Real-time terrain rendering with large geometric deformations

(HS-IDA-EA-03-105)

Anders Dahlbom (c00andda@student.his.se)

Department of Computer Science
University of Skövde, Box 408
S-54128 Skövde, SWEDEN

Final Year Project in Computer Science, spring 2003.
Supervisor: Michael Andersson
Real-time terrain rendering with large geometric deformations
Submitted by Anders Dahlbom to Högskolan Skövde as a dissertation for the degree of B.Sc., in the Department of Computer Science.

2003-06-08
I certify that all material in this dissertation which is not my own work has been identified and that no material is included for which a degree has previously been conferred on me.
Signed: _______________________________________________
Real-time terrain rendering with large geometric deformations

Anders Dahlbom (c00andda@student.his.se)

Abstract

Computer gamers demand more realistic effects for each release of a new game. This final year project is concerned with deforming the geometry in a terrain rendering environment. The intention is to increase the resolution where the original resolution of the terrain is not enough to cater for all the details associated with a deformation, such as an explosion.

An algorithm for extending the maximum available resolution was found, the DEXTER algorithm, but calculations have shown that it has a too high memory consumption to be feasible in a game environment. In this project, an algorithm has been implemented, based on the DEXTER algorithm, but with some structural changes. The algorithm which has been implemented increases the resolution, if needed, where a deformation occurs. The increased resolution is described by b-spline surfaces, whereas the original resolution is given by a height map. Further, graphics primitives are only allocated to a high resolution region, when needed by the refinement process.

It has been found that by using dynamic blocks of graphics primitives, the amount of RAM consumed can be lowered, without a severe decrease in rendering speed. However, the algorithm implemented has been found to suffer from frame rate drops, if too many high resolution cells need to be attached to the refinement process during a single frame.

It has been concluded that the algorithm, which is the result of this final year project, is not suitable for a game environment, as the memory consumption is still too high. The amount of time spent on refining the terrain can also be considered too much, as no time is left for other aspects of a game environment.

The algorithm is however considered a good choice concerning deformations, as the updates needed in association with a deformation, can be kept small and localized, according to the DEXTER structure. Also, the b-spline surfaces offer more freedom over the deformation, compared to using a height map.

Keywords: Terrain rendering, terrain algorithm, deformable terrain, extended resolution terrain, surface extended terrain.
Acknowledgements

I would like to thank my girlfriend, family, and friends for supporting me during the carry out of this final year project. I would also like to give special thanks to my supervisor Michael Andersson, for his constant enthusiasm and helpful comments concerning this project. Further, I would like to thank my examiner Göran Falkman for valuable insights and comments.

I am also most grateful to Mark Duchaineau, for answering my questions concerning the ROAM algorithm. Also, thanks to DICE, for answering my questions about their computer game Battlefield 1942.

Finally, but not least, thanks to all people writing and maintaining the articles at Gamasutra, flipCode, and the virtual terrain project, as these have been valuable resources.
Table of contents

1 Introduction ........................................................................................ 1
  1.1 A brief game history ................................................................. 2
  1.2 Terrain rendering ................................................................. 4
  1.3 Games of today ..................................................................... 5
  1.4 Problem ............................................................................... 7

2 Background ................................................................................ 8
  2.1 The three dimensional graphics rendering pipeline ................. 8
  2.2 View frustum culling ........................................................... 13
  2.3 Level of detail ...................................................................... 15
  2.4 Level of detail for terrain rendering ..................................... 17
    2.4.1 Real-time optimally adapting meshes ......................... 17
  2.5 Dynamic EXTension of Resolution ....................................... 21
  2.6 Parametric surfaces ............................................................. 24
    2.6.1 Beziér curves ................................................................. 24
    2.6.2 B-splines ...................................................................... 25
    2.6.3 NURBS ........................................................................ 26
    2.6.4 Surfaces ........................................................................ 26

3 Problem description .................................................................. 27
  3.1 Problem definition ............................................................... 28
  3.2 Hypothesis .......................................................................... 29
  3.3 Aims and objectives ............................................................ 29
  3.4 Expected result .................................................................... 29

4 Method ...................................................................................... 30
  4.1 Algorithm analysis ............................................................... 30
  4.2 Strategies for changes .......................................................... 31
  4.3 Strategy selection ............................................................... 32
  4.4 Test environment ............................................................... 32
    4.3.1 Test cases ................................................................. 33
    4.3.2 Measurement ............................................................. 33
    4.3.3 Test architecture ......................................................... 34
  4.5 Analysis .............................................................................. 34

5 Implementations ......................................................................... 35
  5.1 Run-time refinement ........................................................... 35
1 Introduction

Three dimensional graphics is a part of computer graphics that have become very popular in the last decade. The development of high speed processing units together with fast graphics hardware, have made this technology available for common people at a reasonable price. Three dimensional graphics has a wide area of use. Amongst them are computer games.

A common type of computer game is the first-person shooters. These games have some common properties, which include explosions, realism, and interaction. A high level of interaction is needed since split second decisions can make it or break it for the player of a first-person shooter. If the opponent is faster, then you loose, but if you are faster, then you win. The games should also respond in a realistic manner to the decisions made by the user, stand on an exploding grenade, and you loose. It is not only the player’s character that should be affected by the interaction with the player. The environment should also react on external changes such as exploding grenades.

A common technique used today for visualizing effects on the environment when an explosion occurs, is to have a couple of textures available that can be mapped on top of the polygons which are affected. In case of an explosion, a decal with an explosion mark is placed on top of the original texture to visualize that something has happened. This is illustrated in Figure 1. It is obvious that a black explosion mark is not a realistic output of an explosion. Instead, there should be a small crater where an explosion has occurred. The goal of this project is to investigate how to make deformations of the terrain, due to external forces such as explosions, in a terrain rendering environment.

Figure 1: A collection of screenshots from the computer game Counter-Strike, released by Valve in 2000. In the upper left corner, a grenade is lying in the unchanged scene. Next the grenade explodes and a decal is applied. The screenshot in the lower right corner clearly shows the decal which is the result of the explosion.
1.1 A brief game history

Computer games are constantly pushing the limits of computer hardware in order to achieve more and more stunning effects, and many important advances in computer graphics are made by people involved with computer games (Luebke, et al. 2003). Games have over the last couple of decades evolved from two dimensions to three.

Little more than a decade ago, computer games was limited to display a two dimensional scenery. In 1992 a game called Wolfenstein 3D was released by id Software. This was the beginning of a new era in the computer game industry. The developers of Wolfenstein 3D had created an illusion of three dimensions in real-time, on the personal computer, and it ran at a reasonably high speed. Wolfenstein 3D also introduced a new type of computer game, the first-person shooter. Id Software continued with their innovative style, and Wolfenstein 3D was followed by Doom 1993, Doom II 1994, and Quake 1996. A screenshot from Wolfenstein 3D can be seen in Figure 2.

![Figure 2: Screenshot from the computer game Wolfenstein 3D.](image)

The computer game Quake, released by id Software in 1996, marked a huge step in computer game development (Bryce & Rutter 2002). This game was one of the first games to take the full step into a completely three dimensional game environment. The three dimensional scenery consisted of a huge set of polygons which represented the world inside the game. A screenshot from Quake can be seen in Figure 3. The games released by id Software during the 1990’s did also introduce other aspects to the game environment.

![Figure 3: Screenshot from the computer game Quake.](image)

The games started to obey to the same physical rules as can be experienced outside the computer. Together with sound effects and stunning graphics, this created an
atmosphere that trapped many players. “One of the reasons Doom was such a massive success was the sense of unease and anxiety created as the gamer’s character travelled through deserted corridors” (Bryce & Rutter 2002). The game, Doom, was also first among the first-person shooter games to include multiplayer capabilities. The multiplayer part in first-person shooters has since then evolved, and today there are large online gaming communities for games like Counter-Strike, Quake III, and Battlefield 1942. These games are played by many players worldwide.

Most of the first-person shooters developed during the mid 1990’s had to take place inside buildings and in restricted areas. The number of polygons which were displayed was kept at a minimum because of the restricted hardware available and the fact that other aspects of a computer game needs processing time. When visualizing a scene, objects which are obscured by other objects, can be removed from the actual rendering. An illustration of this can be seen in Figure 4. By having a scene that is restricted by walls and other large objects, a huge amount of polygons are obscured and can be removed, resulting in more processing power for other aspects of the game.

Figure 4: In this figure, the viewer is positioned at the dot marked Eye, looking towards object A. Objects B-E is obscured by walls, and can be removed from the rendering process resulting in more processing power for other aspects. The field of view is denoted FOV.
1.2 Terrain rendering

In order to visualize an outdoor environment in real-time, which can be used in games, military mission planning, flight simulations, etc., there has to be some sort of information about the geometry of the terrain. A multi-variable polynomial can hold this information, but a more common technique used is height fields. Information about the geometry can be stored in a two dimensional matrix, where every element holds the elevation for a particular point in the landscape. To retrieve a landscape based on such a matrix, vertices are created at the elevation points, and edges are connected between them in order to create polygons. Figure 5, Figure 6, and Figure 7 illustrates the creation of a terrain based on a two dimensional matrix of elevation points. The polygonal structure used is a triangle.

Figure 5: A two dimensional matrix with 17x17 elevation points.

Figure 6: The vertices are connected in order to create triangles.

Figure 7: The triangles are filled and shaded in order to produce a more realistic image.
During the last decade, hardware for computer graphics has made major advances. Special graphical processing units have been deployed on the graphics cards, and they can now handle millions of polygons per second. Three dimensional graphics is preferably built up by polygons, since efficient algorithms and hardware have been developed for displaying these structures at great speed (Watt 2000). Even though hardware is capable of rendering polygons at great speed today, there are still too much polygons involved in order to use a “brute force” algorithm, where all polygons are sent to directly to hardware, as in Figure 6. In order to visualize a large open landscape with the help of polygons, techniques have been developed to limit the actual number of polygons needed (Lindstrom & Pascucci 2001).

Many algorithms for terrain rendering have been developed during the last decade. Most of these algorithms are based on a technique called level of detail (Röttger, et al. 2002). Level of detail algorithms build upon the fact that geometry far away from the camera, as well as regions with flat geometry, can be visualized with less number of polygons. This due to the fact that objects appear to be smaller when distance is increased. Regions far away from the camera will only be represented by a few pixels on screen, and it is therefore not necessary to waste a huge number of polygons on these regions. More polygons are instead placed in areas with high ruggedness, as well as areas close up to the camera. An illustration of level of detail can be seen in Figure 8.

Figure 8: The same height field as Figure 6, but a level of detail algorithm has been applied. The flat parts are visualized with fewer triangles than the bumped parts.

1.3 Games of today

By today, games have become very impressive, and consist of large scenes with realistic effects such as reflections and silhouettes, to mention a few. Games like Unreal II by Epic Games, Quake III by id Software, and Battlefield 1942 by DICE, are built upon powerful three dimensional graphics engines that handle large amounts of geometry, and incorporate stunning particle engines for explosions and other effects. These games are played by many gamers online, which demand more and more realism and better game play for every release. A screenshot from the game Battlefield 1942 can be seen in Figure 9.
Something that has almost become a norm within the game industry, particularly amongst first-person shooters, is that anything that is contained within a game environment should be destroyable (Saltzman 1999). By implementing deformable terrain, the underlying geometry of a game scene can also be destroyed.

“Existing methods focus either on static terrains or on time-varying geometry where all changes are known prior to any rendering.” (He, et al. 2002) This can be a negative aspect when for example one area of use can be in a warfare game, where grenades and missiles are applied on the landscape. To retrieve a realistic simulation of this, the underlying geometry of the landscape should change when an explosion occurs. Depending on the strength of the explosion, a rather large part of the terrain might need to be deformed at the same time as the rendering proceeds without visual artifacts.

In 2000 Yefei He at the University of Iowa, USA, wrote a Ph.D. thesis on the subject of online dynamic terrain. His work was concerned with the interaction on the terrain in vehicle simulations, and the idea was that tire tracks should be created where a car had passed. Hence his report focused on small localized changes on the geometry, and especially that the resolution of the underlying height field might need to be increased in order to give the correct appearance. His conclusions did not include how well his approach cooped with larger geometric deformations, but he mentioned that this could need some further investigation.

One of the most basic factors of computer games is the interaction with the player (Crawford 1982). When the player of a computer game takes an action, something should happen. For instance, when a missile is launched from your missile equipped car upon a hill, a huge crater should appear. The player can after that, drive into the newly created hole, and examine it. This could increase the realism of a game, and computer games have been found to be successful when a high level of realism is present (Bryce & Rutter 2002).
1.4 Problem

The aim of this final year project is to investigate how to visualize a three dimensional landscape, where rather large deformations are possible in real-time.

The main contribution of the algorithm presented by He (2000), is that the run-time refinement structures used for rendering a terrain, are statically extended at run-time, where the highest resolution of the terrain needs to be extended. If a rather large terrain is used and the resolution is extended everywhere, the algorithm is very memory intensive. The main focus of this project is to investigate if the memory consumption of the algorithm presented by He (2000), can be lowered by introducing curved surfaces and dynamic refinement structures.

In chapter 2, general background material will be given. It is necessary to have knowledge of this material in order to appreciate the following chapters. Chapter 3 will present the problem and define a hypothesis. This will be followed by the aim of the project, together with the objectives on how to achieve the aim. Chapter 4 will present a method, which are to be used when analyzing, implementing, and testing the found material. Chapter 5 will present implementation specific material which is used for building a small test engine. This will be followed by results and an analysis of the results, in chapter 6. This is followed by a conclusion in chapter 7, and a discussion in chapter 8. Finally, chapter 9 states some future work.
2 Background

In this chapter, relevant background material will be presented. First the three dimensional graphics rendering pipeline will be described, as it is important to have knowledge of the basic concepts in order to understand three dimensional graphics. After this view frustum culling will be presented, which is an important technique concerned with polygon reduction. This will be followed by a brief overview of how systems for level of detail work. Next a technique for view-dependant continuous level of detail concerning terrain rendering, Real-time optimally adapting meshes, will be presented. This will be followed by a description of He’s technique for extendible resolution dynamic terrain. Finally, parametric surfaces will be described, which is an alternate method for describing geometry.

2.1 The three dimensional graphics rendering pipeline

A three dimensional graphics rendering system is known as the three dimensional graphics rendering pipeline. This pipeline can be divided into three conceptual stages known as the application, geometry, and rasterizer stage (Akenine-Möller & Haines 2002). This is illustrated in Figure 10. Data can be processed independently within the different stages, but a stage must wait for input from its preceding stage. Hence the rendering speed, often measured in frames per second, FPS, is as fast as the slowest of the three stages.

![Figure 10: The three conceptual stages of the graphics rendering pipeline. Adapted from Akenine-Möller & Haines 2002.](image)

The first of the three stages is the application stage. The developer always has full control over this stage as it is driven by the application and always implemented in software, hence the name. The other two stages can either be partially or fully implemented in hardware, and it is therefore more difficult to affect the overall performance of the application within these two stages. However, since the output of the application stage is the geometry to be rendered on screen, the performance of the final two stages can be affected by limiting the amount of geometry passed on. Hence it is here, within the application stage, that techniques for high level culling and level of detail are implemented, which are techniques for limiting the number of primitives. It is also within the application stage that other aspects of the application are handled, such as input devices, collision detection, communication, etc.

The second stage of the three dimensional graphics rendering pipeline, is the geometry stage. This stage is further subdivided into a pipeline of functional steps, as illustrated in Figure 11. The functional steps of the geometry stage will now be described in more detail.

![Figure 11: The functional steps within the geometry stage, as a pipeline. Adapted from Akenine-Möller & Haines 2002.](image)
In three dimensional graphics, all objects or models are associated with their own local coordinate system, known as model space, and it relates the different vertices of an object to each other. In order to relate different objects to each other, a second coordinate system is also used which is called world space. All objects are transformed into world space prior to any rendering, since this coordinate system holds the relationship between different objects and how they are aligned towards each other. Figure 12 shows an illustration of how model space-, and world space coordinate systems are related to each other.

![Figure 12: Two object space coordinate systems within the world space coordinate system.](image)

The first functional step within the geometry stage, Model & View Transformations, is concerned with transforming objects from model space to world space, and from world space to camera space, which is a third coordinate system used when transforming objects into screen space. A model can be associated with one or more model transformations, which places the model within the world space coordinate system with the desired orientation. In case of more than one model transformation, the model can be duplicated to several locations in world space. Figure 13, illustrates a cube being transformed from model space to world space.

![Figure 13: Illustration of a cube being affected by three model transforms. a) The cube resides in its model space. b) The cube is being scaled. c) Rotation is applied to the cube. d) Finally the cube is being translated into its correct position in world space.](image)

The third coordinate system within three dimensional graphics, known as camera space or eye space, is the world space coordinate system transformed so that the camera resides at the origin, looking down along the negative z-axis, for right-handed coordinate systems, and along the positive z-axis for left-handed coordinate systems. The camera space is used in order to make clipping and projection operations faster.
(Akenine-Möller & Haines 2002). The camera is also treated like an object, and to transform the world space into camera space, a view transformation is applied. An illustration of a view transformation can be seen in Figure 14. The model transformations and view transformation are often concatenated into one single transformation, using matrices, in order to become more efficient. In case of one single transformation, there exists no world space coordinate system; objects are converted directly from model space to camera space via world space.

Figure 14: On the left, camera positioned and oriented in world space. To the right, after view transform, the camera relocated at the origin looking along the negative z-axis. Adapted from Akenine-Möller & Haines 2002.

In the next functional step of the geometry stage, lighting is calculated for the three dimensional scene. Lighting will provide a more realistic appearance to a model (Akenine-Möller & Haines 2002). In the real-world, photons are emitted from different kinds of light sources. When the photons reach a surface they are either absorbed or reflected by it, and this interaction makes up our vision of the world. In order to approximate this interaction in a three dimensional graphics environment, a lighting equation is used, which computes the color at each vertex. This equation takes account for the position and properties of light sources together with the material, position, and normal vector of the vertex. Later when a surface made up of several vertices are to be rendered, it is common to interpolate the colors of the vertices, over the surface. This interpolation technique is known as Gouraud shading (Akenine-Möller & Haines 2002).

