Simulating Arcade Style Explosions in Game Environments using 2D Sprite Positioning L-Systems and Random Tree Structures.

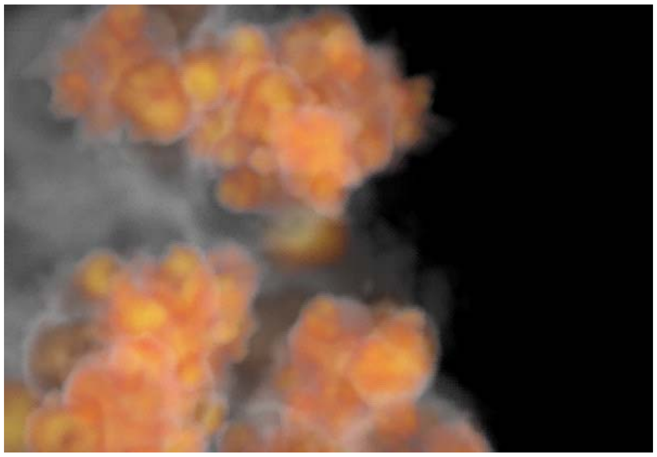
Andrew M Phelps
Assistant Professor
Information Technology Dept.
Rochester Institute of Technology
Rochester, NY, 14623
http://andysgi.rit.edu/
amp@it.rit.edu

Aaron S Cloutier
Multimedia Graduate Assistant
Information Technology Dept.
Rochester Institute of Technology
Rochester, NY, 14623
ascloutier@mac.com

Abstract

There are several instances in arcade games and similar entertainment titles where it is desirable to simulate the expansion of an explosion or similar event (plasma ring, dust cloud, etc). While some modern engines can devote processor cycles to calculating the real-time expansion of a simple cloud, more often than not in web and downloadable games the resources are not available. We describe a reasonable approximation of expansion that seems to stand up to the 'visibility test', meaning that users find such effects to be believable within the game world, and richer than a single animated sequence. These techniques are in no way physically accurate to the expansion of a cloud due to explosive force, nor are they based on any physical model whatsoever.

Instead, we propose the use of classical tree algorithms, and describe both Lindenmayer-Systems [1] and random fractal trees as competing methodologies. Using these trees as layout mechanisms has been largely successful in producing visually acceptable results within our own engines, and the techniques should be adaptable to a range of other games and applications. We provide an example of a random fractal tree explosion and the accompanying code-base. L-System explosions are discussed but are not presented as a demonstration due to the wealth of existing materials on their creation. Finally, several basic special effects are explored, as well as adaptation from two-dimensional test environments to Shockwave 3D.


A random-tree explosion generates from a single animated sequence, composted in real-time with tree-layout, alpha-blending, and wind shear effect.

1 THE SINGLE EXPLOSION SPRITE

1.1 A LOOK AT NON-TREED APPROACHES

A number of non-treed approaches exist for the simulation of explosions-like events, meaning the expansion of gas and debris over time. The most simplistic of these is a single sprite animation. Using this approach, a number of tiles are pre-rendered that depict the event, and then played back, one after another to produce the animation. This is in fact at the core of any of the more advanced techniques: the ability to animate over time. That the sprite technique does so using pre-rendered frames is its defining characteristic, and in fact we use the sprite technique as the basis for our trees.

There are also several non-sprite-based techniques worth mentioning. There was a great deal of work done with regard to the modeling of expanding gases by David Ebert [2], which proposes the use of noise functions and real-time compositing to produce effects in a three-dimensional graphics system. Also noteworthy were the color algorithms for expanding mist proposed and illustrated by F. Kenton Musgrave [3]. Regardless of claims of real-time performance, however, we have found such solutions to be too computationally intensive for interpretive frameworks such as Shockwave, and have had only minimal success in adapting these techniques.

