EPCC-SS2001-15

An Alife Demonstrator

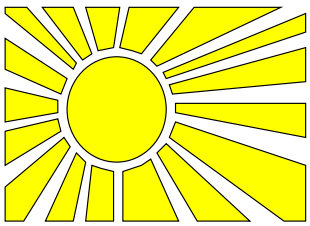
or

Monsters II

Dáire Stockdale

Abstract

The aim of this project is to develop a virtual world inhabited by evolving autonomous animal-like creatures, the ‘Monsters’ of the title. They are to be displayed using 3D graphics in real time moving about and interacting with each other and the terrain. The environment is to be a virtual terrain composed of hills, valleys, trees, plants and water, with its own seasons, climate, day and night. The purpose of the world is to demonstrate the principals of evolution and natural selection at work.



Contents

List of Figures

1 Introduction

The project aims to create the framework for a graphical virtual environment supporting a variety of virtual plants and autonomous, adaptive creatures. This will be achieved by first creating a rendering engine to render a landscape which has water, seasons and a climate.

2 Background

2.1 Herbivore World, Monsters II

The aim of this project is to develop a virtual world inhabited by evolving autonomous animal-like creatures, the 'Monsters' of the title. They are to be displayed using 3D graphics in real time moving about and interacting with each other and the terrain. The environment is to be a virtual terrain composed of hills, valleys, trees, plants and water, with its own seasons, climate, day and night. The purpose of the world is to demonstrate the principals of evolution and natural selection at work.

The project is based upon a SCOPE [1] proposal to develop an educational artificial life demonstrator for children. Although the SCOPE proposal named the world 'herbivore world', at present the project is provisionally named 'Monsters II'.

2.2 The original Monsters

Dáire Stockdale wrote the original Monsters [2] in Spring of 2000. It was a top down 2D-bitmapped graphical population simulation. The world was of fixed size, and was populated by three apparently different species. The world was a grid of squares, each of which could be one of three terrain types, water, rock or plant. A plant type could be one of three plant types, which could in turn be in one of four states, from young to mature.

The original monsters had a limited sequence of genes which were used at birth to determine the monsters operating parameters, i.e. its speed of motion, its graphical species, its tolerance to weariness, its ovulation period (in females).

The monsters in the original game could have their gene sequence displayed, although it was intentionally never made clear which genes were for what behaviour. However in every monster the same gene was responsible for any given attribute.

The intention of the 'game' was to demonstrate population dynamics. The monsters were placed under pressure from their environment and by other monsters, and this forces Darwinian adaptation to occur, favouring in each generation the monster best suited to survive. It worked, and if left run the monsters would converge to the DNA sequence optimal to their environment.

Limitations of the game from an evolutionary point of view were that the environment was not sufficiently varied to produce two or more ecological niches, and one species always 'won' at the expense of the others. Aesthetically, the evolution was not apparent onscreen either.

2.3 Jinigrid

The ALife demonstrator was the second project the author worked upon whilst at the EPCC. The first project, to package up and expand upon work done in a previous 'Jinigrid' project was found to be unworkable. The decision was made in the sixth week to change projects.

3 Design goals

The following are the aims of the project:

3.1 The World

- The World system should be extremely flexible, and allow the end-user to customise all aspects of the game.
- Whenever possible the World should model the real world in its workings, albeit in a simplified manner.
- The World will have a working climate system dependant on the makeup of the plant and animal population and the terrain. This will act upon the World by altering the available water and the global temperature maximum and minimum.
- The World will have a believably organic looking terrain, with graphical rendering that appeals to young children.
- The World will have two viewable states: top-down (a birds eye view), and first-person (walking on the surface). These need two very different approaches to rendering.
- The World will have a working daytime and nighttime, with varying lighting.

3.2 The Plants

- The World can have an almost infinite variety of plants, which can be created offline by the user introduced into the World at run-time by the user.
- The Plants are sensitive to their environmental parameters. They may or may not tolerate extremes of temperature, or humidity. They may prefer to reside at a certain altitude, or within the proximity of another plant or water
- The Plants propagate according to a variety of flexible rules. Some may 'shoot' seeds, some might just seed joining grid-squares depending on the make-up of its neighbours. They may wait for a given season or temperature to occur.
- The Plants do not evolve graphically! That is a game unto itself. The designer decides the graphical appearance of a given plant off-line.
- The Plants have a biology that includes nutrients and minerals necessary for the monsters to live. This should be sufficiently complex to prevent its limited nature becoming noticeable within the game. It should also be varied enough to support a wide variety of monsters which have each evolved to fit a given ecological niche.

- The Plants will never die of just aging. This is a simplification of the real world.

