# Deformable Terrain Generation for Real-time Strategy Game

Christopher Davison and Wen Tang

School of Computing and Mathematics, The University of Teesside, Middlesbrough, United Kingdom

## Abstract

*In this paper, we present a system that has the ability to deform terrain for Real-time Strategy Game with general PC hardware specifications. Various effects could be simulated in real-time such as raising and lowering the ground, creating a large chasm, or levelling the terrain. Effects such as ordering a unit to fire that then destroys part of a mountain could also be replicated. We present our implementations and techniques in using terrain deformation algorithms, Real-time Optimally Adapting Meshes, texture design and terrain generations. An analysis of the speed and memory usage of the system with respect to different PC hardware systems is also presented.*

## 1. Introduction

Deformable terrain is a feature rarely found in current Real-time Strategy Games (RTS). A system that allows to deform the terrain and to see the results on screen has great potential in terms of both fun and strategies of game-play.

Computer games are continuously evolving. Technology that is new and fancy today will be old and worn tomorrow and the PC games industry is always looking for new ways to improve their games; How can they take advantage of the never ending advances in CPU and graphics card speed? What can be done in a game depends upon many factors, however the speed of average users' computers is possibly the most limiting of all.

Advances in graphics tend to happen first followed later by advances in game play features. In this paper, we present our system, which is focused on adding significant deformable terrain game play features to an existing genre.

In many computer games, the traditional isometric view has been changed into a fully 3D environment. Total Annihilation by Cavedog Entertainment [1] was released in 1997 and was the first 3D RTS game. Upon release, it was considered one of the best RTS games and was a benchmark against which many others were measured. The terrain was 3D with accurate line of sight and arcs of fire. The units were also 3D models, however the rest was very much simpler. The terrain was based upon a tile system and it was not possible to deform the terrain at all.

Some of the weapons in the game were huge weapons of mass destruction. One in particular was extremely powerful, the 'Big Bertha'. It was a very long range and deadly, but quite inaccurate artillery gun. The building in the game took a long time to build and each shot used up an amazing amount of resources, but if it hit a building, the building was destroyed. One of the key considerations was where to build the building. Other than making sure it was in range, the most vital consideration was that its arc of fire could hit the enemy base. Getting this wrong would mean that the shells trajectory would cause it to hit the top of the mountain rather than go over and into your enemy's base. One of the disappointments with the game was that if it could destroy any building in one shot, why would the mountain be unharmed?

Populous 3: The Beginning by Bullfrog [2] was released about 2 years ago and was among the first 3D RTS game to allow the terrain to be deformed. This game had a unique terrain system within which players can raise and lower the terrain, create earthquakes and even create volcanoes. However, there were many restrictions to what was possible. There are a limited number of heights for the terrain, about six. The terrain works by having each effect use a kind of template. There are many places where a deformation cannot be done, such as a volcano on high ground. In addition, the effects are not very accurate. This is because the terrain works by recognising set combinations and patterns and then knows how to translate this into an appropriate grid of polygons. This method works very well, but it relies upon constraining the possibilities. One that would not be possible with this technique would be a small dent in the terrain, for example.

Tread Marks [3] is an arcade style 3D battle tank combat and racing game written by Longbow Digital Arts. This game proved very interesting due to its impressive terrain and how well it makes use of it. This is a good example of where adding deformable terrain to a game has raised the enjoyment level and 'fun factor' of the game.

To develop a RTS game with incorprated terrain deformation features would raise a number of challenges in both technique implementations and system design. From terrain generation point of view, there are areas to be considered:

- How to define the terrain.
- How to store the terrain internally.
- How to render the terrain.
- How to deform the terrain.

Terrain generation and rendering is certainly not a new topic and many research methods have been published in the past [4, 5, 6, 7]. These methods vary greatly in complexity and quality, most of them emphasise on terrain visualization and, however, few of them deal with terrain deformations. This is where the challenge lay.

In this paper, we present a system that has the ability to deform terrain for Real-time Strategy Game with general PC hardware specifications. Various effects could be simulated in real-time such as raising and lowering the ground, creating a large chasm, or

levelling the terrain. Effects such as ordering a unit to fire that then destroys part of a mountain could also be replicated. We present our implementations and techniques in using terrain deformation algorithms, Real-time Optimally Adapting Meshes, texture design and terrain generations. Our system design and implementations are described in the next section. An analysis of the speed and memory usage of the system with respect to different PC hardware systems is described in section three. The final section of the paper, section 4, presents the results and discussion.