Another interesting technique is the real-time deformation of spheres using the per-vertex and per-pixel pipelines found in modern graphics hardware. This is demonstrated in part in the planet-creation routines of Jesse Laeuchli [4][5], although it is obvious how these routines could be useful in simulating explosion and similar effects using similar geometry, essentially deforming a sphere in real-time through pre- and post-render effects based on noisy textures.

The technique with perhaps the most current known implementations is the use of a formal particle system for simulation and rendering. These systems are built into most rendering software (Alias|Wavefront's Maya®, Kinetix 3dsMax®, etc). They are also available in real-time on modern graphics hardware, provided that they are not overused. There have been several examples recently of how to simulate explosion-style effects using particle-systems. [6][7] But these simulations rely on direct access to the GPU though shader languages (generally either Cg from Nvidia or the DirectX 9.0 HLSL). Given that neither of these is directly available, and that the available particle systems objects are a significant performance drain on Shockwave 3D environments, this technique was not fully explored. It should be theoretically possible to implement the particle routines in software and use sprite-based techniques for rendering.

A final approach is to use textured quads or spheres with full-motion video of actual explosions. This technique attempts to simulate none of the random, chaotic nature of an explosion and instead relies on successfully compositing a real explosion into the scene. The advantages of this technique are obvious: it has the capability to look the best of any possible simulation because it is, in fact, the real thing. There are, however, a number of issues related to correctly scaling and placing the video in the scene, masking and clipping the explosion from the rest of the footage, and performance problems that arise from the size of video footage even in compressed form. It was not deemed appropriate, in a real-time game, to attempt this using the current crop of video tools, although such an attempt may be useful, either as a direct composite or as a texture extraction exercise.

1.2 DUPLICATING ANIMATION SEQUENCES

In the creation of what we believe to be convincing animation system, we first looked at generating the explosion graphics algorithmically, but this proved to be a significantly arduous task within the confines of Director. Instead, we turned to Kinetix 3D Studio Max® and pre-rendered a series of slides as high-quality PNG images with alpha channel. This is not as elegant as generating them via the code base, as they have to be shipped with the game rather than generated post-download, but the visual appeal was much greater than any attempt at a lingo-based render (and significantly faster). As such, the demo movie contains an "EXPLOSIONS" cast in which our pre-rendered slides are stored as cast members (it should be noted that the size of these cast members is quite large, which will be addressed in section 3.X).

Once all of the image tiles have been created, it is necessary to display them in sequence to produce animation. This is done using imaging lingo techniques, in the "Base Animator" script. This script is essentially a re-creation of the sprite functionality of the classic Director engine, only using imaging techniques instead of traditional sprite based ones. It operates under the following rules:

1. Each member of the EXPLOSIONS cast has its image duplicated and places in an array (EXPLOSTION_WORLD[#g_aExpImages]
2. Each separate explosion created receives a new instance of the "Base Animator" script. This object contains an mUpdate method.
3. Each frame, the mUpdate method of all available Base Animators is called. This is akin to a "stepframe" for objects placed on the ActorLIst
4. The mUpdate method draws the current image in the array (starting with the first) onto a buffer image the size and shape of the stage by calling the objects mDraw method. Following the draw, the current image counter is increased, such that the next frame drawn will correspond to the next image in the array.
5. The mDraw method uses standard copyPIxel commands to copy the image from the g_aExpImages array onto the buffer (see the commented "straight blit" code for details)

NOTE: All explosions, be they the first or last, draw from the same set of images in the array: this is essentially a glorified version of the way sprites instance cast members without needing to duplicate the media itself. Thus while the initial set of images is rather large, they exist in one selected memory space and do not change position or size, and can thus be thought of as at least partially optimized in the sense that no direct calculation needs to occur on a per-frame basis other than the copy into the stage buffer. Also note that our initial tests used the image of the stage itself instead of a buffer, and while this consumed less memory, it was significantly slower than rendering to an off-screen image and performing a single copy onto the stage.

