Introduction

Many algorithms in computer graphics consume considerable machine resources and are difficult and time consuming to implement. University research laboratories and similar facilities often have a minimum of hardware and human resources, yet still wish to contribute to this growing area. An animation infrastructure is required to support in depth research in any one specialty within this field. We present here some examples of practical graphics techniques which help provide efficient realism though not necessarily of the highest quality without the investment of large machine and human resources. We start with a description of the underlying animation system and then present some practical graphics examples as further building blocks.

An overview of Graphicsland

Animation can be broadly split into three areas:

Modelling
Motion control
Rendering

Graphicsland is a layered system which provides an animation substructure in the above three areas. On top of this substructure may be built tools for particular applications. Figure one shows how the practical graphics examples discussed in this paper relate to the animation

substructure.

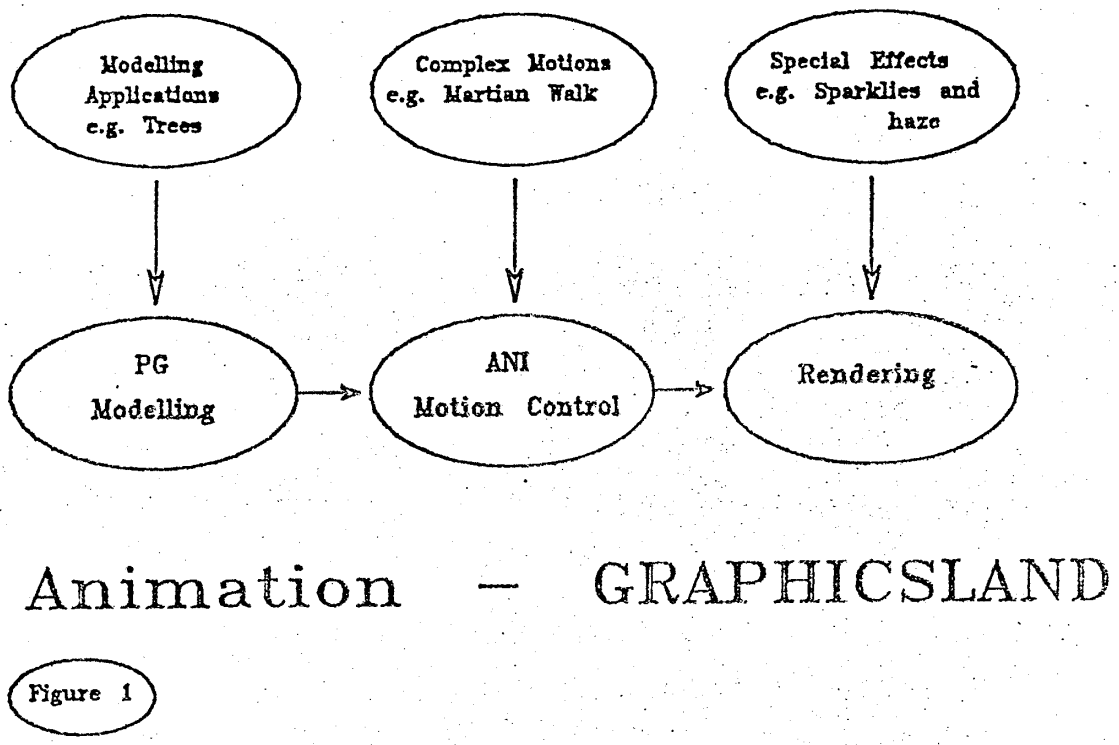

Figure 1

The modeling and motion control tools are based on a hierarchical graphics system called Polygon Groper (PG) [Wyvill 84a]. PG allows a user to define and edit models. The models are stored as a hierarchy of objects, where each object represents a geometrical transformation of another object, a primitive polygon, or a parameterised description. Various systems have been written which take such descriptions and return primitives. In this way the system is extensible. For example an object may be described in terms of particles. The parameters which drive the particle system are the leaf nodes of the hierarchy. The object behaves like any other object in the system and may be included in a model consisting of polygons, particles or any other primitive. Objects may also refer to themselves. A numerical limit defines the termination of recursion. Thus rows of objects, or rows of rows, circles, and spirals are defined as singly recursive; objects with a single self reference. Objects can also be defined as multiply recursive, such as trees.

