Genetic Algorithms for Graphics Textures

Andrew Hobden

EPCC–SS94–03

Genetic Algorithms for Graphics Textures

Andrew Hobden

EPCC–SS94–03

September 1994

Abstract

This report describes how evolutionary techniques of variation and selection can be used to create complex three dimensional textures for use in computer graphics. Generations of textures are calculated and displayed, allowing the user to select one or more individual textures and thus direct the simulated evolutions in preferred directions. The textures are represented by a tree of functions which acts as genotypes. The trees will consist of simple, mathematical functions which may be combined to produce complex effects. Two dimensional textures can also be used as functions to give a more realistic effect to a texture. The simulated genotypes can then be mutated and evolved with the user deciding on the 'fitness' of each texture. The texture resulting from each genotype is displayed on the surface of a sphere to allow the user to see the effect.

It is proposed that artificial evolution has potential as a powerful tool for achieving flexible complexity with a minimum of user input and knowledge of details.

1 Introduction

Procedural models are becoming more common in computer graphics to generate complex graphical effects. This complexity must be controlled, in some way by the user. Most systems for doing this, either require a high degree of knowledge and input from the user or simplify things so that the user loses some control over the output. The techniques detailed here aim to allow the user to retain a high degree of control over the output while having little, or no, knowledge of how the effects are generated. The process of artifical evolution is used to achieve this. The user is presented with a randomly generated collection of images and prompted to select one or more that may contain some effects of interest. These will then be used to construct another generation of images via artifical mutation and sexual combination. These techniques have been used previously for similar purposes. This project builds mainly on the work done by Karl Simms [Simms, 1991] of Thinking Machines Corporation. The main thrust of Simm's work was geared towards 2-D textures, while this project works with 3-D textures. The sections in this report are split into the following sections.

Background	Introduction to textures, artifical evolution and genetic algorithms.
Implementation	Overview of the coding for the project.
Results	Including static and animated textures.
Future Work	Some suggestions and work done with 3-D objects.
Conclusions	Conclusions.

2 Background

Textures, in computer graphics, is a term for the visual texture that a given object has. An orange is a sphere with an orange texture, a golf ball is a sphere with a golf ball texture and so on. Many textures can be given to a relatively simple, mathematical, shape like a sphere but the problem becomes a lot more complicated if the shape itself is complicated. In order for a user to create complicated textures, a lot of knowledge of and around the subject would normally be required, however if a genetic algorithm is used to generate the textures then the user may retain control, without the need for so much specialist knowledge. Using a genetic algorithm a texture is 'evolved'. This is achieved by the user deciding, for each generation, which texture(s) are most useful and only allowing characteristics from those textures to carry on to the next generation. Textures and genetic programming are discussed in more depth in the following section.

2.1 Textures

Mapping textures onto three dimensional computer generated surfaces is a well understood problem to which there are two basic solutions.

- Take a 2 dimensional image and wrap it round the object.
- Use a function with x,y,z as input which outputs the colour of the texture at that point.

The first of these, wrapping a texture around an object, can distort the texture completely, see figure 1. Some object shapes lend themselves to this technique but it is not a good generic method. The textures themselves can be very realistic, however, as they can be scanned in from photographs. They can, however, occupy a large amount of storage which can be undesirable. The second method is easily mapped onto any object but the function that describes the texture is difficult to discover. To find a function that produces a particular effect would be time consuming due to the size of the search space that needs to be examined - it is nearly infinite as there is an infinite amount of textures. This project proposes to solve this problem by using genetic programming techniques to evolve textures with the desired effects.

Figure 1: The Warping effect of mapping a 2-D brick like texture onto a 3-D sphere

2.2 Artifical Evolution

Biological and artifical or simulated evolution have many concepts in common. These include the genotype and phenotype, the processes of expression, selection, and reproduction via different schemes.

In biological terms the *genotype* is the DNA which carries the information required to create an individual. In the case of this project the genotype refers to a tree datatype, each node of which represents a function. The *phenotype* in a biological system is the end result of using the genotype and a set of procedural rules, which normally produces a lifeform of some sort. In this simulation of evolution the phenotype is the 3-D texture that is produced by the tree according to a set of rules for displaying textures on objects. This process of producing the texture, or lifeform, is called the *expression* of the phenotype from the genotype.

Selection is the process by which the genotypes, and hence phenotypes, are evaluated to see which ones are fit enough to survive and breed to produce the next generation. In biological evolutionary terms this is commonly known as '*The Survival of the Fittest*'. This will be implemented in this project by the user choosing certain images over others in order to direct an evolutionary path, hopefully, to the texture that is desired. This is unusual in artifical evolutionary schemes. The selection process is normally carried out by a fitness function which evaluates each phenotype and gives it a fitness factor. This can then be used to chose the genotypes which can reproduce to form the next generation.

Reproduction is the process by which the new generation of genotypes are created from the fittest members of the previous generation. In order for evolutionary progress to be made there must be variation from the old to the new. This variation can be introduced in several different ways. *Mutation* and *sexual combination* are methods which occur in biological systems and these can be modeled in artifical evolution. These variations are done at random and it is up to the user to guide, deterministically, the evolutionary process through the repeated cycle of reproduction, with variation, and selection. In this manner generations evolve towards lifeforms that are more likely to survive or, in the case of this project, more interesting textures.

2.3 Genetic Algorithms

The process of simulated evolution was first developed by Holland [Holland, 1975] as a technique for searching through large search spaces by testing the fitness of evolving populations in order to find a solution to a problem. Since then they have been used in a large variety of projects involving finding the optimum solution within a large search space.

The technique involves using artifical evolution and evaluating the fitness of each genotype to select which ones may reproduce. The genotype is typically represented via a character string which is used by fixed expression rules to produce the phenotype. The fitness function uses the genotype to produce a fitness factor which is a measure of how well the phenotype solves the problem. There are many different breeding strategies but the simplest takes the genotype with the best fitness factor and uses them to produce the next generation.

The rate of mutation need not normally be large (5be reasonably large (100 to >1000) to allow the mutations to occur often enough to affect the evolutionary process.