1.3 ROTATION FOR ADDITIONAL VARIETY

Even with the ability to add several explosion sequences from the same image set, the system exhibited too much similarity, given that the clouds in one sequence would exactly match the expansion of the second. To combat this, a system was developed that allows for the rotation of the "sprite" around its own origin. Since there is no actual sprite object, this system implements its own rotation, using methodology similar to that presented in the author's collision system [8] using the formula for rotation in two dimensions presented in Figure 1.

$$[x' \;\; y'] = [x \;\; y] \times \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

Figure 1: Equation describing rotation of a two-dimensional point around the origin in Cartesian space where [x' y'] describes the position of [x y].

The points of the non-rotated bounding rect of the image are rotated around a given angle, and the resulting points are used to form a quad that is fed to the copyPixels command in place of the target rectangle. For performance reasons, the sin() and cos() of each angle is pre-generated and placed in a lookup table for easy retrieval, as this yield significant speed increases when compared to computing the values every frame. A further optimization that could be implemented would be to not rotate the quad at all if the target "sprite" did not rotate that frame. The code for the sin/cos lookup table generation is presented in Figure 2. The code for the actual computation of the rotated image quad is contained in the "BASE_ANIMATOR" and "ROTATION_HELPERS" scripts (see the "mRotateQuad" handler for specifics).

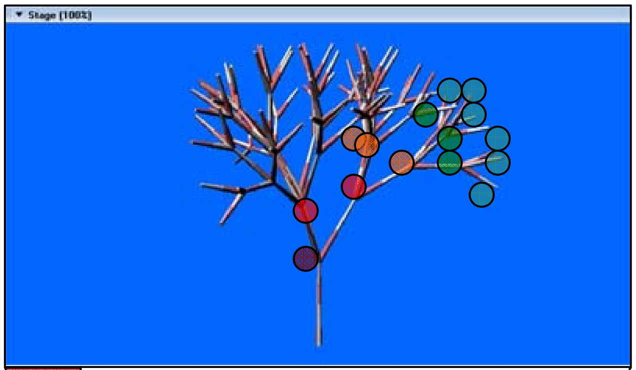
```
--generate sin / cos lookup tables.
 D3D_WORLD[#g_aSin] = []
 D3D_WORLD[#g_aCos] = []
 repeat with iCounter = 1 to 360
  D3D_WORLD[#g_aSin][iCounter] = \
    sin(iCounter * \
    D3D_WORLD[#g_fDegrad])
  D3D_WORLD[#g_aCos][iCounter] = \
    cos(iCounter * \
    D3D_WORLD[#g_fDegrad])
end repeat
```

Figure 2: Lingo handler for pre-generation of sin / cos lookup tables where integer angles serve as keys to the individual values.

2 LAYOUT & EXPANSION ALGORITHIMS

2.1 L-SYSTEMS AS A LAYOUT TOOL

The first methology used for the layout of the individual explosions is an Lindenmyer system. Such systems are commonly used for rendering plants, grasses, and bushes [9], but can also be thought of as a way to generate point lists in time. The authors do not present the underlying structure of their L-System implementation as it does not differ significantly from the basic technique described by Lindenmyer and implemented in [2], and there are several good documentable sources describing other Shockwave3D implementations that our as advanced if not superior to the ones we present here. The L-Systems in this case are not the focus of the application, but rather a way to generate a set of points that are stored in subsequent arrays, and serve as the basis for explosion sequences at those locations. This idea is presented in Figure 3.



🟫	Explosion Event Level 1 – Array 1
🟥	Explosion Event Level 2 – Array 2
🟧	Explosion Event Level 3 – Array 3
🟩	Explosion Event Level 4 – Array 4
🟦	Explosion Event Level 5 – Array 5

Figure 3: Generation of event levels via an L-System Methodology. Explosions are generated at level 1 first, then 2,3,etc. at specific intervals.