## 2. Algorithms for Terrain Generation

The manin consideration in choosing which method in our terrain deformation system is two fold. The method should allow fast and efficient terrain generations and deformations on general PCs avaiable to most players. The techniques should also allow the most terrain rendering task to be taken effectivly by graphics card so that more CPU time available for other areas of the game such as 3D sound and artificial intelligence.

### 2.1 Rendering consderations – voxels rendering vs. polygon rendering

Voxel rendering method [8, 9,] has been used in many games for terrain generation [10] with great success. The advantages of the method are that it gives landscapes an extremely detailed and "organic" look due to the ray casting method. Whilst, with polygons, the ground of terrain is often flat, or at best, it is made up of flat areas that give the ground a looking unevenness. It takes too many polygons to draw natural looking ground for current 3D cards. Once the terrain has been updated, the rays cast for rendering will reflect the new terrain shape. A second advantage is that with some optimisations, such as the one introduced by Cohen-Or et al [11], is that a form of level of detail (LOD) is implemented automatically. Despite many advantages in voxel techniques and the advances in speed [12, 13], they remain too slow. Determining where the ray intersects is highly processor intensive. However the major drawback of the algorithm is the lack of hardware acceleration. Modern consumer level graphics cards only accelerate polygon data. With the huge difference that hardware acceleration can make, voxel rendering does not look to be the way for our implementation This is becoming increasingly true every month as faster graphics cards are released that allow a greatly increased number of polygons to be used at any one time. Now those terrains that once would not have been feasible, are gradually becoming achievable. Consequently our implementation is a polygon based rendering system.

### 2.2 Binary Triangle Trees for Terrain Deformation

Duchaineau et al [4] presented a method for maintaining an optimal polygon mesh. The basic structure is using binary triangle trees. There are several advantages of this structure, firstly, each node only has two descendants, and secondly it uses triangles, which are simpler to deal with than voxels. Duchaineau et al. went on to show how this structure could be used to maintain an optimal mesh that includes LOD techniques. Their method splits and merges triangles as necessary according to terrain variation, distance from camera and maximum polygon count. Finally, as an added on advantage of the data structure, highly efficient frustum culling can also be achieved.

The main concern with this technique is that the generated terrain is not in an efficient form for optimal rendering speed on modern graphics hardware. However, it has been suggested that by reordering the triangles slightly, either triangle fans or triangle strips could be created which is far more efficient. Overall, this method is highly suited to our task. Not only is it simple, efficient and suitable for games, it also can easily cope with deformations.
Our terrain deformation implementation is based on Real-time Optimally Adapting Meshes. In our system, the terrain could be loaded from a height-map, converted to a polygon mesh and then, during the game, be updated and continuously optimised. It also allows users to configure the detail settings in order to run optimally on their personal machine. The algorithm could even be aware of current frame rates and consider this when optimising the terrain [5].

The binary triangle structure is the key component of the program. It deals only with right-angled isosceles triangles. At the top level, there is one triangle, which then has two descendants, and each of them have two descendants etc as shown in figure 1. This is fixed; each node either has no descendants or two and the shape of the triangles are always the same.

Since each level of subdivision does not replace the level above, the number of triangles after n levels of division is: $2^{(n+1)} - 1$. Although this is quite a high number of triangles, only the bottom level are drawn while the rest can be used for frustum culling and collision detection algorithms.

Each triangle is stored as a node, with each node being aware of its 'parent' and its two 'children' as shown in figure 2. It is important to mention that since the structure contains only triangles, and the terrain to be built is square, there must actually be two trees at all times. These structures are linked together and are often treated as one. Each node has to keep track of several things in order for the structure to be maintained. Each node needs to know its three neighbours. These are highly important for coping when a triangle has to be split. Details of the algorithm are in reference by Duchaineau et al [4]. The details of our implementation are listed in Appendix A.