When working with three dimensional graphics, the world must be looked upon in some direction from a specified position. This is often described using a camera analogy, where the viewer is positioned at a reference point looking at the scene through a camera. The camera is associated with a view volume, and all objects that are fully or partially inside the view volume should be rendered on screen. The view volume is shaped differently according to which type of projection is used. There are mainly two different projection methods used, orthographic, and perspective projection. In orthographic projection, the view volume is shaped like a rectangular box, and its main characteristic is that parallel lines remain parallel after projection. Figure 15 illustrates an orthographic projection.
The other type of projection, which is also the most frequently used within computer graphics, is the perspective projection (Akenine-Möller & Haines 2002). The view volume of perspective projection is formed like a bottomless pyramid that stretches out from the camera according to view direction. The main characteristic of perspective projection is that an object appears smaller when the distance to camera is increased. Both types of projections are associated with a near and far clipping plane, outside of which objects are not visible. The perspective projection becomes a polyhedron when the clipping planes are associated, and it is called the view frustum (Akenine-Möller & Haines 2002). Perspective projection and view frustum are illustrated in Figure 16. Together with the view volume, a view plane is associated with a projection. The view plane is where everything that resides within the view volume will be mapped to in the screen mapping step of the geometry stage.

The projection step in the geometry stage, transforms the view volume into a unit cube, which is called the canonical view volume. When everything is transformed into the canonical view volume, clipping which is the fourth step becomes consistent since objects always have to be clipped against the unit cube (Akenine-Möller & Haines 2002).

As mentioned, the fourth step of the geometry stage is clipping. In this step, objects that are not partially or fully inside the canonical view volume can be removed. Hence they are not passed to the screen mapping step. The primitives that are fully inside the
view volume can proceed to the next step as is, but the objects that are partially inside
the view volume needs to be clipped before they can be sent on. When clipping a
primitive, the part which is not inside the view volume will be removed, and a new
primitive is created to replace it. Figure 17 illustrates the clipping process.

![Clipping Diagram](image)

Figure 17: The left figure illustrates three triangles after projection, and the left figure shows
what is left after clipping. The triangle that is completely outside of the unit cube will be
removed. The triangle fully inside the unit cube is left as is. The triangle partially inside is clipped
and the part of it that is outside will be replaced with a new primitive on the boundary. Adapted
from Akenine-Möller & Haines 2002.

The final step in the geometry stage is screen mapping. Primitives reaching this step
are still three dimensional, but now their x- and y-coordinates will be transformed into
screen coordinates. The screen coordinates are now ready to be sent to the next stage
of the three dimensional graphics rendering pipeline, the rasterizer stage, but in order
to perform depth comparisons so that objects closer to the camera will obscure objects
further away, the z-coordinates needs to be sent along as well.

Finally, the third and final stage in the three dimensional graphics pipeline, the
rasterizer stage, will be described. The purpose of this stage is to assign correct colors
to all pixels on screen. This is referred to as rasterization or scan conversion. Given
screen coordinates and z-coordinates for all two dimensional vertices, depth testing is
performed in order to resolve visibility. Figure 18 gives an illustration of why depth
sorting is necessary. The two dimensional vertices are associated with colors, and
perhaps texture coordinates in order to glue an image on to the projected primitives.
2.2 View frustum culling

As described in the previous chapter, rendering speed can be increased if less data is sent from the application stage in the three dimensional rendering pipeline, to the geometry stage. Everything that is contained within the view frustum will be mapped on screen. Thus objects and primitives that are not intersecting the view frustum can be removed. The removal of an object or primitive in three dimensional graphics is known as culling. Hence the name, view frustum culling. There are also other types of culling techniques, for example back-face culling, where all primitives whose normal vector is opposite of the view direction will be removed from the graphics rendering pipeline. Back-face culling is usually not performed in the application stage of the three dimensional graphics rendering pipeline.

In the geometry stage of the three dimensional graphics rendering pipeline, all primitives which have reached so far will be tested against the view frustum. Hence it is not feasible to do this test at an earlier stage, as in the application stage, since it will be performed later. Instead, if all primitives that an object consists of are grouped together within a bounding volume, the bounding volume can be tested against the view frustum. When a bounding volume is found to not intersect, the view frustum, all primitives that are grouped within it, can be removed from the three dimensional graphics rendering pipeline. An object is considered intersecting the view frustum when it is either fully or partially inside it. Objects that are located close to each other can also be grouped in a bounding volume, resulting in that even more primitives can be ruled out by one simple test. In three dimensional graphics it is common to use spheres and boxes as bounding volumes (Akenine-Möller & Haines 2002). Figure 19 illustrates a bounding box, which contains a group of objects.
A bounding volume can contain other bounding volumes, and together they form a hierarchy of bounding volumes, known as bounding volume hierarchy (BVH). A BVH can then be tested against the view frustum, and this is known as hierarchical view frustum culling. In case a volume does not intersect the view frustum, nor will its children intersect the view frustum. If a bounding volume is found to intersect the view frustum, then its children needs to be tested against the view frustum as well. The bounding volume hierarchy is the most common type of hierarchical view frustum culling, and the hierarchy can be ordered into trees or other common spatial data structures (Akenine-Möller & Haines 2002). A BVH is illustrated in Figure 20.
2.3 Level of detail

In three dimensional graphics, it is common to use the perspective projection (Akenine-Möller & Haines 2002). One aspect of the perspective projection is that objects appear to become smaller when the distance to the camera is increased. When an object resides close to the camera, it will be relatively large when mapped on screen, but when it is far away from the camera it will only be represented by a few pixels on screen. This is illustrated in Figure 21.

![Figure 21](image)

Figure 21: When a primitive of height \( h \) is at a distance \( d \) from the camera, it will be represented by an amount of pixels relative to \( l/d \). When the distance to the camera is increased to eight times \( d \), the amount of pixels on screen that represents the object will be relative to \( l/8d \), which is much smaller.

Techniques for level of detail, LOD, are based on the fact that objects far away from the camera will only be represented by a few pixels when rendered. Thus when distance to an object is increased, an approximation of the object can be used which consists of fewer primitives, without compromising the visual quality of the rendered image. “The basic idea of Levels of Detail (LODs) is to use simpler versions of an object as it makes less and less of a contribution to the rendered image.” (Akenine-Möller & Haines 2002)

There are mainly three different frameworks for LOD techniques, discrete-, continuous-, and view-dependant LOD (Luebke, et al. 2003). In discrete LOD, DLOD, several copies of an object, with different levels of detail, are created in a preprocessing step before the actual rendering takes place. The difference in detail from a higher level of detail to a lower level of detail is typically reduced uniformly across an object. This since the simplification is made prior to any rendering, and it can at that time not be predicted at what angle the object will be viewed upon (Luebke, et al. 2003). An illustration of DLOD applied to a triangle mesh is seen in Figure 22.
Later during rendering, one of the representations for the object will be chosen, based on for example distance to the camera, known as view distance. When switching the level of detail during rendering, details may vanish from a finer approximation to a coarser one. This might produce a noticeable popping effect, which is not visually appealing. In the context of terrain rendering, this popping effect may consist of hills and holes suddenly appearing and disappearing in front of the camera.

The second framework for level of detail is the continuous LOD, CLOD. Instead of creating a specified number of individual approximations for an object before rendering, a continuous spectrum of detail is encoded into data structures (Luebke, et al. 2003). This information can then be extracted during rendering to approximate the object according to appropriate factors such as view distance. The level of detail for each object rendered with CLOD is exact, and only those primitives that are needed will be used. This will result in a better granularity of the object, and that no more primitives than necessary are used (Luebke, et al. 2003). Since the levels of detail are continuous, only a few primitives will be changed from a finer level of detail to the next coarser level. This makes the popping effect, associated with DLOD, less noticeable.

Finally, there is view-dependant LOD, which is an extension of the CLOD. In the view-dependant LOD, a dynamic triangulation is used to select an appropriate level of detail according to the current view, it is said to be anisotropic (Luebke, et al. 2003). Figure 23 shows how the angle to an object affects the projection, and Figure 24 shows how a triangle mesh in a view-dependant LOD could look. Each object can contain different levels of simplification in the same frame, resulting in an even better granularity than the CLOD since primitives will be allocated where they are most needed (Luebke, et al. 2003). This will also minimize the number of primitives used, which is good as memory is a limited resource in a three dimensional graphics environment (Luebke, et al. 2003).
2.4 Level of detail for terrain rendering

According to Luebke, et al. (2003), view-dependent LOD techniques are of critical importance when concerned with terrain rendering in real-time. The continuous nature of terrain data makes it possible for large parts of the terrain to be visible at any point, and terrain meshes can also be extremely dense. Thus it is of high importance to use techniques for dynamic level of detail during rendering. The DLOD technique is based on static data structures that are created offline before the actual rendering takes place. Deformable geometry in real-time is not possible with static terrains; hence CLOD techniques must be used.

A common problem when using techniques for CLOD when used in terrain rendering is that t-junctions might appear where neighboring blocks meet, which have different levels of detail. The t-junctions might turn into cracks if care is not taken. This is illustrated in Figure 25.

Figure 25: Illustration of how cracks may appear. If the LOD level of triangle T1 is changed, but the LOD level of triangle T2 is not changed, then there might appear a crack if the newly created vertex is not on the base edge of T2.

During the last decade, a huge amount of work has been done concerning LOD algorithms for terrain rendering. Next, one algorithm for view-depmandent LOD will be presented, ROAM, as this is the one used by He, et al. (2002), for dynamic extension of resolution.

2.4.1 Real-time optimally adapting meshes

Real-time optimally adapting meshes, also known as ROAM, is an algorithm for optimizing the triangulation in a terrain according to a view-depmandent error metric. The algorithm was published by Duchaineau, et al. (1997), and has since then proven to be extremely popular, especially amongst game developers (Luebke, et al. 2003). The algorithm is based on a binary triangle tree structure, bintree, and Figure 26 illustrates the first five levels of a bintree.
The triangles in the ROAM algorithm must be right isosceles triangles: two of the interior angles are equal and the third angle is 90 degrees. In order to create a bintree as in Figure 26, the initial triangle $T$ is split in half, by inserting a vertex, $v_c$, at the center of its base, and then creating a new edge between the vertex at the apex, $v_a$, and the newly create vertex, $v_c$. This creates two new triangles, $T_1$ and $T_2$, which also is of right isosceles type. To continue down the tree, the above procedure is followed in a recursive manner until the desired level of detail is reached. The triangles created by the recursive procedure above are inserted into a binary tree structure, where triangle $T$ forms the root node, having $T_0$ and $T_1$ as its children. This is illustrated in Figure 27.

In the ROAM algorithm, a continuous triangulation is defined as a set of bintree triangles which forms a continuous mesh; this is true when any two triangles either overlap nowhere, at a common vertex, or at a common edge (Duchaineau, et al. 1997). To achieve such a triangulation, a series of split- and merge-operations are performed on the bintree. The consequences of these two operations are illustrated in Figure 28.
As can be seen in Figure 28, the two triangles $T_1$ and $T_2$ are split at the same time, they are said to form a diamond, which is two triangles that share their base edge with each other. In the ROAM algorithm, split operations can only be performed on triangles forming a diamond unless their base edge resided on the border of the terrain, this in order to obtain a continuous mesh without cracks. An important fact about bintree triangulations, which also can be seen in Figure 28, is that the difference in level of detail between neighboring triangles at the most can be one level. For a triangle $T$, all neighbors can be at the same level as $T$, left and right neighbors can be from the next finer level $l+1$, and base neighbors can be of the next coarser level $l-1$.

In the ROAM algorithm, forced splits are introduced so that a triangle that is not currently part of a diamond can be split. The base neighbor of the triangle to be split is first split in order to form a diamond, and this forced splitting can result in many triangles being split recursively. This is illustrated in Figure 29.

In order to facilitate the split operations in the ROAM algorithm, every triangle $T$, needs to have knowledge of its neighbors at each edge. This is illustrated in Figure 30.
Figure 30: Illustration of how a triangle, $T$, is associated with its neighbors, $T_B$ at its base edge, $T_L$ at its left edge, and $T_R$ at its right edge.

The runtime refinement of the terrain is managed by two priority queues, one for triangles that can be split, and one for triangles that can be merged. By enabling both split- and merge-operations, the ROAM algorithm can exploit frame-to-frame coherence, where the triangle operations can be performed on the triangulation from the previous frame. Instead of recalculating the complete triangulation of the terrain mesh, the triangulation from the previous frame is refined to suit the next frame.

The priorities of the two queues are based on a screen-based geometric error for each triangle. Each triangle is covered by a bounding volume, called a wedgie, and the wedgie for a triangle is based on the wedgies of its child triangles. This structure can be compared to the bounding volumes hierarchy in section 2.2, and it provides a guaranteed bound on the error. It is assumed that the vertex-to-world-space mapping $w(v)$ is of the form $w(v) = (v_x, v_y, z(v))$, where $(v_x, v_y)$ are the domain coordinates of the vertex $v$, and $z(v)$ is the height at $v$. The affine height map for a bintree triangle $T$ is denoted $vwxT$.

A wedgie for a triangle $T$ is then defined as a volume in world space containing all points $(x, y, z)$ such that $(x, y) \in T$ and $|z - z_T(x, y)| \leq e_T$, where the wedgie thickness $e_T \geq 0$ for a parent triangle $T$ is computed bottom-up according to Equation 1, where $e_{T_0}$ and $e_{T_1}$ are the wedgie thicknesses for the children of $T$, and $z_T(v_i) = (z(v_0) + z(v_1))/2$, is the interpolated height at the mid point on the base edge of $T$.

$$e_T = \begin{cases} 0 & \text{for leaf triangles} \\ \max\{e_{T_0}, e_{T_1}\} + |z(v_e) - z_T(v_e)| & \text{otherwise} \end{cases}$$

A hierarchy of nested wedgies is illustrated in Figure 31.
The priority for a triangle $T$, is calculated as the screen space error, or distortion, that occurs when projecting its wedgie into screen space, as illustrated in Figure 32.

The distortion of a triangle's wedgie when projected on screen can be calculated with equation (2), which can be rewritten to form equation (3), where $(p,q,r)$ is the camera-space coordinates of a vertex $v$ and $(a,b,c)$ is the camera-space vector, corresponding to a world space thickness vector $(0,0,e_r)$.

$$\text{dist}(v) = \left| \frac{p + a}{r + c} - \frac{p - a}{r - c} \right|$$

$$\text{dist}(v) = 2 \frac{r^2 - c^2}{r^2} \left( (ar - cp)^2 + (br - cq)^2 \right)^\frac{1}{2}$$

Hence the priority of a triangle increases with the size of the projected screen error.

2.5 Dynamic EXTension of Resolution

Dynamic extension of resolution, DEXTER, is an algorithm that extends the ROAM algorithm, and introduces deformable geometry at an extended resolution. The idea behind DEXTER was suggested by He (2000), in his Ph.D. thesis, and it was later
published in a paper by He, et al. (2002). The authors of the paper argue that the method can be applied to enhance several multiresolution surface algorithms. In the thesis presented by He (2000) the method was tested on two terrain algorithms, ROAM (Duchaineau, et al. 1997), and Real-time, Continuous Level of Detail Rendering of Height Fields (Lindstrom, et al. 1996). He stated that the results obtained were good. In the paper published by He, et al. (2002), the main focus is an enhanced version of the ROAM algorithm, and this one will be described next.

In order to modify the ROAM algorithm to be suitable for deformable terrain, two extensions are introduced. The first extension is concerned with updating the mesh data, such as the internal errors of the triangles, in every frame where deformation has occurred. This is necessary since the triangulation algorithm uses the errors when approximating the terrain, and without these updates, the deformations will not be reflected very well. The second extension incorporates a run-time extension of the hierarchical structure, which will increase the resolution at the deformed parts, if necessary.

In the first extension, the mesh data that needs updating when a deformation has occurred includes the world-space errors, and the altitudes of all vertices. The updating of world-space errors will be performed bottom-up since this is how the error values are computed in the original ROAM. The errors computed in the ROAM algorithm, the wedgies, are associated to the different triangles in the bintrees. However, the errors are computed based on the geometry of the vertices a triangle consists of, and it can easily be determined which triangles have been deformed by checking which vertices have been modified. Hence update flags are added to the data structure for every vertex. The error update procedure is performed accordingly to the ROAM algorithm, but the error update will only be initiated on those leaf triangles that consist of vertices that have been modified.

The authors argue that the resolution in deformed regions needs to be increased, in order to capture the details of the deformation, hence the second extension. The idea is to introduce cells, with a more detailed height field attached, on top of the ROAM structure. The leaf triangles which are covered by the cell are further subdivided in order to expand the bintree structure so that the more detailed height field will be taken account for. An illustration of this can be seen in Figure 33.

![Figure 33: Illustration of how cells are introduced. The original ROAM triangle pair in the top left corner, which is covered by a newly introduced cell, is split further to take account for the more detailed height field in the cell.](image)

The cells introduced are axis-aligned and can be of any resolution that is a power of 2. Initially the whole terrain grid is covered by one single cell. Hence, cells of different resolution may overlap each other. This is illustrated in Figure 34.
The introduction of cells with different resolution also introduces problems with continuity on the boundary of the cells. This can be seen in Figure 34, where t-junctions are present at the boundary of cell 3. The continuity between the triangles in cell 3 and cell 2 will not be kept, as the triangles with the highest resolution in cell 3 are more than 1 detail level away from their neighboring triangles in cell 2.

In order to solve this problem, transition zones are introduced. A transition zone is defined along the boundary between two regions $R_1$ and $R_2$, with grid resolution $\delta_1$ and $\delta_2$ respectively, when $\delta_1 > \delta_2$ and the higher resolution $R_1$ meshes cannot be matched by $R_2$ meshes. He (2000) proves by mathematical induction that by creating a transition zone consisting of a number of cells of the same resolution as the newly created cell, but with zero error, along the border of a newly added high resolution cell, the mesh will always stay continuous. This is illustrated in Figure 35.

By adding a high resolution cell and extending the original bintree, the world-space error for all triangles affected in the bintree must be updated. Some of the former leaf triangles will now have descendents themselves, and hence their error is generally no longer zero. A bottom-up approach for recalculating the errors is once again used as this is the way errors are calculated in the ROAM algorithm. However, when extending the transition cells, the former leaf triangles will still have a zero error, even though they now have descendents. This due to the fact that the newly added triangles
in the transition cells must only be split when a forced split is requested from a
triangle inside a newly created high resolution cell. An illustration of how an
extension of the resolution will affect a triangle mesh can be seen in Figure 36.

![Figure 36: Illustration of how the resolution is extended in a triangular mesh. The mesh is
illustrated before any change to the left, and after resolution change to the right. A cell with
higher resolution is inserted in the upper left corner. It is surrounded by three transition cells.
When the highest resolution available is used in the inserted cell, the transition cells are affected
as can be seen in the figure to the right, but the complete mesh is still continuous.]