Using these “Event Levels” an animation controller is constructed that places a large animation at Level 1 using the “base animator” presented earlier. After some number of frames have elapsed, smaller explosions are started at Level 2, at a slightly smaller scale, and possibly at a difference per-frame rate. After yet more time has expired, explosion sequences are started at the positions stored in Level 3, etc, etc. This process is repeated to the desired depth. When no noise is introduced into the system with regards to variability in scale, position, or play-rate a semi-ordered explosion ensues, as can be seen in Figure 4.

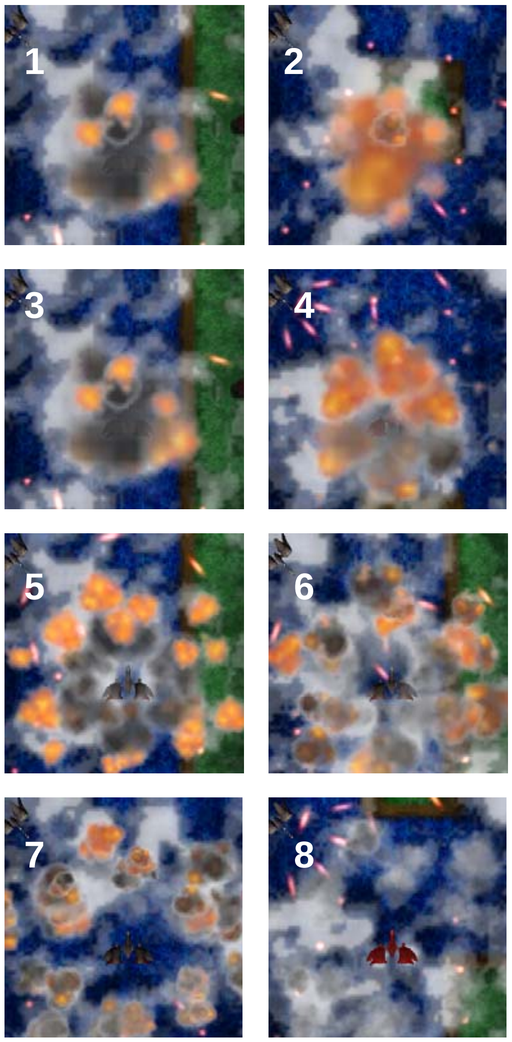
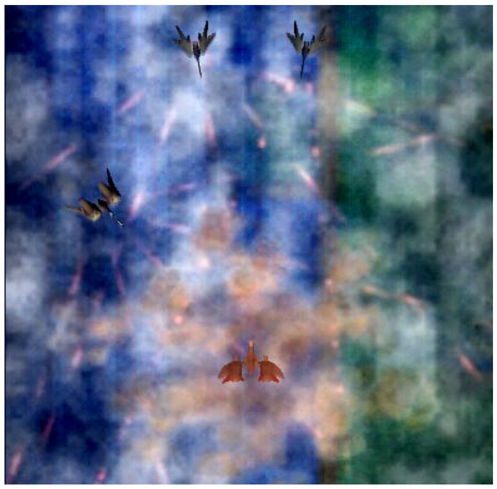


Figure 4: Top-down L-System explosion tree over 64 frames captured every 8 frames. Screenshot taken from the Broadsword Engine, alpha 0.82. Copyright A. Cloutier and A. Phelps 2003-2004.

The explosion above stores the positions of an L-System similar to that presented in Figure 3, but with greater radial symmetry. The explosion is seen from the top down, with entities towards the root-node rescaled to a larger size, where scale is diminished per-level, and rotation of the explosion in 2D space is random, but always bill-boarded to the camera. A time-lapse view of the explosion that demonstrates this symmetry is presented in Fig. 5.



.

Figure 5: Time-Lapse view of top-down L-System explosion tree over 60 frames. Screenshot taken from the Broadsword Engine, alpha 0.82. Copyright A. Cloutier and A. Phelps 2003-2004.

