LingoLand: Simple 3D Terrain Simulation in Lingo.

Andrew M Phelps

Information Technology Dept.
College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY, 14623
http://andysgi.rit.edu/

Abstract

This article describes the implementation of a simple terrain generation system in Lingo. This material was used to teach the basics of lists and objects at the Rochester Institute of Technology to a graduate class of students pursuing a Multimedia Programming track. As such, while it implements a fully featured 3D Engine, the engine is not optimized to its fullest extent, and the data structures used are the most logical, not necessarily the fastest. Optimization techniques will be discussed at the end of this document, and references given to faster simulation environments written in Lingo.

In sprite of this focus, the engine does perform soundly, offering implementation of traditional computer graphics approaches to rotation, translation, and scale as well as basic lighting. Additionally it offers a trails based solution to overcome the limitation of the upper limit of a thousand sprites in Macromedia's Director. Finally, this engine could very easily be combined with the file saving system presented by this author on another project [Phelps, February 2001] and was also used as a simulation environment for experimentation with Artificial Life and Genetic Algorithms within the Director framework [Phelps & Kunkle forthcoming]

1 3D ENGINE IMPLEMENTATION

1.1 TUTORIAL FILE SETUP

Terrain simulation has been a long standing issue in the computer graphics community, with applications ranging from simple game worlds to advanced applications that simulate terrain based off of large data sets like satellite mapping data. This project seeks to implement absolutely none of that complexity, instead it is concerned with rendering a very simple randomly generated terrain, controlled by a few simple variables that are set by the user, primarily for students writing their first simulation environment.

The first thing to do is download the files associated with this document and open the Director movie. Play it. Click the 'Make Land' button until you get one you like the lay of. Experiment with moving the light around. Click the 'subdivide surface' button, and watch as it divides the surface to display at a finer resolution. You should see something like the screen depicted in Figure 1, although your actual results will depend on light placement and the original terrain, which is random.

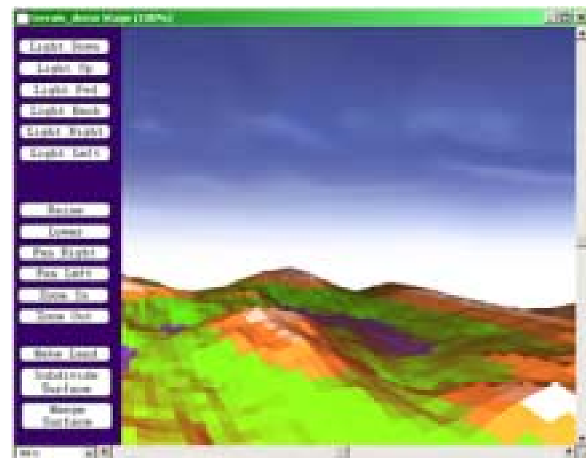

Figure 1: Terrain with 4 subdivisions.

Next, you can achieve very different effects by creating the random land point generation at a subdivision level other than '1'. To accomplish this restart the movie if necessary, click the 'subdivide' button, and then click 'Make Land'. Note that the engine will create the new land at this higher level of detail, producing a more turbulent surface. Finally, experiment with the global variables that control the simulation to get a feel for how the engine works and its capabilities. These variables control the way that the engine performs, and the type of

land that is produced. A table listing these values is provided for reference in Table 1.

Table 1: Engine Control Variables

VARIABLE	DESCRIPTION
max_row max_column	The size of the original land matrix. This is the number of 'points' in X and Z that you see the original land created with when the movie starts.
square_size	Size in pixels of the original tiles, the space between the points defined above.
min_height max_height	The minimum and maximum heights allowable for random generation of land elevation. Values closer together will produce rolling hills, disparate values produce mountainous regions.
sea_level rock_level snow_level	Values relative to min and max height that dictate when to change the base color of the ground (below sea level is rendered blue, etc).
sub_var	Original level of surface subdivision, almost always set to 1.
gSun	The primary light object, see section 2.2 for details.
gCamera	The viewpoint for the scene, see section 1.1 and 1.2 for details.
gZoom	Original zoom level used by the projection matrix relative to the camera.
gAmbient	Base level of ambient light. 0 will leave terrain to be lit only by the sun, 1.0 will result in solid white tiles (maximum saturation).