3 Implementation

3.1 Overview

This project has been implemented by two programs, a genotype manipulator (Visualiser) and a phenotype manipulator (Displayimage), both written in 'C' for Silicon Graphics workstations. They make extensive use of GL functions to open windows and create menus and also use the twenty four bit colour available to these machines. A library of useful graphics routines called GraphicsGems [Glassner, 1990] is used to implement a lot of functionality common to many graphics programs.. This library has been written in the Kernighan and Ritchie form of 'C' and so the other programs in this project have been written in this format to allow easy use of this Library. The project may have been a lot easier to write in ANSI standard and would probably be more portable. Some functions written to produce particular graphical effects like noise and turbulence are also used, these will be described in more detail later. The reason for implementation on these machines is the high graphical capability and GL. It was considered that, within the time scale of this project, implementation in X would take too long. GL provides an easy to use interface to a windowing environment to allow the quick creation of simple user interfaces.

It was necessary to adapt the genetic algorithm used in this project to overcome several problems.

The size of each generation produced by a genetic algorithm is usually large, unfortunately, each generation of textures must not only be calculated but also displayed, which puts limits on the population size. The image needs to large enough for the user to make an accurate evaluation of it's fitness; the larger the image the more calculations need to be done and the longer it takes to calculate a generation. This also limits the size of the population. The time taken to calculate and display a generation is important because the program requires the user's regular input and must not make the user wait too long.

Many genetic algorithm implementations use a bit string as a genotype which contains all the information to solve the problem within it. This project does not use this method for two reasons.

1. The functions themselves could not easily be contained within a datatype. The genotype constructed does not contain all the information needed to generate the phenotype; it just contains the names of the functions to call and the order in which to combine the results.
2. The search space is not just large but infinite and the use of a tree datatype for the genotype means that it is infinitely extendable merely by adding more nodes to the tree.
3. Another attribute of genetic algorithm implementations is the constant length of the bit string genotype. This was considered to be unacceptable in this project as this would mean that the functions could only be executed in a linear fashion. A tree datatype for the genotype, therefore, gives a greater flexibility in the genotype, which is desirable in this case.

This particular implementation of artifical evolution is known as genetic programming.

Sexual combination has also been implemented in the form of crossover. Sexual combination involves the sharing of information between two separate genotypes. This allows genotypes with different but interesting effects to combine and produce one genotype with both effects. The crossover method takes the two genotypes and determines a sub-genotype within each one. These sub-genotypes are then swapped thus exchanging the 'genetic' information. There are several more ways to implement sexual combination but this was the only one that was implemented in this project. The reason for this is that mutation is generally recognised to be more useful under this particular method of simulating evolution and implementing other, more complex methods of sexual combination would take a lot of time for little or no useful results.

Visualiser takes two inputs (*n* and *size*) from the command line. Where *n* is an integer which is the square root of the total number of images to be displayed and *size* is the number of pixels on the side of an image. The reason for *n* being the square root of the number of images is to force the images to be displayed in a square window which makes the placement of images easier to calculate.

Visualiser displays a generation of images on the screen. These images are the result of displaying the texture on a sphere, to allow the user to evaluate each texture. Each point on the surface of the sphere, that the user can see, is calculated via an equation. This method was chosen over a polygon representation of a sphere because the textures need to be mapped pixel by pixel onto the sphere and a more chunky texture would result it were mapped onto polygons. Each generation is displayed at once and the user is required to choose one or more images that will survive to breed, and produce the next generation of textures and therefore images. The genetic algorithm was implemented in this program.

The images can also be produced by displaying the textures on a flat plane. These two surfaces were chosen because they show the texture in two very different environments. The sphere shows a 3-D slice through the texture while the flat surface shows what a texture would look like on a polygon. This is useful as these textures will hopefully be used in a ray tracer. In order to allow the textures to be evaluated within a ray tracing environment each point within the texture needs to be independent of every other point. This excludes the use of image processing functions which can give very interesting graphical effects.

Displayimage is the phenotype manipulator program and takes just one input from the command line which is the filename of a file which stores a genotype saved via visualiser. Displayimage loads in the genotype and displays the resulting phenotype either on a sphere or a flat surface. The image can be enlarged and even animated by this program.

3.2 User Interface

A large part of the visualiser program has been devoted to the user interface. The user may select and deselect images quickly and simply and a feature has been included to allow the user to back up a generation if the result was not useful. The user selects and deselects images by positioning the cursor on the image and pressing the left mouse button. A selected image has a green border. The right mouse button is used to bring up a menu, shown in figure 2, with various options. The user can use this menu to control the methods of variation. The options are to just use mutation or sexual combination (crossover) or both to produce the next generation. This was considered to be necessary as the methods of variation do give different effects and the small size of each generation means that the more control the user has over the process the more chance that a certain texture can be produced. Figure 3 shows the user interface without any images selected.

3.3 Trees and Functions

The tree datatype is defined in trees.h as treenode. Each treenode (or node) contains a random vector, a integer for the number of arguments for the function that this node represents and an array of pointers to the nodes in the tree below the current one. Each node represents a function that takes the output of the nodes below it and passes an the output of it's function to the node above it. There is a special case to this - the bottom level functions which, conceptually, take no arguments. However, the bottom level functions do use the coordinates on the surface of the shape onto which the texture is being mapped, in the case of the visualiser this is normally a sphere. The coordinates passed to the tree are those that are on the part of the sphere that the user can see.

Thirty six functions in all have been implemented in the files functions.c and functions.h. These are given a brief description in appendix A.

A sample function, f1blur, is shown in figure 4.

