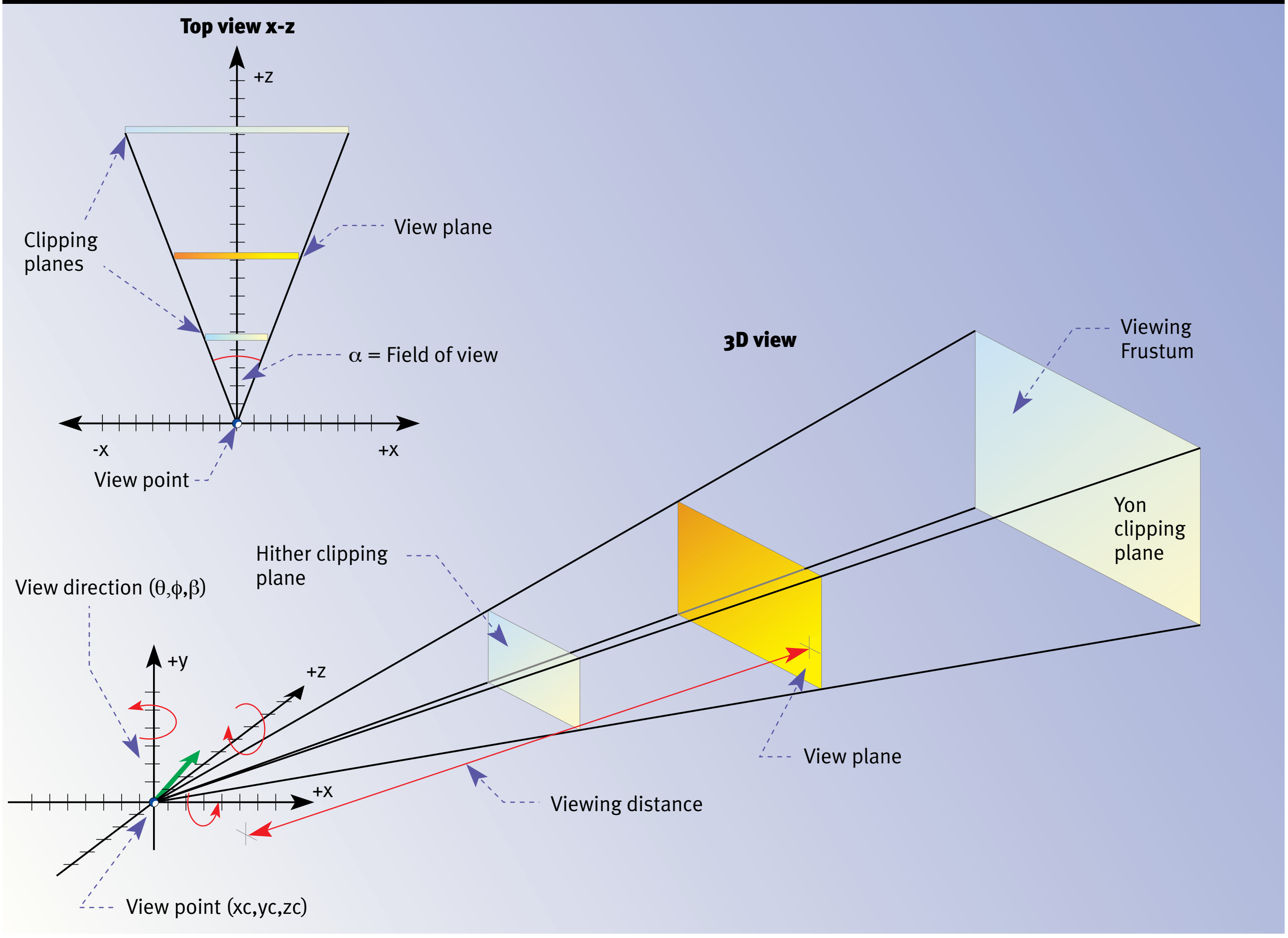
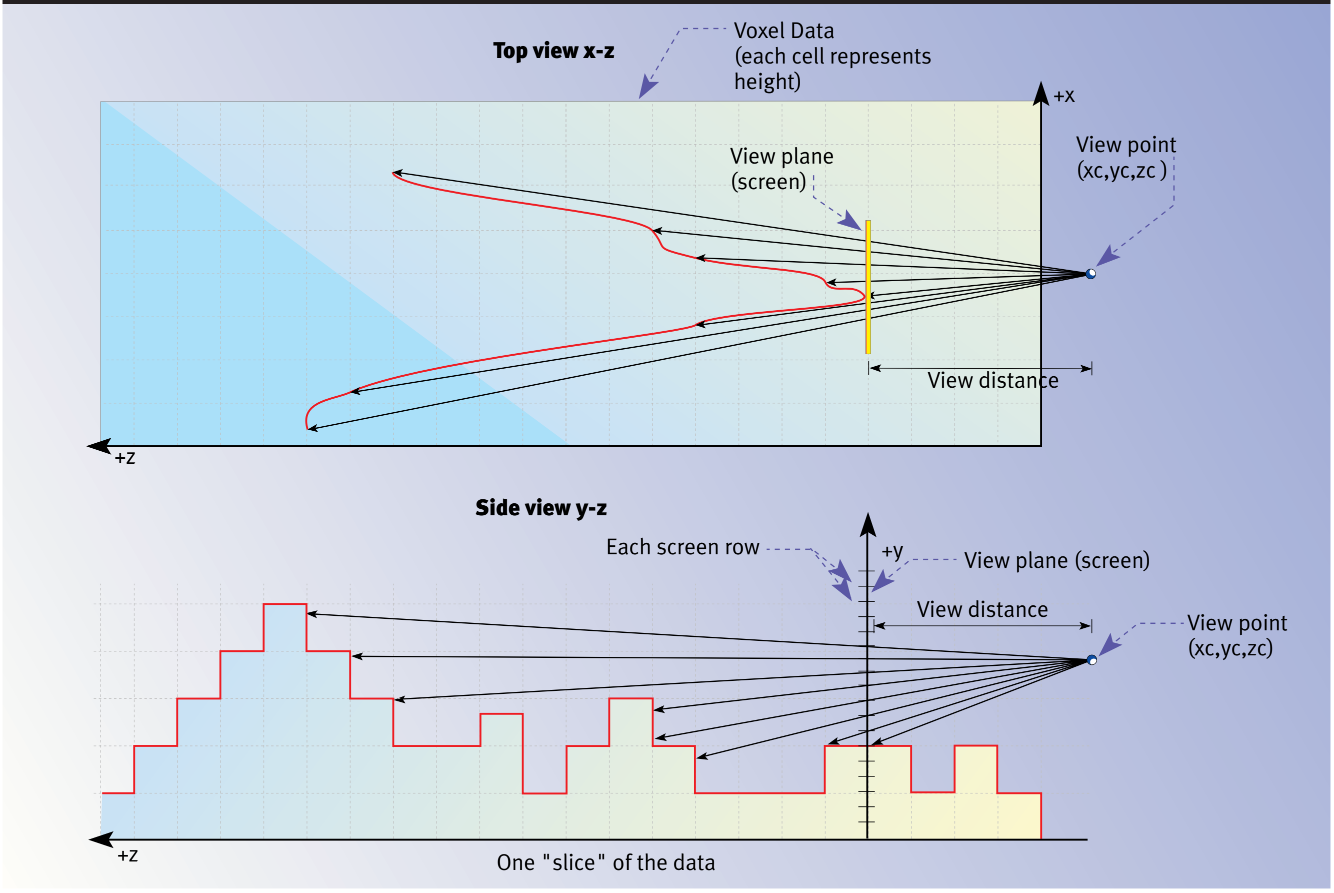