The following practical graphics examples demonstrate some of the ways the system has developed as a testbed for graphical ideas in each of the three primary animation areas. The tree example shows how the fundamental structure of the system can be easily used to test out growth ideas. The Martian walk example demonstrates how tools for particular motion specification and control applications can be tested. The *Sparklies* example shows how rendering algorithms can be adapted for special effects. Some of these examples, such as haze, increase picture quality while being both simple to implement and without adding noticeably to the execution time of the rendering process. The pictures accompanying this article have been

produced without *cheating*, that is all the objects in the pictures are three dimensional, they can be easily animated, no painted backgrounds or texture maps have been used and each picture was produced by a single rendering process.

Summary of the practical graphics

Modelling

- Particles, Fractals and Polygons
- Modelling Trees
- Fractal Mountains

Motion Control

- A three legged gait

Rendering

- Sparkly particles
- Sphere and Particle Calculation in Raytracing
- Haze

Modelling

Graphicsland currently uses polygons, particles and fractals, as primitives. A particle system after [Reeves 83] and a fractal mountain system after [Fournier 82] have been integrated into the model hierarchy by defining primitives which are known as *Procedural Image Generators* (PIGs). A PIG contains all the necessary information to drive the particle or fractal programs. They also inherit properties passed down the hierarchy such as geometrical transformations and attributes like colour. Provision has been made to extend *Graphicsland* to include further PIGs in the future. Various utilities also exist to enable particles to be mapped onto polygons. This is useful for making particle leaves on the end of a polygonal tree or particle fire on a polygonal building.

Modelling Trees

A tree may be simply defined as a multiply recursive object in PG. In contrast to dedicated systems such as in [Kawaguchi 84] and procedural models as used by [Marshall 80], we use a general purpose modeling system, this has the advantage that a tree-like object is treated in a consistent fashion with other objects in the system. The growth rules are currently limited to geometric transformations although experiments are being carried out on including

stochastic controls.

The PG editor allows the user to examine and change each level of the hierarchy along with various attributes. The attributes include lighting constants such as transparency, colour and whether smoothing is required for a polygon mesh.

The tree shown in slide 1 is defined by the following code segment:

```
open  tree     #Define tree
   add cylinder scale 0.6 1 1  #Defined previously
   #Now define four branches as four recursive references to tree
   add tree scale 0.5 0.7 0.5  rotate z 30  rotate y 60  origin 0 8.7 0
   add tree scale 0.5 0.7 0.5  rotate z 45  rotate y -60  origin 0 9.3 0
   add tree scale 0.5 0.7 0.5  rotate z 33  rotate y 180  origin 0 8.1 0
   add tree scale 0.4 0.5 0.4  rotate z 60  origin 0 6 0
   limit 5
close     #End Definition of tree
```

These instructions set up the data structure representing the tree. When the structure is traversed, the transformation commands; scale, origin rotate; cause the coordinate axes to be transformed. The tree is defined in terms of four transformations of itself plus a transformation of an object called cylinder. Cylinder can be defined as a singly recursive object composed of a rectangle and a cylinder rotated 72 degrees with a limit of 5 giving a five sided pentagonal cylinder. The attribute *smooth* informs the rendering process to smooth the cylinder.

