# Raytracing in Lingo: Silly Spheres with Sexy Shadows

**Andrew M Phelps**

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

## Abstract

This article describes the implementation of a fully featured ray-tracing system within the Director programming environment, complete with shadows, reflection, specular highlights, and transparency. Much of this engine is adapted from earlier work by Stephens [1997] with additional feature set and reorganization of code by the author, based primarily on graphics formalisms presented in what now are classic texts by Glassner [1989] and Foley [1987, 1996]. The discussion of formal (and well published) algorithms will be kept to a minimum, except where the particular implementation in the Lingo language is noteworthy. Further, this article assumes basic knowledge of the Lingo programming environment, Object Oriented Programming (OOP) techniques of the Lingo language, and basic mastery of programming constructs such as functions and if/then/else logic.

The article is split into 4 major sections, the first describing the mathematical basics of the ray-tracer itself, the second concerned with the display mechanism of a render window and the process of color calculation, the third which discusses advanced or additional growth of the algorithms for additional features, and the fourth which talks about sampling mechanisms. Also included is a 'Future Work' section describing directions for this project to grow.

## 1 BASIC RAYTRACER THEORY AND IMPLEMENTATION

### 1.1 TUTORIAL FILE SETUP

Ray-tracing has now been viable in computer graphics for quite some time, with improvements to renderers happing at the large firms such as Pixar and Alias | Wavefront almost daily, and advanced techniques like motion-blurring and super-sampling quite common. This project implements a simple ray-tracer, and is intended for students and other professionals to learn the basic fundamentals of how the theory works. As such, this is essentially a throwback to the ray-tracers of the early 1990's, and it must be noted that it performs like one. Ray-tracing in general, because of the number of calculations involved, is a relatively slow process, and this issue is exacerbated by the fact that Lingo is an interpreted language. Nonetheless, this demo hopefully provides a clear and concise picture of the inner-workings of a rendering system, if not in real time.

The first thing to do is to play with the demo files. Download and unzip the files into the same directory, open the raytracer.dir file, start the movie, and type in the message window the following command:

```
-- Welcome to Director --
RayTrace(300,300,4,0.75)
```

Press 'Enter'. After a short amount of time you will see a window pop up labeled 'Render Window', and it will begin to trace the default scene line by line (referred to in the formal literature as a 'scanline renderer'). The command you gave renders a scene 300 pixels wide, 300 pixels high, with 4 levels of ray reflection, and a 0.75 pixel blur. Note the time it takes on your machine, it is certainly not a real-time engine.

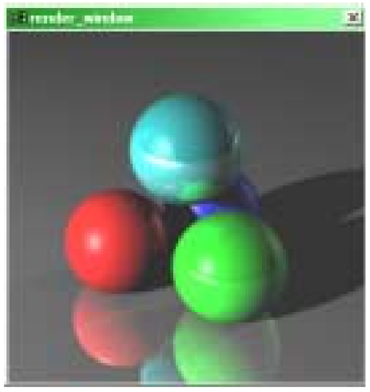

Figure 1: Sample output from Raytracer.dir

## 1.2 BASIC MATHEMATICAL TRANSFORMS

The basis of the renderer comes from defining a three-dimensional world. Most of this implementation is based on the discussion of 3D Transforms by R. Stevens [1997]. While Stevens is implementing in Visual Basic, it is interesting to note that most of the implementations are completely language independent because they are mathematically based, only the array syntax changes for specifying a matrix.

Essentially this program begins with the basic object of a Point, constructed in the `Point3D` function under the "Matricies" script. The other key method in that script is the Mat3Identity function, which creates the classic Identity Matrix [1987]. Manipulation and projections of the point then involves using the rest of the function in this script to transform and project the point as needed by the engine. This is accomplished, in essence, by creating the points, creating a matrix, and then using the `Mat3Apply` or `Mat3ApplyFull` function depending on whether or not the fourth (scale) value of a point array is 1. If a point has a scale of 1, then it is, by definition, using global world coordinates, if it has a value other than 1, then it needs to be re-expressed using a value of 1 ('re-normalized' in the commented code), and then transformed.