3.3 The Monsters

- The Monsters should be autonomous, learning, adaptive virtual machines.
- They will contain their physical and behavioural parameters within virtual genes that are passed on to their offspring by a process mirroring the real world.
- They should reproduce by the female laying a coloured egg, and the males fertilizing it. Gestation periods, attractivity of the eggs etc should be features of the species.
- The Monsters must be able to perceive their environment in a variety of ways. They must at least be able to see, hear and smell.
- The Monsters will be graphically interesting to observe. They will have a variety of markings distinctive to the species but varied enough to distinguish individuals.
- The Monsters must also be physically varied, and this physical variety must endow them with certain advantages and disadvantages sufficient to allow adaptation to their environment. This variation will be specific to individuals but species dependent.
- The behaviour of the Monsters must not be hard-coded but should be determined by them themselves. This will prevent a cyclic looking game.
- **The Monsters must not use random number generators to determine any aspect of their behaviour, save when it agreed a human or animal would also make a random action.**
- The Monsters will have a biology system that mirrors the real world. They will be required to find nutrition, and they will expend energy through all actions. The system must take into account their weight and the work done when they move.
- The Monsters will have a metabolism that needs certain elements in varying amounts, e.g. little of some elements may be useful but too much harmful.
- There can exist carnivorous Monsters that must effectively hunt other monsters. These must not be lumbering dreadnoughts of death, but instead as in real life they must be often hungry, with only the good hunters surviving the season.
- The Monsters will die naturally after a given randomly moderated lifespan.
- The Monsters may be named by the end-user, to keep track of them. The monster can also record its lineage, although this data should be stored on disk and loaded upon request.

3.4 The Game

- The Game will allow saving and loading of worlds, as well as insertion of new monsters and plants at run-time. The world can be any size so long as they conform to some minimum requirements.
- The Game will allow the user the control over certain aspects of the graphics of the game, as has come to be expected of games in the last few years, e.g. colour bit-depth, screen resolution, high-medium-low graphical detail settings etc.

- The Game will record and saver users preferences, reloading them at start-up.

4 DirectX

The DirectX API is Microsoft's multimedia interface. It provides a shallow level of abstraction between the programmer and the system's sound and graphics hardware. It is updated in major releases about every two years, and these updates are usually quite drastic. It is easy to criticise Microsoft for this, however it must be remembered that development of graphics hardware's speed and capabilities is greatly outperforming even Moore's optimistic predictions, and these regular updates allow programmers in a 'eye-candy is everything' market easier access to new hardware features. At the time of writing DirectX is on major release version eight (DirectX 8). The graphics and sound aspects of this application are designed for use with DirectX 7. DirectX has guaranteed backward compatibility, so the end user must have at least version seven on their system. Microsoft's Windows 98 shipped with version five, however any commercial game installation of the past two years will have forced an update, so it is expected that most systems will be compatible. The DirectX runtime library is free to distribute, so it is an option to package it with this application when it is a distributable product.

4.1 Direct3D Immediate Mode

DirectX is broken into several APIs, each covering a different aspect of the PC multimedia spectrum. With respect to game graphics, DirectX 7 had three API's, DirectDraw, Direct3D Immediate Mode, and Direct3D Retained mode. The DirectDraw API was primary concerned with bitmap based graphics. It provided very fast BitBlt (bit block transfer) routines, as well as alpha and colour-keyed special effects. The Direct3D Retained mode is a historical quirk. When DirectX was first released, it was to provide the games market with easy access to the graphics capabilities of a PC running Windows. Microsoft created the Direct3D Immediate mode API as a low-level interface with the hardware. On top of this they built Direct3D Retained mode (this may not be its original name) as an OpenGL style API. It contained higher level primitive commands such as 'Scale' and objects such as 'IDirect3DRMMesh', which was a complete logical visual object. However upon release, it was found that commercial companies were favouring to use the Immediate Mode interface, and writing their own wrapper classes and functions, tailored to their game's needs.

With the release of DirectX 8, Microsoft has also dropped the DirectDraw API, to the consternation of many amateur users, as it was easy to use and also the bed-rock of the 3D API architecture. An instance of a DirectDraw object had to be instanciated to get access to any 3D objects prior to DirectX 8. However as any 2D effect can also be done using the 3D API this was a logical withdrawal.

4.2 DirectX SDK

The DirectX SDK is available as a free download from Microsoft [3]. Previously it had also been available as a free CD-ROM, although this has not emerged for version eight as yet. There is a possibility that future versions of DirectX may not be free but instead Microsoft may release

it as a commercial product, possibly if the X-Box platform is successful.

The SDK comes with the headers and libraries necessary for writing DirectX applications, as well as debugging libraries for testing, and distributable run-time libraries. It also only comes with documentation on the current API, which is usually quite different and incompatible with previous versions.