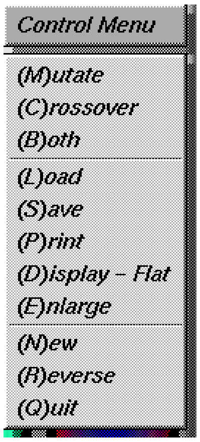


Figure 2: Visualiser's Menu

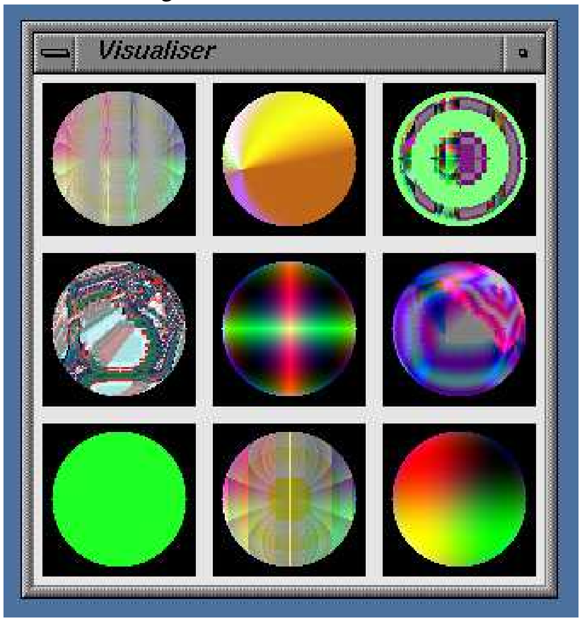


Figure 3: Snapshot of Visualiser in Action

The fact that approximately the same thing is done to each element in the input vector is typical of the functions that have been implemented. Notable exceptions are f3cross which returns the normalised cross product and f1rgb2hsl which uses the values in the input vector to calculate three totally different colour values.

The range of the arguments to and from each function needs to be the same under this implementation, because the functions will take arguments from functions below them and pass output to functions above them. The range from -1.0 to 1.0 seemed to provide good results. Originally the range was from 0.0 to 1.0 but this produced textures that were not symmetrical about the centre. Symmetry about the centre is considered to be a useful feature because this tends to produce textures which can be tiled. Another attribute of the functions which is considered useful is that they should be atomic, that is, that they can not be made up of more, smaller functions. The idea behind this thinking is that the genetic algorithm will 'construct' the larger functions from the smaller ones. An example of this would be an add function which took three arguments and added them together. The same result could be achieved by using a two argument add function twice. To make it possible for the genetic algorithm to produce as many effects as possible the functions have been made general rather than specific, the exception to this is scale. There are two scale functions, one which scales by a factor of two and one which scales by a factor of ten.

```
int f1blur(a,result)
Vector3 *a; /* the input vector(s) */
Vector3 *result;  /* the result of this function */
{

#ifdef DEBUG
fprintf(stderr,"IN f1blur\n");
#endif

result->x = a->x + a->y/10.0 + a->z/10.0;
result->y = a->y + a->x/10.0 + a->z/10.0;
result->z = a->z + a->x/10.0 + a->y/10.0;

V3Clamp(result,-1.0,1.0);

return 1;
}
```

Figure 4: The Function f1blur

A generic function was written at first with another argument which represented the scale factor. This function tended not to produce useful results even when the scale factor was mapped to the range 1 to 10. It seems that, for this application, the scale by two and the scale by ten functions produce the most useful results.

A large design decision that was taken when implementing the trees was in the method that a node represented a function. Two solutions were considered.

1. The tree nodes could hold pointers to their respective functions and invoke the function via this pointer.
2. The tree nodes could hold a function ID code which must be unique for each function. The functions could then be called via a special function which checked the function ID code and called the appropriate function through a switch statement.

The second of these solutions was implemented for several reasons.

- It was easier to debug than the function pointer method.
- It was easier to code and therefore faster for a non ‘C’ programmer to implement.
- It allowed the functions to be defined with as many different arguments as is necessary because the switch statement to call them is hard coded and can call them with the correct amount of arguments.

The functions f2readpic and f1read3Dpic were implemented in an attempt to give more realistic textures. They read, from disk, several standard pictures in ppm format and stores them in fpimages. The input vector is then treated as coordinates to access the picture at that point. The value returned is the colour at those coordinates. The attempt to produce realistic textures via this method did not succeed as the pictures read in became too distorted to be of any use. They sometimes acted as a colour map if they were near the top of the tree but this is an inefficient way to implement that feature.

3.4 Variation

Two main types of variation were used; mutation and crossover, a type of sexual combination. These have been implemented in the trees.c file which contains all functions dealing with the tree datatype.

Six different ways of mutating a tree have been implemented, one of which only mutates the bottom level functions.

1. Change the random scalar vector. This only happens to the bottom level functions as this is where these vectors are stored. A small random amount (-0.1 to +0.1) is added to each element.
2. Add a new tree to a random node. This has the effect of deleting the tree below the node and can, if the top node is chosen replace the entire tree. It can be a drastic mutation and tends to make the tree larger.
3. Change the function a node represents. The tree structure is left intact and the new function must have the same number of arguments as the old one.
4. A node is deleted entirely. If the node has only one branch below it then that is moved up to replace the current node, otherwise one branch is chosen at random.
5. Insert a new node. Creates a new node at random and inserts it below the current node. The new node may only have one argument.
6. Duplicate a sub-tree. Takes a sub-tree of main tree and add this sub-tree to another part of the tree. This allows effects represented by the sub-tree to duplicate themselves throughout the tree.

Since mutation is generally seen to be more useful with this type of genetic evolution, only a simple method of sexual combination called crossover was implemented. The crossover function takes two 'parents' and returns two 'children'. The children are constructed by taking a sub-tree from each parent and swapping them. A definition of crossover states that if two identical parents are 'mated' then the children with be identical also. In this implementation, the sub-tree is chosen at random from each parent and so, even the parents were identical the same sub-tree would not normally be chosen. This was considered to be acceptable, mainly due to the importance of mutation over crossover. For most of this project the mutation rate has been set to one hundred percent, for testing purposes, but with a smaller mutation rate the textures become easier to guide towards desired results.

The different affects of crossover and mutation are shown in the following figures. Figure 5 shows the first, randomly generated, generation with two images selected from it. Figures 6, 7 and 8 shows the results of variation on those two selected images. Figure 6 shows the result of selecting mutation only for those two images. Figure 7 shows the result for crossover only while figure 8 shows crossover and then mutation.

3.4.1 Colour

The program uses a library called fpimage to display images on the screen. This library needs the red, green and blue components of the colours in order to display them. This means that the tree must be capable of producing three values from a given set of coordinates (x,y,z). The method to achieve this must not produce large shifts in colour from small numerical shifts in the input. This problem comes partly from the type of functions used and partly from certain combinations of them. The method of producing colour can make the problem worse, so several methods were tried to find the best one.