A number of additional methods are built up in the 'Matricies' script that combine these function to produce the actions common to 3D worlds, namely transform, rotation, and scale. This script also provides the functionality for points to be projected onto a 2D plane, using the `Mat3Project` function, or, using a wrapper for spherical coordinates, the `Mat3PProject` function. These, combined with the global variable declarations in the 'StartMovie' script, provide the basis for the world. Most of these functions contain associated comment, where V is an argument of type `Point3D`, and M is an argument of type `Matrix`. For a more in-depth explanation of the math involved in creating a world such as this, refer to the references section at the end of this document, as there are countless materials on 3D graphics, matrices, and graphics engines available, many implemented in Lingo already, and for additional documentation and geometric proofs of these precise methods, see Stephens original derivations [1997].

## 1.3 SCENE OBJECTS AND OBJECT HEIRARCHY

The next step in creating the application is the creation of objects in the scene, namely the collection of spheres and the ground plane. Each object is created in a script name obj(*name_of_object*), so a sphere would be created in 'ObjSphere', a plane in 'ObjPlane', etc. Additionally, each of these objects set as its ancestor an object of type ObjGeneric (see figure 2). This class encapsulates the basic methods for setting attributes for the lighting calculations, and also sets Boolean values for coding clarity with regard to `IsReflective` and `IsTransparent`. Each object implements four functions, Initialize, Apply, RayDistance, and HitColor. Initialize should be fairly straightforward; it is called from the 'DrawData' script to instantiate the object. Incidentally, objects are organized on an Objects list, so the can be looped through later. The `Apply` function takes an argument of type Matrix, and applies that matrix to the coordinates of the object using the `Mat3Apply` function described earlier. The object stores these transformed coordinates in a separate `Point3D` object. When the movie is started, the 'DrawData' script is called. In this script, projecting the camera onto the Stage creates a transform matrix, and this transform is then applied to all of the objects in the list.

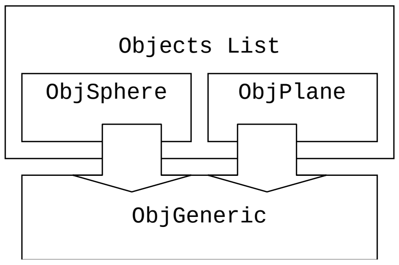

Figure 2: Object hierarchy of the scene-graph.

Objects are loaded with values to create different colors and effects, according to the following table. Each of these properties is set before the object is added to the list.

Table 1: Attributes of Scene Objects

| PROPERTY | DESCRIPTION |
|---|---|
| IsTransparent | Boolean Flag, is the object Transparent. |
| IsReflective | Boolean Flag, is the object Reflective. |
| PSpecN, pKs | Attributes for specular highlights. |
| pHitX, Y, Z | Coordinate data for where an object is hit with a ray. |
| Kdr, Kdg, Kdb | Red, Green and Blue components for diffuse lighting calculation. |
| Kar, Kag, Kab | Red, Green, and Blue components for ambient light calculation. |
| Krr, Krg, Krb | Red, Green, and Blue components for reflected light calculation. |
| Ktr, Ktg, Ktb | Red, Green, and Blue components for transparent light calculation. |
| n1, n2, nt | Constants for reflection and refraction calculations. |

Once these objects exist in their list, and hold their project coordinates, we are ready to move on in creating the rest of the elements necessary to complete the scene, namely the lights and the viewpoint.

### 1.4 SCENE LIGHTS AND LIGHTING

The lights in the scene are very simple objects, that hold 2 Point3D values, one for the original coordinates, and one for the coordinates once projected by the viewpoint matrix. This is different from many lighting systems that store color and decay information directly within the lights themselves. In the effort of simplicity, this system uses single global values to control much of the lights behavior, namely the ambient values, and the `LightKDist` value, which is basically a mechanism for how 'bright' the light will appear on object, and how quickly it will fall to shadow. The following table descripbes the lighting values and their effects on the system.