2.6 Parametric surfaces

The world space in three dimensional graphics is the container for everything that
eventually will be rendered on screen. The world space can contain objects, terrain,
players, etc., and each of these will be based on some sort of geometry. Each object
consists of a set of polygons, which are grouped in such a way that they form the
geometry of the object. The methods for rendering terrain described earlier have all
been based on two dimensional matrices, containing elevation points. The polygons
created from such a matrix, are all based on static vertices, which are predefined
according to a height map. In order to get a fine granularity of the geometry, when
using height maps, the height map might need to be rather dense, in order to take
account for all details which is needed. There is however another way for describing
the geometry of which objects and terrains consist of, and this is by using parametric
surfaces. Parametric surfaces are described by polynomials of varying degree, which
serves as a mapping from $\mathbb{R}^2(u,v)$ to $\mathbb{R}^3(x,y,z)$. Akenine-Möller & Haines (2002),
states that the beauty of using curved surfaces are at least fourfold. They are
represented in a more compact way than a set of polygons. The geometry becomes
scalable, as any point on a curve can be calculated; yielding that an infinite amount of
primitives can be created. The primitives that can be created are smoother and more
continuous. Animation and collision detection may become simpler and faster. There
is however also some computational overhead associated with curved surfaces, as the
vertices have to be extracted from the surface, compared to having them stored in a
height map. In order to understand parametric surfaces, parametric curves will first be
presented.

2.6.1 Beziér curves

Beziér curves, named after the French engineer Pierre Beziér, are a way of describing
every point along a curve by the means of a subset of points, called control points. A
A point on a beziér curve is defined in terms of the control points and a basis function, which tells how much each control points should influence the wanted point. The basis functions are Bernstein polynomials of degree $n$, where $n+1$ is the number of control points. A point on a beziér curve can be found with the following equation:

$$P(t) = \sum_{i=0}^{n} B_i^n(t)P_i,$$

where $B_i^n$ represents the Bernstein polynomials as follows:

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}, \text{ where } i = 0, \ldots, n, \text{ and } 0 \leq t \leq 1.$$ 

The basis functions in a cubic, degree 3, beziér curve are expanded as follows:

$$B_0(t) = (1-t)^3$$
$$B_1(t) = 3t(1-t)^2$$
$$B_2(t) = 3t^2(1-t)$$
$$B_3(t) = t^3$$

A beziér curve passes through the first, and last control points, but necessarily not the others, they are simply pulling the curve towards them.

2.6.2 B-splines

B-Splines, short for basis splines, are a generalization of the beziér curves. The b-splines take use of a knot vector, which describes the range of influence for each control point. In the beziér curves, the degree of the Bernstein basis functions is determined by the number of control points, $k = n - 1$, where $k$ is the order of the curve, and $n$ is the number of control points. It can also be seen that every control point affects every point on a beziér curve. The b-spline curves decouple the degree of the curve from the number of control points, and also, every control point does not need to influence every point on the curve. Each point on a b-spline curve is only affected by a subset of the control points, which is determined by the order of the curve together with a knot vector. As already mentioned, a knot vector is a set of values that describes the range of influence for each control point. A knot vector must be $n+k$ elements long, and it must be monotonically increasing, $x_i \leq x_{i+1}$, for each element $x_i$ in the knot vector. As an example, consider the knot vector $[t_0, t_1, t_2, t_3, t_4, t_5, t_6]$, which is to be used on a third degree curve with four control points. Then the range of influence for each control point on the curve would be $[t_0, t_3], [t_1, t_4], [t_2, t_5], [t_3, t_6]$. A knot vector which has $k$ repeated values at the beginning and at the end is called an open knot vector, and its characteristics are that the curve will pass through the first and last control points, which is of interest when continuity is needed between two adjacent curves. A uniform knot vector is defined as a knot vector having its values evenly distributed.

A point on a b-spline curve can be calculated with the following equation:

$$P(t) = \sum_{i=1}^{N} P_i N_{i,k}(t),$$

where $N_{i,k}(t)$ is the basis functions for the b-spline, which is calculated by using Cox-de boor recursion formulas seen below.
2.6.3 NURBS

NURBS, non-uniform rational b-splines, is a superset of the b-spline curves, which also includes weighted control points. The NURBS is rational since the equation for a NURBS curve is defined as a quote between two polynomials. A point on a NURBS curve is defined as follow

\[ N_{i,k}(t) = \begin{cases} 1 & \text{if } x_i \leq t < x_{i+1} \\ 0 & \text{otherwise} \end{cases} \]

\[ N_{i,k}(t) = \frac{(t - x_i)N_{i,k-1}(t)}{x_{i+k-1} - x_i} + \frac{(x_{i+k} - t)N_{i+1,k-1}(t)}{x_{i+k} - x_{i+1}} \]

The weight values of a NURBS curve, defines how much each control point will pull the curve towards it. By setting all weight values equal to 1, a b-spline is formed.

2.6.4 Surfaces

The bézier curves, b-splines, and NURBS, can be extended to describe surfaces instead of curves, by using two parametric directions, \( u \) and \( v \), instead of \( t \). The equation for a bézier surface is given by

\[ P(u, v) = \sum_{i=0}^{N} \sum_{j=0}^{M} P_{i,j} B_i(u) B_j(v), \]  where \( B \) is the Bernstein basis function.

The equation for a b-spline surface is given by

\[ P(u, v) = \sum_{i=0}^{N} \sum_{j=0}^{M} P_{i,j} N_{i,k}(u) N_{j,l}(v), \]  where \( N \) is the basis functions, \( k \) is the curve order in the \( u \) direction, and \( l \) is the curve order in the \( v \) direction.

Similarly, the equation for a NURBS surface is given by

\[ P(u, v) = \frac{\sum_{i=0}^{N} \sum_{j=0}^{M} P_{i,j} W_{i,j} N_{i,k}(u) N_{j,l}(v)}{\sum_{i=0}^{N} \sum_{j=0}^{M} W_{i,j} N_{i,k}(u) N_{j,l}(v)} \]
3 Problem description

As stated by Bryce & Rutter (2002), games have been found to be successful when a high level of realism is present. The idea of supporting deformable geometry could increase the realism. A crater after an explosion that occurs at a location $x$, could increase the realism, but if the crater disappears after a certain time $t$, then that would perhaps not be as realistic as if it remains as long as the current session is active. Hence the regions that are deformed need to stay deformed. If a crater is created at location $x$, but not at location $y$, which both have been exposed to equal explosions, then that would maybe not either be as realistic as if craters are created at both locations. Hence the geometry need to be deformable everywhere.

He, et al. (2002), argues that when simulating tire tracks in a vehicle simulation, it is not enough to merely change the height of existing elevation points in the terrain, the resolution needs to be increased where a deformation should take place in order to take account for all new details associated with the change. This fact is true for larger deformations such as explosions as well. If the explosion is of small size such as for example a grenade, it is not enough to visualize this with a terrain density of 1 m or more between the elevation points. An illustration of how it could look if only the existing elevation points are changed, compared to the insertion of extra, can be seen in Figure 37.

Figure 37: Illustration of how an explosion could change the landscape with and without increasing the resolution of the terrain. In (a) a part of the geometry before an explosion occurs above its center point. The figure in (b) shows how the terrain could look after the explosion, if resolution stays the same. The final figure in (c) shows how the terrain could look after the explosion if the resolution is increased.

The goal is to find an algorithm that can visualize rather large terrains in real-time. The algorithm should also support real-time deformations on the terrain, and the resolution in a deformed region should be extendable if needed. The terrain requirements should be similar to those found in recently released games. Further, the algorithm should be deployable in a game on a home PC, fulfilling the requirements on processing power for other aspects of the game besides the terrain rendering.
3.1 Problem definition

Since the algorithm developed by He (2000), is the only one found concerning deformable terrain with increased resolution, the problem will be delimited to investigating this algorithm, and how deployable it could be in a game environment.

The algorithm presented by He, et al. (2002) could be implemented with a large grid size in order to be useful in a computer game environment, but this would assume that a limited amount of deformations occur throughout the landscape, or that deformations cancel each other over time, so that dynamic resolution cells can be removed from the geometry. As stated by Saltzman (1999), players of computer games are fond of blowing things up. This aspect introduces the possibility that an enormous amount of deformations could occur which will not cancel the effect of each other.

In his thesis, He (2000) presented an equation for calculating the memory consumption of the DEXTER algorithm implemented on the ROAM algorithm. The equation is approximately:

\[ M_{ROAM} = (n+1)^2 M_p + 2(2n^2 - 1)M_t + c(M_e + s^2 2^{2t}M_p + 2s^2 (2^{2t+1} - 2)M_t), \]

where \( n \) is the width of the terrain, \( M_p \) is the memory footprint for a terrain post, \( M_t \) is the memory footprint for a triangle, \( c \) is the number of dynamic terrain cells, \( M_e \) is the memory footprint for a dynamic cell, \( s \) is the number of original pair of leaf triangles covered by a dynamic cell, and \( t \) is the degree to which the resolution is extended.

By using this equation in a terrain of size 1024x1024, without extended resolution, gives two bintree structures with over 4 million triangles. If each triangle has a memory footprint of 50 bytes, including error value, pointers to its parent, neighbors, and children, and pointers to its vertices, and each terrain post has a memory footprint of about 12 bytes. Then this would yield a total memory usage of more then 220 Mb, which is close to the amount that most home PC’s contain today. If deformations resulting in an increased resolution of 8, \( t = 3 \), are created all over the landscape, and the memory footprint for a dynamic cell is 24 bytes, then over 268 million triangles would be needed, which in turn yields a memory usage of over 14 GB. This value is perhaps a little overkill, but a PC of today is only likely to have around 256-512 Mb of physical memory available. This could be solved by paging techniques, but as the maximum address space on a 32-bit architecture is around 4 GB, an increase of resolution of 8 could only be permitted in less 30% of the terrain.

Further, He (2000) highlights that when implementing his algorithm on the original ROAM algorithm; much more time is spent on retriangulating the terrain, and processing the split-, and merge queues. A lot of time was also spent on creating dynamic terrain cells, and extending the bintrees, but He, argues that the gain in resolution outweighs the cost of additional computation time.

He, et al. (2002), states that a part of future work could be to investigate how well the algorithm works with larger deformations of extensive terrain areas.
3.2 Hypothesis
The hypothesis is that the algorithm presented by He in 2000, can be improved concerning memory requirements, by:

1. Using curved surfaces, such as b-spline surfaces, to describe the geometry of the higher resolution cells.
2. Only allocating triangles for higher resolution cells when necessary, where necessary is defined as when the camera is close enough and directed towards a high resolution cell.

3.3 Aims and objectives
The aim of this final year project is to investigate if the hypothesis is valid.
To achieve the aim of this final year project, the following objectives will be performed:

1. Analyze the algorithm presented by He (2000).
2. Investigate different strategies concerning how the problem could be solved.
3. Chose a strategy for improvements based on suitability in a game environment on a home PC.
4. Implement a simple test environment based on (3).
5. By means of (4), evaluate if the suggested improvements proposed in (3), are in conjunction with practical use with regards to the performance needs in a game environment.

3.4 Expected result
It is expected to find out if the algorithm developed by He (2000), can be improved in such a way that the terrain still can be deformed, and that the deformed parts can be described with a higher resolution, whilst at the same time lowering the memory requirements of the algorithm, with the help of dynamic bintree allocation and curved surfaces for describing the higher resolution.
4 Method

This chapter will start with a recapitulation of certain issues that are likely to become a problem when using the DEXTER algorithm in applications such as games and simulators. In chapter 4.2, suggested changes to the algorithm will be described. Chapter 4.3 will state which changes are to be incorporated, and chapter 4.4 will describe how testing is to be performed in order to evaluate if the suggested changes are feasible concerning timing requirements. Finally, in chapter 4.5 statements regarding how to interpret the test results will be given.

4.1 Algorithm analysis

The ROAM algorithm, described in chapter 2.4.1, is a view-dependant LOD algorithm that refines the level of detail according to the current view. The maximum level of detail available in the ROAM algorithm is the level of detail described by all leaf triangles in the bintree structures. This can be seen as the original resolution of the terrain. The DEXTER enhancement for the ROAM algorithm is concerned with extending the resolution at parts in the terrain where the original resolution is not enough to cater for all details in a deformation, such as an explosion. The resolution is extended by means of extending the bintree structures at regions where more details is needed. The refinement process can then refine the terrain in certain areas to a much higher level of detail than the initial maximum detail level.

Studying the equation for memory usage given by He (2000):

\[ M_{\text{ROAMX}} = (n+1)^2 M_p + 2(2n^2 - 1)M_t + c(M_c + s^2 2^{2i} M_p + 2s^2 (2^{2i+1} - 2)M_t) \]

It is clear that there are approximately 4 times more triangles than terrain posts; He (2000) refers to vertices as terrain posts, and they can also include vertex normal vectors and texture coordinates. If each terrain post should store at what \((x, y, z)\) a vertex \(v\) is located in world space, and what normal, \(n = (a, b, c)\), is associated with the vertex, then approximately 24 bytes are needed for a vertex. A triangle on the other hand, has to contain pointers to all neighboring triangles, parent triangle, child triangles, and vertices, this in order to maintain a continuous triangle mesh when trimming the structure according to the current view. A triangle also has to include the internal structural error for using it, this gives that approximately 48 bytes are needed for a triangle. Hence the memory usage for each triangle is approximately 2 times the memory usage for a terrain post, and as there are 4 times as many triangles as terrain posts, this yields that the triangles needs about 8 times more memory than the terrain posts. For example, if the terrain posts need 12 Mb of memory, than the triangles would need about 96 Mb of memory. This gives that the amount of RAM used by the algorithm, is dependent on the amount of triangles allocated. Hence the ROAM algorithm would work on terrains of sizes which contain bintree structures that are as large as the amount of available physical and virtual RAM. This limitation is valid for the DEXTER algorithm as well; the maximum size of the bintree structures allocated can be as large as the amount of available RAM. This yields that the degree to which the resolution can be extended is directly dependent on how many triangles is needed.

A typical game environment would to day consist of a regular home PC with some extra power in the graphic card. Hence the available memory would be in the range of about 256-512 Mb.
The DEXTER algorithm could be deployed in a game environment, depending on how large the initial terrain is. The game Battlefield 1942, released by DICE in the fall of 2002, gives a good estimate of how large the terrain should be in a modern game environment. The different campaigns in the game, contains maps of sizes from 1.5 km² x 1.5 km² to 4 km² x 4 km², which is described by height maps of sizes from 256x256 to 1024x1024 (Hejdenberg, DICE, E-mail contact 2003-05-09). The DEXTER algorithm could work with the smaller of these resolutions, but in the higher setting, the amount of RAM consumed would be too much for a game environment to run smoothly, and a too low frame rate will most certainly destroy the playability of a game. Hence to use the DEXTER algorithm in a game environment, the memory usage of the algorithm needs to be lowered.

4.2 Strategies for changes

If level of detail could be exploited by the underlying structure of the bintrees as well as for the runtime refinement of the bintrees, memory would only be consumed by higher resolution cells that are needed according to the current view. If the view is located far away from a cell that has been deformed, then the extended resolution is not needed since the details will be too small when rendered on screen. The basic resolution will be enough to visualize the changes on a greater distance. Instead of creating all triangles in a higher resolution cell, a pool of bintree structures could be created, which at run time will be connected to the cells that needs them. This however introduces a problem with error computations, since these are performed in a bottom-up fashion in the ROAM algorithm.

As mentioned earlier in chapter 2.6, parametric surfaces such as b-spline surfaces, can hold very much information in a very compact manner. In the DEXTER algorithm, the extended resolution is contained in a two dimensional array of elevation points, as in the original ROAM algorithm. If instead, the extended resolution could be described by a parametric surface, memory could be saved as only the control points in the parametric surface would need to consume memory. By introducing parametric surfaces for describing the extended resolution, the problems with computing the errors in a dynamic bintree, described in the previous paragraph, can be solved as well. A parametric surface is described by polynomials, which have derivatives. Calculus gives the answer to the error computation problem, by means of finding the local maxima and minima in a certain interval. In parametric surfaces described by control points the maxima and minima of the parametric equation will be confined to be at the control points.

Another approach when using a parametric surface for describing the higher resolution, is to let the underlying graphics API tessellate the higher resolution cells on its own. If the bintree triangulation reaches a cell with higher resolution, the parametric surface would simply be passed on to the graphics API, to be tessellated there. This could pose problems with mesh continuity, since the unity of the bintree structures would be broken, but this could be solved by using a knot vector, which makes the curve pass through all points on the border of the surface. However, the underlying graphics API contains functions for efficiently evaluating and tessellating parametric surfaces, and by using the functionality of the API, calculations are moved further down the graphics rendering pipeline. On more modern graphics cards, functionality for evaluating curves and surfaces are implemented in hardware, postponing the calculations even further. Hence by using the surface routines of the graphics API, the CPU could be relieved from a lot of computations resulting in more
processing power for other aspects for a game environment, which have been mentioned earlier.

4.3 Strategy selection

The geometry of the high resolution cells will be described by NURBS surfaces with constant weighting factors, yielding a b-spline. The reason for having a NURBS is the amount of local control given over the resulting surface to a lower price. In a bézier surface, the degree of the surface would be directly dependant on the amount of control points, yielding that more calculations are needed as the amount of control points are increased. This is not the case for NURBS surface, as the surface order is decoupled from the amount of control points used. Hence NURBS surfaces will be used, as more control points can be utilized at a lower surface order, resulting in less computational cost. There was no functionality found in the weighting factors at this time, therefore it was chosen to keep them constant and achieving a b-spline. However, by using the NURBS basis functions instead of the b-spline basis functions, the b-spline can easily be turned into a NURBS if there is a need for this at a later stage.

A pool of dynamic bintree structures will be maintained and shared amongst all dynamic high resolution cells, created. The triangles of the dynamic bintrees will at need be attached to the original refinement structures. When a dynamic bintree is attached to the original resolution, that is the original static triangles described by the initial height map, the refinement process can use the amount of triangles from the dynamic bintree as it needs. By keeping the structures at the application level of the graphics pipeline, the top-down approach of the ROAM algorithm can be maintained. This can be good as the amount of frame time used by the refinement process can be kept at a constant level. Hence the API functions for rendering curved surfaces will not be used. It should however be noted that it could be more effective to have the API process a surface instead of the CPU, and that is in case there exists hardware support for tessellating surfaces, and that this hardware support is efficiently used by the graphics API.