4.3 Graphical data generation

Generating the complex data required to display 3D data is usually one of the more difficult aspects of 3D applications. For this application all 3D objects will be derived from one of two sources.

4.3.1 The terrain

The terrain data is stored in an array of data structures, which contain all the information, needed to render a given square. As the world is divided into a grid, the X and Z location (Y being the up vector) of the start of the square is simply its world X and Z location. Each game terrain square is subdivided equally into four squares, each two triangles. So every terrain square is displayed as eight triangles onscreen, and needs twenty-four unique vertices to do so.

Each terrain square stores four heights in it, suffixed _00, _10, _01, and _11. In this system _00 is the lower left hand side height, _11 the middle of the terrain squares height, _10 the point to the right of _00 and _01 the point above _00. As _00, _10, and _01 are shared with its adjoining squares, then to get all of the information about one square we must examine four squares. The normals required to render the square are stored in a separate array. This is because this array will only be accessed during the generation of the graphics data, whereas the terrain data array may be accessed many times for differing reasons.

4.3.2 Other 3D objects

The 3D data used in the game will be loaded from DXA files. DXA is an ascii file format developed by the author which is easily to use in conjunction with DirectX. 3D Studio Max is the de-facto industry standard application used to generate 3D data. It allows for the exporting of scenes in ascii format as ASE files. A convertor from ASE to DXA is available from http://www.solosnake.fsnet.co.uk/applications.htm. The DXA file format contains only the data needed to render polygon based textured objects. When 3D plants are added to the world, it should not be necessary for the user to have a copy of a 3D data generator. A few basic shapes of common plants can be supplied, and textured as required by the user to create new looking plants, trees and rocks.

The monsters will have as single set of 3D data, shared by all monsters. This data will take the form of body sub components, e.g. head, eyes, feet, tail etc. The genetic material of the monster will detail how this should be rendered in relation to a centre point. Warping matrices can be used to scale, rotate and distort the bodies into species specific traits. This will be a difficult programming task, as the distortions will have to effect the look and behaviour of the

monsters in some way. This is**not** merely to look convincing, but also because for the game to work the monsters must receive some evolutionary advantage or disadvantage from the body configurations. This will allow for Darwinian selection to occur.

A difficult programming task is also at the same time to link the monsters appearance at render time to its activity. This might be accomplished by limiting the monster to several 'routines'. These routines must be applicable to any distortion of the basic monster template.

5 The Game

Hereafter the application and code in its present state will be referred to as 'the game'.

5.1 Programmers foreword

The project is very much 'work-in-progress', and should be regarded as the beginnings of a larger project. The game has two very distinct compilations, release and debug. Conventionally, to compile the debug version just define _DEBUG. The most noticeably difference is the creation of a GE_Debug_Output.txt file when run. This will contain information about the DirectX environment created, including the exact bit-depth and z-buffer depth used, and information about textures created. If an error DirectX error occurs at runtime, its output will often have the source of the error.

The game is quite fault tolerant. Four exceptions may be thrown, and the entire game is run inside a 'try - catch' scope in the main() function. This catches exceptions as below. Unhandled exceptions by definition call the 'terminate' function. In the game a new handler is defined which ensures that the screen is returned safely to windows upon an exception occurring. The game itself will throw only four classes of exceptions, as follows:

- `GECException` The base class exception.
- `GECASMException` Thrown by assembly language routines.
- `GECDirectXException` Thrown (when possible) by incorrect use of DirectX.
- `GECGameLogicException` Thrown by an error that should not have happened if the game was running correctly.
- `GECFileIOException` Thrown to indicate a file error. This includes files not being found, or containing corrupt data.

The intention is to have the game follow the exception rules as above. This allows for different levels of error handling, e.g. file io problems need not crash or stop the game, although game

logic errors perhaps should. When the application is working correctly only io errors should be able to occur. The exceptions also append their output to a text file, to allow for analysis of problems. There are no asm routines in the game as it stands, although the GE library (see below) does contain a few routines for changing the FPU setup.

Note also that much of the program as it stands may be completely change later, depending on speed constraints etc. Unfortunately at the time of writing I have been testing it on an old laptop without graphics hardware, and have no idea of its performance. If you are developing correctly, I would suggest you need two networked PC's, running Windows 98, both with graphics hardware. If you are using Microsoft's Visual Studio, you can easily run the application on one PC whilst stepping through the code on the other. This is invaluable for finding tricky fullscreen bugs.

5.2 The GE Code used

All classes and code files prefixed by GE e.g. GEConsole were written by the author Dáire Stockdale in months prior to coming to the EPCC, and are part of a longer term project to develop a generic game engine to use as a 'demo'. As such, this code should be considered the property of the author. The code is provided for use with the project as is, and any problems or suggestions should be addressed to the author[4]. Dáire Stockdale is committed to supporting the code, and will be improving and expanding on it over the next two years.