FIGURE 5. *Ray tracing voxel data.*



A Ray Casting Terrain Algorithm

Before we write the terrain generator, we're going to look at how to do it the slow way and then we'll make a number of optimizations to make it about 100 to 1,000 times faster. The key here is to understand the geometry and what is happening in the terrain generator. Don't worry, we're not going to plunge into gradients and vector calculus. We'll use nothing more than basic geometry with a little bit of trigonometry.

Let's begin with some terminology — refer to Figure 4 as we go. The position that we're going to view the scene from is called the "view point," or "camera." It has a position (xc,yc,zc) and orientation (θ, φ, β). The position is in "world coordinates," and the orientation coordinates are angles analogous to the pitch, yaw, and roll of the camera. The direction that the camera is pointing is called the "view direction." The object we're looking at is called the "view plane," which is a surface representing the window onto which we're mapping the 2D information to generate the illusion of 3D. The view plane is usually the screen, but doesn't necessarily have to be. Also, the perpendicular distance from the view plane to the camera is called the "view distance." Finally, the camera has a "viewing volume," or "frustum." This is the area that is visible to the camera. For example, most people have about a 140-degree horizontal field of view (FOV) and a 90- to 120-degree vertical FOV. So we should make our computer models within the same order of magnitude if they are to look at all real. For example, DOOM and QUAKE use a 90-degree FOV. A game with a 60-degree FOV would show less, and a 180-degree FOV would be like looking out of a wide-angle lens — that is, lots of distortion.

Imagine that the camera is flying above the terrain at some height, and we want to see what the resulting image would be on the computer screen. This can be accomplished by brute force ray tracing. For every pixel on the screen, we'll construct a ray that originates from the camera, pierces the screen (at the pixel), and continues on until it hits the terrain. Figure 5 illustrates this using multiple views. Listing 1 shows the pseudo-code.

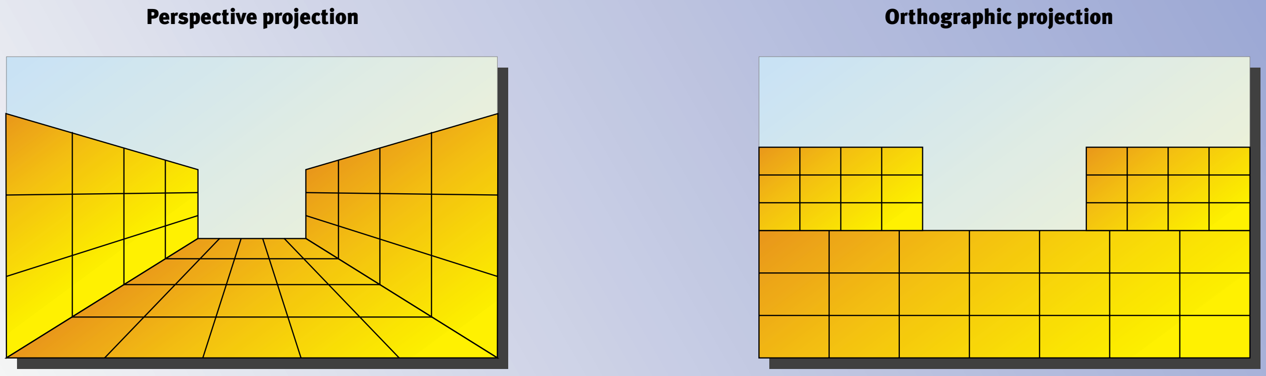
That's all there is to it. Because of the algorithm's design and its close analogy to the physics of photons and real

LISTING 1. *Pseudo-code for brute force ray tracing.*

```
For (each pixel on the screen)
    {
    1. Compute ray originating from camera thru pixel.
    2. Project the ray.
    3. If (Projected ray intersects terrain)
          {
          Color screen pixel color of intersected terrain data.
          }  // end if

    } // end loop
```

FIGURE 6. *Screen projections.*



$x' = D * \frac{x}{z}$

$y' = D * \frac{y}{z}$

z coordinates are used to complete (x',y').