Table 2: Global Lighting Variables

| VARIABLE | DESCRIPTION |
|---|---|
| LightKdist | Numerical integer describing the fall. |
| LightIar, LightIab, LightIag | Red,Green,and Blue attributes for ambient light. |
| BackR, BackG, BackB | Red, Green, and Blue components of background color (before applying ambient contribution). |

### 1.5 CAMERAS AND CLIPPING PLANES

There really in essence is not a 'camera' in the sense of most traditional 3D packages. There is simply a viewpoint that is expressed in spherical coordinates, and a matrix that is created from projecting this point onto the 2D viewing plane (in our case, the Render Window). Additionally, it is possible to move the point that the viewpoint is 'looking at' by changing the Focus(X,Y,Z) coordinates. All of the viewpoint variables are set in the 'StartMovie' script, the Projector Matrix is created by calling `Mat3Pproject` and feeding it the coordinates of the viewpoint, the coordinates of the viewpoint focus, and the nature of the world coordinate system (in this case we are using a 'Y-up' world, where the Y axis points up, the X axis moves across the screen, and the Z axis comes out towards the viewer).

### 1.6 WORLD CREATION : PUTTING IT ALL TOGETHER

So in essence these are the parts of the program that are responsible for the creation of the scene and getting the scene to a point that it can be rendered at the desired resolution. All of this occurs without any input from the user of the software, and occurs in the modules of the program as demonstrated in figure 3. First, global variables are set that describe the viewpoint and the center of the scene, scene objects are instantiated, as well as light objects, and both objects and lights are held in their respective lists. This occurs inside the DrawData script. All of the objects and lights have their respective coordinates projected and stored in their internal structures as the TransPoint[x] property. Following through the code should be relatively straightforward based on the flowchart and the associated comments in the lingo code. At the end of the completion of the DrawData script, all of the required globals and objects for rendering are ready for rendering, and the movie awaits user input, namely the `RayTrace( )` command that you gave when starting the movie.

Table 3: Global Viewpoint Variables

| VARIABLE | DESCRIPTION |
|---|---|
| EyeR | The first part of the viewpoint coordinates, expressed in spherical coordinates. This value represents the distance from the origin of the world (set in Focus X, Y and Z) |
| EyePhi | This is the angle between the viewpoint and the XZ plane (assuming a Y-up world, which this is). |
| EyeTheta | The angle representing rotation of the viewpoint around the Y axis. |
| FocusX, FocusY, FocusZ | Point in 3D space that the viewpoint is centered on. . |
| Projector | The Matrix returned by projecting the coordinates above onto the viewing plane. This matrix is then applied to all objects and lights such that when drawn on the plane they appear in proper perspctive. |

## 2 THE RENDER WINDOW AND PIXEL PLOTTING

### 2.1 RENDER WINDOW AND LINGO APAPTION TECHNIQUE

Typical ray-tracers have what is called a 'render window' that hold and display the scene as it is being calculated. This movie tries to emulate that in Director using a separate Movie In A Window (MIAW). This is a very simple movie, containing only 2 scripts and 1 sprite. The first script is what I consider a standard way to keep a movie running on a single frame, which is the standard 'go to the frame loop' located in the 'go_frame_loop' script. The sprite is a cast member who is exactly 1 x 1 pixel square, and whose color is solid black (rgb 0,0,0).