5.3 File dependancies

The game requires several files of differing types. Upon startup it attempts to load a file named 'M2.dat'. This contains the games default startup parameters, as bit flags in a 32 bit unsigned integer, and also the names of the bitmaps used for the loading and splash screens. This file is saved again upon exit. The purpose of this is to 'remember' the user's settings here. If the game is stated using the 'Start Menu' program (see below) then the flags passed are stored in the M2.dat file. The M2.dat file can be generated from scratch using the 'Restore.exe' that comes with the game.

Once successfully started, the game reads a list of 'world' directories (levels) from a text file named 'M2.worlds'. These are displayed for the user to choose which game to play. No more then the first ten will be read from the file. If an error occurs at this stage the program throws an exception and terminates. Later versions should remove the offending level from the file, and present the user with a safe loading screen again. Later versions of the 'WorldMaker' application should prompt the user to create or modify the current 'M2.worlds' file.

5.4 Command line parameters

Upon execution the game parses the command line for two unsigned integers. The first of these contains startup data (bit depth, verbose on/off, screen size, speed etc), the second is included should further information need to be passed. If these are not found the flags contained in the M2.dat file are used. The final version should include the ability to start a level by double clicking on it. This needs a registry entry to associate a file extension with the game, and also

modification of the games loading function. At present it is tied to the displaying of the levels. This should be separated into two functions, one of which takes only the name of the level to load, and returns true if it was successful.

5.5 The Game start menu program

The ‘Start Menu’ program is a modified version of a MFC application written to accompany any games developed using the GE code (See above). It checks the hardware of the system, and displays the compatible bit depths and resolutions possible on the users system. In the Monsters2 version of the menu it also has a ‘verbose’ checkbox. When the ‘Start’ button is pressed the game is started, and passed the flags chosen by the user. These flags can be guaranteed to be compatible with the system, and relieves the game of the need to enumerate the hardware. Note that the enumeration does still occur, and is visible in the output of the debug mode. However it is enumeration looking for the chosen setup, not enumerating all the permutations.

5.6 The Restore program

The ‘restore.exe’ creates the M2.dat file. This file is the first file looked for upon startup, and contains the users default screen configuration, and the names of the bitmaps used during loading and starting. The application only allows you a limited choice of ‘safe’ options. The M2.dat file is saved again upon normal termination of the game, to reflect any changes that may have been made during the game.

6 The World

The original intention of the author was to have the class named M2World be completely responsible for the games running, and the class named M2Game be responsible only for the creation of the DirectX environment, and the user interface. However this would have meant the world rendering itself, and any class that is required to do any rendering must be tied closely to the graphics API used, and part of the design intention was to avoid overly tying the game logic to the graphics. For this reason much of the game workings have been drawn into the M2Game class, and the M2World class is now little more then a data structure. This may be only temporary, depending on how the application evolves.

Once a level is loaded, the game loop looks something like this:

```
M2Game::Play()
{
Advance game clock;
Check user input and update ‘controls’;
Change viewpoint and game based on ‘controls’;
Update World;
Render World;
}
```

6.1 Time

The world has a day length, measured in clock ticks. Each game clock tick is 10 milliseconds. This may change if finer granularity is found to be useful. The default day length is 360,000, or one human hour per game day. Each call of 'Update()' to the world will pass to it the elapsed time in game ticks since the last call, to allow the world to keep time with the real world. To 'fast-forward' the world, all that is required is a loop that calls Update continuously with small intervals, without rendering in between.

It is a design goal to have the world lighting change visibly with the time of day. The main source of light is the m_sunlight of the M2Game class. The position, colour and intensity of this light can be altered to reflect the time of day, and it will be possible to have nighttime in the game too. Provision for a night-time texture has already been made in the world file format. See Rendering for details on how this might be implemented.

When the M2World class records that a day has passed it will call the Update function of the M2Climate, which advances the date. If it notes that a new season has been entered it will call the MakeTemperatureArray() function of the Terrain Manager. However this could lead to large temperature jumps, and sudden graphical changes onscreen if the seasonal variations are large. A better approach is to have the temperature try to rise or fall to meet the new maximum and minimums, incrementing the change each world day.

6.2 The terrain grid

The terrain is a grid of n x m squares, where m%n == 0, and n is a power of 2. This is needed for to make the fractal generation process easier. See Terrain generation. The singleton M2TerrainManager (see below) owns the grid. The terrain grid is held in a TVEC2DWRAP<> class. This is a templated array that emulates a 2D array, and also allows access via negative indexes. This allows the world grid to 'wrap', and the user can walk into the negative indices without causing any problems. The down side to this is that each access to the grid must translate the indices into a single safe index. For this reason when I know multiple accesses will be made to the same grid square in a routine, I create a const reference to it and reuse this.