1.2 THE VECTOR OBJECT

Now that we have seen the engine in action, how does it work? The setup for this entire simulation is based on two object, a VBLF and properties of that object of type vector. We will dissect the vector object first. This object holds the x, y, and z coordinates of a point in three-dimensional space, and also contains the methods necessary to transform that point using standard math familiar to those with a computer graphics background. Therefore, these methods will not be discussed as they are based on the standard mathematical formalisms inherent to the filed [Foley et al, 1987, 1996].

The vector object stores its coordinates in the properties pX, pY, and pZ as floating point values. Integer values could be used for optimization purposes, with little to no visual effect, the engine is left un-optimized to show its capability for floating point precision.

The only other responsibility of a vector object is to provide the methods inherent to vectors as mathematical constructs, namely dot product, vector addition, subtraction, scale, normalization etc. [Hall 1999]. These methods are described in Table 2. While the basis of this 3D engine is being glossed over here, more detail is provided in the complete engine description [Kurtz & Phelps, forthcoming 2001].

Table 2: Vector Object Methods

METHOD	DESCRIPTION
new	Create a vector.
print	Print the X,Y,Z coordinates.
initialize	Reset vector to new coordinates.
cross	Take the cross product of two vectors.
unitize	Unitize the vector.
scale	Uniformly scale the vector by some value.
copyOf	Create a copy of the vector.
addvec	Add a vector to the vector this handler was called from.
rotate	Rotate the vector around an axis.
vector _offset	Reposition the vector relative to another vector.
average	Reposition the vector based on the average of two other vectors.
normalize	Normalize the vector
invert	Multiplies a vector by negative one.
abs_dist	Returns the absolute distance between two vectors.
abs_sum	Returns the sum of the absolute values for all three coordinates.

1.3 THE VECTOR BASED LIFE FORM

The second object of primary importance to this engine is the Vector Based Life Form (VBLF), so termed by professor Steve Kurtz at the Rochester Institute of Technology [Kurtz & Phelps, forthcoming]. Essentially a VBLF combines a group of three vectors with a series of methods that enable them manipulate the x,y,z (the fourth vector) of the VBLF. This enables the object to roll, pitch, yaw, and move relative to its current position, using the notion that these points carry their own local coordinate systems with them relative to their center. This is based in part on the original 2D Turtle Engine [Kurtz, forthcoming] which is in turn based on the Turtle Graphics work by M..I.T. in the late 1960's, which began with Seymour Papert and continued throughout their

LOGO based projects, and is now seen in the StarLogo project from the Epistemology group [MIT, 2001].

The VBLF also contains a method that allows the point to be projected onto the 2D stage, by using a reference to the `gCamera` object, which is in turn a slightly modified VBLF. This method sets the h and v properties of the object which represent the x and y on the stage, or, in the case that the VBLF is used to control a single sprite, the sprites `locH` and `locV`. The rest of the methods available to a VBLF are summarized in Table 3.

Table 3: VBLF Object Methods

METHOD	DESCRIPTION
new	Create a VBLF.
print	Print the location vector of a VBLF
RollLeft	Roll counter-clockwise along the pForward vector, X-Axis upon creation.
RollRight	Roll clockwise along the pForward vector, X-Axis upon creation.
PitchUp	Roll counter-clockwise along the pLeft vector, Z-Axis upon creation.
PitchDown	Roll clockwise along the pLeft vector, Z-Axis upon creation.
YawLeft	Roll counter-clockwise along the pUp vector, Y-Axis upon creation.
YawRight	Roll clockwise along the pUp vector, Y-Axis upon creation.
Move	Move a VBLF along its forward vector a unit equal to its speed property.
MoveTo	Reposition a VBLF to an absolute coordinate value.
Project	Reset the h and v properties of a VBLF so that it draws on the 2D stage correctly.

1.4 CREATING SURFACES

Now that you have at least a rudimentary understanding of the engine involved underneath the hood, the true work of this application begins, namely the creation of the land. This begins with the call to `createArray()` and `GeneratePoints()` in `StartMovie`. These handlers create a 2 dimensional list, each element of which contains a VBLF. These VBLF's are spaced apart by the `tile_size` global variable, and there are the number or rows and columns in the array as specified by `max_row` and `max_column` (see Table 1 for more detail).