$x' = x$

$y' = y$

z coordinates are thrown away.
Projection is totally parallel.

light, the scene will be rendered almost perfectly (of course, there won't be shadows or reflections). The only problem is that Steps 1, 2, and 3 take a lot of computer cycles. For example, if you want to render a screen image that is 320 × 200, you'll have 64,000 pixels. Hence, we must compute 64,000 rays or parametric lines, project these until they intersect with the terrain data, compute the points of intersection, and then plot the pixels on the screen in the correct color. That's a lot of computing. We need a way of doing the same thing with less math and fewer computations. This can be accomplished by taking advantage of the regularity of the data and the behavior of perspective transforms.

Optimizing the Algorithm

We're now going to derive an algorithm and associated optimizations so that voxel terrain data can be rendered at faster rates. However, the algorithm is so simple, you really need pay attention to all the subtle details. Let's begin by taking a look at the derivation of the perspective transform and what it does for us.

When drawing 3D computer graphics on a 2D computer screen, we only have two dimensions. Therefore, to convey the data that is lost in the translation of 3D to 2D, various projection techniques have been created. One such technique, perspective projection, is based on the premise that as an object moves away from the view plane and camera, it should get smaller. Similarly, as the same object gets closer to the view plane and camera, it should get larger. Therefore, we can deduce the projection from the z coordinate of each vertex making up a 3D object.

Figure 6 represents a perspective transformation as well as a nonperspective, or orthographic, projection. As you can see, the perspective projection looks a lot better — it looks 3D. The key to generating a perspective projection is using the z component of each vertex to scale the position of each (x,y) to arrive at a new (x,y). It might be easier to understand this relationship by looking at how real light works. Take a look at Figure 7a. Here we see a line segment at some position on the left side of the view plane. To generate the perspective projection of the line on the view plane, all we need to do is draw rays from the view point through the view plane to each end of the line segment. The point where each of these lines intersects the view plane is the perspective-correct line that we would see if we were viewing the scene at the given viewing distance through a window.

Figure 7b shows the derivation of the perspective transform. The figure shown is a side view of the z-y plane, so there isn't any x information; a similar derivation can be done for the z-x plane. Anyway, the view point lies on the z axis at a distance V_d from the view plane, and the point P = (Px,Py,Pz) is at some distance Pz and at some height Py from the view point. Finally, P' = (Px',Py',Pz') is the point of intersection on the view plane. Using similar triangles, we extract the relationship

$$\frac{Py}{Py'} = \frac{Pz}{V_d}$$

or,

$$Py' = V_d * \frac{Py}{Pz}.$$

Thus, it looks like the perspective transform of a point (x,y,z) at a viewing distance of V_d is

$$\begin{aligned} \text{x\_perspective} &= \left(V_d * Px/Pz\right) \\ \text{y\_perspective} &= \left(V_d * Py/Pz\right). \end{aligned}$$

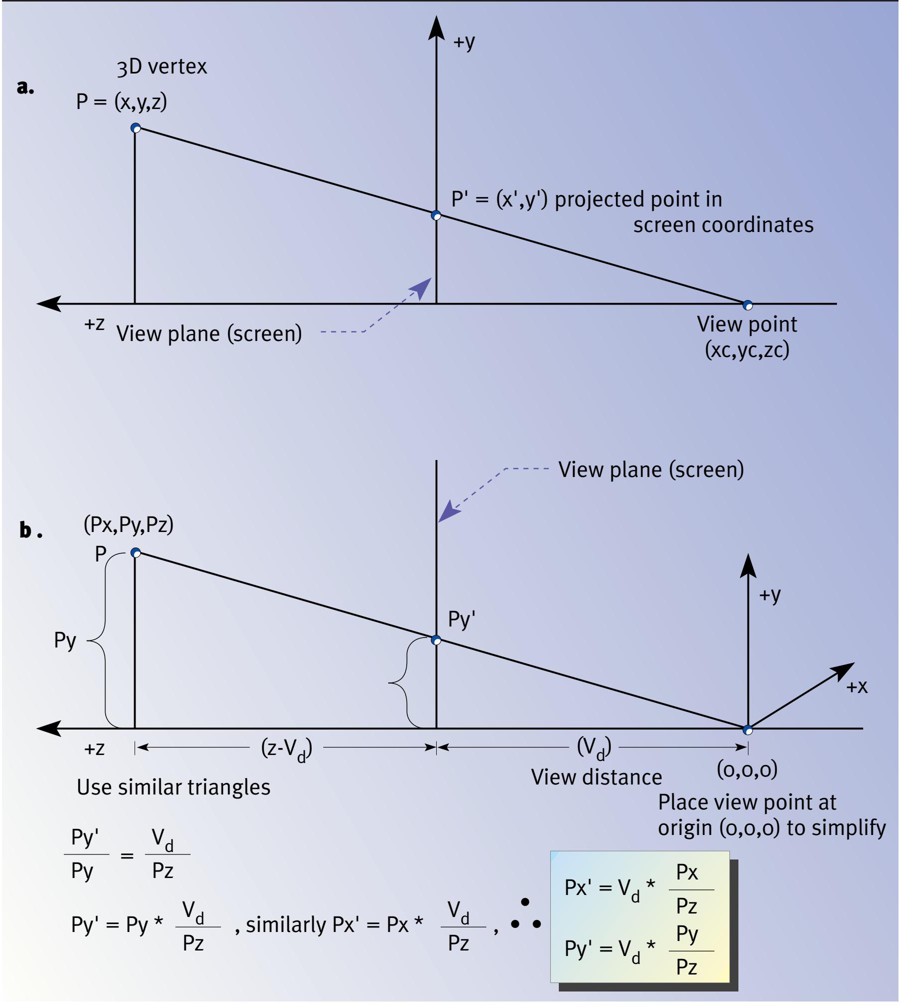
This tells us a couple things: First, the height or size of a line decreases as it gets farther away from the view plane. We therefore know that the scale of an object is also proportional to the z distance that it is from the view plane. We're going to base our entire algorithm on this important fact.