Single value This was the first method tried. The functions took and returned floating point numbers as arguments instead of vectors. The single resulting number that came out the top of the tree then under went bit manipulation to give 3 distinct numbers one for each colour component. The diagram 9 on the following page shows how the bits were split up to try and give a smooth colour gradient for each colour component. Despite these attempts this method did not give good results with the images tending to have large shifts in colour for small numerical changes.

Three Trees This method is perhaps the simplest. The functions worked with floating point numbers as described above but three trees were produced, one for each colour component. This method did occasionally give interesting results but mostly when the trees were similar in structure. If the trees did not have a similar structure then the pictures tended to look like three textures superimposed over each other in different colours. Since this is not a desirable effect this method was abandoned.

Three Element Vectors This method is the one used in the program and described is this report. Each function was modified to take and return 3 element vectors as arguments - each element in the vector

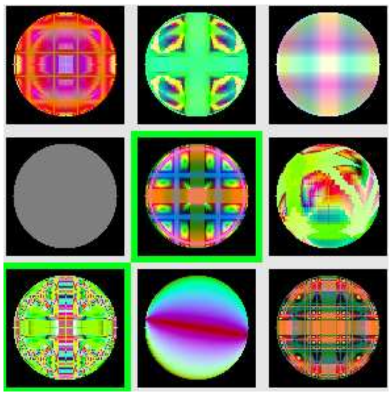


Figure 5: First Generation

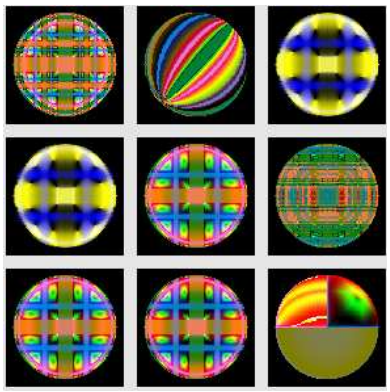


Figure 6: Result of Mutating Only

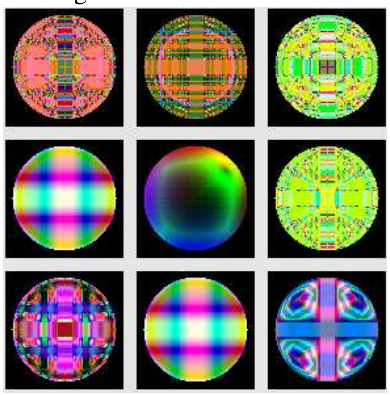


Figure 7: Result of Crossover Only

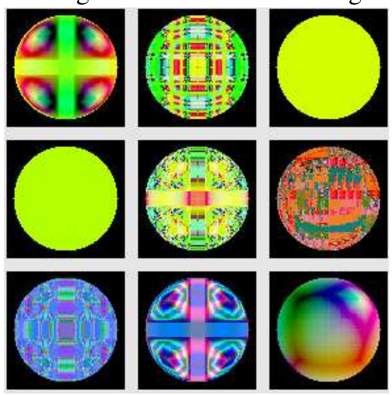


Figure 8: Crossover then Mutation

was matched to a colour component. This means that each colour can be shifted separately from one another and that numerical shifts match colour shifts. In order to allow colour components to be changed simultaneously three more functions were added to the function pool - f1swap, f1shift and f1duplicate. These functions allow a texture to be predominately a single colour or even black and white, if all three elements are mapped to the same element. This method also allows black and white images to appear as a sub-image within a main colour image. It was therefore considered to be more flexible and provide a range of textures than the other two methods of implementing colour.

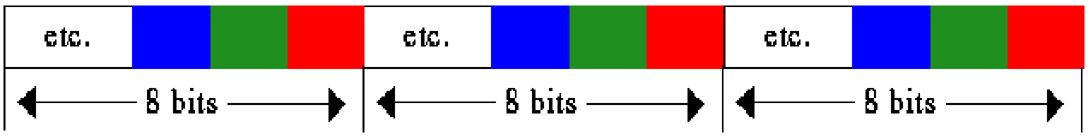


Figure 9: A Method of Implementing Colour

3.5 Phenotype Manipulator

It is impossible to examine a genotype produced by visualiser to discover if it will produce a good texture. The genotype must be expressed into a phenotype first. Another problem arises from the versatility of the texture, a texture which looks good on a sphere may not look as good on another object or at a high resolution. In order to help to evaluate the textures they must be examined further by the user. The program Displayimage has been written to allow this. Another problem that can arise if that of frequency sampling whereby the image is completely different when displayed at higher resolutions. Displayimage allows the user to detect the trees which are prone to this effect.

The visualiser tends to be used with quite small images, typically 100 pixels across, which means that the textures are not seen to best effect. The enlarge option on the visualiser menu does allow the image to be displayed at 200 pixels across but it is not very flexible. Displayimage allows the image to be resized to any size the user wishes. This is done when the user resizes the screen. This program can also be used to animate the textures.

Displayimage takes just one argument, the file name of the genotype for the texture to be displayed. The menu allows the user to save the image of a single frame or all frames of an animation in ppm format. The time step option allows the user to decide how many frames the animation will have. This is useful if the user wishes to get a rough idea of what the animation looks like before asking the program to calculate all the frames. The smaller the time step the longer it takes to calculate the images and, even at quite small resolutions, this can take a long time.

The phenotype manipulator may have many extra features written for it that would be very useful to the user when trying to achieve specific effects. One feature which would be very useful would be a colour editor of some type, whereby the genotype could be manipulated to get different coloured phenotypes. The figure 10 is an example of where this would be useful. The image has a woody texture but is not realistic because of it's colour. Writing this sort of manipulator would be very difficult in practice because the elements that give colour to an image come from many different conditions in a tree. One method might be to allow the user to edit the random constant vectors within a tree and interactively display the results.