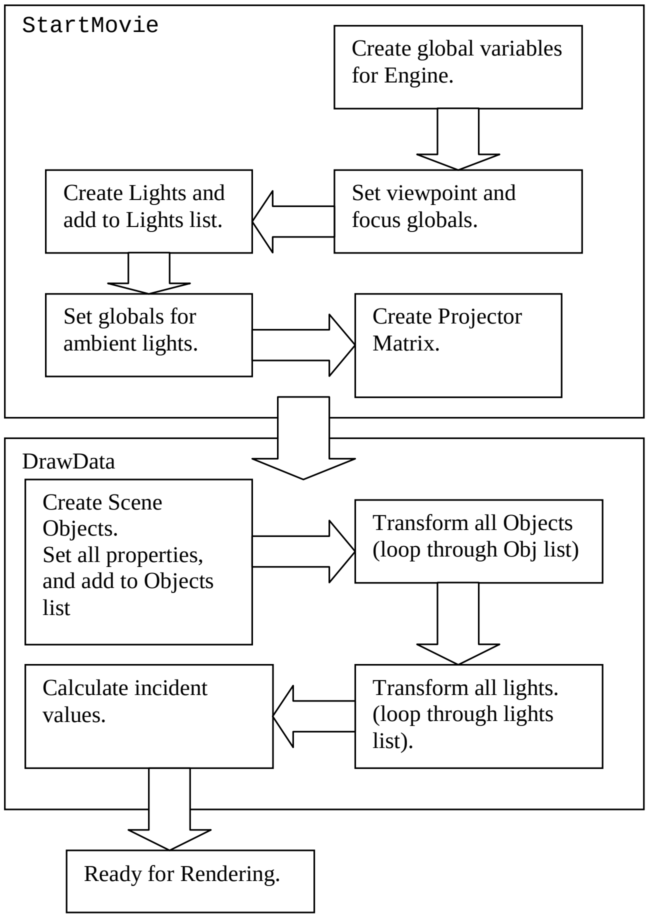
StartMovie
Create global variables for Engine.
Set viewpoint and focus globals.
Create Lights and add to Lights list.
Set globals for ambient lights.
Create Projector Matrix.
DrawData
Create Scene Objects. Set all properties, and add to Objects list
Transform all Objects (loop through Obj list)
Transform all lights. (loop through lights list).
Calculate incident values.
Ready for Rendering.

Figure 3: Flowchart of world creation and startup.

The third and final script is a method that allows a given pixel to be plotted at a precise RGB value. This is accomplished in Lingo by allowing the script to position the locH and locV of the pixel sprite (in essence placing it on the stage) and setting the RGB color value of this pixel (see figure 4). Because the image size is variable, this movie uses the same sprite over and over again rather that a separate sprite for each pixel. To achieve this illusion, you will notice that the pixel sprite has **trails turned on**, thus presenting the illusion that we are drawing separate pixels to the render window when in fact we are drawing the same one over and over. The movie starts with the pixel sprite located off the stage.

```
on plot_pixel x, y, r, g, b
  Sprite(1).locH = x
  Sprite(1).locV = y
  Sprite(1).color = rgb(r,g,b)
  updatestage
end plot_pixel
```

Figure 4: sample code to plot a pixel on the render window stage.

The main rendering loop (called from the `RayTrace` command you issues at the start of this article) uses a double loop to plot every pixel on in the render view, from left to right, top to bottom. This produces the effect of the image being 'built in lines' and is indicative of this generation of ray-tracers, where each pixel value is sampled individually.

## 2.2 THE RENDERING LOOP

The real underlying issue of course, is not looping through the pixels and plotting them but in knowing which color to plot them with. This is finally the intersection of the 3D world with the 2D image plane, and the idea is to some degree a simple one. What this program does, in its simplest form is send out rays from the center of projection, through each pixel, and into the scene. The program then checks to see how far into the scene the ray travels, if it reaches an arbitrary infinite value, then it can be assumed that it did not hit anything, and the background should be rendered. If the distance comes back as less than infinite, then the color is calculated based on which object was hit and how that object was lit (see figure 5).

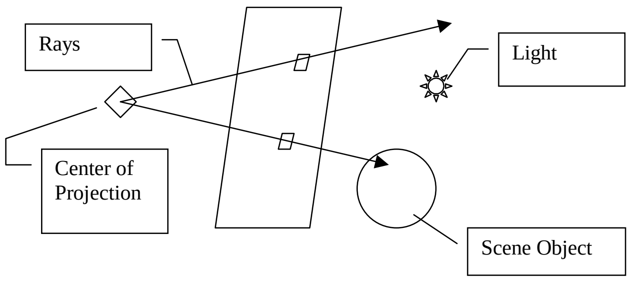

Figure 5: Basic Ray-Tracing Theory.

This system then, has the added bonus of built in overlapping and culling of objects, since rays can hit one and only one object first. The process of calculating the ray distance, and the color of an object hit, is exceedingly intensive, and is implemented in the sample file as follows:

For each pixel, a call is made to the `RayColor` function, which in turn calls the `TraceRay` function. `RayColor` is used to average the results from multiple calls to `TraceRay`, sampling a pixel more than once based on the number of lights in the scene. TraceRay is truly the start of the ray-tracing process, as it loops through each object in the scene, and recursively tries to find the shortest distance by using the `RayDistance` method of each object on the list. The `RayDistance` method is built using the classic quadratic for this purpose, derived in full in Foley's master work [1987, 1996] and implemented by Stevens [1997] and many others. The comments in this section of code should explain to some degree the purpose of each calculation.