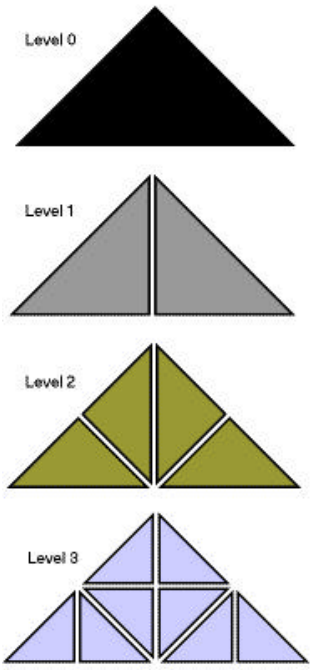


**Figure 1.** *This diagram demonstrates how the triangles are split.*

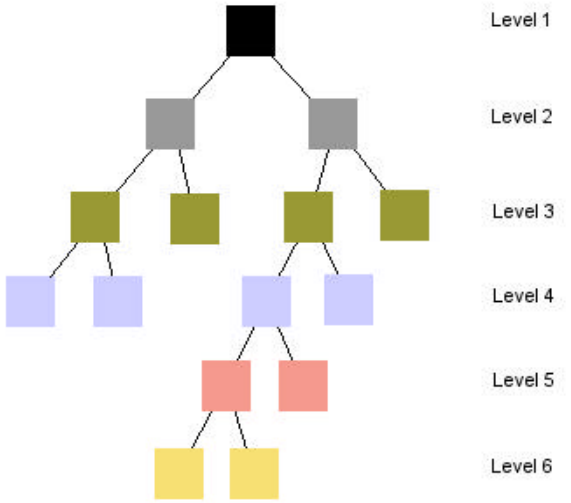


**Figure 2:** *The resulting tree shaped pattern*

## 2.3 Texturing

The second consideration for our system design is texturing. Texturing a terrain can be a complex task. "Texturing has been the single biggest headache with landscapes in the game - allowing the mission editor to select a texture for every square and dynamically generate the combination textures for squares that are at a lower detail level, as well as automatically texturing the terrain based on vertex material." [14]

Applying a texturing to each polygon will not work satisfactory unless the textures are applied with great precision. The challenge is that the textures are square but the triangle that it is being applied to it is not, so obvious texture can occur. The second concern with texturing is that because the terrain is deformable, the textures would need to be updated accordingly. A final consideration is the number of textures applied to the terrain, up to four textures might need to be applied to each triangle. This would result in a large performance hit even on state of the art graphics cards as the most number of textures the cards can process in one pass is three. If four were used, then two passes would be needed for the graphics card to draw the triangle. This would reduce the speed of the rendering time by half. Therefore, a new approach has to be taken.

An alternative is to make use of unique textures for the terrain instead of tiling textures [15]. With modern graphics cards having up to 64MB of memory, it seems sensible to use this memory on the card and using more than one or two textures for terrain, which will add visual richness to a game. This is the approach taken for the implementation in our system. The benefits are numerous.

- The terrain can be textured using just one texture. This leaves a second or even third textures to perform special effects such as dynamic lighting using light-maps or even moving water effects, without incurring a major speed hit.
- The generated textures can be made by blending any number of other textures together. For example, rock, grass, mud and shrapnel damage textures could all be blended into one.
- By generating the terrain at load time and only updating a small part of it at any one time, texture generation is kept to a minimum during the game.
- The generated textures can be created once and then saved on to the hard disk for future use.
- Incorporating mipmapping is as simple as enabling it in a standard graphics API.

Rather than generate one big texture, the system should be able to split up the texture into several smaller parts. For example, a 512x512 terrain could be split into 4 square generated textures or a 1024x1024 might be split into 16. However, the size of these textures needs to be considered. Using too many or too few could be highly inefficient and therefore the size of these generated textures should be configurable. It is also worth noting that some older graphics cards, such as the Voodoo 3, have a maximum texture resolution of 256x256. Therefore, this should be the default size.

A simple image height map can be used to store the terrain height information. The advantages are convenience, rather than having to write ones own file tools to createthe maps, standard image editing packages can be used, such as Paint Shop Pro or Adobe PhotoShop. There is however one major drawback with a standard height map. The problem is from the fact that the system is only storing a height value at each location, no texture information is included. This indicates that texture information has to be derived from the height values alone. A more conventional way of generating terrain with textures involves linking certain heights with certain textures. For example, low terrain might be sand and water while high terrain might be snow and rock. While this system is adequate, its main draw back is that mountain lakes for example would never be generated. Although this is not a huge problem, it does limit the freedom and variety of the maps. To overcome this drawback a two-part height map is used in our system. The first image is a standard grey scale height map therefore it allows 256 different heights. The second map uses several different key colours to represent different terrain types. The advantage of this method is that terrain designer can place complex terrain types in a game such as a sandy beach and a lake high up in the mountains.

The terrain generated in our system uses two types of image maps: actual height map and the second is the terrain type map as shown in figure 3. Figure 3a shows the actual height map and the figure3b shows the terrain type map. Green represents grass, yellow represents sand and blue as water. The rendered result is shown in figure 4.