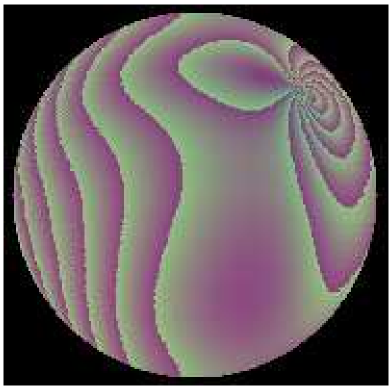


Figure 10: Woody Texture with Wrong Colour

3.5.1 Animation Techniques

There are several possible ways to produce animation from the textures, some of these are outlined below. The one used in Displayimage is the time variable method.

A Time Variable The animation technique used here is the simplest one of the methods that have been outlined in this section. The image is displayed at various slices along the Z axis. Another way of looking at this is to regard Z as a time variable for the animation. As Z changes so does the image; not in space but in time. The time step option is used to control the distance between each frame, so the smaller the distance between the frames, the smoother the animation. Animations made with this method can make large shifts in colour and pattern quite suddenly or not seem to move at all, so these are probably not the best animations that can be achieved from these textures. A set of five frames of a texture animated on a sphere are shown in figure 11.

Interpolation Between Two Textures Two textures are used in this method, one for the starting image and one for the end image. The frames to get from one to the other are calculated by interpolating between the results. If the interpolation is at small steps the resulting animation will be smooth. An advantage of this method over the time step method is that is a lot easier to see and guess the result of the animation. A disadvantage is that movement is only produced if the trees contain similar subtrees otherwise the first image simply dissolves into the second image.

Change the Input Values The images are a texture mapped onto an object, if the object is manipulated in various ways and the new position of the object is then given as input to the tree this may produce movement within the texture.

Same Object at Different Sizes If the texture is on a virtual sphere of a large radius but the image displayed is a small window onto the virtual sphere, then movement over the surface of the sphere could result in animation of the image similar to moving over a surface.

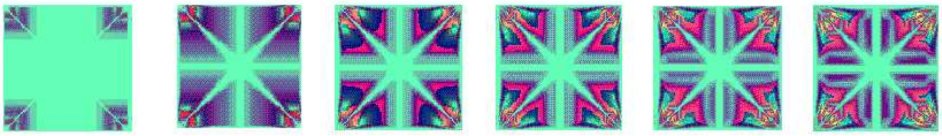


Figure 11: Frames from Time Variable Method of Animation

3.6 Problems Encountered

Several problems needed to be overcome to allow the successful completion of this project.

At some times during the course of the project it became very difficult to get access to SGI machines and progress was slowed because of this. This is obviously a direct result of the decision taken at the onset of the project to program using GL.

Another problem came as the result of NaNs and out of range numbers being created at certain points in the evaluation of the tree of functions. This caused two types of error - incorrect mapping of output to colour, as a certain upper and lower limit is assumed and by far the more serious, the program sometimes crashed. On further examination the stack appeared to be being corrupted. This proved to be because of the math libraries on a machine with version 5.2 of the operating system installed. The manual entries gave misleading information about return values for certain inputs to library functions and the stack seemed to be being corrupted by passing certain numbers into math library functions. For example, the sqrt function was at some time passed negative numbers which returns a quiet NaN but it also seemed to cause the stack to be corrupted. Whether this was because the math library was incorrect or because another function in the tree above the sqrt function could not handle NaNs properly is unknown but the later seems most likely. This was solved by not passing sqrt negative numbers. Another, similar problem arose when using the Black and White noise function. This seems to return a NaN for the x element of its result vector for certain inputs. Both this problem and the out of range problem was solved by putting checks in the calling function to

4 Results

4.1 Textures

Many different types of textures were created by this method. There were abstract patterns like figure 13 which could become very complex and intricate and some natural ones. There always seemed to be more abstract ones.

It was discovered that the same texture could have very different trees of sizes from 10 to 100 nodes. This applies to complex trees as well as straight forward ones. This suggests that the functions produced via this method may not be the most efficient functions to achieve a given texture.

The textures displayed in this section have been enlarged by Displayimage to about two times the size they were generated at.

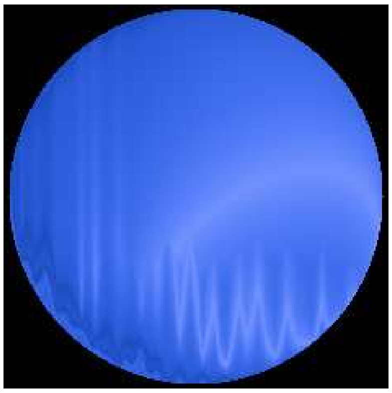


Figure 12: Quite Subtle Colouring

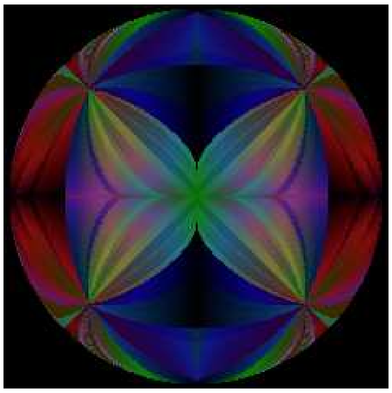


Figure 13: Abstract and Symmetrical

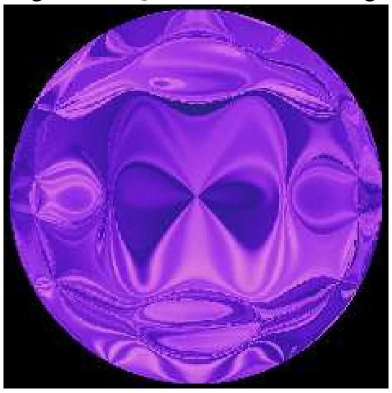


Figure 14: Folded Silk

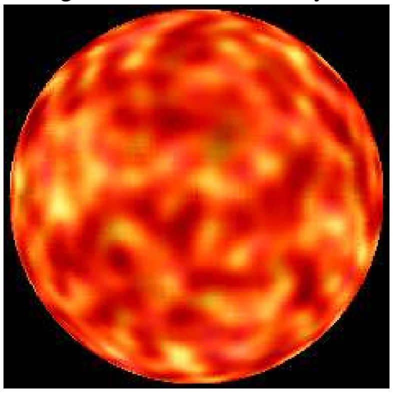


Figure 15: Fiery Planet

All these textures have been evolved from scratch in several minutes, a much shorter time than it would take to design the expressions that they represent. Five to thirty generations are normally required to achieve interesting images. Many different types of textures can be generated via this method, even ones with surprising subtle colouring like figure 12. Completely unexpected images were generated, like the folded silk one in figure 14. If you have difficultly in seeing this just think of the silk cut adverts. The tree that generates this image is in figure 16. This tree only contains thirty nodes and only uses ten of the thirty or so functions available to the program. This proves that complex effects can be generated form relatively small trees of simple functions. The fiery planet image in figure 15 is from a very small tree but the colour noise function f2noise produces the bulk of this effect and is thus very important if natural looking textures are to be produced.