Now it's time to get down to the details. We have all the tools that we need to really analyze the situation and come up with a better algorithm than the brute force method of computing a ray for each pixel on the screen. What we're going to do is trace out, or contour map, each vertical scanline of the screen by first casting out a single ray for each column on the screen. Based on the first voxel column that the ray hits, we'll then follow the contour of the terrain from that point on without any further rays being cast out in that particular column (Figure 8).

But before you do that, let me offer you a warning. The following figures are a bit cavalier in their labeling of the axes. This is due to the fact that we're trying to mix world coordinates, polar coordinates, and screen coordinates. The bottom line is that when we finally get to writing the engine, the x-y axis will be the ground plane, and the z axis will vertical. However, for a number of the figures, I used z to go into the screen and y for vertical just to make things easier to understand.

It's possible to find the first intersection of a ray cast from the view point through the bottom of the view plane into the terrain, and then follow the rest of the terrain's contour and render it based on the math and consistency of the perspective transform. How? Well, we need to make a few observations before we can construct the algorithm. Figure 9 shows the view point with a number of rays being cast through each pixel in the same screen column. This figure contains a lot of information, so let's review it slowly.

FIGURE 7. *The derivation of the perspective transform.*



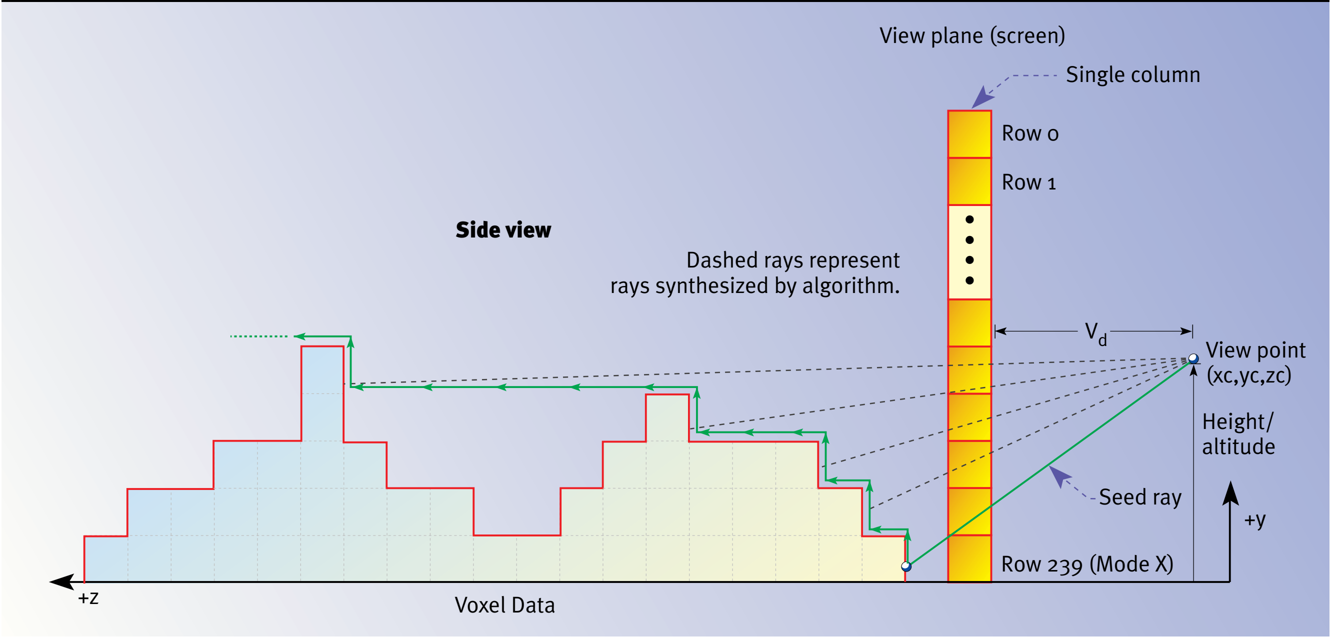
$$\frac{Py'}{Py} = \frac{V_d}{Pz}$$

$$Py' = Py * \frac{V_d}{Pz}, \text{ similarly } Px' = Px * \frac{V_d}{Pz}, \therefore$$

$$Px' = V_d * \frac{Px}{Pz}$$

$$Py' = V_d * \frac{Py}{Pz}$$

FIGURE 8. *Contour ray casting.*



The first thing I want to bring to your attention is the slope of the lines. The first line has a slope m_1. Remember that

$$\text{Slope} = \frac{\text{Rise}}{\text{Run}}.$$

And remember, slopes don't always have to be in terms of x and y — they can be in terms of anything. A slope is the rate of change of one variable in relation to another. In our case, we're interested in the rate of change of z with respect to x or dz/dx. In Figure 9, the first line with slope m_1 has dz = 1 and dx = V_d. Therefore, the slope must be

$$m_1 = \frac{dz}{V_d} = \frac{1}{V_d}.$$

That's interesting. The slope of a line that originates at a distance V_d from the view plane and pierces the first pixel below (or above) has a slope of $1/V_d$ (the reciprocal). If you're with me, then hold onto your hat, because here's the clincher; in addition to m_1 having slope 1/dz, m_2 has a slope of $2/V_d$. Hence,

$$\begin{aligned} m_1 &= 1/V_d \\ m_2 &= 2/V_d \\ m_3 &= 3/V_d \\ &\vdots \end{aligned}$$