4.4 Test environment

As stated earlier one of the most basic aspects of a computer game is the interactivity with the player. A low frame rate gives a low response rate from the game on the actions made by the user, which results in a low interactivity. A good estimate for if an algorithm would be functional in a game environment is at what frame rate the algorithm can perform its task. The ROAM algorithm has become very popular amongst game developers; hence it can probably approximate the terrain at a feasible frame rate. Besides calculating if the suggested changes can lower the memory requirements for the ROAM/DEXTER algorithm, the frame rate at which the algorithm can tessellate the terrain, should also be measured. The algorithm to be implemented, based on the ROAM algorithm with the DEXTER algorithm for extending the resolution, will from here on be referred to as SEROAM, for surface extended ROAM. It should be investigated if the SEROAM algorithm can perform its task at a feasible frame rate, and to estimate how the suggested changes affect the tessellation process. According to Akenine-Möller & Haines (2002), differences in the frame rate above 72 FPS, is effectively indetectable, and hence this is a good comparison value. It should however be kept in mind that other aspects of a game environment, such as artificial intelligence, also need a share amount of frame time.
In order to get a good estimate if the algorithm could be feasible in a game environment, several test cases should be performed on the algorithm, and these will now be explained in more detail.

4.3.1 Test cases

Three different height maps will be used, and these will be of sizes 257x257, 513x513, and 1025x1025. These three different height fields are chosen, since they are common sizes used in game environments, see chapter 4.1.

As the focus of this final year project is deformable geometry, deformations should occur in all test cases. A common sequence of deformations will be used in all test cases. The sequence of deformations will be described by script file created before any simulations are conducted, and it will contain a series of deformations to occur at a designated location and at a specified time. The deformations in the script file will be randomly generated. If a deformation is to occur at a time $t$, and in the location $x$, $y$, $z$, a bomb will be released at the location $x$, $y + 400$, $z$ at time $t$. It will accelerate with $-G$ until it collides with the terrain, and at collision, an explosion with an effect $e$, and radius $r$, will be simulated at the location of impact.

When tessellating the terrain in the ROAM algorithm, an accuracy value is associated. This accuracy value defines when a higher level of detail is needed in a certain region. As the algorithm tessellates through the landscape, for each triangle, which is in the current triangulation, a priority will be calculated. This priority will be tested against the accuracy value, and if above, the triangle will be replaced with its two children, and if below, it will be replaced with its parent. The accuracy value determines how fine grained the terrain should be. For each of the three height fields, three different accuracy values will be used.

This results in a total of 9 test cases for the SEROAM algorithm, of which all concerns deforming the terrain.

For each test case, an average set of variables will be calculated by recording a set of variables when moving through the landscape five times on a predefined path.

4.3.2 Measurement

In each frame rendered, a set of variables will be recorded in memory, which are later written to disk, these variables contains timing results and triangle counts as follows:

**Variable**: Total frame time

This variable is needed in order to calculate the FPS for each frame. This is the time elapsed between two consecutive frames.

**Variables**: Time to triangulate, time to render, time to deform.

These variables are needed in order to analyze which sections of the process are affected when extending the resolution.

**Variable**: Number of deformations that have occurred in the frame.

This variable is needed in order to calculate how the timing results relate to the number of deformations that has occurred.

**Variables**: Number of triangles in active triangulation, number of triangles rendered.

These variables are needed in order to analyze how the amount of triangles is affected when extending the resolution.
The variables listed above will be used to construct diagrams in order to analyze the effects of introducing the suggested changes to the ROAM/DEXTER algorithm.

4.3.3 Test architecture

All simulations will be conducted on a test environment consisting of an AMD XP2000+ processor running at 1667 MHz. The amount of physical RAM is 256 Mb. The environment is equipped with an Nvidia Geforce2 MX graphics card, with 32 Mb onboard graphics memory.

The operating system under which all simulations will be conducted is Windows 2000 Pro.

OpenGL, and the utility library GLUT, will be used as the API to the graphics card.

All programming will be done in C++, and the code will be compiled with Borland C++ Builder 5.

4.5 Analysis

Diagrams will be created for all simulations according to the variables described earlier.

The diagrams will be compared to see how the total frame time, and the time to conduct each specific part of the refinement process, as well as the render process, are affected by introducing extended resolution, which is described by curved surfaces.
5 Implementations

This chapter will be concerned with describing how the test environment will be implemented. First, design issues concerning the original ROAM part of the algorithm will be described in chapters 5.1-5.5. This will be followed by a brief explanation of aspects concerning extending the resolution in the ROAM algorithm, in chapter 5.6. Chapter 5.7 will present how the extended resolution will be described, and how this will be used at runtime. This will be followed by a description of how the higher resolution will be attached to the original resolution at runtime in chapter 5.8. Finally, chapter 5.9 is concerned with how the deformation of the geometry in the landscape will be conducted. The term accuracy, refers to how large height faults are currently allowed in the terrain.

5.1 Run-time refinement

In the original ROAM algorithm, by Duchaineau, et al. (1997), the triangulation of the terrain is achieved through an incremental greedy algorithm, which repeatedly splits the highest priority triangle, or merges the lowest priority diamond. This process is continued until the highest priority triangle in the split queue has a lower priority than the lowest priority triangle in the merge queue or until a certain triangle count or accuracy is reached.

He (2000) argues that the splitting and merging of triangles can be broken in two different parts, where all the elements in one of the two queues are handled separately from the processing of the elements in the other queue. That is, either all triangles in the merge queue are processed before all triangles in the split queue, or all the elements in the split queue are processed before any elements in the merge queue are processed. According to He (2000), no triangles which form a diamond that can be merged due to a split operation, will have a priority low enough to be considered for a merge operation based on the same view. This is true the other way around as well, no triangle which have been merged in a frame \( f \), will have a priority high enough to be considered for a split operation in the frame \( f \). In the following section, some algorithmic material will be given. In all listings, \( Q_s \) is the split queue, and \( Q_m \) is the merge queue. The general structure of a triangulate and render algorithm with separated split-, and merge operations, can be seen in Listing 1.

<table>
<thead>
<tr>
<th>Triangulate and render algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> Current view</td>
</tr>
<tr>
<td>Target accuracy</td>
</tr>
<tr>
<td><strong>Output:</strong> Renders the terrain on screen</td>
</tr>
</tbody>
</table>

1. begin
2. if \( f = 0 \) then begin
3. initialize terrain
4. initialize bintree structures
5. insert root triangles to \( Q_s \) and set \( Q_m \) to \( \emptyset \)
6. end
7. recalculate priorities for all elements in \( Q_s \) and \( Q_m \)
8. update merge queue \( Q_m \)
9. update split queue \( Q_s \)
10. render triangles, either by traversing the bintre or by using \( Q_s \)
11. end

Listing 1: Algorithm for triangulating and rendering the terrain in a frame, \( f \).
As can be seen in Listing 1, the merge queue is processed before the split queue. The functionality for update merge queue can be seen in Listing 4, and the functionality for the update split queue can be seen in Listing 2.

**Update split queue algorithm**

**Input:** A queue with all triangles in the current approximation mesh  
**Current view**

**Output:** A queue with a more optimal triangulation

```
1. begin
2. while there are triangles above the target accuracy in Qs and the number split operations performed is not to many do
3. begin
4. let T be the first element above the threshold of Qs
5. Split T
6. end
7. end
```

**Listing 2:** Algorithm for processing and updating the split queue.

The algorithm in Listing 2 continues to split triangles from the split queue as long as there are triangles with a priority above the wanted accuracy. In each iteration of the while loop, a split operation, which can be seen in Listing 3, is called for a triangle $T$. The split operations does not really split any triangles, it simply removes split triangles from the split queue, and inserts their children on the split queue. It also removes diamonds which are no longer mergable, from the merge queue, and inserts newly mergable diamonds to the merge queue. The split operation also takes care of updating all triangles neighboring connections in order to preserve a continuous triangle mesh.

**Split triangle algorithm**

**Input:** A triangle to be split, $T$

**Output:** $\emptyset$

```
1. begin
2. if $T$ has children then do
3. begin
4. let $T_b$ be the base neighbor of $T$
5. if $T_b$ is not NULL and dont have $T$ as base neighbor then do
6. Split $T_b$
7. update the neighbor connections for the children of $T$
8. remove $T$ from $Q_s$
9. insert the children of $T$ to $Q_s$
10. if $T_b$ is not NULL then do
11. begin
12. update the neighbor connections for the children of $T_b$
13. remove $T_b$ from $Q_s$
14. insert the children of $T_b$ to $Q_s$
15. insert the diamond formed of $T$ and $T_b$ to $Q_m$
16. else
17. insert the half diamond formed of $T$ to $Q_m$
18. end
19. if the parent of $T$ is in a mergable diamond $D$ then do
20. remove $D$ from $Q_m$
21. if $T_b$ is not NULL and the parent of $T_b$ is in a mergable diamond $D_b$ then do
22. remove $D_b$ from $Q_m$
23. end
24. end
25. end
```

**Listing 3:** Algorithm for splitting a triangle, $T$. If the triangle has a base neighbor, then the triangle and its base neighbor will be split together. If the base neighbor is not at the same level as $T$, then the base neighbor will first be forced split in order to keep the mesh continuous.
Update merge queue algorithm

<table>
<thead>
<tr>
<th>Input: A queue with all diamonds that can be merged in the current approximation</th>
<th>Current view</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: A queue containing all mergable diamonds in a more optimal approximation mesh</td>
<td></td>
</tr>
</tbody>
</table>

1. begin
2. while there are diamonds below the target accuracy in \( Q_m \) and the number of merge operations performed is not too many do
3. begin
4. let \( D \) be the last/first element below the threshold of \( Q_m \)
5. Merge \( D \)
6. end
7. end

Listing 4: Algorithm for processing and updating the merge queue.

The update merge queue algorithm, in Listing 4, merges all diamonds in the merge queue which have a priority less than the current accuracy value. The merge operation which is called can be seen in Listing 5. It has similar functionality as the split operation, but the other way around. It removes the children of the triangles in the diamond to be merged, from the split queue, removes the triangles forming the diamond to be merged from the merge queue, and inserts them to the split queue. It also handles the updating of neighboring connections for the triangles in the mesh. Finally it also inserts new diamonds to the merge queue.

Merge diamond algorithm

<table>
<thead>
<tr>
<th>Input: A diamond to be merged, ( D )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: ( \emptyset )</td>
</tr>
</tbody>
</table>

1. begin
2. let \( T_0 \) and \( T_1 \) be the two/one triangle/s forming \( D \)
3. remove the children of \( T_0 \) from \( Q_s \)
4. insert \( T_0 \) to \( Q_s \)
5. update neighboring connections for \( T_0 \)
6. if \( T_1 \) is not NULL then do
7. begin
8. remove the children of \( T_1 \) from \( Q_s \)
9. insert \( T_1 \) to \( Q_s \)
10. update neighboring connections for \( T_1 \)
11. end
12. remove \( D \) from \( Q_m \)
13. if the parent of \( T_0 \) is now part of a diamond \( D_0 \) then do
14. insert \( D_0 \) to \( Q_m \)
15. if \( T_1 \) is not NULL and the parent of \( T_1 \) is now part of a diamond \( D_1 \) then do
16. insert \( D_1 \) to \( Q_m \)
17. end

Listing 5: Algorithm for merging a diamond consisting of either one or two triangles.

5.2 Triangle queues

In the original ROAM algorithm, when a triangle is to be split, the highest priority triangle in the split queue will be selected for splitting, and when a triangle is to be merged, the lowest priority triangle in the merge queue will be selected for merging. This is good when there is a limited amount of time available for the update procedures, or when the approximation mesh may not become larger then some maximum mesh size. If the update procedure is stopped before a given accuracy is met, the approximation mesh will still be as close to a satisfying mesh as can be reached in the amount of split-, and merge operations that have been performed. He (2000) states that maintaining two priority queues adds to the computational cost, and if there is no restriction on the number of operations performed or any restriction on
the final mesh size, the priority queues can be replaced by two special FIFO queues. These FIFO queues will separate triangles with a priority above and below the target accuracy, and it can be seen as a priority queue where all elements above the accuracy has the same priority, and all triangles below the accuracy has the same priority. An example of such a queue can be seen in Figure 38. This enhancement yields a constant time for inserting triangles into any of the two queues, and can be effective when used together with modern graphics hardware since the rendering of primitives has moved from software to hardware, and that a couple of hundred more or less triangles will not affect the overall performance.

As can be seen in Listing 3 in chapter 5.1, there are approximately 4 remove operations and 5 insert operations performed on the queues for the split operation. Likewise, it can be seen in Listing 5 in chapter 5.1, that there are approximately 5 remove operations and 4 insert operations associated with the merge operation. By having the special FIFO queues, the time complexity for all insert operations have been made constant, but there is still a lot of processing associated with removing an item from any of the two queues. The time complexity for removing an object from a FIFO queue is at worst case $O(n)$. This can however be avoided by delaying the removal of items from the queues until the next time they are visited.

Each triangle will have two flags associated, dirtySplit and dirtyMerge, where the first is associated with the split queue, and the later with the merge queue. When a triangle is not currently in any of the two queues, both flags are set to true. When a triangle is inserted into one of the two queues, the flag corresponding to that queue is set to false. When a triangle is to be removed from any of the two queues, it is not actually removed from the queue, rather its flag associated with that queue is set to false. This yields constant time for removing items from a queue, since all queue items has to be traversed anyway.

The algorithm for updating the merge queue, in Listing 4, chapter 5.1, have to be complemented with one more check to see if the triangle fetched from the merge queue is actually a part of the merge queue. The element retrieved from the merge queue also has to really be removed after it has been processed. Similar changes in the code will also have to be conducted on the algorithm for updating the split queue in Listing 2, chapter 5.1. The algorithm for updating the priorities for all elements in any of the two queues can be seen in Listing 6.
**Update queue algorithm**

<table>
<thead>
<tr>
<th><strong>Input:</strong></th>
<th>A queue with triangles</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Output:</strong></td>
<td>A queue with triangles</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>List 6: Algorithm for updating a queue. All elements will have their priorities recalculated.</th>
</tr>
</thead>
</table>
| 1. **begin**
| 2. **while** elements left in old queue **do**
| 3. **begin**
| 4. let E be the first element of the old queue
| 5. if this is a merge queue **then do**
| 6. **begin**
| 7. if E is not dirtyMerge **then do**
| 8. **begin**
| 9. recalculate priority for E
| 10. insert E into new queue
| 11. **end**
| 12. **else**
| 13. if E is not dirtySplit **then do**
| 14. **begin**
| 15. recalculate priority for E
| 16. insert E into new queue
| 17. **end**
| 18. **end**
| 19. remove E from old queue
| 20. **end**
| 21. **end**

**5.3 View frustum culling**

In the ROAM algorithm, a priority will be calculated for every triangle that is processed. The priority computation involves projecting the error for a triangle on screen, and measuring how large the error will be in screen space. This calculation involves a great deal of processing cycles, and should be avoided as often as possible. As described in chapter 2.2, view frustum culling can greatly speed up the graphics rendering pipeline. By testing a sphere, containing each triangle, against the view frustum, a lot of processing cycles can be saved, as too much time does not need to be spent on a triangle which is not going to be visible on screen. The priority for a triangle which is not in the view frustum can be set artificially low, resulting in fewer priority calculations. Setting the priority for a triangle below the target accuracy also yields that it will not be considered for splitting unless it is forced to by its neighboring triangles. According to He (2000), merge operations should be welcome even if a triangle is not inside the view frustum, as this reduces the size of approximation mesh, this is also the case when the priority is set artificially low.

In chapter 2.2 it was also stated that by using a bounding volumes hierarchy, BVH, even more processing power can be saved, as whole blocks of primitives can be ruled out at an early stage in the hierarchy. Hence, if a triangle is found to not be inside the view frustum, then its children does not need to be tested against the view frustum, as they are automatically also outside the view frustum since their parent covers them completely. A set of regions, called patches, will be created for the complete terrain, and each such patch will contain two bintrees. Before the triangulation of the terrain starts for a frame, the patches will be tested against the view frustum. For each patch found intersecting the view frustum, a BVH of spheres, containing all triangles for the patch, will be processed recursively. If a sphere for a triangle is found completely inside or completely outside of the view frustum, its status will be propagated to all triangles descending from it.
5.4 Aging priority

According to Duchaineau, et al. (1997), it is too costly to recalculate the priorities for all triangles in every frame, since the number of split-, and merge operations performed in a frame, is much smaller than the number of triangles in the current approximation mesh. Instead, it is suggested that the priorities only should be recalculated when they potentially can affect a split-, or merge decision. Duchaineau, et al. (1997) mentioned the incorporation of priority deferral, which postpones the recalculation of a triangles priority until a frame, $f+x$, in which the view point has moved such a distance that the triangle is due for a split-, or merge operation. A deferral list should be kept for the next few dozens of frames, and triangles on the current frames deferral list should have their priorities recalculated. For how long a triangles priority recalculation can be postponed is given by the velocity of the view point. In case of a high velocity of the view point, there would be a large number of split-, and merge operations, which would need to be conducted during each frame, resulting in a low priority deferral rate, but Duchaineau, et al. (1997), also stated that in case of a situation where the view point has changed in such a way, that it would yield a large number of split- and merge operations, the algorithm should fall back on the top-down algorithm, and reinitialize the split-, and merge queues as if was the first frame.

As stated, most of the triangles in the approximation mesh for the previous frame will be present in this frame. Instead of using the suggested priority deferral by Duchaineau, et al. (1997), a kind of aging priority computation will be incorporated. Consider the split queue. As the changes in the current view is typically small between two consecutive frames, a triangle which is in the current approximation mesh, and had its priority calculated in the previous frame, and which was found to be below the current threshold value, will probably still have a priority below the threshold value in this frame. A triangle that had its priority calculated two frames ago, are much more likely to be due for a split operation, than a triangle that had its priority calculated in the previous frame. Hence the priorities for triangles needs to grow with the amount of frames that have passed since their priorities was last calculated. When a triangles priority is found to be above the current accuracy, its priority needs to be recalculated in order to know if it does need to be split. The same arguments go for the merge queue, but the other way around. The priorities for elements in the merge queue needs to get smaller for every frame, and if the priority for a diamond on the mere queue is found to be less than the current accuracy, then this diamonds priority needs to be recomputed. The amount by which the priorities needs to increase and decrease should be relative to the current accuracy value. Special care should be taken for triangles, which was previously completely outside of the frustum, but which have now been found fully or partially inside it. These triangles need to have their priorities recalculated immediately.