Then the final call is to `RenderGround()` which is the primary method used in the application. This is the primary rendering loop of the application, and it should be noted that it is not built for performance at this time (see the Optimization discussion in section 4). This method makes calls to `RenderHeight()` which sets the base color of the tile based on elevation, `RenderLight()` which performs the lighting calculations which are described in the following section, and then sets the four points adjacent in the array to the corners of a quad, drawing the sprite to the stage (see figure 2).

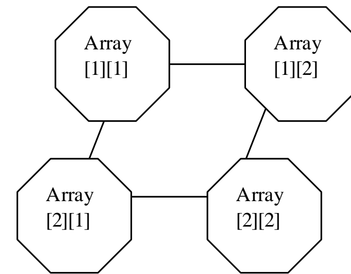

Figure 2: Array lookups to produce quad surface, order of points past to `sprite.quad` is clockwise from top left.

It does this by passing in the VBLF's h and v properties to a sprite's `quad()` function, and setting that sprite's color to the calculated RGB property. This is generally the easiest way to store the data for first time engine developers, although not the best way to store the data for efficient rendering.

1.5 SUBDIVIDING SURFACES

After all of this setup work and the initial plot to the screen, the application sits and waits for input from the user. Most of the other controls, such as moving the light, the camera, etc, simple alter the property of the light or camera object as defined in Table 1 and the re-call the `RenderGround()` method. Subdividing the surface, however requires a little more work: namely the insertion of new data points within the `PointArray`.

This insertion is relatively simple to visualize given the following example of what we mean by subdivision of the surface. Essentially every quad is split into 4 new quads. This involves first placing a new data point between every existing data point in each row of the array, averaging the height of the points to the left and right to produce the new height value. Next, a new row is inserted between every row in the array, taking getting its height values from the average or the points before and after the new points vertically in the column. Through this algorithm (see commented code for details) the original 4 points are now a tighter mesh of 9 points, producing 4 quads from 1. To produce some slight variation in the land, these averages are modified by a diminishing random value to produce more realistic results.

Subdividing the surface also produced a decidedly problematic issue when it was first implemented, in that after relatively few subdivision level (2 or 3), the engine

would quickly exceed the available number of sprites, which is currently has a hard ceiling of one thousand. To get around this issue, the engine uses one row of sprites with trails turned on, effectively using the sprite as a "brush" to "stamp out" the quads and then reuse the stamp to draw the next one. You could conceivable draw the entire scene with one sprite, the demo packaged with this paper reuses rows only, not single sprites, so it is possible to eventually run out of sprites, when you reach a 1000 x 1000 array or greater. Since this will most likely run so slowly as to be ridiculous this is not seen as a drawback of the demo.

2 LIGHTING

2.1 BASIC LIGHTING THEORY

Computer lighting models have been around for years, and nearly all of them start with the same first steps: ours is no exception. First, the engine calculates a normal to the surface, and then compares this normal to a vector from the same surface to the light source (see figure 3).

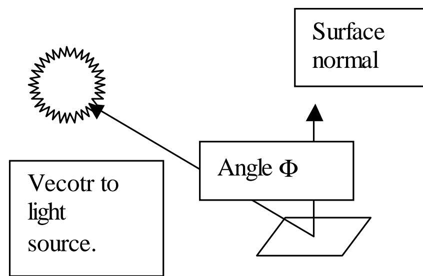

Figure 3: Simplistic Lighting Theory based on surface normals.

The engine then computes the color based on the distance of the surface away from the light, and the angle between the two vectors described above, and most engines add in the ability to add some small percentage of color based on an ambient light value [Foley et al 1987, 1996]. Thus, the basic mathematical implementation of a lighting system (for a single light source) is relatively simple.

2.2 LIGHTING THE LAND

There are, however, some rather particular features of our Lingo engine that make the lighting calculations slightly more complicated, namely the fact that we are using quads instead of triangular surfaces. This is unfortunate in that while a triangular face can produce a single correct normal to the surface, a quad can make no such guarantee, because there is no mathematical certainty [Edgerton, Hall, 1999] that all four points lie along the same plane, indeed in our engine we almost would desire that they do not if we are attempting to get mountain tops that end in sharp peaks.