Adding The Leaves

An *if* command in the language allows objects to be added at a particular level of recursion. It is desired to map particle leaves on the outer branches of the tree. These can be obtained by using the *if* command with the same definition as above:

```
add cylinder if 0
```

When the structure is traversed, the cylinder is added only if the last (0th) level of recursion has been reached. These (0th) level polygons are then passed to the particle program along with a number of parameters which instruct particle generators to be distributed along the length of the branch (cylinder). The particle program uses the (0th) level cylinders as a skeleton, only particles, not cylinders, are passed on to the rendering process to make the leaves. Slide 2 shows how this technique is used to map level 8 leaves onto a level 7 tree. The scene also shows purely polygonal objects, such as the British telephone box and fractal mountains.

Fractal Mountains

The method for generating fractals in our fractal mountain package is largely based on [Fournier 82]. A recursive algorithm is employed to subdivide an initial polygon. After each subdivision the coordinates of the vertices of the new polygons are modified by a random value constrained by a roughness factor. The main advantages of our approach are the use of integer arithmetic for parts of the subdivision process, and a data structure that guarantees no gaps in the generated polygon mesh without attempting to generate identical coordinates for corner points shared by two or more polygons, which is what Fournier suggests.

Polygon Subdivision Algorithm

Input is a list containing the polygon to be subdivided, output is a list of the four resulting polygons. Polygons consist of lists of pointers into a global points list. A centre point is generated, based on the averages of the vertex coordinates. Each edge of the input polygon is split into two edges by generating a point based on the average of the two endpoints, if the edge has not already been split previously while subdividing an adjacent polygon. Each point has its own list of "partner" points used previously to spit edges; this list is checked first before an edge is split. If such a point is found, then the previous result is used. The order in which the new points are used to construct new polygons is important if the orientation of all polygons is to be preserved. The final output routine writes out two triangular polygons for each four-sided one passed to it so that non-planar polygons will not have to be rendered.

This scheme is rather more general than required for this particular application, and probably could be optimised to decrease the number of cross-reference pointers. Note also that a non-recursive algorithm may be more efficient. However, for our purposes the current performance is sufficient. To generate a mountain consisting of 32K triangular polygons took 2 minutes 40 seconds CPU time on our VAX 780, and used 3.7 Megabytes of memory, while a 128K polygon mountain took 13 minutes 4 seconds, using 10 Megabytes. See slides 3 and 4.

Motion Control

The motion specification and control system, *ANI*, supports animation of object types in *Graphicsland.* Currently this includes particles, polygons, fractals and heirarchies of any combination of these objects. Motion may be applied to camera(s), light(s), or to any object in the hierarchy. Thus a node representing a hundred objects could be given a global motion that would be added to any local motions applied lower in the hierarchy. Motion controls include geometric transformations, special effects such as *wobble, bounce, scrape, fade,* etc. Each motion effect follows a chosen acceleration curve which are useful in a large variety of different animation situations. There is also the facility of using beta splines to define a path, for motions that are more complex similar to the scheme outlined by [Kochanek 84].

The system is implemented to run as a co-process with PG, and in fact uses PG as would any other user. It is very easy to switch from the animation system to PG, and communicate directly with that package. The animation system was designed to be an open system, one which could easily accept the output from more specific, task oriented systems. The reason for this is that the command set is too low level for defining something as complex as say a walk

motion. Thus it is recognised that special purpose programs will be necessary to describe these motions and provision has been made for communication between the animation system and such programs as described below.

For any serious animation project it is essential to prototype the sequences before going to the expense of rendering each frame. We achieve this by loading low resolution monochrome images into the frame buffer. Using 128 by 128 pixel resolution, provides 384 frames (about sixteen seconds) of preview. Such a system is by necessity largely device dependent, we take advantage of this in our system by emulating the frame buffer, using as input code from any system which can produce output for that device. (Raster Technology Model 1/20 24 bits/pixel)