6.3 The terrain manager

The terrain manager is a singleton (there is only one). Singletons are used in the game code to enforce uniqueness, and also to allow global access from any point in the application without actually creating globals. It is likely that everything in the game will at some stage need access to the terrain data, as everything in the world will be interacting with the terrain in some way.

The terrain manager is responsible for loading and saving the terrain grid, the array of normal indices, and the normal palette. To save memory, instead of storing each vertices normal, which requires 12 bytes of data, the normals have been palettised, and only a single byte sized index has been stored. The normal palette contains 256 normals; these are all 'up' normals, and are equally spaced out over the surface of a hemisphere.

The terrain manager is responsible for the water table of the terrain, and also the temperature at each terrain square. The terrain's temperature varies along the x-axis (think of it as latitude) according to the following formula:

$$Tsquare = Tmin + (Tmax - Tmin) * \sin(\Pi * \frac{X}{Width})^2$$

This should give extremes of hot and cold equal to the seasonal maximum and minimum at the 'poles', with smoothly varying temperatures in between.

The terrain manager also has two variables named fHeightHumidFactor and fHeightTempFactor. These are used to vary the humidity of any given square based upon its height above sea level. As the height above sea level rises, the temperature at that square will drop. The fHeightTempFactor is the amount it drops by per unit above sea level, and similarly for the humidity.

6.4 Climate

The M2Climate class is a singleton responsible the world's seasons. It has an array of four M2Seasons, each of which has a maximum and minimum temperature, a base humidity, and a length (in world days). The base temperature at any latitude on the world (its X co-ordinate) is found by indexing into the terrain managers array with that squares X co-ordinate. The seasonal maximum and minimum do not have to be static, but instead can themselves be functions of the world population statistics. It should be possible to mimic greenhouse effects, just by having some or all types of plants moderate the world maximum and minimum temperature. This may require per tick or per day calculation, which may be computationally expensive. Per season change calculation could be used; if so the world will seem to react slowly but with momentum to changes in its makeup. The change takes the form of an additive modifier to the seasonal maximum and minimum.

6.5 Water

Water in the world is implemented by the concept of a water table. All points below this height, which is a member of the terrain manager, are below the waterline. At the moment, the fact that a square or sub square is below the water table is signaled in two ways: either by comparing its heights with the water table height, or by checking its flags. As the comparison with the height may be slow, each square maintains an 'underwater' flag for each of its four sub-squares. If the water table changes, and hopefully this will not be often, the entire grid must be updated.

6.6 Terrain generation

The terrain is generated by the accompanying application 'WorldMaker'. The 'WorldMaker' program allows the user to input the parameters required to create an entire world, under four categories: world, viewer, terrain and climate. When generating terrain, the user must input the width, the height, the fractal dimension and a shared angle value.

6.6.1 Width and Height

The terrain is generated as a continuous fractal terrain, based on Stephen Booth's 'xmountains' [5]. The terrain wraps about itself in both the X and Z axis. I have assumed that the height will always be less then or equal to the width. The width must be an integer multiple of the height, to allow the world to be divided into squares during the fractal generation. Also, to allow continuous recursive division, the world height must be a power of two.

6.6.2 The fractal dimension

During creation, each new point's height is based on the average of its surrounding point's heights. This average is modified by an offset as follows:

$$H = U + F * D^{2*Q}$$

Where H = the new height, U = the mean of the four surrounding heights, F = a random number from a Gaussian distribution, D = distance to the surrounding heights, Q = fractal dimension.

This controls how 'smooth' or 'rough' a surface is. The height scaling factor should be used to generate flat or hilly terrain.

6.6.3 The shared angle value

This value is concerned with how the terrain looks when rendered. As our polygon mesh is not extremely dense, and is intended to represent an organic surface, we would like it to be smoothly curving in appearance. In real time 3D graphics APIs the lighting is calculated most often per vertex, and interpolated across the triangle surface, using either Phong or Gouraud shading. The colour at the vertex is in turn calculated by the interaction between the world's lights and the vertex normal. If triangles meeting at a point all share the same normal at that point (and are of the same material), they must also share the same colour at that point. Thus the colour will vary smoothly over the terrain at that point. If the normals are different, then different colours will appear to meet at that point, and it will appear angular.

During terrain generation, the normals to the surfaces of all the triangles that share any point are averaged. If the actual normal to the current vertex being set differs from this average by more then the shared angle value, then the vertex does not get this average as its normal, but instead a normal moderated towards the average is given.