This engine takes the following solution: it divides the quad into two triangles, calculates the normal for each triangle, and uses the average of those vectors as the normal for the lighting calculation. This is certainly not an optimal solution the engine was attempting a smooth shaded look which involved lighting calculation within the quad, however since each quad is one and only one color, this has the visual effect of being believable, if not perfect. A more robust engine could in essence use quads to simulate triangles, thus wasting a data point, or any number of other solutions, including the notion of calculating the light color at the points and blending inward towards the center of the face.

In any event, we now have the angle of incidence between the surface normal and the vector that points to the light source, in our case the `gSun` object. Using this angle, we compute the color of the tile, modifying the base color we receive from elevation based on the ambient light present, and the properties of the light describing its color, intensity, and decay (see commented code for more detail on the exact calculation). The basic formula for these calculations is described in Foley's definitive work on the subject [Foley et al, 1987, 1996]. The lighting can be used to produce a variety of effects, based on placement and color (see Figure 4).

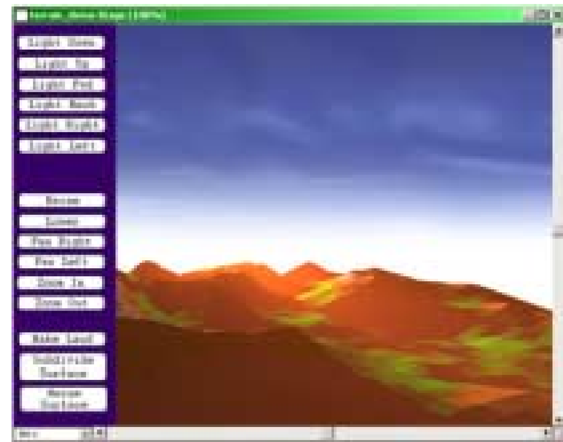

Figure 4: Land at Sunset (gSun at low Y large negative Z) at 5 subdivision.

3 FUTURE OPTIMIZATION

3.1 DATA STRUCTURES AND OBJECT METHODS

This engine is not designed for speed. More to the point, it is not *structured* for speed. There are a variety of reasons for this, not the least of which is legibility for students experimenting with systems like this for thie first time. One of the primary bottlenecks in this application is the calling structure for the `VBLF.project()` method. This is caused by the delay in Director's interpretation of

calling methods of objects, which is greatly exaggerated by the fact that this call is placed in a double repeat loop. The following syntax inside the repeat loop would offer a significant speed increase (see figure 5).

```
-- current technique
-- QuadPoint1.project()

-- new technique
-- call (#project, QuadPoint1)

-- optimal technique (no repeat loop)
-- call (#project, list_of_objects)
```

Figure 5: Enhanced method-calling structure using alternate Lingo syntax.

While this is a large speed increase, the optimal method to use the call #handler syntax for multiple objects is not to use the call function directly but instead to pass to the call function a list of object. This can offer very significant performance savings, however, it involves a complete reorganization of the data structures that we use to house the VBLF's, since it requires a flat list as an argument. Likewise, referencing properties from outside of an object is slower than referencing the same property internal to the object, much of the code could be reorganized to fit more directly within the objects and thus cut down on the number of external references. This could have significant impact with the lighting routines and the way the render functions are separated (and whether or not they should be).

4 CONCLUSIONS

In this article we have used a simple 3D engine to create a simple random surface, used lighting algorithms to modify the color of that surface, and used that surface to simulate a simple terrain. This engine, or any similar scale application is capable of much more, this application serves as a basis for understanding the concepts of terrain simulation, without the complexity overhead of a more traditional language or terrain-generating algorithm. This work is not highly optimized; it should instead serve as a clear, precise example for others to follow and extend to a more fully featured engine. This application is in turn only one possible extension of the engine created by this author, another more robust version of which is also planned for publication.