A three legged gait

Many computer animation systems allow a user to describe motions for three dimensional objects either by geometric descriptions or by inbetweening. However to define a realistic walking sequence is difficult since the graphics must be controlled by the motion dynamics of the walking object. Others have approached this problem by assigning constraints to the limbs that relate to the constraints found in nature and using goal directed techniques to compute individual movements on the way to a final motion goal. [Korein 82] Such techniques are computationally expensive. A simple approach to the problem is described which falls between these two extremes. A limited set of parameters is provided which give the animator control over particular facets of the gait but allow him to describe physically unlikely gaits. However, the program allows quick previews of gaits, permitting easy selection of those which appear natural. The system is called *MAWP* since it originally applied to a 3 legged martian and thus named; *Martian Walk Program*; supplies a crude wire frame skeleton figure which will walk in real time. *MAWP* has been integrated into *Graphicsland* so the walk previewed as a wire frame can be transferred into a complete scene for rendering, providing an interactive interface between the animator and *Graphicsland.*

The martian itself has a small elliptical body with 3 legs which come out of the equator of its body at equal angles of 120 degrees, with the ***back*** leg going straight back. There is one knee joint in each leg.

The ability to generate virtually any conceivable Martian gait through a small set of user-defined parameters permits design flexibility. The user can quickly approximate the gait he desires and refine the parameters until thoroughly satisfied with the results, or he may alternatively experiment with various gaits until one or more catch his interest.

MAWP is given eight parameters to define the martian gait. A wide spectrum of gaits can be defined, including some which are physically unlikely.

The first pair of parameters defines the resting stance of the martian:
bodyx - the horizontal component of distance between foot and body
bodyy - the vertical component of distance between foot and body

The next pair defines the magnitude of the step:
stepx - the distance travelled in one step
stepy - the height the foot is lifted

The last four parameters define the phase relations between the three legs and the body.

MAWP provides immediate feedback on the gait by displaying a stick figure through one walking cycle. Code is automatically generated for each gait, in a form suitable for input to ANI. This generates a short sequence of the gait on a fleshed-out martian, prototyping the martian's gait as it would appear in an actual film sequence. If the user is satisfied with the results, the code can then be incorporated as part of the final animation sequence.

The use of a parameterized specification of the gait much reduces the computational overhead and thus allows real-time checking of the gait. Although some experimentation is necessary to get a realistic and natural gait the immediate feedback makes this easy.

MAWP has been extended to allow for definition of turns. In addition, a second version of *MAWP* has created a four-legged creature model with which to emulate animal gaits. Features such as flexibility (namely the parameterization of the gaits), immediate feedback and automatic generation of animation code have significantly facilitated the accurate and straight-forward definition of gaits.

Rendering

Graphicsland offers the user a choice of several rendering algorithms. These have to deal with all the primitives that are currently supported.

Particles

Particles present an interesting problem particularly when ray tracing. A sophisticated treatment of ray tracing particles using density models is given by [Kajiya 1984]. Our approach is directed towards producing special effects at low computational cost.

Each particle is considered to be spherical, and optionally a light source. Since it was desired to gauss filter the particles, the distance of any given ray from the centre of the particle is required. Deriving this distance has also lead us to an efficient equation for calculating sphere

intersections in raytracing. Both results are summarised below.