Once the minimum distance is obtained, and assuming that distance is not infinite, the object that produced the closest intersection with regards to the center of projection is now the 'active object' as any other object encountered would, by default, be behind the active one. The `TraceRay` function then calls this objects `HitColor` routine, to determine, at that point of intersection, which is stored in the objects `pHitY`, `pHitY`, and `pHitZ` properties from the previous call to `RayDistance`. `HitColor` calculates the color of that point be determining what color the object would be at that location, again through algorithms well published and documented [1997]. The first step in creating the illusion is tracing a ray from the point of intersection to the light source, and seeing if it gets there without interruption. If it does, then it can be assumed that the light is shining on the surface, if it does not then the surface is in shadow, and as such there should be no diffuse value other than the standard ambient value for the scene. Assuming that the object is lit, the process for diffuse color is very simple, it involves calculating the normal of the surface at point of intersection, calculating a vector from that point to the light source, and then examining the angle of difference between these two vectors. If the angle is large, then the surface faces away from the light and is likely in very dim, otherwise, the reverse is true. To measure this, the traditional approach is to take the dot product of the two vectors and multiply them by the base constant of the color of the object, modifying the colors both positively and negatively to produce shading.

The specular highlights found in the system are simply one further step along this idea, using a vector calculated back from to the viewpoint from the point of intersection, and using the dot product of this vector and a vector in the mirror direction to the one described above, multiplied by a constant to determine how 'shiny' the object is. This produces the 'plastic look' common to ray tracing systems before the addition of texture mapping techniques.

## 3 ADDITIONAL FEATURE SET

### 3.1 REFLECTION

Reflection is usually the first 'goodie' that people implement in systems such as these, and it is implemented faithfully here. The reason for this is that it is the first logical follow-on to the idea of tracing rays. Essentially, instead of only calculating the color at the point where a ray hits an object, a new ray from that 'hit location' is traced back through the scene to see if it intersects another object. If another object is encountered, that color is calculated and averaged with the first color using a constant that is set to determine how reflective we want the object to look. The greater the constant, the greater the effect of the second color, and thus the less influence the first has. To some degree, this mimics the way light is bent by reflective objects in the real world, and can provide a convincing illusion if the constants and lighting are set at believable levels [1989].

To accomplish this, this program implements a technique often found in graphics systems and more traditional computer science program architecture, but one that this author has rarely used in standard multimedia programming: recursion. The user of this system will pass in the maximum number of times a ray is allowed to 'bounce', referred to as 'ray-depth'. This attribute is passed from the `TraceRay` function to an object's `HitColor` method when we calculate the color of an initial ray intersection. The program will then (assuming that the depth is not infinite) reduce this value by 1 and call `TraceRay` again, this time with a starting location not from the center of projection, but from the last intersection of ray and object. This continues (assuming that none of the rays run to infinity) until the ray-depth value is depleted to zero. Note, that this can drastically effect the overall performance of the system because tuning this value to a high number causes the number of calculations to grow, while there is very little visual effect beyond a relatively low number of recursive iterations. Also note the possibility that without the level being user set, the possibility for 2 perfectly parallel, perfectly reflective planes to trap a ray forever, and thus cause an infinite loop.

### 3.2 TRANSPARENCY & REFRACTION

The final 'illusion' presented in this demo is that of transparency, with refraction. This is achieved by allowing the ray to pass through the object, and adjusting the angle of the ray based on the inverse normal of the surface and the constants set for the index of refraction. The larger the difference between the first and second constants (n1 and n2) the more the rays will 'bend'. Stevens describes this process in much greater detail [1997], as does Foley [1987] and many other classic texts on computer graphics techniques.