All these textures are displayed on a sphere because they seem to be more interesting when viewed in this way. The texture can change drastically when displayed in flat mode. Figure 17 shows what the folded silk image looks like displayed on a flat surface.

```
f2cross
  f2stripes
    f1shift
      f2add
        f1normalise
          f2sin
            f2cross
              f1shift
                f2add
                  Different Random, -0.182400, 0.589804, -0.165220
                  Different Random, 0.290887, -0.481077, -0.010539
              f2sin
                f2cos
                  f1normalise
                    f2add
                      f2sin
                        f2cross
                          f1shift
                            f2add
                              Different Random, -0.182400, 0.589804, -0.165220
                              Different Random, 0.290887, -0.481077, -0.010539
                          f2sin
                            f2cos
                              f1normalise
                                Allcoord
                      Allcoord
        Allcoord
  f2sin
    f1hsl2rgb
      Different Random, 0.000000, 0.000000, 0.000000
```

Figure 16: Genotype for the Folded Silk Image

4.2 Animation

The animation of textures as discussed before has been done via the time variable method. The results have been interesting although abstract. One of the better animations has been a watery one which could be waves crashing on rocks. Some frames from this are shown in figure 18. The small size of the images is to keep the amount of calculations down to minimum.

Animation is possible via this method but the results are very unpredictable and another method may need to be implemented in order to produce consistently viable animations.

5 Future Work

This method of genetic programming can be applied to many areas in the field of graphics and this scheme of a program to generate the genotype and another to examine the results in detail seems to be valid. The method could be adapted to any or all of the projects listed here.

- Using the Genotype to warp a 3-D object to create another 3-D object
- Input points on a line to the trees to produce a new line which could be used to move an object in 3-D space.