Setting the shared angle value very high (measured in radians) means that most vertices will have the same normal, and the terrain will appear smooth, but it will have strange shadowing artifacts. These are produced when one of the faces meeting at a vertex differs greatly from the others, and throws off the average by enough so that the new, average normal subtends greater then $\Pi/2$ with the angle of the light. All of the vertices now appear to be in shadow, although in fact no shadow could be possible.

If the shared angle value is set low, many of the vertices will have their own individual normal, different to its neighbours. This creates a very angular and unrealistic looking landscape.

Shown below is two landscapes which differ only by their shared angle value. The left landscape has a value of 3.0 Pi, the right is using 0.1 Pi.

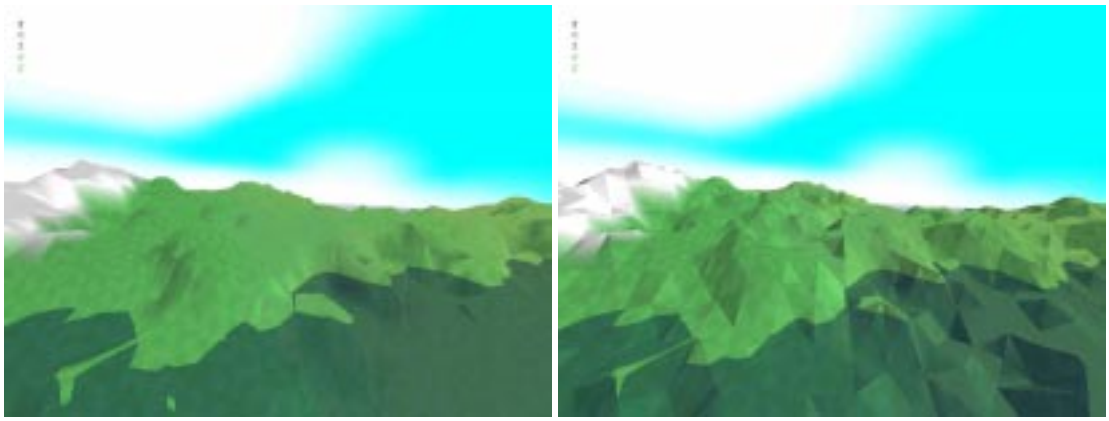


Figure 1: Demonstration of different shared angle values.

The only cure for this is to increase the resolution of the terrain so that any glitches are too small to be of concern.

6.6.4 Height scaling factor

The height scaling factor simply scales the heights generate by the fractal functions by a factor entered by the user. The default scaling factor is 1.0. The fractal dimension controls how ‘rough’ the surface appears, and the height scaling factor controls how ‘hilly’ that surface is. Scaling a smooth surface produces worlds that are most compatible with the planned creatures. Overly undulating surfaces may cause movement problems and/or unrealistic behaviour.

7 Rendering

7.1 The terrain

To render the terrain we must first generate the data. To do this we need to know exactly what data falls into the view frustrum. Although not implemented yet, to do this is simply to multiply the view frustrum extents, held in the M2Viewer object, by the current view matrix. This will translate the view frustrum into the world. We need only be concerned with the X and Z components. Find the boundaries of the minimum box that contains this projection; this is the area to be rendered.

The data generated will be held in an IDirect3DvertexBuffer7. This is an array of renderable data that can be processed and optimised. Processing a vertex buffer means transforming it into screen space. It is useful to process vertex buffers which will be rendered multiple times per frame, as ours will.

Later we will need to sort the renderable data according to what textures it needs. To actually sort the renderable data structures would be very expensive, as each renderable ‘unit’ is composed of 24 vertices, each 36 bytes in size. Instead we can sort indexes into them, and make use of

DirectX's ability to render indexed arrays. Only the location of the start of each squares data in the vertex buffer need be recorded, along with whatever texture is associated with it. Sort these, then generate the indexes, and pass these to the renderer.

The maximum number of vertices DirectX can render in a single call is 65,535 or 0xFFFF. This can quickly and easily be exceeded. If the visible area is 53 x 53 in size, this needs 67,416 vertices.

The base texture applied is the default texture specified by the world, loaded as m_szBaseTexture and associated with m_BaseTexD.

7.2 Textures

Textures are images applied to the primitive's surface. They are commonly loaded from bitmaps. Most are square, as some hardware requires this, and some hardware requires them to be of side-length a power of 2, for mip-mapping. When a world is loaded, the game expects to find in the worlds sub-directory named 'Textures' the textures listed in the world file. In this directory is created an html file by the 'WorldMaker' application, listing the files it expects to find there.

During the third week the author experimented with texturing the colder areas with snow, and also texturing the surfaces lying underwater with a 'pebble' texture. Neither was found to look very satisfactory, and also they incur costly sorts upon the secondary texture.