2.2 RANDOM TREES

The explosions presented previously are desirable if and when a more “ordered” explosion is needed. To get a more chaotic feel, a second approach was used, based on a more random, less-governed tree. This is not to say that L-System are incapable of producing less ordered results: techniques such as random pruning, selective offspring, and pseudo-random noise tables can produce significantly altered visuals from the base tree. In the interest of time and performance however, a second approach was implemented that simply computes a number of children from the parent at completely random angles. This is significantly easier to calculate, and can be visually appealing if somewhat controlled. This technique is the basis for the FRACTAL script in the demo file, and a render of non-explosive spheres, color coded by age, can be seen in Figure 6. This technique is based in part on Keith Peters’ work in fractal generation, but without any of the advanced rulesets applied [10]

The FRACTAL script begins by creating an explosion at the base position, and then waits until the next pass to create the children to that parent. Recursively, it then creates the children to those parents, and continues to iterate to its maximum depth, pausing and waiting *N* frames between creations such that the explosions appear to cascade out from the center. Each time an explosions is created, a new “Base Animator” is

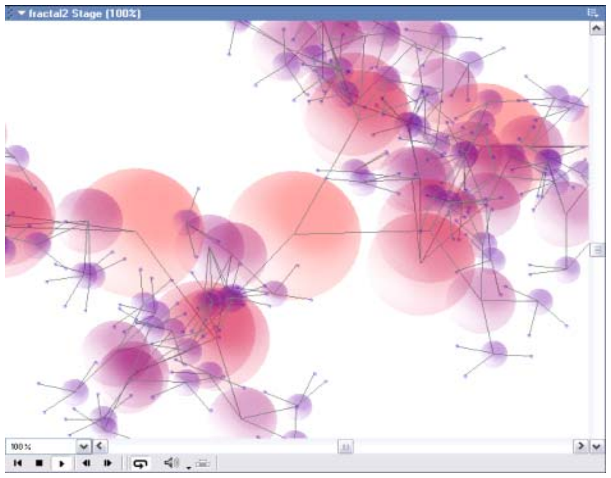


Figure 6: A rendering of the same fractal system that is used in explosion generation, with all nodes rendered as spheres. Color and size represent age in the system, with large / light red [center] being the root node and the smallest purple nodes being the last. Connection between nodes are child-parent and represented with grey lines.

created, and stored in the EXPLOSION_MGR. This manager keeps track of all of the currently playing explosions, and updates each of them per-frame on its own mUpdate method. Thus, once an individual "base animator" is created at a certain position, it is totally independent of the fractal – it will animate, rotate, and drift on its own without interaction with the other nodes in the fractal. A final explosion sequence using this technique is presented in Fig 7.

3 EFFECTS AND 3D

3.1 WIND AND OTHER SPECIAL EFFECTS

Several effects are added to the base explosion system, the most noticeable is the "wind" effect. The effect is actually dead simple, and involves a hack to the "base animator". Every frame, as the animator calculates the quad into which it draws the image for the frame, the quad is offset by a number of pixels in the x and y direction. This can be done linearly for a flat "slide" of the explosion, or exponentially such that the wind has no effect at first, and has a greater and greater effect based on the number of frames the animator has already calculated. The formula used in the wind example provided is p = p + (f1/ft)*c, where p represents the position of the quad-point, f1 represents the current frame of the animation, ft represents the total number of frames in the animation, and c is a constant value that represents the wind strength. (2D coordinate positions pX and pY are computed individually). This could be modified such that either c or (f1/ft) is raised to an exponential power to further slant the curve of the winds effect if needed.