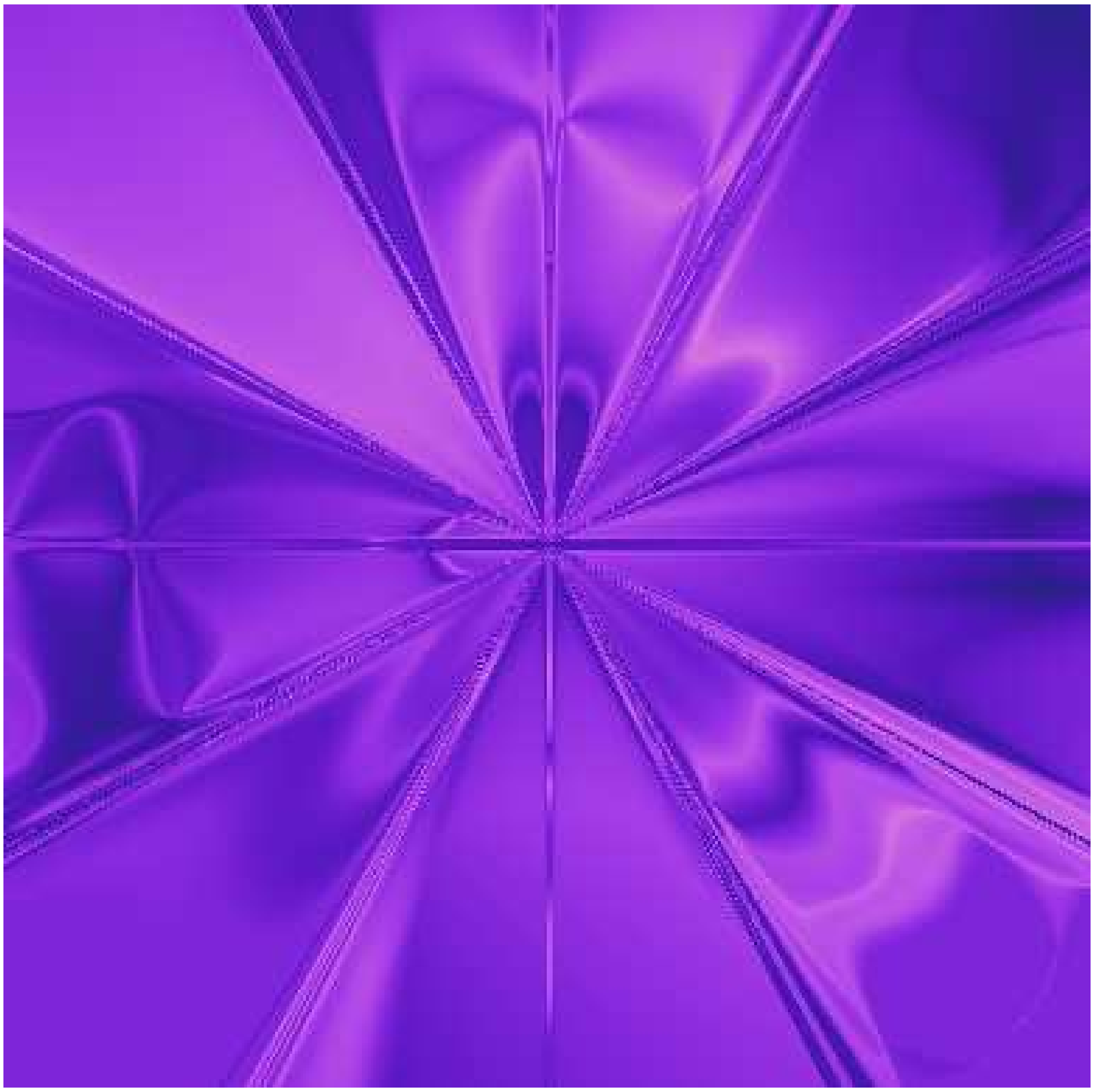


Figure 17: The Folded Silk Genotype on a Flat Surface

- Instead of trees representing the genotypes, directed graphs could be generated. This would allow repetition of nodes which could lead to fractal like imagery being created.
- Use an automatic fitness function instead of a user. This could take the form of an image, perhaps scanned in, that the fitness function would compare the phenotype to. In this way a genotype could be generated which gives a particular texture.
- Implement another genetic algorithm system on top of this one to discover common functions or function combinations that produce interesting textures. In this way a program could be written that learned what an interesting texture was and tried to produce them automatically.
- The texture project could be parallelised. This could be achieved in several ways.
    1. Each pixel in a fpimage could be calculated in parallel as each pixel is totally independent from the others.
    2. The package written at EPCC called RPL-2 could be used. This could also be used to implement parts of the genetic algorithm also and would construct a task farm of textures to be calculated in parallel. This does produce some problems, however, as the package has

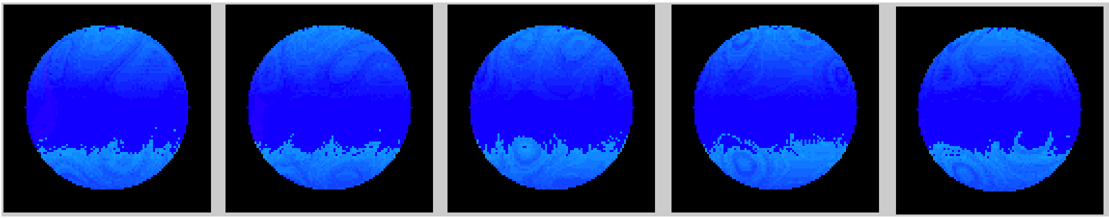


Figure 18: Some Frames from the Water Animation

been written to use a fitness function. Therefore there will need to be a master and worker architecture to collect the data and display it for the user to select textures.

3. The task farm could be implemented directly without RPL-2.

The parallelisation part of this project was not completed due to the discovery that up to a hundred textures could be displayed, evaluated and selected in a reasonable space of time, if the fastest SGI available was used. Since the parallelisation of the project was not necessary at this stage another project above was implemented.

The first project listed has been implemented quickly. The program takes a sphere made up of triangles and uses the tree to warp the vertices of the triangles, changing the shape of the triangles, and thus changing the shape of the sphere. The resulting 3-D object is then displayed on the screen using the 3-D display capabilities of the SGI machines. The objects were lighted and rotated so that the user could see the shapes clearly. The user interface needed to be re-written and the functions that read in images, f1read3Dpic and f2readpic, were removed. A new datatype was created called shape which held information about each 3-D object and contained functions to create, delete, load and copy them. The rest of the program remained the same and the phenotype manipulator was not implemented.

The results, shown in figure 19, were not very interesting. One reason for this is the choice of functions has not been changed much from the texture project. The functions seem to need to be tailored for the type of output required. This was an unexpected result as the genetic algorithm technique was assumed to be able to select ‘good’ functions over ‘bad’ ones. The small population size may partly explain this. Another extension to this project could be the ability to use a particular library of functions so that this effect can be avoided.

A method of defining a goal would be useful. A user could decide which aspects of a texture are desired. These could include the main colours, whether the desired texture is natural or abstract, etc. This could be used to chose the functions that are more likely to produce these effects.

6 Conclusions

This method of artifical evolution has been demonstrated to be effective at producing various types of textures and could be useful in generating 3-D objects. Allowing the user to pick more than two parents and decide on which method of variation is to be used does seem to allow a good degree of control over the evolutionary process. The genotypes would be very difficult to produce by hand and would require a lot of mathematical and graphical knowledge. The user needs very little knowledge of the underlying problems of graphical textures; all that is required is an idea for the desired image. This could therefore be the basis for a graphical package be used by artists for graphical design work. Functionality could be added to this package to include 3-D objects and animation paths as described in the future work section.

There appears to be two ways to use the visualiser; opportunistic or goal oriented. The opportunistic user will keep generating textures until an interesting one is found and then try to direct the evolutionary path from there. The goal oriented user is only looking for a particular effect and tries to direct the evolution from the start. Both methods work, but the goal oriented one usually takes considerably longer to achieve a good effect.