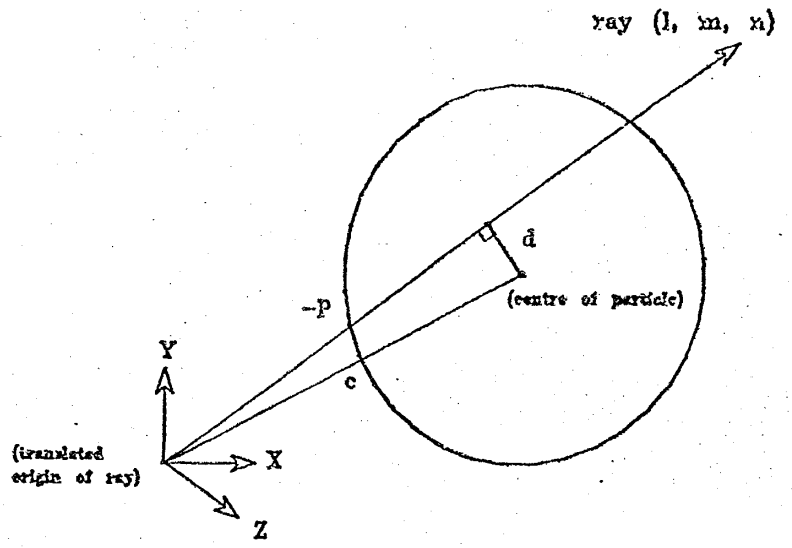

Figure 2

To find the distance d in Figure 2. from the centre point C to the ray (l, m, n) we observe that the line d lies in a plane with (l, m, n) as a normal, and passing through point C (X_c, Y_c, Z_c). The length of the ray from it's origin (X_o, Y_o, Z_o), to it's intersection point with the line d is then:

$$-P = (l * (X_p - X_o)) + (m * (Y_p - Y_o)) + (n * (Z_p - Z_o))$$

where (l, m, n) are the normalized direction cosines of the ray and -P is the distance of the plane from the origin of the ray, or rather the length of the ray to it's nearest approach to the particle's centre.

At this point, if the simple test $P < -(r)$ is true, where r is the radius of the particle, then the particle is behind the origin of the ray.

The distance from the centre of the particle to the origin of the ray is obtained from

$$d^2 = c^2 - P^2.$$

where

$$c^2 = (X_p - X_o)^2 + (Y_p - Y_o)^2 + (Z_p - Z_o)^2$$

If

$$\sqrt{d} > r_{particle}$$

then the ray does not intersect the particle.

A careful implementation of the above requires only 15 floating point operations and one square root to find if the ray intersects. In order to eliminate the square root, the radius of each particle may be squared at the outset. This is reasonable since the intersect calculation is the most costly in ray tracing and particles in general will share a common radius.

Spheres

This algorithm may easily be extended to sphere intersections. The calculations described above are identical, but for the sphere, the intersection points are needed. Therefore, the distance *s* in Figure 3 must be known in order to calculate the length of the ray when it intersects the sphere. As above,

$$S^2 = r^2 - d^2$$

and

$$L1 = P - S$$

$$L2 = P + S$$

After determining which distance, L1 or L2, is nearer the origin of the ray, the point of intersection is:

$$X_i = X_o + l*L$$

$$Y_i = Y_o + m*L$$

$$Z_i = Z_o + n*L$$

Where L is the nearer of L1 and L2.

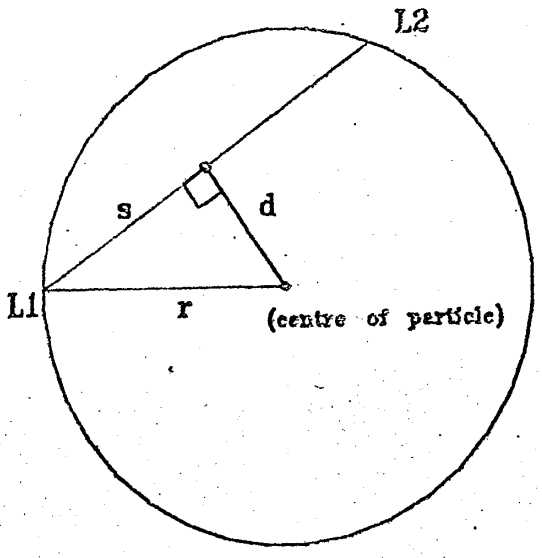

Figure 3

Rendering Particles and *Sparklies*

In [Reeves 83] particles are rendered by adding their value into the appropriate pixel. In our system particles are treated as spheres and can have an associated filter function to control their affect on neighbouring pixels. The transparency of the particle increases as the point that is being rendered gets farther away from the center of the particle. This is done in a uniform manner regardless of the orientation from the center. We have extended this idea to include particles that are not uniform, that have a structure. This can be used to give a sparkle to fires or explosions, give variety to clouds, details to any system that is using particles in their description. A *sparkly* is a particle with an associated filter function. In slide 5 the filter function is not uniform, the diagonals of the particle are given a higher intensity. Experiments are continuing on stochastic methods of describing, and modifying filter functions for *sparklies.*