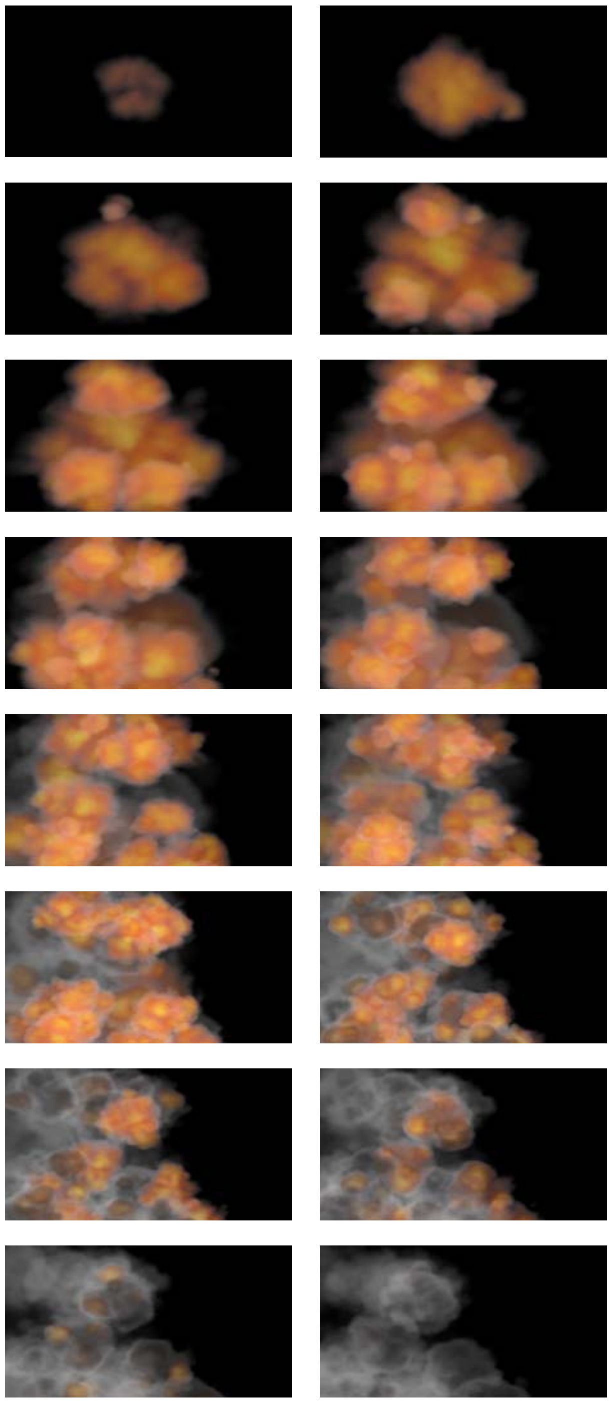


Figure 7: Explosion sequence based on a fractalized, random tree structure and animated explosion tiles.

A second small touch that was added is an ever increasing rotational force such that explosions that do not appear to rotate when they begin do approach noticeable rotation as they fade into smoke and debris. There is a small amount of randomization in the rotational speed to avoid visual repetition.

3.2 ANIMATING IN PERFORMANCE CRITICAL ENVIRONMENTS

It is worth noting that in performance critical environments there are two major factors that can effectively speed up the system. The first is the size of the pre-rendered explosion images, and smaller images increase performance because there is less information to copy per frame. In general, explosion tiles can be rendered 1/3rd to 1/4th of the size that they will be seen on screen and the results will still be visually appealing. Rendering the tiles too small relative to size on screen results in over-pixelation and visual degradation.

The second available optimization occurs in trees that advance far enough such that there are a large number of children towards the end of the tree. In such systems, it is generally unnecessary to draw all of the children towards the end of the tree, as they cannot be visually distinguished from their peers at this level. Thus, either linearly or exponentially culling the number of children per node or simply capping the growth algorithm at an upper bound can allow explosions to continue visually without degrading performance due to the number of child objects. A balance between the cap and the visual results of the system should be obtainable. See the comments at the bottom of the mUpdate method in the "FRACTAL" script for further details.