Or, in terms of m_1,

$$\begin{aligned} m_1 &= 1 * m_1 \\ m_2 &= 2 * m_1 \\ m_3 &= 3 * m_1 \\ &\vdots \end{aligned}$$

What this means is that the change in slope per vertical screen pixel, or per dy, is constant and equal to $1/V_d$, which we'll call ds. Referring again to Figure 9, we see that the lines with slope $m_1, m_2, \ldots m_5$ each intersect lines parallel to the view plane at constant intervals. In other words,

$$\begin{aligned} y_1a &= 1 * ds * (V_d + z_a) \\ y_2a &= 2 * ds * (V_d + z_a) \\ y_3a &= 3 * ds * (V_d + z_a) \\ &\vdots \\ dy_a &= ds * (vd + z_a) = ds * \text{constant}_a, \end{aligned}$$

and

$$\begin{aligned} y_1b &= 1 * ds * (V_d + z_b) \\ y_2b &= 2 * ds * (V_d + z_b) \\ y_3b &= 3 * ds * (V_d + z_b) \\ &\vdots \\ dy_b &= ds * (V_d + z_b) = ds * \text{constant}_b. \end{aligned}$$

So the points of intersection of a ray along any vertical line at a given distance from the view point are also at constant intervals. Moreover, this constant increases at a rate of (ds*z), where z is just the distance between the line and the view point along the z axis.

That's all there is to the algorithm. We can now create an incremental ray caster that casts a single ray for each column of the screen and then generates a vertical column of pixels that represent what the display would have looked like if we had cast a ray for each pixel in the column. Let me repeat, the algorithm we're going to come up with only casts a single ray from the view point through the bottom of the view plane (the screen) for each column. The ray's intersection is then computed, and from that point on, the rest of the contour for that column is generated incrementally using the math that we've derived.

That sounds great, but how? Our plan of attack is as follows: First, we'll need to generate a ray that has a dx,dy, and dz for each column of the screen. We've talked about the dz part, but what about dx and dy? The dx and dy components are computed by finding the endpoints of a unit vector that lies in the ground plane and is perpendicular. For each column of the screen, we want to cast a ray. So for a screen that is 320 pixels wide, such as Mode X, we need to cast 320 rays, each at a slightly different angle. To do this, we first need to decide on our horizontal FOV. Assuming that we want a 60-degree FOV and that our screen is 320-pixels wide, we'll need to break 360 degrees up into smaller subangles so that from any viewing direction (that is, heading), we can cast out 320 rays: 160 to the left and 160 to the right. So we create virtual degrees, based on the FOV and the screen width. The formula is

$$\text{Number of Virtual Degrees} = \text{Screen Width} * \frac{360}{\text{Field of View}}.$$

In our case, this is equal to 320*(360/60) = 1,920 virtual degrees. With that computation out of the way, let's generate the dx,dy part of the ray

FIGURE 9. *The geometry of height mapping.*

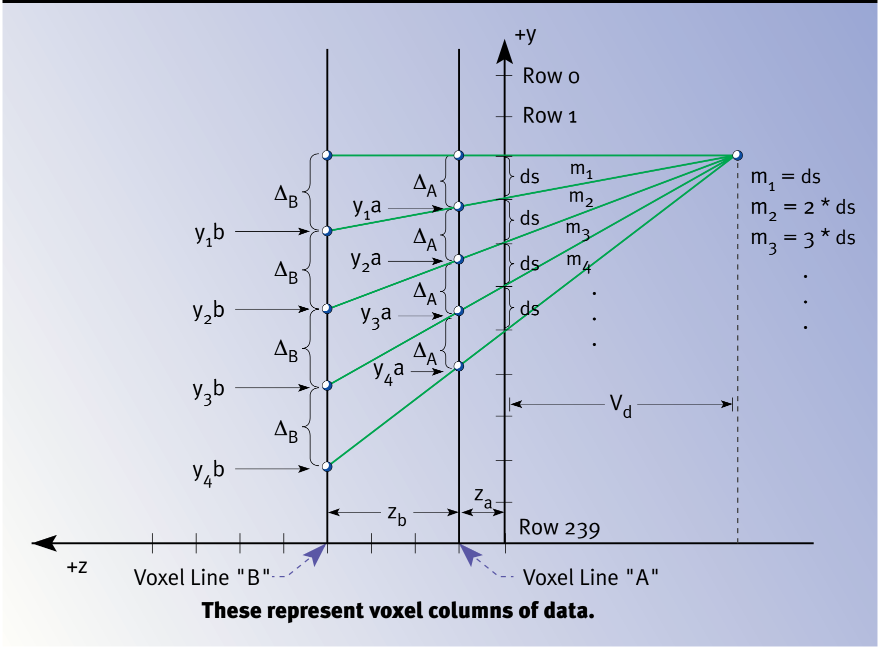
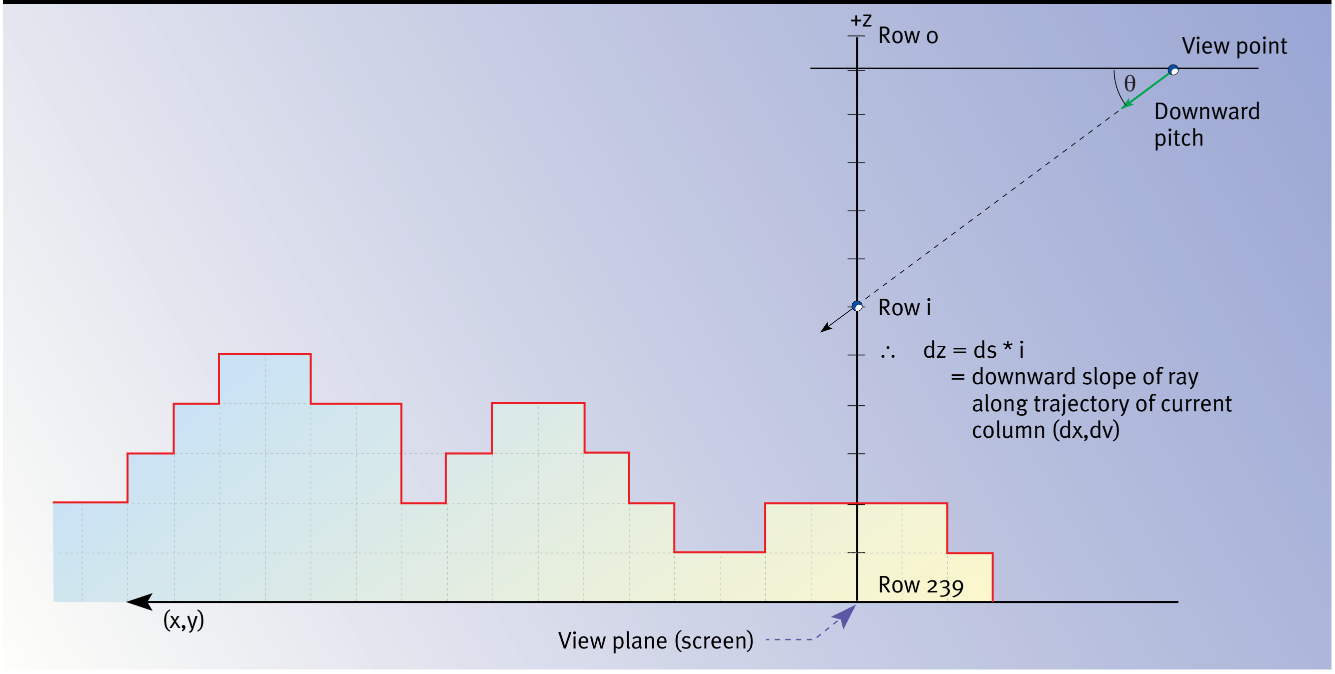