Rendering Haze

The apparent realism of pictures such as landscapes which show some sort of distant background scene can be greatly enhanced by introducing a very simple hazing algorithm. A haze colour is defined by the user and then added to the frame in proportion to the distance from the yon plane. Best results have been found with a medium grey haze colour. This method is very easy to implement in Z-buffer without adding significantly to the rendering time. The formula used for haze at each pixel is as follows:

$$Haze = Z * H_{red}$$

$$Pixel_{red} = Haze + ((1 - Haze) * P_{red})$$

where

Z - The depth of the pixel, this value is between 0 and 1.

H_{red} - The haze colour for red.

P_{red} - The red component of the pixel before the haze calculation.

$Pixel_{red}$ - The colour of the pixel after the haze calculation.

Similarly for green and blue. Integer arithmetic can be used throughout with minor modifications.

The Z depth used in calculating haze is based on a linear function, that is, a depth halfway between the hither and yon planes should be assigned one half of the maximum depth value. This is not a characteristic of the depth in the screen coordinate system outlined in [Newman 79].

Conclusion

We have presented a collection of practical graphics which offer some refinements to various methods for modeling and rendering scenes for 3D animation. These build on the animation system, *Graphicsland*. The system has been used to make several short sequences of animated film amounting to about three minutes.

Acknowledgements

The *JADE* project at the University of Calgary has been supportive of our work in distributed graphics. This work and *JADE* is supported by the Natural Science and Engineering Research Council of Canada. We would also like to acknowledge the help of all the students who have written software for *Graphicsland* and built models. In particular Breen Liblong, Norman Hutchinson and Danny Freedman.

References

Cleary, J., Wyvill, B.L.M., Birtwistle, G., and Vatti, R. (1983) "MULTIPROCESSOR RAY TRACING" *Research Report No. 83/128/17 University of Calgary, Dept. of Computer Science..*

Dippe, M. and Swensen, J. (July 1984) "An adaptive subdivision algorithm and parallel architecture for realistic image synthesis." *Computer Graphics. Proc. ACM SIGGRAPH 84,* 149-158.

Fournier, A., Fussell, and Carpenter (June 1982) "Computer Rendering of Stochastic Models" *CACM, 25* (6).

Kajiya, J. and Von Herzen, B. (July 1984) "Ray Tracing Volume Densities" *Computer Graphics. Proc. ACM SIGGRAPH 84,* 165-175.

Kawaguchi, Y. (July 1982) "A Morphological Study of the Form of Nature" *Computer Graphics 16(3). Proc. ACM SIGGRAPH 82,* 223-232.

Kochanek, D. (July 1984) "Interpolating Splines with local Tension, Continuity and Bias Control." *Proc. ACM SIGGRAPH '84 pp33-41.*

Marshall, R, Wilson, R, and Carlson, W (July 1980) "Procedure Models for Generating Three-Dimensional Terrain" *Computer Graphics, Proc. ACM SIGGRAPH 80,* 223-232.

Newman, W.M. and Sproull, R.F. (1979) "Principles of Interactive Computer Graphics" *Second edition, McGraw-Hill..*

Reeves, William. (Apr 1983) "Particle systems- A technique for modeling a class of fuzzy objects" *ACM Transactions on Graphics, 2,* 91-108.

Wyvill, B.L.M., Liblong, B., and Hutchinson, N. (June 1984) "Using Recursion to Describe Polygonal Surfaces." *Proc. Graphics Interface 84, Ottawa..*