This type of aging priority will be incorporated into the algorithm for updating the priorities for all elements of the split queue. The update queue algorithm in Listing 6, of chapter 5.2, will be revised, and a new version of it can be seen in Listing 7.
Update queue algorithm

<table>
<thead>
<tr>
<th>Input: A queue with triangles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type of queue</td>
</tr>
<tr>
<td>Current threshold</td>
</tr>
</tbody>
</table>

| Output: A queue with triangles |

1. begin
2. while elements left in old queue do
3. begin
4. let E be the first element of the old queue
5. if this is a merge queue then do
6. begin
7. if E is not dirtyMerge then do
8. begin
9. P = P - accuracy / constant
10. if P <= accuracy then do
11. recalculate priority for E
12. set priority of E to P
13. insert E into new queue
14. end
15. else do
16. insert E into new queue
17. end
18. remove E from old queue
19. end
20. end
21. end

Listing 7: New algorithm for updating a queue. Aging priority will be used as well as actual removal of elements that should not be in the queue that the algorithm is processing.

The original priority calculation for the ROAM algorithm, presented in chapter 2.4, will not be used. Instead an alternate way for computing the priority is used, which only calculates the screen distortion at the center of a triangle. The alternate priority calculation technique can be found at Duchaineau (2001). Also, as the special FIFO queues are used, there is no actual need for computing the exact priority; it is enough to know when a priority is above or below the target accuracy.

5.5 Data structures & rendering

All triangles in a bintree will be stored in an implicit bintree structure. The two root triangles in a patch will be stored at indexes $i = 2$, and $i = 3$ in the implicit bintree array. A triangle's left and right children can be found at $i = i*2$, and $i = i*2+1$, respectively, and the parent of a triangle can be found at index $i = i/2$. This kind of structure eliminates the pointers to parent and child triangles in every triangle. These are replaced by a pointer to the patch which owns the triangle, together with an id which indicates at what position in the array the current triangle is stored. This structure will not reduce the memory usage, but it can speed up the retrieval of triangles, as every triangle can be reached through an index. An example of the binary tree structures used can be seen in Figure 39.
As stated in chapter 4.3, the geometry needs to be changed in the algorithm. In order to change the shape of the terrain, some of the elevation points in the height map need to be relocated. The ROAM algorithm uses regular grid spacing between all elevation points; hence the vertices may only be displaced vertically. When a deformation has occurred in the terrain, the errors for all affected triangles must be updated, otherwise the triangulation in a deformed part would not take account for the new locations of the vertices, rather it would triangulate according to the old positions of them. The updating of triangle errors will proceed in the same way as in the DEXTER algorithm, update flags will be added to the vertices, then all leaf triangles which have vertices that have been changed, will propagate an update message up the bintree structure. Finally, the original ROAM part of the algorithm will have a class composition which can be seen in Figure 40.

The rendering of the triangles in the current mesh, can be performed by either visiting all elements in the split queue, or by traversing the bintree structures. Traversing the bintree structures should be avoided, as this yields computational overhead if it is performed with the help of recursive functions. Visiting all elements in the split queue, means that one primitive at a time has to be sent onto the graphics pipeline. In order to push more primitives, modern graphics cards likes to have the data in a streaming manner, such as strips, or fans of triangles. Another approach is to use vertex arrays, where a buffer of vertices is sent to the graphics API. When using vertex arrays, a buffer of vertices is created which is sent directly to the graphics hardware, via the graphics API. An array of indexes is also sent along that indicates which vertices are to be rendered. The graphics API, or hardware, then constructs
graphical primitives from the vertices, according to the information in the array of
indexes. Vertex arrays will be used in the algorithm, and in order to maintain them,
functionality will be added to the split-, and merge operations presented in Listing 3,
and Listing 5 of chapter 5.1, which inserts and removes the vertices associated with
the triangles that are split-, and merged. As a triangle consists of three vertices, each
triangle to be rendered will be given three positions in the index list.

5.6 Extending the resolution

In the DEXTER algorithm, an extension of resolution is achieved through extending
the bintree structures at locations where more resolution is needed. This is done by
recursively subdividing the original leaf triangles, which cover a region that needs
more resolution, until a new higher resolution is achieved in the region. The newly
created triangles are from the time of their creation until they are removed, a static
part of the bintree structure. As stated in chapter 4.1, it is the triangle structures that
are the main memory consumers, hence the need for dynamic bintree allocation. A
region which needs higher resolution does not need to allocate triangles for the higher
resolution until the triangles describing the higher resolution is needed by the
triangulation process.

In a bintree structure, the first level, \( l=1 \), is described by the two root triangles, which
share the same base edge. At level \( l+1 \), some triangles will not share a base edge with
another triangle, but at level \( l+2 \), all triangles will again share their base edge with
another triangle. Continuing down the structure, it can be found that at every uneven
level, all triangles share their base edge with another triangle. The height fields used
with the ROAM algorithm must be of the form \((2^n+1)^2\). The bintree structures
used in the ROAM algorithm takes the form of perfect binary trees, with \(2^{(2^n+2^n)}-1\)
number of nodes. The tree depth in a perfect binary tree can be calculated with the
formula:

\[
Depth_r = \frac{\ln(x + 1)}{\ln(2)}.
\]

In the equation, \( x \) represents the number of nodes present in the tree. This can be
simplified as follows:

\[
Depth_r = \frac{\ln\left(2^{(2^n+2^n)}-1+1\right)}{\ln(2)} \Rightarrow Depth_r = \frac{\ln(2^{2n+1})}{\ln(2)}
\]

Now, using the laws for logarithms, this can be solved to yield

\[
Depth_r = \frac{(2n+1)\ln(2)}{\ln(2)} \Rightarrow Depth_r = 2n + 1
\]

Hence the depth of all legal bintrees in the ROAM algorithm must be an uneven
number greater than zero, which together with the knowledge that at every uneven
tree level, all triangles share their base edge with another triangle, yields that at the
finest level of triangulation, all triangles are a part of a perfect diamond.

This information can be used to state that if the resolution is to be increased, a high
resolution patch can be created to cover pairs of two triangles at the finest level of
detail, which share the same base edge. This patch however, does not need to create
all triangles which it needs to contain in order to describe the higher resolution. It will
instead have knowledge of a manager that statically has created a number of bintrees,
together with a set of corresponding height maps, which can be retrieved during runtime, by any high resolution patch that needs them. The class composition for the original ROAM algorithm in Figure 40, chapter 5.5, will be complemented with a bintree manager, and the incorporation of two different patch classes. The modified class composition can be seen in Figure 41.

5.7 Describing the higher resolution

In the DEXTER algorithm, the extended resolution is described by a two dimensional matrix of elevations points, as the original resolution is. In this implementation, the higher resolution will be described by b-spline surfaces, captured in NURBS surface objects. If all high resolution patches are created with the same number of control points in each parametric direction, and if the number of control points in a high resolution patch does not change during run-time, and also, if a pre defined maximum amount of extractable points for each b-spline surface is defined, then all b-spline surface objects can share the same set of basis functions. Figure 42, gives an illustration of two b-spline surfaces described by triangles, and their associated control points, which holds the geometry of the surface. In the illustration, 5 control points in each parametric direction is used, and from the surface, a maximum of 17x17 vertices are extracted.

---

**Figure 41:** Class diagram showing how the original ROAM composition has changed in order to support high resolution patches with dynamic bintree retrieval.

**Figure 42:** Illustration of two b-spline surfaces described by 5x5 control points each. The triangles in both surfaces, is created from 17x17 vertices, extracted from each b-spline object. The black dots in the illustration, represents the control points for each surface.
As previously mentioned, all triangles in the original mesh which are affected by a deformation, needs to have their errors recomputed when a deformation has occurred in the terrain. When extending the resolution, all triangles covering the extended resolution need to have their errors based on the extended resolution. As described in chapter 2.4.1, the errors for the triangles in the ROAM algorithm are computed bottom up, this will become a problem when using dynamic bintrees, which is not allocated until the triangles in a high resolution patch is requested by the triangulation process. Without the triangles allocated, their errors can not be computed and propagated up the tree, and hence they will never be used.

If the error present in a high resolution patch could be calculated without having the triangles allocated, this problem could be solved. Then all leaf triangles in the original resolution, which have high resolution patches beneath them, can update their errors from the high resolution patch. If the higher resolution in a high resolution patch is described by a b-spline surface, then a bound on the total error in the high resolution patch can be established. A parametric surface is built by control points, which pulls the surface towards them. The farther away from a control point, a vertex on the surface resides, the less it will be affected by that control point. It can be established that the vertices on the surface that are pulled the most, is the vertices that are closest to the control points, one for each control point. In chapter 2.6 it was described, that by having an open repeated knot vector when building the parametric surface, the surface would pass through the control point, which has the repeated values. The control points defined at the corners of the high resolution patch needs to be at the same positions as the vertices building up the leaf triangles covered by the high resolution patch. Hence the knot vectors for both u-and v directions needs to be open repeated knot vectors. Now it can also be established that the maximum deviation of the parametric surface is the difference between the surface, and the two triangles which are covered by the surface. A maximum bound for the deviation between the parametric surface and the triangles covered by it can be calculated by comparing each control point’s position with a point that is calculated by interpolating across the two triangles which cover the surface. This gives a summation equation:

\[
d_{\text{max}} = \sum_{j=0}^{M} \sum_{i=0}^{N} c_{i,j} - \left( A + \overrightarrow{AC} \frac{j}{M} \right) + \left( A + \overrightarrow{AC} \frac{j}{M} \right) \left( B + \overrightarrow{BD} \frac{j}{M} \right) + \frac{i}{N}
\]

where \(A, B, C, D\) is corner vertices, and \(M, N\) is the number of control points in the u- and v direction, respectively, and \(c_{i,j}\) is a control point on the parametric surface. This maximum deviation of a dynamic patch will be propagated to the two leaf triangles of the original structures that are covered by it. During run-time, the refinement process needs to know the height fault for every triangle it is to compute a priority for. The dynamic structures are to be attached to the original triangles, in such a way that they are treated equally by the refinement process; the refinement process refines the triangles of the dynamic structures in the same way as the static structures. In order for this to work, a height fault for every triangle in a dynamic bintree also needs to be established. An exact height fault as those computed bottom-up according to the ROAM structure, could be established for every triangle in a dynamic structure at time of attachment, but this is not the strategy chosen, as it is considered being too time consuming. Instead, the triangles in a dynamic bintree, is assigned height faults as follows:
\[ d_{\text{tri}} = \begin{cases} \frac{d_{\text{max}}}{(\text{Tri}_{\text{level}})^2} & \text{if } (\text{Tri}_{\text{level}} = 0) \\ \frac{d_{\text{max}}}{\text{Tri}_{\text{level}}} & \text{else} \end{cases} \]

where \( \text{Tri}_{\text{level}} \) represents the level in the bintree at which the triangle in focus resides. The reason for this height fault calculation is that the maximum bound is a highly overestimated value, and the true maximum deviation has to occur at the control points, which in this implementation is usually much fewer than the total amount of points that can be extracted.

The class composition in Figure 41, chapter 5.6 is refined to include NURBS objects, and a NURBS basis function object. The resulting class composition can be seen in Figure 43.

**5.8 Attaching the higher resolution**

In chapter 5.6, it was stated that the high resolution patches needs to allocate triangles when they are needed by the triangulation process. This can be constrained further: when the triangulation process can reach any triangles which belong to a high resolution patch, then the high resolution patch needs to allocate triangles. Equally, when the triangulation process can no longer reach any of the triangles within a high resolution patch, then the triangles can be deallocated. So when are the triangles within the high resolution patch in reach by the triangulation process?

A triangle belonging to a high resolution patch, \( P \), is defined as \( T_P \). The two leaf triangles covering a high resolution patch, \( P \), is denoted \( T_o \) and \( T_f \). The split queue, \( Q_s \), can now be used to establish when the triangles of a high resolution patch are in reach for the triangulation process, as the split queue contains all triangles in the current approximation mesh. The patch \( P \) needs to have triangles allocated as long as the following logical statement is true:
\((T_p \in Q_t \land T_p \in P) \lor (T_0 \in Q_t \lor T_1 \in Q_t), \land P \subseteq (T_0 \land T_1)\),

which translates to: A high resolution patch needs to be have its dynamic structures attached if either of its two parent triangles is present in the split queue, or if any of the triangles in the dynamic bintree allocated, is present in the split queue.

It is not feasible to check the split queue to see if the statement is true, since that would be too costly. It can however be checked when triangles are inserted and removed from the split queue. Further the only two triangles that need to be checked are the two leaf triangles covering the high resolution patch, as long as they have knowledge of the patch, if any, which are attached to them. If a triangle that is covered by a high resolution patch enters the split queue in association with a split operation, then that high resolution needs to be attached. Further, if both leaf triangles which are covered by a high resolution patch are removed from the split queue in association with a merge operation, then that high resolution patch can be detached.

The bintree manager, which statically creates several dynamic bintrees, which can be requested by any dynamic high resolution patch, also creates a corresponding height map for each bintree it creates. The height map describes all vertices associated with the triangles in the dynamic bintree. At run-time when a dynamic patch needs to be attached to the refinement structures, the dynamic patch in question, requests a dynamic bintree with the corresponding height map from the manager. The height map is then filled in with values extracted from the b-spline surface for the dynamic patch in focus. In order to attach the newly retrieved dynamic bintree to the static structures, there is one important aspect to take account for. All the dynamic bintrees received from the manager are oriented the same way in world space, but all pairs of leaf triangles is not oriented the same way in world space. This is illustrated in Figure 44, where the orientation of leaf triangle pairs, comes in four flavors.

As can be seen in Figure 44, there are four cases for rotating the dynamic bintree in order to match the associated pair of leaf triangles. As the height map describes all corner points for all triangles in a dynamic bintree, rotating the height map would also yield a rotation of the dynamic bintree. Hence when the values are extracted from the b-spline, the current rotation is also taken account for.
The continuity problem of extending the resolution of the terrain will be solved in the same way as in the DEXTER algorithm. He (2000) proves by mathematical induction that if cells which have been introduced with a higher resolution are surrounded by transition cells, then the continuity is preserved. The transition cells should have zero error and be of the same resolution as the newly introduced cell. Hence high resolution transition patches will be introduced, surrounding all high resolution patches. These transition patches will have their height faults set to zero, and their control points will be an interpolation over the vertices of the leaf triangles covered by them.

5.9 Deformations

This section will describe how the terrain will be deformed according to simulating the detonation of a bomb at a specified location. The material in this chapter is based on an article by Formitchev (2000), which is concerned about 2D surface deformations.

An explosion at a position, $p_{Source}$, will affect all vertices within its range of influence, $r$. The range of influence can be seen as the radius of a sphere, and all vertices present within this sphere will be affected by the force exerted by the explosion. The amount of force that will affect a given vertex, $v$, will be relative to the distance between the vertex and the center of the explosion. The direction of the force acting on a vertex is the vector created from the vertex to the center of the explosion.

The terrain consists of several vertices which are connected to form triangles. If a force $F$ is acting upon a vertex, $v$, then all other vertices which are connected to it via the triangles will also be affected in some way. Each vertex, $v_i$, exerts a force on every vertex, $v_j$, which is connected to it. These forces will be equal but in opposite direction, and they are said to be in a stable state when no external forces are present. When the external force, $F$, is acting on the vertex, $v_i$, then the state between the two vertices are no longer stable. The forces exerted from both vertices will try to move them in such a way that equilibrium is reached again.

The deformations in the ROAM algorithm, can only affect the height of vertices, since the algorithm is built upon a regular grid with even spacing between all points. Hence the vertices will only be displaced vertically.

If a deformation occurs in the terrain, which has a radius of influence that is less than two times the horizontal grid spacing, then a high resolution patch will be introduced in between all original vertices that are affected. The b-spline surface of the high resolution patch will be initialized by interpolating across the corner vertices of the triangles covered by the high resolution patch. The control points will after that be treated as any other vertex, and their displacement will be calculated in the same manner as the original vertices of the terrain. Transition patches will be introduced surrounding all high resolution patches, and the control points in a transition patch, will keep their control points unaffected by the deformation.
6 Results & analysis

In this chapter, the algorithm, SEROAM, which has been implemented, will first be analyzed, and then the results from the simulations conducted on the algorithm will be presented and analyzed. In chapter 6.1, a memory analysis of the extended algorithm will be performed. This will be followed by the results from the simulations, in chapter 6.2. Each diagram presented will briefly be discussed. The source code for the implementation of the SEROAM algorithm can be found in appendix A.

6.1 Memory analysis

This chapter will begin with an analysis of the memory consumed by the extended algorithm, SEROAM, which have been implemented. This will be followed by some example calculations of the total memory usage in the SEROAM.