The screen shot in figure 5 shows a large terrain based upon a 1024x1024 height map. Some of the main features include snow peaked volcano with lake in crater, jagged rocks jutting out of sea, and small rivers running down the sides of the mountain can be seen. All of these are possible with the high-resolution height maps and pixel level accuracy of the texture generation. Fine details can easily be incorporated in the map. The two maps, height map and terrain type map, which created the terrain, are shown in figure 6a and figure 6b respectively.

## 2.4 Control system

The camera and control system designed in our system are to be simple and easy to use as it is intended for using in a game. Several camera movements are needed to be controlled. They are:

- Moving the camera forward, backwards, left and right.
- Moving the camera up and down.
- Rotating the camera left and right.

The keys chosen are based upon several factors, but mainly those which would be obvious, convenient and close together as shown in table 1.

| | |
|---|---|
| Camera Forward | Up Arrow |
| Camera Back | Down Arrow |
| Camera Left | Left Arrow |
| Camera Right | Right Arrow |
| Camera Up | End |
| Camera Down | Home |
| Rotate Left | Delete |
| Rotate Right | Pg Down |

**Table 1:** *Control keys*

## 3. Speed and Memory Analysis

The speed and memory analysis procedure to test the system has been run on a variety of other machines.

Two elements were tested, firstly the terrain rendering speed, in frames per second and secondly the time taken to perform a terrain demonstration program, timed in milliseconds. Both of tests were performed at 640x480 in windowed mode using the smaller 512x512 world with the cameras in the same place. The reason for using the smaller window and world size is so that all machines have a possibility to run the program at an adequate speed. The desktop colour depth was set to 16-bit colour if possible.

The first test was run on ten PC's with varying specifications. The objective was to try as wider range of CPU's, graphics cards and memory configurations as possible. The test machines, listed in order of CPU MHz speed, are as follows:

| | CPU | Graphics Card | Mem. | OS |
|---|---|---|---|---|
| **A** | Intel PII 300 | TNT2 M64 | 196MB | Win98 |
| **B** | AMD K6-2 350 | TNT | 128MB | Win2k |
| **C** | 2 Intel PII 350 | GeForce DDR | 384MB | Win2k |
| **D** | Intel Celeron 366 | TNT | 64MB | Win98 |
| **E** | Intel Celeron 400 | TNT2 Ultra | 320MB | Win2k |
| **F** | 2 Intel PII 450 | 3DLabs Glint | 512MB | Win2k |
| **G** | Intel PIII 600 | ATI Rage Pro 128 | 128MB | Win2k |
| **H** | Intel PIII 800 | GeForce2 GTS | 256MB | WinME |
| **I** | AMD Duron 900 | GeForce2 MX | 196MB | WinME |
| **J** | AMD Athlon 1.3 | GeForce2 GTS | 512MB | Win2k |

**Table 2:** *Ten PC specifications*

There are a large variety of specifications in the test machines, with all factors varying greatly. Also included in the table 2 are the different operating systems run on each machine. This information is included because benchmarks were significantly higher on machines running Windows Millennium Edition (WinME) than that on Windows 2000 Professional (Win2k). However there are still many other variables to be taken into account such as differing driver versions, memory speeds and so forth. The results table 3 shows the number of frames per second that each of the test machines obtained.

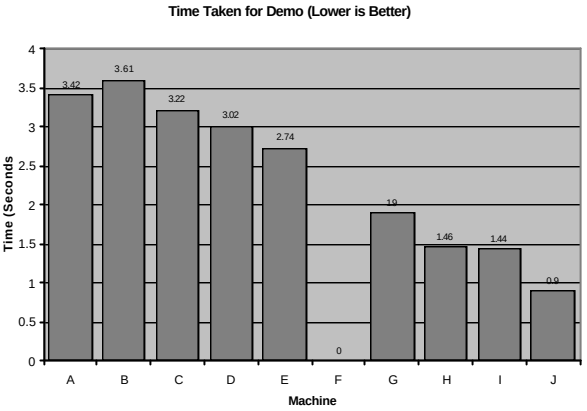


**Table 3:** *The number of frames per second on each of the test machines*

A demonstration program is used and timed to test the terrain deformation time. The results are shown in table 4 in which the graph shows that the biggest influence on frame rate is the speed of the graphics card. This is not surprising as when nothing is moving, there is very little for the CPU to process. However, the reverse is true.