FIGURE 10. *Computing dz.*



trajectory. From basic trigonometry, we know that any point on a circle with radius r is equal to

$$x = r * \cos(\theta)$$
$$y = r * \sin(\theta)$$

where θ is the angle made with the positive x axis.

Since our camera is always flying above the terrain at an altitude of z, we can use this equation to generate our dx,dy deltas. However, instead of using the standard sine and cosine, we must use versions of them that can take angles from 0 to (1920 - 1). That's why we'll need to make look up tables. The pseudo-code for our outer loop is in Listing 2.

Now back to the computation of dz. You'll recall that dz is the slope of the downward ray that we are casting to compute the very first intersection (Figure 10). Referring to the figure, the downward pitch of the camera relates to the starting position of the ray cast. This means that if the camera is pointing down to row 100 of the screen, then the slope of the ray is 100*ds. Therefore,

$$dz = \text{screen row} * ds$$

Of course, since the screen's y coordinate gets more positive on the way down, we'll need to alter this equation a bit; in fact, we'll need to do a lot of little tweaks to make things work.

Now that we know how to compute dx,dy, and dz, we're finally ready to complete the algorithm. Once we compute these values, we enter a loop to render the current column. The loop will project the ray from the camera's view point (xc,yc,zc) at a rate of dx,dy,dz. However, we must track one other variable — the "current projected scale." Recall that we derived that the deltas between ray intersections piercing adjacent pixels in a single column will intersect any vertical column beyond the view plane at constant intervals. These intervals are equal to the current distance from the view point multiplied by the change in slope between adjacent pixels on the screen in a single column, which is always the same and is called ds in our derivation. This means that if you're some distance d from the view plane and you're trying to draw a voxel column, then you need to scale the column by this amount to take into consideration the perspective scaling that exists at the given distance from the view plane. And the amount of scaling or difference between adjacent intersections in a vertical column gets larger as we move farther away from the view plane.

Now that we know that we need to track this "scaling" variable, how do we use it? When the main rendering loop is entered, the position of the ray is updated with the deltas dx,dy,dz, and then the current z position of the ray is compared to the height data located in the height map at the current ray's downward projected (x,y). If the height in the height map is higher than the ray, we enter a second inner loop. In this inner loop, the goal is to draw pixels in the column until we have drawn enough pixels so that the perspec-

LISTING 2. *The outer projector loop.*

```
// start ray off at far left column
curr_angle = curr_heading + 160

for (int column = 0; column < 320 column++)
    {
    dx = COS_LOOK[curr_angle];
    dy = SIN_LOOK[curr_angle];
    dz = ?

    // draw the column

   // adjust the trajectory angle, rotate ray to right
   curr_angle--;

   } // end for
```

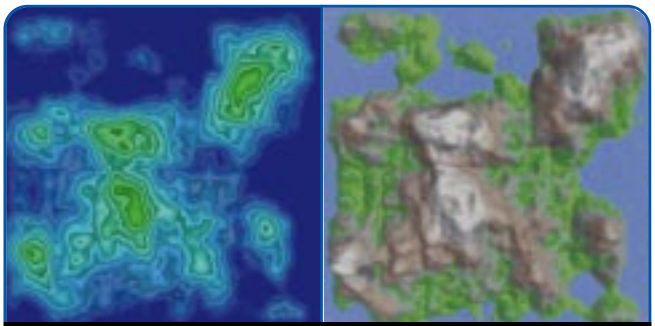


FIGURE 11. *Using VistraPro 4.0 to render terrain. 11a (left) is the terrain height data exported as a 256-color gradient contour map. 11b (right) is the rendered terrain based on that contour map.*

tive projection of the voxel column looks correct on the display. This is accomplished by having an exit strategy based on the current projected scale. At each iteration, we add the current projected scale to the z of the ray, which has the effect of "climbing" up the voxel column. At the same time, we adjust the slope of the ray, because each pixel we draw means that the new ray would have a slope change of ds. Therefore, dz is incremented by ds. Finally, when z is above the voxel column (or when we've hit the top edge of the screen), we bail out of the loop and continue tracing the contour. The tracing process must be done for a number of steps from "hither" to "yon" in Figure 4 to cast far enough to see reasonable detail.