6.1.1 Memory consumption

The terrain is divided into static patches, which have a width of 64 triangles. Each patch contains two bintrees, descending from its two root triangles. A static patch class has a memory footprint of \(M_{PS}\), and a triangle has a memory footprint of \(M_T\). Each static patch also constructs a rendering list, which contains all vertices to be rendered; the vertices are grouped three and three according to which triangle they belong to. An index list is also created, which associates the elements of the rendering list with triangles. Both the rendering-, and index lists, is constructed from unsigned integers. Now if an unsigned integer use \(M_{UINT}\) memory, then the total memory usage for a static patch, \(M_{PATCH_S}\), is given by:

\[
M_{PATCH_S} = M_{PS} + 2(2 \times 64^2 - 1)M_T + 2 \times 64^2(1 + 3)M_{UINT}.
\]

For every pair of two leaf triangles, which are base neighbors, there can be a dynamic patch created to cover them. A dynamic patch contains a b-spline object with a memory footprint of \(M_{B\text{-}\text{SPLINE}}\), and it also has a set of \(c^2\) control points, \(c\) vertices in each parametric direction. The control points are evenly distributed in the x-, and z-planes, over the dynamic patch; hence a control point only consists of one float to describe the height. If the memory footprint for the dynamic patch it self is \(M_{PD}\), then this yields a total memory usage for each dynamic patch, \(M_{PATCH_D}\), as follows:

\[
M_{PATCH_D} = M_{PD} + M_{B\text{-}\text{SPLINE}} + c^2(M_{\text{FLOAT}}).
\]

Each b-spline object created in a dynamic patch must have access to a set of basis functions. All b-spline objects share the same basis functions, and it is created by the terrain class at startup. The b-spline basis class itself, has a memory footprint of \(M_B\), and it contains a set of pre calculated basis values together with the knot vectors it need. From each b-spline object, a total of \(n_{mv}\) vertices in each direction may be created. The number of control points in each direction, is \(c\), and the maximum surface order is denoted \(s_o\). The precalculated values are stored as floating point values. This yields a total memory usage for the b-spline basis, \(M_{BASIS}\), as follows:

\[
M_{BASIS} = M_B + 4n_{mv}cs_oM_{\text{FLOAT}} + 2(c + s_o)M_{\text{FLOAT}}.
\]

The dynamic b-spline patches must be able to quickly allocate bintrees at need. The bintree manager is created by the terrain class at startup, and it creates a number of dynamic bintrees, which can be retrieved by the dynamic patches at runtime on demand. The structure for the bintree manager has a memory footprint of \(M_{BM}\), and it

49
allocates \( n_{db} \), dynamic bintrees. For each dynamic bintree created by the manager, a set of vertices are created, \( n_{mv} \) in each direction, and triangles are created to form bintrees matching the vertices. Each dynamic bintree also creates a rendering list, and an index list, in the same way as the static patches. The internal structure for a dynamic bintree has a memory footprint of \( M_{DBT} \). This yields a total memory usage, \( M_{MANAGER} \), for the bintree manager, as follows:

\[
M_{MANAGER} = M_{BM} + n_{db} \left( M_{DBT} + (n_{mv})^2 M_V + 2(2(n_{mv} - 1)^2 - 1)M_T + 8(n_{mv} - 1)^2 M_{UINT} \right).
\]

In order to facilitate the split-, and merge operations, two triangle queues must be maintained. The structure for a triangle queue has a memory footprint of \( M_{TQ} \), and it contains queue elements with a memory footprint of \( M_{QE} \) each. The split queue contains \( n_s \) number of elements, and the merge queue contains \( n_m \) number of elements. This gives a total memory usage for the split queue, \( M_{SPLIT} \), as follows:

\[
M_{SPLIT} = M_{TQ} + n_s M_{QE}.
\]

, and the total memory usage for the merge queue, \( M_{MERGE} \), is given by:

\[
M_{MERGE} = M_{TQ} + n_m M_{QE}.
\]

The terrain class is the heart of the algorithm, and it is here that most of the processing is performed. The structure for the terrain class itself has a memory footprint of \( M_{TE} \), and it contains all static patches \( M_{PATCH_S} \), the b-spline basis \( M_{BASIS} \), the bintree manager \( M_{MANAGER} \), and the triangle queues \( M_{SPLIT} \), and \( M_{MERGE} \). The terrain class also creates all vertices used by the static patches, and the memory usage for a vertex is denoted \( M_V \). If the width of the terrain is given by \( w \), then the total memory usage for the terrain, \( M_{TERRAIN} \), is given by:

\[
M_{TERRAIN} = M_{TE} + w^2 M_V + \left( \frac{w-1}{64} \right)^2 M_{PATCH_S} + M_{BASIS} + M_{MANAGER} + .
\]

\[
M_{SPLIT} + M_{MERGE}.
\]

Finally, if \( d \) is the number of dynamically created patches, then the total memory usage, \( M_{ROAMx2} \), is computed as follows:

\[
M_{ROAMx2} = M_{TERRAIN} + d M_{PATCH_s,D}.
\]

Expanding the above yields a total memory usage of:

\[
M_{ROAMx2} = M_{TE} + w^2 M_V + \left( \frac{w-1}{64} \right)^2 \left( M_{PS} + 2(2 \times 64^2 - 1)M_T + 2 \times 64^2 (1 + 3) M_{UINT} \right) +
(M_{B} + 4n_{mv} c \sigma_{s} M_{FLOAT} + 2(c + s) M_{FLOAT}) +
(M_{BM} + n_{db} \left( M_{DBT} + (n_{mv})^2 M_V + 2(2(n_{mv} - 1)^2 - 1)M_T + 8(n_{mv} - 1)^2 M_{UINT} \right) +
(M_{TQ} + n_s M_{QE}) + (M_{TQ} + n_m M_{QE}) + d \left( M_{PD} + M_{B-SPLINE} + c^2 (M_{FLOAT}) \right).
\]

6.1.2 Comparing values

In chapter 3.1, a memory calculation for the ROAM/DEXTER algorithm was performed on a 1025x1025 terrain. This chapter will perform a memory calculation on the SEROAM algorithm, on the same size of terrain, in order to compare if the changes lowered the memory consumption.
The total memory consumption for the algorithm, which was given in the previous chapter, will be used. The calculations will be performed in steps in order to enhance reading, but first the variables used in the equations needs to be defined.

The classes of the compiled SEROAM algorithm have memory consumptions as follows:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Class type</th>
<th>Base memory consumed per Instance, in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M_B )</td>
<td>Nurb basis</td>
<td>48</td>
</tr>
<tr>
<td>( M_{BM} )</td>
<td>BinTree manager</td>
<td>20</td>
</tr>
<tr>
<td>( M_{DBT} )</td>
<td>Dynamic BinTree</td>
<td>48</td>
</tr>
<tr>
<td>( M_{FLOAT} )</td>
<td>float (not a class)</td>
<td>4</td>
</tr>
<tr>
<td>( M_{B-SPLINE} )</td>
<td>B-spline</td>
<td>16</td>
</tr>
<tr>
<td>( M_{QE} )</td>
<td>Queue element</td>
<td>20</td>
</tr>
<tr>
<td>( M_{PD} )</td>
<td>Dynamic patch</td>
<td>100</td>
</tr>
<tr>
<td>( M_{PS} )</td>
<td>Static patch</td>
<td>104</td>
</tr>
<tr>
<td>( M_T )</td>
<td>Triangle</td>
<td>48</td>
</tr>
<tr>
<td>( M_{TQ} )</td>
<td>Triangle queue</td>
<td>24</td>
</tr>
<tr>
<td>( M_{TE} )</td>
<td>Terrain</td>
<td>184</td>
</tr>
<tr>
<td>( M_{UINT} )</td>
<td>unsigned integer (not a class)</td>
<td>4</td>
</tr>
<tr>
<td>( M_V )</td>
<td>Vertex</td>
<td>24</td>
</tr>
</tbody>
</table>

Table 1: Table showing the base memory consumption for the classes constructed.

The other variables used in the memory equation, can be split in two groups, static and dynamic variables. The static variables include, the width of the terrain, \( w = 1025 \), the maximum number of vertices in each direction for a dynamic patch, \( n_{mv} = 9 \), the number of control points in each parametric direction for a b-spline, \( c = 5 \), and the maximum surface order for a b-spline, \( s_o = 4 \). The dynamic variables are: the number of dynamic bintrees created, \( n_{db} \), the number of elements in the split queue, \( n_s \), and the number of elements in the merge queue \( n_m \). All values in the calculations to follow, is regarded as being bytes. In the memory consumption for a triangle, \( M_T \), 4 bytes is added for the triangle variance when solving for \( M_{PATCH,S} \). This since the static patches also creates an array of variance values for all triangles.

Now, calculating \( M_{PATCH,S}, M_{PATCH,D}, M_{BASIS}, M_{MANAGER}, M_{SPLIT}, M_{MERGE} \), gives

\[
M_{PATCH,S} = M_{PS} + 2(2*64^2 - 1)M_T + 2*64^2(1 + 3)M_{UINT} = 104 + 2*8191*52 + 32768*4 = 104 + 851854 + 131072 = 983040
\]

\[
M_{PATCH,D} = M_{PD} + M_{NURB} + c^2(M_{FLOAT}) = 100 + 16 + 5^2(4) = 216
\]

\[
M_{BASIS} = M_B + 4n_{mv}cs_oM_{FLOAT} + 2(c + s_o)M_{FLOAT} = 48 + 4*9*5*4*4 + 2*9*4 = 48 + 2880 + 72 = 3000
\]
\[ M_{\text{MANAGER}} = M_{BM} + n_{db} \left( M_{DBT} + (n_{mv})^2 M_V + 2(2(n_{mv} - 1)^2 - 1)M_T + 8(n_{mv} - 1)^2 M_{UINT} \right) = \\
20 + n_{db} \left( 48 + 9^2 \times 24 + 2(2 \times 8^2 - 1) \times 48 + 8 \times 8^2 \times 4 \right) = \\
20 + n_{db} \left( 48 + 1944 + 12192 + 2048 \right) = \\
20 + 16232 n_{db} \]

\[ M_{\text{SPLIT}} = M_{TQ} + n_s M_{QE} = \\
24 + 20 n_s \]

\[ M_{\text{MERGE}} = M_{TQ} + n_m M_{QE} = \\
24 + 20 n_m \]

Calculating \( M_{\text{TERRAIN}} \):

\[ M_{\text{TERRAIN}} = M_{TE} + w^2 M_V + \left( \frac{w-1}{64} \right)^2 M_{\text{PATCH-S}} + M_{\text{BASIS}} + M_{\text{MANAGER}} + \\
M_{\text{SPLIT}} + M_{\text{MERGE}} = \\
184 + 1025^2 \times 24 + \left( \frac{1024}{64} \right)^2 \times (983040) + (3000) + (20 + 16232 n_{db}) + \\
(24 + 20 n_s) + (24 + 20 n_m) = \\
184 + 25215000 + 256 \times 983040 + 3020 + 16232 n_{db} + 24 + 20 n_s + 24 + 20 n_m = \\
184 + 25215000 + 251658240 + 3068 + 16232 n_{db} + 20 n_s + 20 n_m = \\
276876492 + 16232 n_{db} + 20 n_s + 20 n_m \]

Finally, put it all together to achieve \( M_{\text{ROAMX2}} \):

\[ M_{\text{ROAMX2}} = M_{\text{TERRAIN}} + d M_{\text{PATCH-D}} = \\
276876492 + 16232 n_{db} + 20 n_s + 20 n_m + 216d \]

In order to establish a worst case scenario, the maximum value of \( n_{db} \), \( n_s \), \( n_m \), and \( d \) must be established. The maximum number of dynamic patches that can be created, is equal to the half the number of original leaf triangles. In a terrain of width, \( w = 1025 \), this would be: \( d = (1025 - 1)^2 \). The maximum number of dynamic bintrees that could be needed occurs if dynamic patches have been created all over the terrain, and if all these dynamic patches are needed at the same time. Hence the maximum value for \( n_{db} \) is equal to \( d \). The maximum number of elements in the split queue, \( n_s \), occurs if dynamic patches are created all over the terrain, and if all leaf triangles of all dynamic patches, are used at the same time. This gives \( n_s = 2((w-1)n_{mv})^2 \). The maximum number of elements in the merge queue, \( n_m \), can at most be half the maximum number of elements in the split queue, hence \( n_m = n_s / 2 \). Now, solving with \( n_{mv} = 9 \) gives:
The worst case scenario would need approximately 17.7 GB of memory. As can be seen above, it is the number of allocated bintrees that consumes most of the memory. The assumption for dynamic bintrees to work, is that the highest available resolution is not needed everywhere in the terrain at the same time. This is also one of the fundamental facts about level of detail algorithms, as their purpose is not to use the finest level of detail everywhere. In case the highest level of detail available would be required everywhere, a brute force algorithm, where all the finest primitives are simply pushed to the graphics pipeline, would yield better performance. Also, the dynamic bintrees would no longer be dynamic as they are all needed at the same time.

The major difference in memory consumption between the SEROAM, and DEXTER algorithms, is that the prerequisites for a worst case scenario has changed. In the DEXTER algorithm, a worst case scenario is achieved when all locations in the terrain have been covered by high resolution patches, whilst a worst case scenario in the SEROAM algorithm, is achieved when the complete terrain has been covered by high resolution patches, and that all these high resolution patches is needed by the refinement process simultaneously.

In the SEROAM algorithm, the worst case scenario can however be limited. The refinement algorithm is defined in such a way that it does not try to use resolution not currently attached even if it exists, and could be needed. The attachment procedure is defined in such a way that a maximum amount of simultaneously attached high resolution patches can be defined. In case of a maximum bound on the amount of simultaneously attached high resolution patches, the algorithm would initially create a given number of dynamic bintrees that can be used. If a high resolution patch requests a dynamic bintree, but there are no more free bintrees, than that high resolution patch would have to wait for a bintree to be released. By having the algorithm defined like this, a maximum bound can be established for the worst case memory scenario, but to some possible loss in visual appearance, in case a patch needs its resolution but is denied it.

Now another calculation will be performed, in which a maximum amount of dynamic bintrees will be created at startup. This example is calculated in order to show that the worst-case scenario can be lowered, while still extending the resolution everywhere. A short discussion will follow in order to estimate how many dynamic bintrees should be created. Assume that the details described by the high resolution patches, is only small scale changes compared to the details described by the static patches. Further assume that the details described by the static patches, is enough, if at a greater distance from the camera.

In the simulations conducted on the algorithm, a horizontal grid spacing of 4 m was used between all vertices of the original resolution. This gives that one dynamic bintree would cover $4 \times 4 = 16$ square meters. To cover an area of 16000 m$^2$, there would be 4000 dynamic bintrees needed. Now consider looking through a camera, then everything behind the camera lens, would not be visible. This gives that the 4000 dynamic bintrees could be used in front of the camera. Now consider the area of half a circle,
distributing these 4000 dynamic bintrees over that area, would give a radius as follows:

\[ A = \frac{\pi r^2}{2} \Rightarrow r^2 = \frac{2A}{\pi} \Rightarrow r = \sqrt{\frac{2A}{\pi}} = \sqrt{\frac{2 \times 16000}{\pi}} = 101. \]

Having a pool of 4000 dynamic bintrees, then all possible high resolution patches within a distance of 101 m in front of the camera could be assigned a dynamic bintree simultaneously. Now, as the high resolution patches is assumed describing only smaller changes in detail, the radius of 101 m, will be considered enough. According to He (2000), the split-, and merge queues can be discarded, as they usually only contains some thousand objects. Calculating with 4000 bintrees using the previously presented memory algorithm, gives:

\[ M_{ROMX2} = 276875492 + 16232n_{db} + 216d = \\
276876492 + 16232 \times 4000 + 216 \times 1024^2 = \\
276876492 + 64928000 + 226492416 = \\
568,296,908 \]

The assumption made about the computer system, on which the algorithm should be deployable, was that the amount of RAM available would be around 256-512 MB. It is clear that 568 MB is more than 512 MB; hence an extended resolution everywhere in the terrain can not be permitted. An extension of the level of detail can only be permitted in 90% of the terrain. It can also be seen in the calculation, that if not creating any dynamic bintrees, the memory consumption would be approximately 503 MB, which leaves almost no memory left for other aspects of the game environment, which is not acceptable.

6.2 Timing results

This chapter will present, compare, and analyze the data collected during the simulations conducted. First, some common properties for all simulations will be presented in chapter 6.2.1. This will be followed by the results collected when simulating on a terrain of size 257x257, in chapter 6.2.2, 513x513, in chapter 6.2.3, and 1025x1025, in chapter 6.2.4. A short discussion will be connected to each diagram, and in chapter 6.2.5, the collected data will be summarized.

6.2.1 Common simulation properties

One script file with explosions will be crated, and this will be used in all simulations. The script file will contain 1000 explosion entries, and each entry will have the following five variables:

1. Activation time for this entry in the script file. The time \( t \), when line, \( l \), will be activated, must be greater or equal to the time, \( t-1 \), which represent the activation time for line \( l-1 \).
2. At what x-position, in world space, a bomb will be released. This is measured in 0-100.00% of the terrain size.
3. At what z-position, in world space, a bomb will be released. This is also measured in 0-100.00% of the terrain size.
4. The radius of influence for this particular explosion.
5. The effect of this particular explosion.

There will be four types of explosions created, large, medium, small, and extra small. The difference amongst them will be the radius of influence, and the effect, described in chapter 5.9. For a large explosion, the radius of influence, \( r \), will be 20.0, and the effect, \( e \), will be 15.0. A medium explosion will have a radius of influence, \( r \), of 10.0, and an effect, \( e \), of 7.0. The radius of influence for a small explosion, \( r \), will be 6.0, and it will have an effect, \( e \), of 4.0. Finally, an extra small explosion will have a radius of influence, \( r \), of 4.0 and an effect, \( e \), of 3.0.

An example of two rows from an explosion script file follows:

```
30.55 19.29 10.51 20.00 15.00
30.97 88.17 19.88 6.00 4.00
```

Having a terrain with 257x257 elevation points and 4 m between the posts, this can be translated into English as follows: a bomb will be dropped at world x-, and z-coordinates (193.30,108.04), when 30.55 seconds have elapsed. When that bomb impacts with the terrain, it will affect vertices within a spherical radius of 20.00 m, with \( \frac{15d}{r} \), where \( d \) is the distance between the vertex of interests, and the location of impact. When 30.97 seconds have elapsed, another bomb will be dropped at position (906.4,204.4), and when it impacts, it will affect vertices within a radius of 6.0 with an effect of 4.0.

When creating a script file, the time between two explosions, will be a random value between 0.20, and 0.50 seconds. The x-, and z-positions, will be random values between 0.00 and 100.00%.

In each of the simulations, a grid spacing of 4.0 will be used. This means that the distance in the xz-plane, between all vertices in the original height map data, will be 4.0, for vertices having either the same x-, or z-value, or \( \sqrt{32} \), for vertices having different x-, and z-values. An illustration of the grid spacing used, can be seen in Figure 45.

![Figure 45: Illustration of the grid spacing used in the simulations.](image)

In all simulations, the camera moves at a velocity of 40 m/s, 40 m above terrain. The path used for the camera can be seen in Figure 46, and by using it, the camera will visit all parts of the terrain. The camera will have a vertical field of view, FOV, of 45 degrees, and a horizontal FOV of 60 degrees. All simulations will be carried out in a window of size 800x600. When each simulation is initialized, 500 dynamic bintrees will be created. If this amount is not enough, more will be allocated at run-time. In the simulations, a maximum of 17x17 vertices can be extracted from each b-spline object.
The elevation data that will be used is represented in the Binary Terrain, BT, file format Discoe (2002a), and the files that are used can be downloaded at Discoe (2002b). The BT files have been converted to BT format from USGS DEM, US Geological Survey Digital Elevation Model, files.

For each of the three terrain sets that have been used in the simulations, three different accuracy settings have been used, 1.0, 0.5, and 0.25. This approximately represents a pixel error of 4, 2, and 1, respectively. Hence, lowering the accuracy value yields a smaller pixel error, and a higher refinement level using more primitives. For each of these three values, five independent flyovers have been conducted. A set of variables have been recorded for each flyover, and an average have been calculated from the five simulations in concern. The average values, which have been calculated, are used to construct diagrams showing the frame rates and triangle counts. Diagrams have also been constructed showing the total frame time for a frame, divided by the amount of triangles rendered in that frame. These diagrams will show how the average time spent per triangle is affected during the simulations. The average file for a simulation setup is later used to calculate average values for each accuracy setting. These average values are used to construct diagrams, which are to be used for comparing the different terrain-, and accuracy settings.