At present the game requires the following textures per world:

- A day texture, for the sky.
- A night texture, for the sky.
- A base texture, for the ground surface.
- A subwater texture.
- An ice texture.
- A water texture.
- A sun texture.
- A flare texture.

7.2.1 Day and Night textures

The day texture is the daytime sky texture applied to the skysphere. Later this will be modulated with the night texture during twilight, and then the night texture applied at night.

7.2.2 Base texture

This is the texture used to cover all the terrain, upon which secondary plant textures will be applied. It should be subtle to allow the flowers to be displayed against it. Note that the textures also contribute a lot to the effect of perspective, as they can be seen to grow smaller with distance.

7.2.3 Subwater texture

After experimenting with the subwater textures I have decided to discontinue with this idea.

7.2.4 Water texture

The water texture is the texture applied to the surface used to represent the water. Note that this is modulated with a blue material, and transparency is applied.

The following textures are for future development:

7.2.5 Ice texture

The Ice texture is intended to be used on areas of water completely bounded by land that is below zero. However this might be complicated to calculate. See The Water below.

7.2.6 Sun texture

It would be nice to have a visible sun in the sky, whose altitude varies with the time of day. To do this requires calculating the visibility of the sun, which requires per polygon ray intersection testing. If the sun is found to be visible, it must be 'billboarded' into place, i.e. displayed so that it is also facing the viewer. This is done by using pretransformed co-ordinates for its polygons. By creating the texture as an image against a black background and using a material with only emissive value for its red, blue and green, the sun can be made to shine against the background.

7.2.7 Flare texture

A lens flare is the result of light interacting with the lens of a camera, and is not perceived normally by the human eye. However it can be nice to add it to a game, for cinematic effect. If the calculations for the sun have been done, it is trivial to add a lens flare. It should be located at some point along a line joining the centre of the viewport to the sun's location, and rendered in the same manner as the sun.

7.3 The water

The water is rendered as a sheet of squares, presently 30 x 30. It must be rendered as a mesh of reasonable density for the fog to look correct, as the game uses vertex fog. See Fog below. I have animated the water for the moment, and the effect is very nice over the larger open stretches of water. However inland, in areas that are close to the height of the water table, the slight rising and falling of the water causes it to appear and disappear in an unrealistic manner. Choosing not to animate the water will free up some processing time and eliminate this problem.

The water appears to fill the terrain lying below the water level very naturally. We have in a sense got this effect 'for free'. We do not calculate where the water is, just render a square sheet

of translucent water. Where it cuts the terrain and becomes visible is calculated in the z-buffer for us by DirectX. If however we wanted to make the areas of water that were completely surrounded by sub-zero land rendered as ice (textured with ice) we would need to treat the water as individual units, and test each polygon to see if it was in a sub-zero area, and not connected to any above zero areas. This is computationally very expensive.

7.4 The sky

The sky is a sphere that moves with the viewer, so it is always the same distance to it along to X and Z axis. It does not translate along the Y axis however. If the sky appeared to move as we walked (the effect generated by actually not moving the sky with the viewer) then we would perceive that we were inside a finite sphere.

The lighting and fog is switched off rendering the sky. This is to prevent any shading that might demonstrate it is a sphere. As the sky sphere is always the furthest visible object away from us, fogging must be disabled to see it.

The sky sphere is loaded from a DXA format file [6] named skybox.dxa. It is scaled at runtime by a factor dependant on the far plane of the view frustrum, so that it is always far away from the viewer.

7.5 The fog

The fog is used in the game to create the impression of distance. There are various fogs available in DirectX. Here the game uses the most commonly available fog, vertex fog. Vertex fog means that the colour of the fog at the vertex's point is calculated and interpolated across the primitive. This works well for terrain vertices in the distance, as we have a high density of small triangles. However it means we cannot just use two triangles for the water plane, as the furthest vertices would always be in dense fog, and the near in none, resulting in water that clouded as it went up the screen.

The fogs colour is loaded from file by the M2World object, and its start and end points calculated based on the distance to the far plane of the view frustrum.

7.6 The world at present

Currently the world can load and display world created by the WorldMaker application. Movement of the viewpoint is still not implemented, although it is very close. Shown below is an early world; on the left hand side can be seen white snowy regions due to the lower temperature of the left hand side, which in this picture happens to be the worlds arctic regions. Small worlds (this one is 64 x 64) with large temperature ranges may display rapidly changing temperature bands along the X-axis.