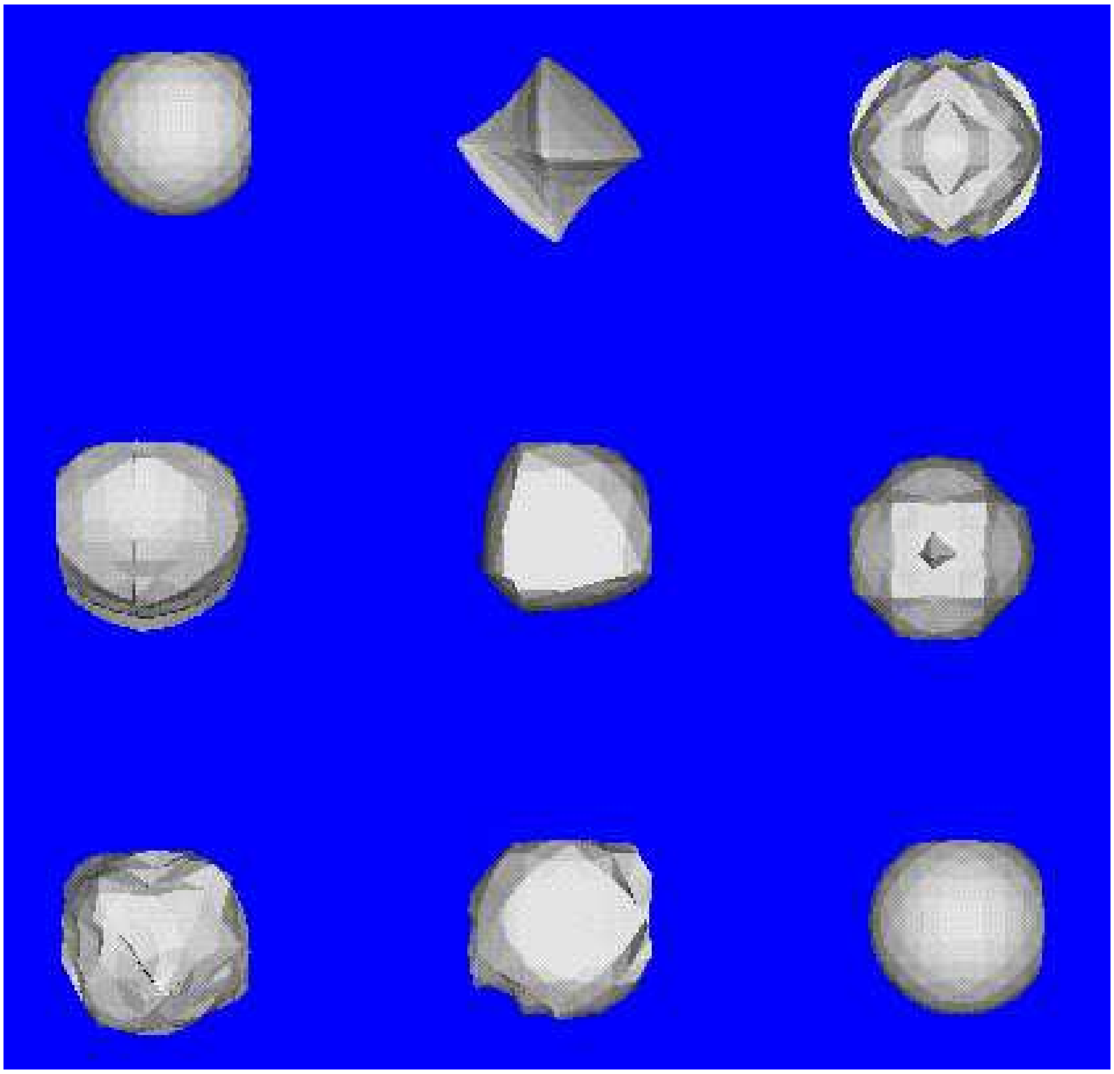


Figure 19: Some Generated 3-D Objects

The mutation rate appears to be critical to this system. Too low and not enough variation is introduced and so a lot textures are the same from generation to generation. Too high and there is too much variation and the evolutionary path becomes difficult to guide towards a desired result. The result of changing the mutation rate is not always obvious as it make take several generations to evaluate the change. Another parameter that is difficult to evaluate is adding a new function into the function pool. The new function may only be useful in certain situations which may not occur for many generations and even then be difficult to detect.

The project did produce realistic, natural looking textures but rarely. This may be due to the choice of functions and the chances of producing a good natural texture could increase with the correct functions. The functions which seemed to produce the best natural effects are noise, turbulence and hsl2rgb. These are all complex functions and it may be that the complex functions are better at producing realistic textures.

The textures resulting from this project may be included in a ray tracer which would be able to show them to greater effect.

The compute power required to calculate textures of a reasonable size is considerable and the project needs a fast workstation, as used here or it could be parallelised via various methods described in the future work section.

A Description of Functions

f2readpic returns: the value of a picture stored at that point

f1read3Dpic returns: the value of a picture mapped onto each surface of an imaginary cube.

f1diffrand returns: a different random constant in each element of the vector

f1allcoord returns: the x,y,z coord vector

f1swap returns: the x,y,z coord vector swapped to x,z,y

f1shift returns: the x,y,z coord vector circular shifted to z,x,y

f1duplicate returns: (x,y,z) -> (x,x,z)

f2stripes returns : returns the inverse of each element of the vector

f2minimum returns : minimum of each element of vectors x and y

f2maximum returns : maximum of each element of vectors x and y

f2noise returns : result which has been injected with colour using the Perlin noise function from Graphics Gems II. This varies the r,g,b components of the inputed triple. It uses the variable noisey->x which as the amount of noise.

f2mult returns : x * y for each element in vector

f3cross return the cross product of the the input vectors

f2cos returns the cosine of each vector element

f2sin returns the sine of each vector element

f1and returns : boolean ANDs corresponding elements of the two vectors

f1or returns : boolean ORs corresponding elements of the two vectors

f1xor returns : boolean XORs corresponding elements of the two vectors

f1not returns : boolean NOT of each element of the vector

f1normalise returns : normalises the vector

f2average returns : the result of averaging the input vectors

f1blur returns : the result of adding 10 percent of each number to the others and then clamping this to the range -1.0 to 1.0

f2add returns : the result of adding the input vectors and then clamping to the range -1.0 to 1.0

f2sub returns : the result of subtracting the input vectors and then clamping to the range -1.0 to 1.0

f1exp returns : the result of doing e to the power of x for each element and then clamping to the range -1.0 to 1.0

f1log10 returns : the result of doing log10 for each element and then

f1abs returns : the absolute contents of each element in the vector.

f2mod returns : returns the modulus of the first vector divided by the second

f1rgb2hsl returns : the hsl equivalent of the rgb input.

f1hsl2rgb returns : the rgb equivalent of the hsl input.

f2scale returns : scales vector 1 by vector 2

f1sqrt returns : the square root of each element in the vector

f1atan returns : the arc tangent of each element in the vector

f2turbulence returns : a turbulent B/W image

f2bwnoise returns : a noisey B/W image

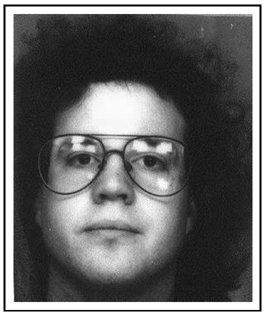
f2scale10 returns : scales vector 1 by vector 2*10.0

References

[Glassner, 1990] Andrew Glassner. Graphics gems. *Academic Press*, 1990.

[Holland, 1975] J. H. Holland. Adaptation in natural and artifical systems. *MI:University of Michigan Press*, 1975.

[Simms, 1991] Karl Simms. Artificial evolution for computer graphics. *Computer Graphics*, 1991.



Andrew Hobden is a student at Napier University studying Computing. After graduation in 1995 he hopes to pursue a career in programming.
This project was supervised by C. Thornborrow (EPCC).