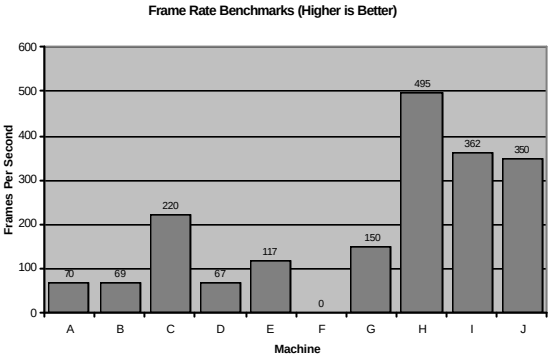


**Table 4:** *Frame rate Benchmarks*

The program ran on almost all tested machines with the most common hardware specifications and the speed was acceptable under the circumstances. The terrain is highly detailed and extremely adapting. As the above testing showed, the frame rates are high, however the time taken to perform the deformations is still long. The reason for this is that our system codes have not been significantly optimised and many improvements can be done in this area.

## 4. Disscussions and Conclusion

Binary triangle trees have been proved to be highly robust and very efficient for terrain deformations. The capability of the method to generate a triangle mesh both quickly and with controllable complexity is very suited to RTS games. Keeping the triangle count down is very important to maintain acceptable frame rates. The end result of the terrain deformation implementation can be seen in the screen shots in figure 7, which shows only a smaller part of a terrain textured. The image in figure 8 shows the same terrain but in wire frame. The different sizes of triangle are clearly visible. The bands of rocky outcrops can be seen where there is a steep increase in the slope of the terrain. Also the darker areas at the edge are where the sand and water meet.

A small built in demonstration has been coded into the system that runs through a series of deformations that allow users to see the terrain adapt when many deformations occur. The terrain used in the demonstration is a small terrain. Ten random deformations occur followed by a series of scripted ones. The terrain is completely flat water mid way through the demo and so the reduction in triangles is clearly visible. At the end there is a small mountain created on an island in the centre of the map as shown in figure 9. Snow effect is also shown in figure 9. The snow starts from high above the ground and slowly falls onto the terrain. When the snow hits the ground it settles over time and so the terrain slowly starts to be covered in snow.

The texture generation in the system involved a great amount of work, but the results show that it was clearly worth it. The end results demonstrate the great control possibilities with our system, the terrain can be completely flattened and then a new terrain created. Meanwhile the texture is constantly being updated to reflect the change in terrain. The effect is further emphasised by the snowing effect. This shows the capability of the system to manipulate the textures with permanent effects. When the snowfall is switched off, the snow still stays on the ground. The whole effect is still done using single pass texturing.

The current system could be improved in several areas. The split only approach currently employed is adequate in the creation of the triangle mesh, but a split and combine approach would be better for the purpose. The next stage would be to traverse the binary triangle tree, updating only those that have been tagged. This alone should result in a large speed improvement due to reduce the number of calculations and the memory bandwidth required when recalculating the whole terrain.

One other area for improvement is the texturing. The current system works very well for most terrain types, however its one weakness is that the texture will get stretched on near vertical cliffs. One possible approach to solve this problem is to use a similar technique as used in Outcast [15]. This simple idea is to mark off special polygons, i.e. those with a very steep gradient. This could be done using a special colour code in the terrain type map or simply checked for when generating the triangles. When such a triangle is to be drawn, an alternative texturing method could be applied.