6.2.2 Terrain 1: 257x257

A height map file with 513x513 entries was loaded, but only the lower left quarter 257x257 height points were used. The reason for loading a larger height map, but only using a part of it, is that no good terrain data was found in the needed size. The height map data represents the west half of Crater Lake, Oregon, and it have been converted from a USGS DEM file. The terrain was chosen, because it has varying geometry, with both flat and roughed regions.

In this data set, the first five frames have been discarded, since they were misleading, as the camera was positioned 1000 m beneath the terrain, due to the fact that the Crater Lake is not at sea level, where the camera was initially positioned. Also, as the camera used smooth terrain following, it took some five frames to adjust the camera according to the geometry. After the first five frames had elapsed, the camera position matched the terrain, and representative data could be collected.
Diagram 1: Diagram showing the frame rate and triangle count, when flying over a 257x257 terrain, at an accuracy of 1.0.

It can be seen in Diagram 1 that the frame rate varies according to the triangle count. When the triangle count is low, the frame rate is high, and when the triangle count is high the frame rate is low. The peaks or maximum values in the triangle count, occurs when the camera is traveling against the center of the terrain. The size of the maximum values on the triangle count curve tends to increase with time, and the size of the minimum values also tends to increase. This can be associated with the number of deformations that has occurred, since they also increase with time. This can be considered normal, as the maximum amount of triangles available has been increased, and also, the geometry where a deformation has occurred, does not tend to be flat. The overall frame rate does not seem to be affected by the deformations at these triangle counts.

Diagram 2: Diagram showing the average time spent per triangle, when using a terrain of size 257x257, and having an accuracy of 1.0.
In Diagram 2, the average time spent on each triangle can be seen. The peaks in the diagram represent a very low triangle count. This since there is always a certain amount of time spent on culling the terrain. Also, the type of vertex array rendering, which have been implemented, comes with a little bit overhead if used on small sets of triangles. The overall shape of the curve in Diagram 2 does not seem to change over time.

![Diagram 2 showing the average time spent per triangle.](image)

Diagram 3: Diagram showing the frame rate, and triangle count, when flying over a 257x257 terrain, with an accuracy of 0.5.

In Diagram 3, similar results as in Diagram 1 can be observed. The main difference is the peak values of the triangle count, which has increased. This can be though of as normal, as a lower accuracy setting means allowing smaller pixel errors, and to meet this criterion more triangles are needed. The peak values for the triangle count, tends to increase with time, as in Diagram 1, but the increase in the peak values also seems to be of the same sizes as before.

![Diagram 3 showing the frame rate and triangle count.](image)

Diagram 4: Diagram showing the average time spent per triangle, when simulating on a terrain of size 257x257, and using an accuracy of 0.5.
In Diagram 4, the average time spent per triangle in the 257x257 terrain, with an accuracy of 0.5, can be seen. The curve looks similar to that in Diagram 2, with the difference that the peaks seem to be lower. This can be explained by the fact that more triangles are needed to meet the lower accuracy. Hence the total time is divided by a higher amount of triangles. The overall shape of the curve does not seem change with time.

Diagram 5: Diagram showing the frame rate, and triangle count, when flying over a 257x257 terrain, using an accuracy of 0.25.

The final accuracy setting for the 257x257 terrain is 0.25, and Diagram 5 shows the average frame rate, and triangle counts for that accuracy setting. Again, there is similarity between the diagrams. Diagram 5 looks like both Diagram 1, and Diagram 3. The peak values for the triangle count are now reaching even higher, and the minimum values for the FPS curve, follows the peak values for the triangle curve. The three diagrams looks very similar; they are only centered at different average FPS values.

Diagram 6: Diagram illustrating the average time spent on each triangle, on a terrain of size 257x257, and with an accuracy of 0.25.
Diagram 6, shows the average time spent per triangle, in the terrain of size 257x257, and with an accuracy of 0.25. Again, the overall shape of the curve does not seem to be affected over time. The difference from the previous average time per triangle diagram, Diagram 4, is that the peak values seem to be lowered even further. This can again be associated with a higher average of triangles.

<table>
<thead>
<tr>
<th>257x257 Heightmap</th>
<th>Accuracy 1.0</th>
<th></th>
<th>Accuracy 0.5</th>
<th></th>
<th>Accuracy 0.25</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of frames</td>
<td>39743</td>
<td>1</td>
<td>35701</td>
<td>1</td>
<td>31491</td>
<td>1</td>
</tr>
<tr>
<td>Total time / Average FPS</td>
<td>245.660233</td>
<td>161.8 FPS</td>
<td>245.650391</td>
<td>145.3 FPS</td>
<td>245.6698</td>
<td>128.2 FPS</td>
</tr>
<tr>
<td>Rendering</td>
<td>9.002339</td>
<td>0.000227</td>
<td>12.97518</td>
<td>0.000363</td>
<td>17.961876</td>
<td>0.00057</td>
</tr>
<tr>
<td>Triangulating</td>
<td>41.96373</td>
<td>0.001056</td>
<td>70.046282</td>
<td>0.001962</td>
<td>106.705719</td>
<td>0.003389</td>
</tr>
<tr>
<td>Deforming</td>
<td>0.342772</td>
<td>0.000009</td>
<td>0.353418</td>
<td>0.000001</td>
<td>0.394162</td>
<td>0.000013</td>
</tr>
<tr>
<td>Number of triangles</td>
<td>85499112</td>
<td>2151</td>
<td>127690768</td>
<td>3576</td>
<td>176063280</td>
<td>5590</td>
</tr>
<tr>
<td>Culling &amp; world error update</td>
<td>33.51239</td>
<td>0.000843</td>
<td>30.103483</td>
<td>0.000843</td>
<td>26.426544</td>
<td>0.000839</td>
</tr>
<tr>
<td>Number of deformations</td>
<td>580</td>
<td>580</td>
<td>580</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Max dynamic bintrees used at once | 245 | 614 | 1375 |
| Number of allocated dynamic cells | 10887 | 10887 | 10887 |
| Recorded memory usage after init ~ | 50.3 MB | 50.3 MB | 50.3 MB |
| Calculated memory usage after init ~ | 49.4 MB | 49.4 MB | 49.4 MB |

Table 2: Table showing the average values calculated for the simulations in the 257x257 terrain.

In Table 1, the average set with frame data have been summed together, and also a per frame average have been calculated. The total number of dynamic bintrees in use at once, increases with the accuracy level, which is a perfectly normal behavior when using a LOD algorithm. After the terrain has been initialized, the amount of RAM consumed by the algorithm, in the 257x257 setting, is approximately 50 Mb, compared to the calculated value of 49 MB. At run-time, approximately 580 deformations have occurred, which has resulted in 10887 new dynamic patches. The assumption that not all dynamic patches needs to have bintrees allocated at once, holds, as the maximum number of dynamic bintrees in use is 1375, in the 0.25 accuracy setting. At most, 12.63% of the dynamic regions need to have a bintree allocated at run time. The average frame rate is 162 FPS, for an average of 2151 triangles at an accuracy of 1.0, 145 FPS, for an average of 3576 triangles at an accuracy of 0.5, and 128 FPS, for an average of 5590 triangles at an accuracy of 0.25.
Diagram 7: Diagram showing how the average values for different variables, changes according to the accuracy level, in a terrain of size 257x257.

A final diagram has been constructed for the 257x257 terrain, Diagram 7, and it illustrates how the average values for the simulations are associated. It can be seen in the diagram, that the view frustum culling and world error updating, is not dependant on the accuracy level. It can also be seen that the time spent on rendering, is only slightly increasing with a lowered accuracy setting. The amount of triangles is affected most by a change in accuracy, and for the time variables, the time to triangulate the terrain, is the variable that is most affected. The time used for deforming the terrain, does not show up on the diagrams, as 580 deformations divided by 30-40 thousand frames is not a very large value.

6.2.3 Terrain 2: 513x513

The same height map file as in chapter 6.2.2, Crater Lake, Oregon, have been loaded, but now the complete set of elevation points is used. The diagrams in this chapter only show the first 63000 frames of the data set recorded. The first five frames were discarded in these simulations as well, as the Crater Lake is still not at sea level where the camera was initially positioned.
Diagram 8: Diagram showing the frame rate, and the triangle count, when flying over a 513x513 terrain, using an accuracy of 1.0.

In Diagram 8, it can be seen that the frame rate is tightly coupled with the amount of triangles rendered. This was also the case for the diagrams in chapter 6.2.2. The FPS curve does not seem to be affected by the deformations.

Diagram 9: Diagram showing the average time spent per triangle, when simulating in a terrain of size 513x513, and having an accuracy of 1.0.

Diagram 9, shows the average time spent per triangle in the 513x513 terrain, having an accuracy of 1.0. As in chapter 6.2.2, the overall shape of the curve does not seem to change over time. The peaks occur at very low triangle counts, as in the average triangle diagrams in chapter 6.2.2, and at the highest peak, the triangle count almost reaches zero. This can again be explained by the fact that the view frustum culling, and rendering, always uses some frame time, even if there are very few triangles to be rendered.
Diagram 10: Diagram showing the frame rate, and the triangle count, when flying over a 513x513 terrain, with an accuracy of 0.5.

Again it can be seen in Diagram 10, that a lower accuracy level moves the FPS curve downwards, and that the minimum values for the FPS curve follows the peak values for the triangle count curve, which tends to increase with a lower accuracy level. The overall shape of the FPS curve, does not seem to be affected by the deformations, but it does however breach the 75 FPS limit at some point around 12-14 thousand triangles.

Diagram 11: Diagram showing the average time spent per triangle, in a terrain of size 513x513, and with an accuracy of 0.5.

Similar results, as in Diagram 9, can be observed in Diagram 11, which shows the average time spent per triangle in the 513x513 terrain, with an accuracy of 0.5. As in chapter 6.2.2, the overall average time spent per triangle does not seem to change with time. The peaks however, have been lowered compared to those in Diagram 9. As in
the previous chapter, this can be explained by the increase in the amount of triangles due to a lower accuracy value, for which more triangles are needed.

Diagram 12: Diagram showing the frame rate, and triangle count, when flying over a 513x513 terrain, using an accuracy of 0.25.

The diagram constructed when having an accuracy of 0.25 in the 513x513 terrain, can be seen in Diagram 12. In this diagram, it can be observed that the 75 FPS line is broken more repeatedly. The diagram looks like both Diagram 8, and Diagram 10. The differences between the three diagrams, is that the offset for the FPS curve gets lower with a lower accuracy level. Also, the peak values get higher for the triangle count, which results in even lower minimum values for the FPS curve. It can be seen in the diagram, that the 75 FPS limit is broken around 10-15 thousand triangles. This is also true for the previous diagrams, both in this and the previous chapter.

Diagram 13: Diagram showing the average time spent per triangle, when simulating in a terrain of size 513x513, and with an accuracy of 0.25.

Diagram 13, illustrates the average time spend per triangle, in the terrain of size 513x513, and when using an accuracy of 0.25. As earlier, the average time per
triangle does not seem to change over time. The peak values are representing very low triangle counts, and as for the earlier diagrams, this can be associated with the time spent on culling and rendering.

<table>
<thead>
<tr>
<th>513x513 Heightmap</th>
<th>Accuracy 1.0</th>
<th>Average Per frame</th>
<th>Accuracy 0.5</th>
<th>Average Per frame</th>
<th>Accuracy 0.25</th>
<th>Average Per frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of frames</td>
<td>70277</td>
<td>1</td>
<td>58583</td>
<td>1</td>
<td>44944</td>
<td>1</td>
</tr>
<tr>
<td>Total time / Average FPS</td>
<td>490.413025</td>
<td>143.3 FPS</td>
<td>490.348389</td>
<td>119.5 MB</td>
<td>490.271454</td>
<td>91.7 FPS</td>
</tr>
<tr>
<td>Rendering</td>
<td>31.763988</td>
<td>0.000452</td>
<td>40.43309</td>
<td>0.00069</td>
<td>47.438198</td>
<td>0.001055</td>
</tr>
<tr>
<td>Triangulating</td>
<td>151.93692</td>
<td>0.002162</td>
<td>229.779648</td>
<td>0.003922</td>
<td>298.9757</td>
<td>0.006652</td>
</tr>
<tr>
<td>Deforming</td>
<td>0.659546</td>
<td>0.000009</td>
<td>0.670216</td>
<td>0.000111</td>
<td>0.696527</td>
<td>0.00015</td>
</tr>
<tr>
<td>Number of triangles</td>
<td>283672224</td>
<td>4036</td>
<td>399616384</td>
<td>6821</td>
<td>481374400</td>
<td>10710</td>
</tr>
<tr>
<td>Culling &amp; world error update</td>
<td>129.4422</td>
<td>0.001842</td>
<td>105.33165</td>
<td>0.001798</td>
<td>78.474434</td>
<td>0.001746</td>
</tr>
<tr>
<td>Number of deformations</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>Max dynamic bintrees used at once</td>
<td>144</td>
<td>418</td>
<td>1324</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of allocated dynamic cells</td>
<td>22489</td>
<td>22489</td>
<td>22489</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Recorded memory usage after init ~</td>
<td>104.5 MB</td>
<td>104.5 MB</td>
<td>104.5 MB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Calculated memory usage after init ~</td>
<td>101.4 MB</td>
<td>101.4 MB</td>
<td>101.4 MB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: A table showing the average values calculated for the simulations in the 513x513 terrain.

The average set of frame data have been summed together, and Table 3, shows the values calculated. The total amount of RAM consumed after initialization is around 104 Mb, compared to a calculated value of 101 MB. During run-time, 1000 deformations have occurred, which have resulted in 22489 new dynamic high resolution patches. At runtime, the highest number of simultaneously requested dynamic bintrees is 1324. This yields that only 6% of the high resolution patches needs to have bintrees associated at one moment. The average frame rate is 143 FPS, for an average of 4036 triangles, at an accuracy of 1.0. For an accuracy of 0.5, the average frame rate is 120 FPS, for an average of 6821 triangles, and for an average of 10710 triangles, the average FPS is 92 at an accuracy of 0.25. Diagram 14, illustrates how the average variables changes according to different accuracy levels. As for Diagram 7, in chapter 6.2.2, the time spent on culling the terrain, is not affected by a change of accuracy. The time to render the terrain, is also like before, only slightly increasing with a lowered accuracy. In Diagram 14, it becomes more obvious that the time to triangulate the terrain is the time which is most affected by a change in accuracy.
Diagram 14: Diagram showing how the average of different variables changes according to accuracy level, in a terrain of size 513x513.

6.2.4 Terrain 3: 1025x1025
A height map file containing 1025x1025 elevation points have been used. It represents the Island of Hawaii. The height map is converted from a USGS 30m DEM file. In all diagrams constructed for the 1025x1025 terrain, the data values beyond frame 63000 are not appearing in the diagrams.

Diagram 15: Diagram showing the frame rate, and the triangle count, when flying over a 1025x1025 terrain, using a accuracy of 1.0.

Diagram 15, shows some peculiar results. The frame rate seems to constantly be jumping between a high-, and low frame rate, even if the amount of triangles is unchanged.
Diagram 16: Diagram showing the average time spent per triangle when simulating in a terrain of size 1025x1025, with an accuracy of 1.0.

In Diagram 16, the average time spent per triangle for the 1025x1025 terrain, at an accuracy of 1.0, can be seen. The diagram shows the same results as Diagram 15, the recorded data seems to be jumping between high- and low values.

Diagram 17: Diagram showing the frame rate, and the triangle count, when flying over a 1025x1025 terrain, with an accuracy of 0.5.

Diagram 17, shows similar results as in Diagram 15, and during run-time the operating system seems to be paging a lot of information to disk. In chapters 6.2.2, and 6.2.3, it could be observed, that the frame rate was centered on different average FPS values. This fact is not as easily detected in the diagrams for the 1025x1025 terrain, as the constantly changing FPS curve makes it very hard to conclude anything from the diagrams.
Diagram 18: Diagram showing the average time spent per each triangle, when simulating in a terrain of size 1025x1025, and with an accuracy of 0.5.

Diagram 18 shows the average time spent per triangle with an accuracy of 0.5, in the 1025x1025 terrain. Similar signs as before can be observed, the data is constantly jumping. Also, when compared to Diagram 16, the peaks do not seem to match, as they did for the 257x257, and 513x513 terrains. The peaks seem to be sporadic.

Diagram 19: Diagram showing the frame rate, and the triangle count, when flying over a 1025x1025 terrain, using an accuracy of 0.25.

Yet again, Diagram 19 shows the same signs of unstableness for the triangle counts and frame rates. This can also be observed in Diagram 20, for the average time spent per triangle diagram. If looking beyond the constantly jumpy data values, some signs in the average time per triangle diagram, shows that the peaks tend to occur when having a very low triangle count. This is also true for the triangle count and frame rate diagrams in this chapter, if looking beyond the jumpy data, the overall frame rate curve seems to be coupled with the triangle count.
For the 1025x1025 terrain, an approximate 321 Mb of RAM was consumed, which can be compared to a calculated value of 309 MB. The test environment only consists of 256Mb of physical RAM, and of this, the Windows operating system is consuming 80Mb. This concludes that the application has to rely on virtual RAM, which resides on the much slower hard drive medium. This can explain the very unstable results achieved when simulating on the 1025x1025 terrain. In the 1025x1025 terrain, an average FPS of 98 was recorded for an average of 3742 triangles, at an accuracy of 1.0. For an accuracy of 0.5, the average frame rate was 75 FPS for an average of 6499 triangles. The average frame rate for an accuracy of 0.25 was 49 FPS, for an average of 9376 triangles.
Diagram 21: Diagram showing how the average values for different variables, changes according to the accuracy level, in a terrain of size 1025x1025.

Diagram 21, shows how the different average calculated variables change according to a change in accuracy. As in chapters 6.2.2, and 6.2.3, the time spent on culling, does only seem to be slightly increasing when using a lower accuracy setting. However, the time to render seems to be more affected in the 1025x1025 terrain. Also of the total frame time, the time to triangulate the terrain is the time most affected by a change in accuracy.