Putting It All Together

Now to write a program that uses our algorithm and generates the 3D voxel display. Here is where all the details come into play. Before we dive into the voxel display generator and the main control module, let's cover a quick check list of what our code needs to do. First, we need to generate height data and a color map, and we need to encode them in some common format such as .BMP or .PCX. The height data can be generated algorithmically, or you can download satellite height data — or if you have an extra $100.00, you can get a copy of VistaPro 4.0 (http://www.callamer.com/vrli/vp.html), which is what I did. VistaPro is a fractal terrain generator that creates height data and 3D views with full texturing of the height data. Thus, you can create any fractal landscape you wish. The program allows you to export the terrain height data as a 256-color gradient contour map (Figure 11a). We can read in this file and use the color index as the height of each voxel column.

Getting the texture information is a little tricky, but basically what you do is place the camera at a (x,y) position in the center of your world with a z altitude of 10,000 units or so. From this altitude, the 3D terrain looks like a 2D texture map. You can export it from VistaPro and then you can use this texture data as the texture color information for your rendering (Figure 11b). So what size should we make the height and color maps? I started off with 1,000 × 1,000-pixel height maps and then realized these files would be too big to download — about 2MB for everything. So I decided to go with 512 × 512 height and texture maps. This works out fine, looks good, and all the data for two worlds is less than 400k after compression.

Now that we know the how and what of the data, let's talk about the program. I've written two versions of the voxel terrain generator program; one with DirectX for Windows 95 (VOXELWIN.*) and the other in DOS 32 protected mode (VOXELD32.*). They're both way too long to list in their entirety here, so we'll only cover the main voxel rendering module and the main control module. Let's begin with the terrain-rendering function `Render_Terrain(...)`.

The function implements our algorithm almost verbatim. The only difference is that everything is in fixed-point math with a 20.12 format — this, of course, is for speed. While the floating-point processor may be faster than the integer processor for multiplication, when you convert floating-point values back to integers you're looking at 40-60 cycles per float. That's horrible. The key to rasterization algorithms is to convert the input into integers or fixed-point and, for the entire algorithm, perform all pixel rasterization with integers — don't mix integers and floats. You can use both, but keep them separate, or the implicit conversions and casts will kill your performance. Anyway, let's briefly cover the function and see what it does.

First, you must send the function the (x,y,z) world coordinates of the view point along with the pitch, yaw, and roll of the camera. Actually, the roll isn't used at all, and the pitch is really more the horizon than an angle. The last parameter is the destination buffer into which the scene frame is rendered. The scene frame is defined in the main program's define section to be 320 × 240. Once the function is entered, the first

LISTING 3.0. *The Terrain Rendering Algorithm.*

```
void Render_Terrain(int vp_x,
                    int vp_y,
                    int vp_z,
                    int vp_ang_x,
                    int vp_ang_y,
                    int vp_ang_z,
                    UCHAR *dest_buffer)
{
// this function renders the terrain at the given position and
// orientation

int xr,             // used to compute the point the ray
    yr,             // intersects the height data
    curr_column,    // current screen column being processed
    curr_step,      // current step ray is at
    raycast_ang,    // current angle of ray being cast
    dx,dy,dz,       // general deltas for ray to move from
                    // pt to pt
    curr_voxel_scale, // current scaling factor to draw each
                    // voxel line
    column_height,  // height of the column intersected and
                    // being rendered
    curr_row,       // number of rows processed in current
                    // column
    x_ray,y_ray,z_ray, // the position of the tip of the ray
    map_addr;       // temp var used to hold the addr of data
                    // bytes

UCHAR color,        // color of pixel being rendered
    *dest_column_ptr; // address screen pixel being rendered
```

thing we do is to convert everything to fixed point and compute the starting video address and ray casting angle.

Now that the starting position of the ray is computed, along with the deltas, the main stepping loop is entered, which will step out on each ray trajectory about 200 times. This procedure is also controlled by a `#define` in the main program. Now, for each iteration of the main stepping loop, the height of the voxel under the ray being cast is tested to see if it is higher than the current ray's z value or height. If so, the voxel column segment `while(...)` loop is entered, and the voxel column is drawn until its projection is as large as its height. Then the `while(...)` is exited, and the ray's position, along with the current voxel scale, is updated with the appropriate deltas. The function can literally be ripped out and used in any program as long as you include the `#define`s and the main global data access pointers `height_map_ptr` and `color_map_ptr`, which point to the data for the height field and texture map respectively. Let's see the function in action with a complete application (a really simple flight simulator).

The Killer App: A 1,000-Line Flight Sim

My original goal was to create a demo that allowed you to fly around the terrain data and change the viewing parameters. However, I got tired of controlling the flight, so I decided to write a little bit of AI to control an autopilot that would fly over the terrain when we left the controls alone for a couple of minutes. I originally wrote the demo in DirectX and Windows 95 (using Visual C++ 4.0), then I decided to port it to protected mode DOS (using Watcom 10.6), because half of the programmers that I know still don't have DirectX on their computers, let alone the SDK. (I never thought I would be porting Windows programs back to DOS.) All of this software is on the *Game Developer* web site.