## References:


1. Ron Dulin, 1997, "Total Annihilation", Review, GameSpot UK, http://www.gamespot.co.uk/stories/reviews/
2. Ed Ricketts, "Populous: The Beginning " Game Review, PC Gamer Issue 65. http://www.pcgamer.co.uk/
3. Sarju Shah, "Tread Marks" Review on FiringSquad, http://firingsquad.gamers.com/games/treadmarks/
4. Mark Duchaineau,Murray Wolinski, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein, 1997, "ROAMing Terrain: Real-time Optimally Adapting Meshes." *Proceedings of the Conference on Visualisation'97*, pp. 81-88, Oct. 1997
5. Bryan Turner, 2000, "Real-Time Dynamic Level of Detail Terrain Rendering with ROAM." Features on Gamasutra. 3 April 2000.
6. Hugues Hoppe. 1997, "View-dependent refinement of progressive meshes", In *SIGGRAPH'97 Conference Proceeding*, August 1997.
7. Peter Lindstrom, David Koller, William Ribarsky, Nick Faust, and Gregory A. Turner, 1996, "Real-time Continuous Level of Detial Rendering of Height Fields". In *SIGGRAPH'96 Conference Proceedings*, pp. 109-118, August 1996.
8. Alan Watt and Fabio Policarpo, 2001, "3D Games Real-time Rendering and Software Technology", Addison-Wesley, 2001
9. Foley and Van Dam et al., Computer Graphics – Principles and Practice, Addison-Wesley, 1996
10. Alex Champandard, " Voxel Landscape Engines", Daily Game Development News and Resources. http://www.flipcode.com/voxtut/
11. D. Cohen-Or, E. Rich, U. Lerner, V. Shenkar Real-Time Photo-Realistic Visual Flythrough. *IEEE Transactions on Visualization and Computer Graphics*, 2:3 (1996) 255-265
12. Thatcher Ulrich, 2000, "Continuous LOD Terrain Meshing Using Adaptive Quadtrees," Features on Gamasutra, 28 February 2000. http://www.gamasutra.com/features
13. Peter Lindstrom et al. 1996, "Real-Time, Continuous Level of Detail Rendering of Height Fields", In Proceedings of *ACM SIGGRAPH 96*, pp. 109-118
14. Mark Allen, "Terrain Texturing " http://www.tashco.com/terraintexturing.html
15. Franck Sauer et al., "Outcast: Programming Towards a Design Aesthetic", http://www.appeal.be/products/page1/Outcast_GDC/outcast_gdc_1.htm


## Appendix A

This section of code calculates the intger points along a line, effectivly resolving the equivilent of scan lines in the rasterisation of polygons. This is done for the three edges in each triangle. Our terrain deformation implementation is based on Real-time Optimally Adapting Meshes and details of the method are described in [4]. In our system, the terrain could be loaded from a height-map, converted to a polygon mesh and then, during the game, be updated and continuously optimised. It also allows users to configure the detail settings in order to run optimally on their personal machine. The algorithm could even be aware of current frame rates and consider this when optimising the terrain [5].

```
Taking each edge {
   Step through from start vertex to finish vertex in y {
      Increase x by slope of edge
      Record point in array.
   }
}
```

Using these 'scan lines' the variation in height is calculated. This is multiplied by the number of height values checked and then compared to an arbitrary detial level. The purpose of multiplying by the number of heights is to decide that smaller polygons are split less.

```
Count = 0

Taking each of scan lines {
   Check height for all points between the start and end (incl.) {
      If height > largest
         Largest = height
      If height < smallest
         Smallest = height
      Count + 1
   }
}

If (highest-lowest)*count > detaillevel
   Return true
Else
   Return false
```

The following pseduo codes extract generate the terrain textures upon load up. Taking each texture grid in turn, the texture is generated and then writeen out to a file for use later.

```
genTextures()
{
        Load terrain textures and abort if any failed.

        For each texture grid square {
                Generate texture
                Write texture to file
        }
}
```

For a given texture grid, the texture is then created. It takes it pixel by pixel and blends two textures at a time. A third or subsequent texture can then be added.

```
createGenTexture ()
{
        Create pointers to terrain types.

        Go through each pixel of generated texture{
                Find the slope for that pixel
                Get the type for that pixel

        First texture is the terrain type of the pixels
                Second texture is usually rock

                Destination pixel is first texture times blend
                plus second texture times one minus blend

                Third texture is blended with above.
        }
}
```

## Appendix B

This is the node structure used for terrain deformations. One of these contains all the information for one triangle. The parent, child and adjacent pointers then link this node to the rest of the tree.

```
typedef struct node_str
{
        float point1[3];
        float point2[3];
        float point3[3];

        unsigned short generation;
        short texnum;

        unsigned int terrain[3][2];
        float texcoords[6];
        float heightaverage;

        bool draw;

        node_str *parent;
        node_str *child[2];
        node_str *adjacent[3];
} Node;
```

The following code extract calculates the blend factor. It uses the diference in height between the point and its right neighbour and difference in height between it and its bottom neighbour. A look up table is then used to find the overall blend value.

```
float findBlendFactor(int x, int y)
{
        int vector1;
        int vector2;
        int vector3;

        float blends[10] = { 0.0, 0.0, 0.0, 0.3, 0.6, 0.7, 0.85, 0.9,
        0.95, 1.0 };

        vector1 = getHeightFor2(x+1, y) - getHeightFor2(x-1, y);
        vector2 = getHeightFor2(x, y+1) - getHeightFor2(x, y-1);
        vector3 = abs(vector1 + vector2);

        if( vector3 >= 10 )
           vector3 = 9;

        return blends[vector3];
}
```