5 FUTURE WORK

My students have already extended this project to include a variety of functions: namely saving and loading terrain files, using a bitmap as an elevation map to create the terrain (using Imaging Lingo's `getPixel()` call), and massaging the engine to perform up to 6 times as fast. Additionally, some of them allowed the creation of 'terrain worlds' on cubes, rectangles, spheres, etc., and at least two of them managed to implement a simple texture mapping system across multiple quads. Most all of them created control panels external to the original window to avoid the clipping problems present in this demo.

One area in which this application could be extended would be in multi-user functionality. Currently there is no support in the engine for this capability, but it is high on the list of desired functionality that most people seem to want out of graphics engines. If the user has the ability to control light placement, tile size, etc, then this present a technical issue in synching the two. Additional support for displaying real time graphics (which would most likely break the 'trails' solution) is inherent to the engine, however as discussed significant portions of the code would need to be reworked.

Acknowledgments

I would like to acknowledge my colleagues at the Rochester Institute of Technology, and in particular Professor Steve Kurtz, who shares my interest and passion for writing graphics programs and extending Lingo into this arena. I would like to publicly acknowledge all the support that this author has received on the DirGames-L mailing list, and thank the University of Georgia for hosting the list. Many thanks to Barry Swan, NoiseCrime, and others who have inspired us all to push Director to its limits: code long, code late, code happy.

References

P.A. Edgerton and W. S. Hall. Computer Graphics: Mathematical First Steps. (1999). Prentice Hall Europe:Essex, England.

James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes. (1987, 1996 2nd revised printing).*Computer Graphics Principles and Practice – 2nd Edition in C*. The Systems Programming Series. Washington, DC: Spartan Books.

MIT Epistomology Group, Media Lab, Massechusets Institute of Technology. *Introduction to StarLogo*. (2001) Online: http://el.www.media.mit.edu/groups/el/Projects/starlogo/index.html.

Steven Kurtz. Turtle World. Forthcoming. Featured Article in Using Director – Director Online. Online: http://www.director-online.com/

Steven Kurtz and Andrew M Phelps. *Vector Based Life Forms, a 3D Engine Based on Turtles*. Forthcoming. Featured Article in Using Director – Director Online. Online: http://www.director-online.com/

Andrew M. Phelps, Daniel R. Kunkle. *Teaching Old Turtles New Tricks: Artificial Life Simulations in Lingo*. Forthcoming. Featured Article in Using Director – Director Online. Online: http://www.director-online.com/

Andrew M. Phelps. *Perspective Based Lingo Mazes: The Director Dungeon Crawl*. February 2001. Featured Article in Using Director – Director Online. Online: http://www.director-online.com/

Annotated Bibliography

[AA] Anti-Aliasing; [E] Everything; [L] Lingo based 3D Code; [M] Mathematics;

1. Cole, David. (2000) Dave's 3D Engine v. 7. Online. http://www.dubbus.com/devnull/3D. [M][L]
2. Edgerton, P.A & W.S. Hall. (1999) *Computer Graphics: Mathematical First Steps* Essex, England: Prentice Hall. [M]
3. Lithium. (1999-2001) Three Dimensional Rotations. Online. http://www.gamedev.net/ [M]
4. McNally, Seumas. (1999-2001) 3D Matrix Math Demystified. Online. http://www.gamedev.net/ [M]
5. Perez, Adrian, Dan Royer. *Advanced 3-D Game Programming Using Direct X 7.0*. Plano, Texas: Wordware Publishing. [M][R]
6. Rodgers, David F. And J.Alan Adams. (1976, 1990) Mathematical Elements for Computer Graphics 2nd Ed. New York, New York: McGraw Hill. [M]
7. Rodgers, David F. (1985) *Procedural Elements for Computer Graphics*. New York, New York: McGraw Hill. [M]
8. Swan, Barry. (2000) T3D Engine. Online. http://www.theburrow.co.uk/t3dtesters/. [L]
9. Tamahori, Che. (1999) How to Cook 3D in Director. Online. http://www.sfx.co.nz/tamahori/thought/shock_3d_howto.html. [L][M]
10. Watt, Alan and Fabio Policarpo. (2001) *3D Games: Real Time Rendering and Software Technology*. New York, New York: Addison-Wesley ACM Press. [M][R]
11. Zavatone, Alex. *Inside Zavs Brain: 3D Director*. Online. www.director-online.com/accessArticle.cfm?id=286. [L]