3.3 ADAPTATION TO SHOCKWAVE 3D AND TEXTURED QUADS OR SPHERES

The final implementation of this system operates in a Shockwave3D, and derives several benefits from the 3D environment. First and foremost, instead of creating an array of images, an array of textures is created through standard S3D methodologies. These textures are then mapped onto models that are single meshes of 2 adjoining triangles that create a square "tile". These tiles can be rotated freely without any of the sin/cos operations above by applying matrix operations to the model through the `model.rotate(vector(0,Φ,0),#self)` where Φ is the desired angle of rotation. This is a significant performance advantage. All of the tiles remain oriented towards the camera, and a "camera render-group" approach can be used to ensure that explosions always render on top of other elements [11].

The second major advantage is the complete lack of copyPixel operations. The explosions animate by swapping textures onto the model one after another, there is no need to copy the image into screen or pixel-space. This is further optimized by the fact that the textures can be very small compared to the rendered textures used in the demo, because of the mip-maps created by the graphics card when the images are used to create textures. In the prototype of the engine, the sample explosions in Figures 4 and 5 were created using pre-rendered explosions that were 16x16 pixels. This is significantly smaller than the 128x96 pixel tiles used in the imaging lingo demo program, but the results are as good if not more appealing, due primarily to the mip-mapping and texture blurring features found in modern graphics hardware.

4 CONCLUSIONS

There are several methodologies available to create more believable explosions and gas-like and particle-like effects in game engines. While several of these rely on advanced techniques using modern graphics hardware, some tried-and true techniques like billboarding sprites and animations are still very effective. Using L-Systems and random trees can help create believable placement of expanding gas-clouds without the complicated mathematics of exactly modeling the physics of rapidly expanding gas and debris. This is particularly suited to arcade style games, in which explosion based effects are often needed, but the processor time that can be devoted to them is minimal.

References

[1] Eric W. Weisstein. "Lindenmayer System." From *MathWorld*--A Wolfram Web Resource. Online: http://mathworld.wolfram.com/LindenmayerSystem.html

[2] Ebert, David S., F. Kenton Musgrave, Darwyn Peachy, Ken Perlin and Steven Worley. Texturing & Modeling: A Procedural Approach. 3rd Edition. San Francisco, California. Morgan Kauffman, 2003. pp. 203-224.

[3] Ebert, David S., F. Kenton Musgrave, Darwyn Peachy, Ken Perlin and Steven Worley. Texturing & Modeling: A Procedural Approach. 3rd Edition. San Francisco, California. Morgan Kauffman, 2003. p. 553.

[4] Jesse, Laeuchli. "Real-Time Generation and Rendering of Planest in 3D". Graphics Programming Methods. Ed. Jeff Lander. Hingham, Massachusetts. Charles River Media, Inc. 2003. p. 193-199.

[5] Jesse, Laeuchli. "3D Planets on the GPU". Shader X2: Shader Programming Tips & Tricks with DirectX 9. Ed. Wolfgang F. Engel. Plano, Texas. Wordware Publishing, 2003.

[6] Celes, Waldemar and Antonio Calomeni. “Simulating and Rendering Particle Systems“. Graphics Programming Methods. Ed. Jeff Lander. Hingham, Massachusetts. Charles River Media, Inc. 2003. p.5-16.

[7] McCuskey, Mason. Special Effects Game Programming with DirectX. The Premier Press Game Development Series. Ed. Andre Lamothe. Premier Press, 2002. 698-706.

[8] Phelps, Andrew M., and Aaron S Cloutier. “Methodologies for Quick Approximation of 2D Collision Detection Using Polygon Armatures“. Macromedia DevNet Center. 2003. Online: http://www.macromedia.com/devnet/mx/director/articles/collision_detection/collision_detection_lingo.pdf

[9] Przemyslaw Prusinkiewicz, A. Lindenmayer. The Algorithmic Beauty of Plants. Springer Verlag. April 1996.

[10] Peters, Keith. Flash Most Wanted Effects & Movies. Friends of Ed. 2003.

[11] Hill, Mark. “Brain Goes Bye-Bye“ DirGames-L listserve discussion. Friday, November, 15th 2002. http://nuttybar.drama.uga.edu/pipermail/dirgames-l/2002-November/020340.html