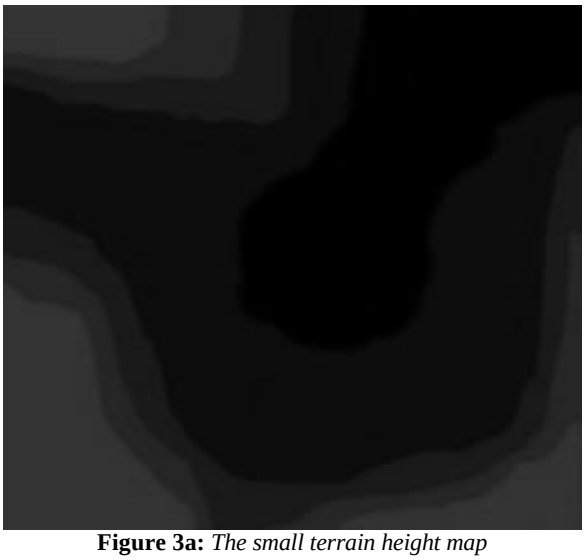

**Figure 3a:** *The small terrain height map*

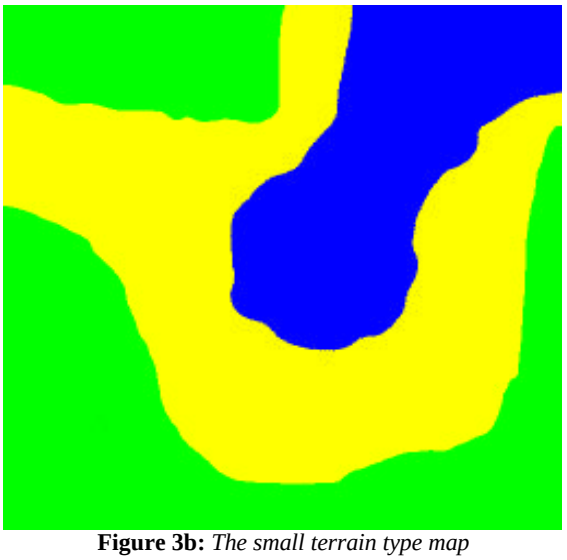

**Figure 3b:** *The small terrain type map*

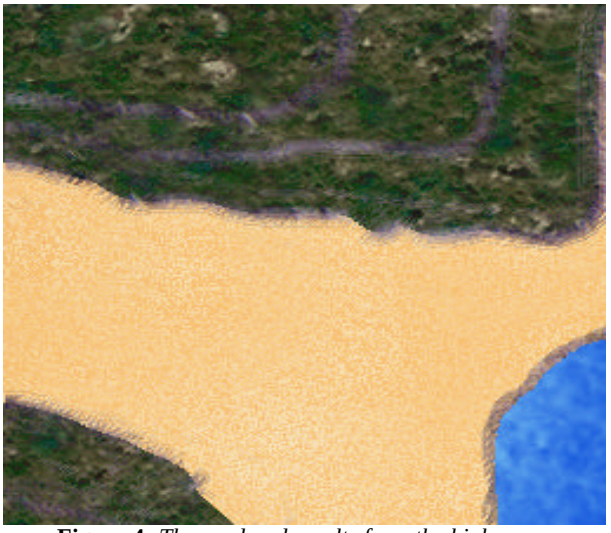

**Figure 4:** *The rendered results from the high maps*

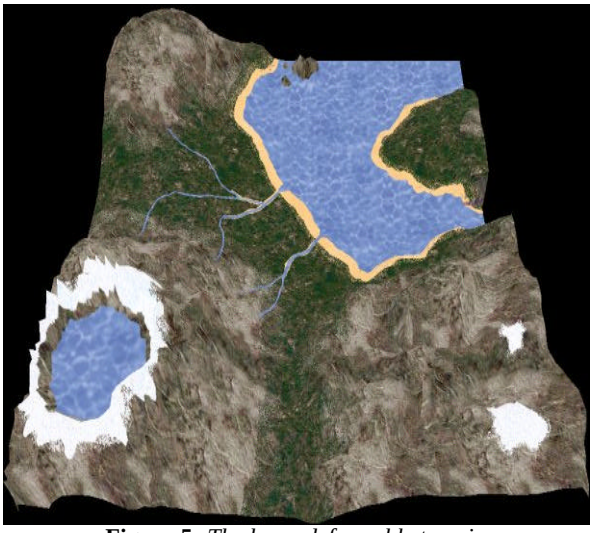

**Figure 5:** *The large deformable terrain*

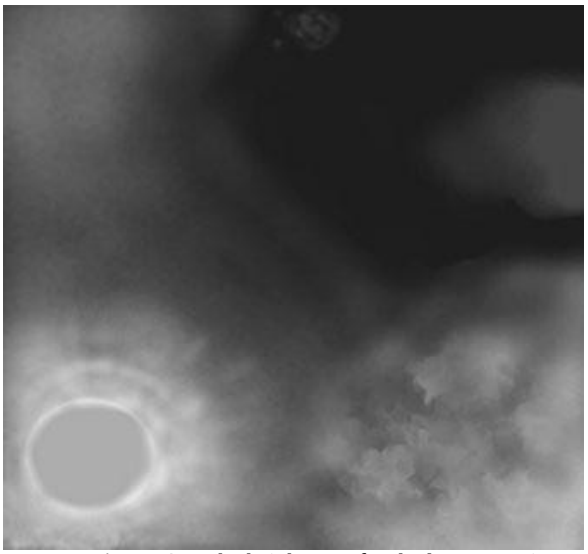