To run to the DirectX version, you'll need the DirectX 3 or better run-time DLLs, and to run the DOS version, you'll only need the DOS extender, which is included in the files. Both demos will load the height and color data by default, so all you need to do is execute either application. If you want to supply your own height and color data, then the command line parameters are

- `VOXEL(WN1|WN2|D32).EXE height_data_file.bmp color_data_file.bmp`
- Both files must be on the command line.
- Both file names must be in 8.3 format.
- Files must be in 512 × 512 in 256 colors.
- Files must be .BMP format with no compression.

The main control module of the voxel engine is called `Game_Main(...)`. `Game_Main(...)` isn't too complex, but I want to peruse briefly its operation. The function is called once per frame and is responsible for taking the input, altering the flight model (or lack thereof), and rendering the terrain. You have control of the view point, its orientation, and the speed of motion. Also, the autopilot that I mentioned earlier will engage in about one minute if you don't touch any keys; to regain control, turn left or right. The autopilot selects random speeds, turns, and follows the height of the terrain, so it's a cool screen saver. The DirectX version of the code (also on the *Game Developer* web site), has a couple of Windows calls, such as the frame synchronization and the keyboard access. Otherwise, its no different from the DOS version.

LISTING 3.0 (CONT.). *The Terrain Rendering Algorithm.*

```
// convert needed vars to fixed point
vp_x = (vp_x << FIXP_SHIFT);
vp_y = (vp_y << FIXP_SHIFT);
vp_z = (vp_z << FIXP_SHIFT);

// push down destination buffer to bottom of screen
dest_buffer += (SCREEN_WIDTH * (SCREEN_HEIGHT-1));

// compute starting angle
raycast_ang = vp_ang_y + ANGLE_30;

// cast a ray for each column of the screen
for (curr_column=0; curr_column < SCREEN_WIDTH-1; curr_column++)
{
// seed starting point for cast
x_ray = vp_x;
y_ray = vp_y;
z_ray = vp_z;

// compute deltas to project ray at, note the spherical cancelation
// factor
dx = COS_LOOK(raycast_ang) << 1;
dy = SIN_LOOK(raycast_ang) << 1;

// dz is a bit complex, remember dz is the slope of the ray we are
// casting, therefore, we need to take into consideration the
// down angle, or x axis angle, the more we are looking down the
// larger the intial dz must be
dz = dslope * (vp_ang_x - SCREEN_HEIGHT);

// reset current voxel scale
curr_voxel_scale = 0;

// reset row
curr_row = 0;

// get starting address of bottom of current video column
dest_column_ptr = dest_buffer;

// enter into casting loop
for (curr_step = 0; curr_step < MAX_STEPS; curr_step++)
    {
    // compute pixel in height map to process
    // note that the ray is converted back to an int
    // and it is clipped to to stay positive and in range
    xr = (x_ray  >> FIXP_SHIFT);
    yr = (y_ray  >> FIXP_SHIFT);

    xr = (xr & (HFIELD_WIDTH-1));
    yr = (yr & (HFIELD_HEIGHT-1));

    map_addr = (xr + (yr << HFIELD_BIT_SHIFT));

    // get current height in height map, note the conversion to
    // fixed point and the added multiplication factor used to
    // scale the mountains
    column_height = (height_map_ptr[map_addr]
                  <<(FIXP_SHIFT+TERRAIN_SCALE_X2));

    // test if column height is greater than current voxel height
    // for current step from intial projection point
    if (column_height > z_ray)
    {
```

What Now?

Well you have a fast terrain generator with which you can alter terrain and texture data at will. I suggest that you try animating the height data, and/or the texture data. For example, you could create a tiled texture and height map and play animations on the data. You could animate real waves on the water, or little people running around the terrain. In addition, we've said nothing of lighting. You could create a light lookup table with 256 shades of each color by using a least-squares method. Then you could have another 2D light map that would describe the light intensity for each pixel in the texture map data. Then, as you're rendering, you could simply do one more look up per pixel to do lighting in real time. This would allow you to do lighting effects such as local spot lights just by painting into the light map each frame. Finally, there's no reason why you couldn't use a 3D polygon engine to place objects above the terrain. This way, you could make a poor man's FURY III. Have fun. ■

André LaMothe has written three best-selling game programming books and is currently CEO of Xtreme Games. He completed his latest project, REX BLADE, a DOOM-style game, in less than six months.There's not much left of him after that! You can reach him at necron@inow.com or http://www.rexblade.com.

LISTING 3 (CONT.). *The Terrain Rendering Algorithm.*

```
    // we know that we have intersected a voxel column, therefore
    // we must render it until we have drawn enough pixels on the
    // display such that their projection would be correct for the
    // height of this voxel column or until we have reached the top
    // of the screen

    // get the color for the voxel
    color = color_map_ptr[map_addr];

    // draw vertical column voxel
    while(1)
    {
    // draw a pixel
    *dest_column_ptr = color;

    // now we need to push the ray upward on z axis
    // so increment the slope
    dz+=dslope;

    // now translate the current z position of the ray by
    // the current voxel scale per unit
    z_ray+=curr_voxel_scale;

    // move up one video line
    dest_column_ptr-=SCREEN_WIDTH;

    // test if we are done with column
    if (++curr_row >= SCREEN_HEIGHT)
      {
      // force exit of outer stepping loop
      // cheezy, but better than GOTO!
      curr_step = MAX_STEPS;
      break;
      } // end if

    // test if we can break out of the loop
    if (z_ray > column_height) break;

    } // end while

    } // end if

    // update the position of the ray
    x_ray+=dx;
    y_ray+=dy;
    z_ray+=dz;

    // update the current voxel scale, remember each step out
    // means the scale increases by the delta scale
    curr_voxel_scale+=dslope;

    } // end for curr_step

    // advance video pointer to bottom of next column
    dest_buffer++;

    // advance to next angle
    raycast_ang--;

} // end for curr_col

} // end Render_Terrain
```