8 Looking ahead

In order of implementation, the following must be done before work can begin on the plant life proper:

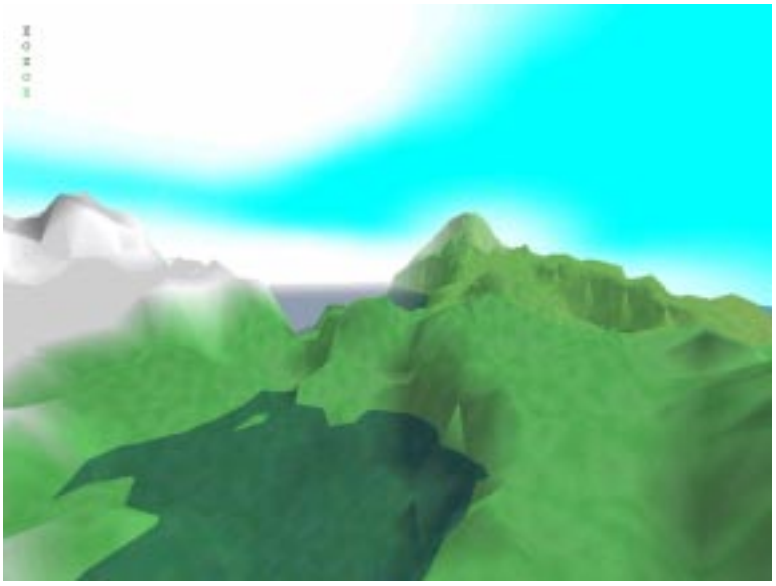


Figure 2: Sample world.

- Movement of the viewpoint must be implemented. This first requires the M2MoveableObject class to be finished, and collision detection implemented.
- Agree on the look of the terrain as being satisfactory, and begin to address other aspects.
- Before the plant file format can be developed the world biology must be decided. This will be the basis for the food chain of the monsters.
- Develop a flexible plant file format capable of holding multiple textures and a method of encoding the plants seeding mechanism and preferances, as well as the plants nutritional values.
- Add a plant manager class responsible for loading the plants in the world. Add loading of this class to the world loading routine and file format. Develop a plant creator program.
- The same must be done but with 3D data attached for renderable plants.
- Incorporate changing seasons and time of day to the game. The nighttime must be convincing whilst still being useable.
- The influence the plants have upon the climate must be decided and tuned.
- Test the world. If it is working then development of the creatures can begin.

8.1 The plants

There are to be two types of plants. The first is similar to the manner in which the plants are represented in the original monsters, simply by overlaying the base texture of the terrain with a plant texture. Using per-colour transparancy, in which one nominated colour is to be transparent, the flowers and low-lying plants can be applied to the ground in a manner pleasing to the eye. During render time if the square being parsed has a plant on it, a referance to it is passed to the plant manager. The plant manager maintains an array of the plant species. The plant species

returns a texture ID based upon the current parameters of the terrain square. The other type of plant is similar to the first, but has 3D data. These can be used to represent trees, shrubs etc. These must be able to render themselves and so will be tied to the graphics API, at least for this call.

8.2 The monsters

The monsters are still a long way off. When the world is working correctly then creation of the monsters can begin. It is essential that they have an environment varied enough to reward specialisation. Within real eco-systems this is called 'niche-specialisation', and it is this which produces the variety we see upon the Earth.

9 Conclusion

This project is very much in its infancy, and the author intends to develop it at home whether or not it receives SCOPE funding. Unfortunately there was not more time to add the many features mentioned. In most respects the code is very close to having varying day and night and movement. Difficult aspects of the project include the fact that the world must simply look nice. This often involves changing small parameters in the code, such as material colour, and recompiling the scene. Changes are made to the code based on nothing more then how the scene looks, and this can become tedious and subjective.

10 Notes

1. Scientific Computing for the Public of Europe.
2. The original may be found here: http://www.solosnake.fsnet.co.uk/application.htm#monsters.
3. The current DirectX SDK is available at: http://msdn.microsoft.com/downloads.
4. E-mail: solosnake@hotmail.com
5. Details of Xmountains, including source code, is available from: http://www.epcc.ed.ac.uk/~spb/xmountains.
6. DXA format is a DirectX friendly ascii file format. Details of the format and a convertor from 3D Studio Max's Ascii Scene Exportor file format to DXA may be found at http://www.solosnake.fsnet.co.uk/main/dxa.htm.

11 Suggested reading

- 'Inside Direct3D', Kovach, Peter J., Microsoft Press, ISBN: 0735606137
- 'The Blind Watchmaker', Dawkins, R., W.W. Norton & Company; ISBN: 0393315703

Dáire Stockdale is a student of Computer Games Technology at the University of Abertay Dundee. His interests outside of programming include reading or watching good science-fiction and looking after his pet rats. Inside of programming his interests include C++, DirectX and game engine design. He maintains a website at http://www.solosnake.fsnet.co.uk, where an explanation of ‘solosnake’ may be found.

The Alide Demonstrator was supervised by Neil Chue Hong and David Henty. The Jinigrid project was supervised by David Henty.