**Figure 6a:** *The height map for the large terrain*

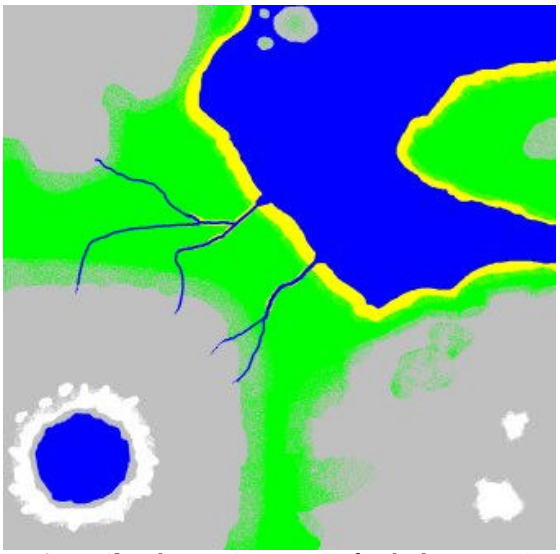

**Figure 6b:** *The terrain type map for the large terrain*

**Figure 7:** *The deformed terrain*

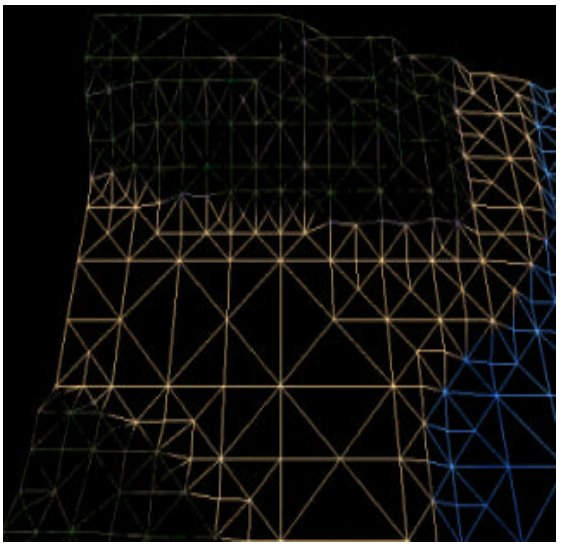

**Figure 8:** *The deformed terrain in wire frame*

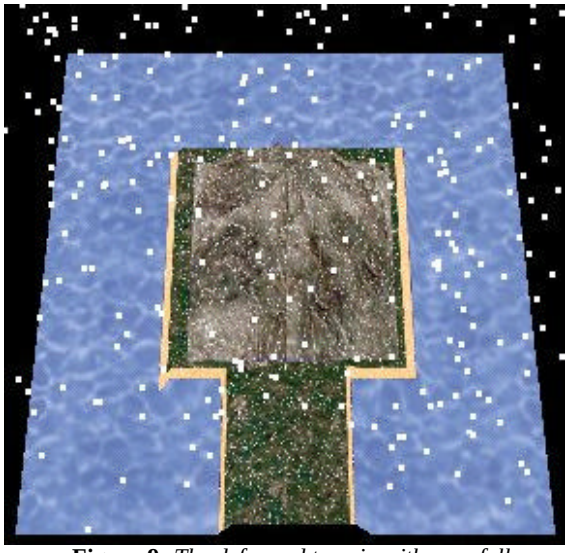

**Figure 9:** *The deformed terrain with snowfall*