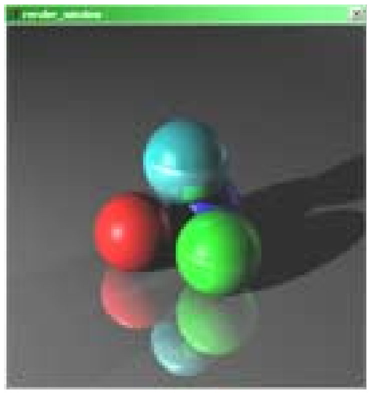

Figure 6: Pre-blur ray-tracer output.

## 4 PIXEL BLUR AND ANTI-ALIAS ROUTINES

### 4.1 THE NEED FOR SAMPLING

The need for sampling becomes apparent when considering an original render of the system (see figure 6). In this picture, the edges of the objects are 'popped out' against the scene, producing 'jaggies' along the edges of the objects. This occurs because a single pixel can be one and only one color value, and when calculating the pixels the system must choose weather a pixel lies inside or outside an object. If the edge of the object is at an angle or curves across the pixel grid, this produces the effect seen above, described by Glassner as 'Spatial Aliasing' [1989].

### 4.2 PIXEL BLUR ROUTINES

The solution to this issue, in its simplest form, is to sample a pixel more than once and then average the results of those samples together to get the final solution. In essence, if the pixel grid is of a finer resolution, then the appearance will be better and the aliasing effects minimized. This solution samples a pixel 5 times, once at its center and 4 times at an offset from the center, one in each direction. This offset is the blur arguments passed to the engine at render-time, and should be greater then zero, but not more than one full pixel, else stippling effects will occur. Note that (a) a system that only used additional sampling rays where it was deemed necessary because of a large variation in the initial sample set would be more effective [1989] and that (b) this solves the issue of single frame aliasing only, it does not solve motion based artifacts should multiple stills from this engine be played back in sequence [1987, 1997][1987].

## 5 CONCLUSIONS

This system is unique mainly in its implementation choice, it is intended as an introduction to the ideas of traditional computer graphics for a non-traditional audience. This paper and the associated application have hopefully provided the reader with a basic understanding of the parts of the system, and the resources and references to explore the formal literature on the subject in a meaningful way. Additionally, is has demonstrated as an aside that the Lingo language is fully capable of implementing the types of systems that are often regarded as a suitable 'level of difficulty' for undergraduate computer science or graphics courses, and that it has grown into a fully featured programming environment. While the performance of the system is certainly not ideal, this performance comes more from the algorithms used for calculation and sampling than any overall deficiency in the language. A much faster implementation could like be obtained with more modern algorithms and recursion techniques, however this may be counter productive to the audience, as it would likely be less understandable to a novice. Finally it has been suggested, and is interesting to this author, to combine this engine with the author's other work in artificial life to allow visualization of scenes that are allowed to evolve on their own through the use of genetic algorithms.

## 6 FUTURE WORK

This work is the first work in creating a whole-scale animation system in director. A real-time wire-frame modeler and time track are also underway. Obviously, such a system is not for production use, but would be a valuable educational tool in teaching students how these systems work, that they are not 'magic'. Additionally, this project serves as an introduction to graphics concepts and algorithms; from here students could easily begin research into real-time solutions in a variety of languages. I would like to add to this a more complete pixel sampling mechanism that discusses Stochastic sampling and weighting of differences for better edge resolution, as well as providing a performance increase. Also of primary import is also the inclusion of texture mapping support, in a most basic sense, to serve as an example for more complex engines. Finally, the ability to load and save scenes from text files in a pre-rendered state, possibly using the VRML or other simple format, coupled with the ability to save the rendered image in a .jpg or .gif format for use on the web, would further the benefit of this application as an educational tool.

**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. He has served as a capable sounding board and mentor for a number of years, and I thank him for it. I would also thank Rod Stevens for his excellent text describing the implementation of graphics algorithms in Visual Basic. His text has served as an invaluable resource since I first acquired my copy in 1999. Finally I would thank Professor Ronald Coleman at BGSU for getting this artist interested in graphics programming in the first place.

**References**

R. Stevens (1997). *Visual Basic Graphics Programming*. 397-617. New York, NY: John Wiley & Sons, Inc.

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.

Andrew S. Glassner, editor. (1989). *An Introduction to Ray Tracing*. San Francisco, California: Morgan Kaufman.

## Annotated Bibliography

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

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