6.2.5 Timing results summary

In this chapter, the observations concerning the previously presented data will be summarized. In Table 5, the average triangle counts, and average frame rates, at a given accuracy, for the different terrains, are listed. An observation from this table is that the terrain size of 1025x1025 is clearly dependant on something besides the triangle count. It can also be seen that to render approximately 10 thousand triangles, an average frame rate of 92 FPS, is given for the 513x513 terrain. This can be compare to an average frame rate of 49 FPS for the 1025x1025 terrain. Looking at the differences between the 257x257, and 513x513 terrains, gives that to display an average of 3600 triangles, an average frame rate of 145 FPS, and 143 FPS, is needed, respectively.
<table>
<thead>
<tr>
<th>Accuracy 1.0</th>
<th>Terrain size</th>
<th>Average triangle count</th>
<th>Average FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>257x257</td>
<td>2151</td>
<td>161.8</td>
<td></td>
</tr>
<tr>
<td>513x513</td>
<td>4036</td>
<td>143.3</td>
<td></td>
</tr>
<tr>
<td>1025x1025</td>
<td>3742</td>
<td>97.8</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Accuracy 0.5</th>
<th>Terrain size</th>
<th>Average triangle count</th>
<th>Average FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>257x257</td>
<td>3576</td>
<td>145.3</td>
<td></td>
</tr>
<tr>
<td>513x513</td>
<td>6821</td>
<td>119.5</td>
<td></td>
</tr>
<tr>
<td>1025x1025</td>
<td>6499</td>
<td>75.3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Accuracy 0.25</th>
<th>Terrain size</th>
<th>Average triangle count</th>
<th>Average FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>257x257</td>
<td>5590</td>
<td>128.2</td>
<td></td>
</tr>
<tr>
<td>513x513</td>
<td>10710</td>
<td>91.7</td>
<td></td>
</tr>
<tr>
<td>1025x1025</td>
<td>9376</td>
<td>48.7</td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Table showing a comparison between the average frame rate and average triangle count, between the three terrain sizes at the different accuracy settings.

The average values of Table 5 have been used to construct a diagram, Diagram 22, which illustrates that the 257x257, and 513x513 terrains, has the same dependency concerning the triangle count. The values for the 1025x1025 terrain, does not match the values of the former two terrain sizes.

![Diagram 22: Diagram showing the average frame rate over the average triangle count, for the different terrain sizes.](image)

During the simulations, it has been observed, but not recorded in any of the collected data, that if a high number of dynamic patches needs to be attached in the same frame, then the frame rate will drop to a very low value for a short moment. Even though this is not presented in any of the collected data, it is an important observation made about the simulations, since it affects the overall usability of the algorithm.

To summarize the collected data, it can be said that the frame rate is tightly coupled with the triangle count, and the triangle count seems to increase as more deformations have occurred. This gives a slightly lowered frame rate, but besides the effect of having more triangles, the frame rate does not seem to be changed by having deformations and increased resolution. It should however be noted, that if a lot of dynamic resolution needs to be attached during the same frame, the frame rate drops for a couple of frames. The average time spent per triangle, increases with fewer
triangles. This can be associated with the amount of time spent on culling and rendering the terrain, as the time spent on performing these two tasks, have been found to be almost constant and independent on the amount of triangles rendered. The overall observation for the average time spent per triangle, is that it is not increasing with time, and hence not particularly affected by the deformations and increased resolution.

The data collected during simulations on the 1025x1025 terrain, was constantly jumping between high-, and low values. This can be explained by the fact that the data structures for the terrain system could not fit in physical RAM, but had to rely on virtual RAM, which resides on a much slower medium.
7 Conclusion

Imagine a game environment, where a missile is launched upon the hill up front. When the missile impacts on the hill an explosion occurs, resulting in a crater. This could increase the realism of the game, compared to not having a crater as a result of the explosion. The goal of this final year project was to investigate how terrain rendering could be performed which had support for deformations on the geometry of the terrain.

The requirements were an algorithm that could visualize a terrain as large as those found in games today. The geometry of the terrain should be deformable everywhere, and a deformed region should stay deformed throughout the session. It has also been found out that the resolution at deformed regions might need to be increased to cater for all details associated with a deformation such as an explosion. The final catch was that the frame rate must not drop too low, as this could decrease the interactivity with the game, which is one of the fundamental aspects of game play.

An algorithm was found for increasing the resolution in a dynamic LOD algorithm, the DEXTER algorithm, and it maps on top of the ROAM algorithm, which is a terrain rendering algorithm that have become very popular amongst game developers. The DEXTER algorithm statically extends the refinement structures of the ROAM algorithm, when a higher resolution is needed in a region in order to describe the details. Memory calculations have been conducted on the DEXTER algorithm, and they have shown that the memory consumption for the algorithm becomes very high if many high resolution cells are introduced. This may pose a problem if used in a game environment, and it was stated that the memory consumption needs to be lowered in order for the algorithm to be feasible in a game environment.

The hypothesis stated that the memory consumption of the DEXTER algorithm could be lowered by using parametric surfaces, such as b-spline surfaces, and by only having the triangles describing the higher resolution, allocated at need.

An extended version of the ROAM algorithm, has been presented, SEROAM, and it follows the layout of the DEXTER algorithm, besides that the geometry in a high resolution cell, is described by a b-spline surface instead of an array of elevation points. The second aspect of the DEXTER algorithm, which have been changed, is that the triangles describing the higher resolution, is not created to be a static part of the terrain structure. Blocks of triangles are attached to high resolution cells on a per need basis, and sets of such triangle blocks are kept in a manager.

Memory calculations on the SEROAM algorithm have shown that the absolute worst-case scenario remains the same, but the prerequisites for it to happen has changed. In the DEXTER algorithm, a worst-case scenario would occur if high resolution cells have been introduced all over the terrain. This has changed in the SEROAM algorithm, and in order for a worst-case scenario to occur, there are two conditions that need to be fulfilled. First, high resolution cells need to be introduced all over the terrain, as in the DEXTER algorithm, and secondly, all high resolution cells needs to be attached to the refinement process simultaneously. If the second condition is true; all the finest level of detail is needed everywhere, then all LOD algorithms, both for terrain rendering and other areas of interest, would be ineffective, as their purpose is not to use the finest level of detail everywhere. It would be much faster to simply use a brute force algorithm, and send all primitives of the finest resolution directly to the graphics hardware, without any time spent on detail reduction.
In the SEROAM algorithm, the worst-case scenario can also be limited. The refinement functionality is defined in such a way that it does not try to use any dynamic resolution which is not currently attached, even if it exists and could be needed. Also, the attachment functionality is defined in such a way that a maximum bound can be established on the maximum simultaneously attached high resolution cells. Then, if a high resolution cell needs to have its structures allocated, but there are no more free dynamic structures, then it would have to wait until a dynamic structure is released by some other high resolution cell. This functionality could pose some loss in visual appearance, but to the gain of lower memory usage.

Testing has been conducted on the algorithm, in order to estimate if the timing requirements could be kept. Three different sets of height maps were used during all simulations, and they were of sizes: 257x257, 513x513, and 1025x1025. In each of these terrain sets, three different accuracy settings have been used, which corresponds to allowing three different sizes of pixel errors to occur when mapped to screen space. Akenine-Möller & Haines (2002), states that changes in the frame rates above 72 FPS are effectively in detectable, and thus, a comparison limit of 75 FPS was chosen. In the smaller two of the terrains that were tested, the algorithm proved to work with triangle counts up to 10-15 thousand triangles. The creation, and attachment of bintrees to the dynamic regions, did not seem to affect the overall performance of the algorithm, as the average time spent per triangle was unchanged. However, if there was a high amount of dynamic bintrees attached in a single frame, then the frame rate would suffer. The amount of triangles increased as more deformations and high resolution terrain was introduced, but that should be considered normal, as the total amount of triangles available increases when extending the resolution, and also, the geometry of a deformed region does not tend to be flat, hence more triangles are needed there.

The final terrain that was tested, the 1025x1025, gave results that were not as good as the previous two terrain sets. The frame rate was constantly jumping between a high-, and low frame rate. This was related to the memory consumed by the algorithm. It have been recorded that at the 1025x1025 setting, 320 Mb of RAM was needed by the algorithm. The test environment only consisted of 256 Mb, of which the windows operating system consumed 80 Mb. This leaves 170 Mb, which is not enough, and virtual RAM has to be used. The virtual RAM resides on a hard drive, and the results of having to page memory in and out from disc made the frame rate go jumpy. It can be concluded that the algorithm only performs well, if its structures can fit in physical RAM.

During all simulations, it was also recorded that only about 10% of the high resolution cells introduced, needs to have triangles attached. This strengthens the statement that the memory requirements could be lowered by using dynamic bintrees.

In the computer game Battlefield 1942 (2002), terrains of the size 1025x1025, is used in some of the campaigns. Now, if the computer on which the game is to run only contains 256 MB RAM, then the SEROAM algorithm would clearly not be a choice, as it consumes more than 320 MB of RAM in a terrain of that size. It is not acceptable for the terrain rendering part of a game environment to consume the complete amount of available RAM, as textures, models, etc. also needs a share amount of RAM. Another important observation is that if an approximation mesh of 10 thousand triangles is needed, then the SEROAM algorithm also consumes too many CPU cycles per frame. Other aspects of a game environment, such as artificial intelligence, collision detection, and particle effects, also needs to be assigned processing time.
To conclude this final year project, it can be stated that the SEROAM algorithm, is not feasible to use in a game environment of today. The memory has in fact been lowered, compared to the DEXTER algorithm, but the memory consumption is still too high to be used in a game. The algorithm could however be useful in another environment, where the amount of physical RAM, and processing cycles consumed, is not a problem, such as in an environmental simulation or similar.

Finally, a set of images will show the effects of not increasing the resolution, compared to increasing the resolution, at a region with deformations of equal sizes. Figure 47, illustrates the geometry before any deformation has occurred. Figure 48, illustrates the same location, but now a deformation of size $x$, has been conducted, and the resolution has not been extended. Finally, Figure 49 illustrates the same deformation of size $x$, at the same location, but now the resolution have been extended.

Figure 47: Illustration of how the geometry looks before any deformations have been conducted.
Figure 48: Illustration of how Figure 47 looks after deformation without extending the resolution.

Figure 49: Illustration of how Figure 47 looks after deformation, with increased resolution.
8 Discussion

It has been found during the time working with the algorithm, that most of the time spent on triangulating the terrain, can be associated with the queue updating, where all elements needs to have their priorities recalculated. Blow (2000), stated that the ROAM algorithm does not perform well on high granularity terrains, due to its queue updating. Blow (2000), presented an alternate split/merge metric to the ROAM algorithm, and he stated that the queue update problem was avoided by using it.

The ROAM algorithm is a top-down algorithm, meaning that it refines the terrain from the top and down. In all top-down algorithms, the refinement can be stopped at any moment, if the available frame time has elapsed. This is an important aspect in a game environment, since the time required by other aspects, tends to vary with time, and at some moments when there is much action, the rendering needs to be given less time.

If considering the deformation capabilities of the SEROAM algorithm, the results obtained in this final year project, can be considered positive. The time to update the height faults when introducing a deformation can be kept low, as it is only the static structures that need to be updated recursively. The updating of the dynamic structures is confined to each single object that have been touched. The means of describing high resolution, by using b-splines, keeps the amount of data that has to be processed in association with a deformation, at a minimum. Still, an infinite amount of primitives can be extracted from the b-spline. By using curved surfaces to describe the higher resolution, a lot of time and space is saved concerning deformations, compared to describing the higher resolution with a height map.

There are however some limitations when using the SEROAM algorithm, compared to using the DEXTER algorithm. In the DEXTER algorithm, several terrain cells may overlap, and one high resolution cell may be covered by an even higher resolution cell. This could be useful if first the resolution at a region needs to be extended so some degree, too later be extended more in a sub-region of the first region. In the SEROAM algorithm, only one level of extension is allowed, as the dynamic regions is not a static part of the refinement structures. There is also some computational overhead when using the SEROAM algorithm, compared to using the DEXTER algorithm. This as every time a high resolution cell needs to be attached in the SEROAM algorithm, its vertices needs to be calculated, and if many high resolution cells needs to be attached in the same frame, then this would affect the overall frame rate. This problem is not present in the DEXTER algorithm, as the high resolution cells are a static extension of the refinement structures.

In order to compare the DEXTER, and SEROAM algorithms, the usability of the SEROAM algorithm, in the same context as the DEXTER algorithm was developed, should be investigated. The focus of the work conducted by He (2000), was to introduce tire-tracks in an off-road driving simulation. It was stated that in order to take account for the small sized tire-tracks, the resolution needed to be extended. Then a height map was used to describe the newly introduced tire-tracks. In the SEROAM algorithm, the height-map is exchanged with a b-spline surface. If only one tire-track exists in every high resolution cell, than only a few control points is needed to describe the tire-track, compared to the amount of evenly spaced height points needed. The control points are inserted surrounding the tire-track, in order to make the surface describe the tire-tracks. If several tire-tracks are needed in the same high-resolution cell, then the same level of detail can be achieved by introducing as many
control points, as there would be elevation points in the height map. The difference is that many more different primitives can be extracted from the b-spline surface, than can be extracted from the height map, resulting in a more continuous terrain.

As stated in chapter 2.6, there are at least four aspects of a curved surface, which are better than for a height map:

1. The representation is more compact
2. The geometry becomes scalable.
3. The primitives created, can be smoother and more continuous.
4. Animation and collision detection may become faster.

There is one final aspect of using curved surfaces that needs to be given some thought, overhangs can be achieved. For example, picture a bomb exploding next to a steep slope, then a hole could be created which digs right into the side of the slope, resulting in an overhang. A regular height map has its points evenly distributed in the horizontal plane. The elevation points may not overlap, and they must be monotonic. Hence height maps cannot visualize such an overhang. Now, consider curved surfaces, and more precise, a b-spline surface. A b-spline surface is described by a set of control points, which pulls the surface towards them. The control points may be located everywhere, as long as the knot vector describing their range of influence is monotonic. Hence, overhangs can be visualized with curved surfaces. The terrain can be formed in any way you see fit.

In the SEROAM algorithm, the control points have been defined as being evenly distributed, this can however be changed very easily, so that they can be located at any position in world space. By having them not evenly distributed, small overhangs can be created. The size of the overhangs is of course delimited by the overlaying evenly spaced height map, but their maximum size can be increased if the overlaying grid space is increased.

To conclude this discussion, it should be mentioned, that if the DEXTER algorithm on top of the ROAM algorithm, is incorporated in a terrain of size 1025x1025, and that dynamic cells are introduced at every possible location, then the algorithm would probably not be functional, as the maximum address space would be exceeded. If using the SEROAM algorithm, in a terrain of size 1025x1025, and creating high resolution cells at every possible location, the algorithm could still be functional, if a highest amount of available dynamic bintrees is defined for which the maximum address space is not exceeded. This clearly shows that the memory requirements have been lowered by using b-splines and dynamic bintree allocations, but to some possible loss in visual appearance.
9 Future work

It would be interesting to investigate if the approach used in this project, could be applied to some other terrain rendering algorithm, besides ROAM, and more precisely to investigating the usability in a terrain rendering algorithm, which is not as memory-, and CPU intensive.

Another aspect, which could be interesting to investigate, is if the performance of the algorithm can be improved by using functionality in the underlying graphics API, to tessellate the high resolution b-splines surfaces.

It could also be interesting to investigate if another activation criterion for attaching dynamic high resolution patches could lower the risk of having a drop in frame rate due to several simultaneous attachments in the same frame. This could be deployed by for example having an activation radius, to decide when to attach dynamic structures to the high resolution patches.
References


Battlefield 1942 (2002) [Computer program], Digital Illusions CE.


Counter-Strike (2000) [Computer program], Valve.


Wolfenstein 3D (1992) [Computer program], id Software, available on the Internet: ftp://ftp.sunet.se/pub/pc/idgames2/levels/v-z/wolf3d.zip [Fetched 03.06.03].
Appendix A – Source code

In this appendix, the source code from the implementation which is the results of this final year project is found. It is not the complete set of source code, used when simulating, instead, only the structures used by the terrain sub-system, is given.

BinTreeManager.h

```c
#ifndef BINTREEMANAGER_H
#define BINTREEMANAGER_H
#include "Triangle.h"
class DynamicBinTree {
public:
 DynamicBinTree(int resolution, int id);
 DynamicBinTree();

 //Get the number of elements which is active for rendering in this bintree
 void GetNumRenderElements(unsigned int re) { numRenderElements = re; }
 //Get this dynamic bintrees busy flag
 void GetBusy(bool b) { busy = b; }
 //Get the patch link, so that triangles in the bintree is attached
 void GetPatch(Patch* lPatch) { patchLink = lPatch; }
 //Check if this dynamic bintree is busy
 bool IsBusy(void) { return busy; }
 //Get the ID for this dynamic bintree
 int GetID(void) { return id; }
 //Get the resolution of this dynamic bintree
 int GetResolution(void) { return resolution; }
 //Get the total number of triangles+2, which this bintree contains
 int GetNumAllocatedTris(void) { return numAllocatedTris; }
 //Get the depth of the bintree structures
 int GetTreeDepth(void) { return treeDepth; }
 //Get the width of this dynamic bintree (resolution+1)
 int GetWidth(void) { return width; }
 //Get the render_list for this dynamic bintree
 unsigned int* GetRenderList(void) { return render_list; }
 //Get the index_list for this dynamic bintree
 unsigned int* GetIndexList(void) { return index_list; }
 //Fetch the bintree structure
 Triangle** GetBinTree(void) { return binTree; }
 //Fetch the vertex list
 VertexExt* GetVertexList(void) { return vertex_list; }
 //Get one specified vertex from the vertex list
 VertexExt* GetVertex(int vID) { return &vertex_list[vID]; }
private:
 //Recursively create child triangles for a given triangle
```
void RecursiveCreateSplit(Triangle *triangle, int thisLevel, int stopLevel);
//Create two child triangles for a given triangle
void CreateSplit(Triangle* triangle);

int id, resolution, treeDepth, numAllocatedTris, width; //Self explanatory
bool busy; //Is tree busy flag
Triangle** binTree; //Array containing all triangles
VertexExt* vertex_list; //A list of vertices for this tree
Patch* patchLink; //Link to which patch the tree is attached to
//The current rendering list, contains triplets of vertices to be sent to
//the graphics API for rendering
unsigned int* render_list; //Back link, which triangles are contained in the rendering list
unsigned int* index_list; //How many triplets of vertices are there to render
unsigned int numRenderElements;

class BinTreeManager {
public:
  BinTreeManager(int in_numTrees, int in_resolution);
  ~BinTreeManager();
  //Fetch a dynamic bintree structure from pool
  DynamicBinTree* GetFreeBinTree(void);
  //Release a dynamic bintree structure
  void ReleaseBinTree(DynamicBinTree* releaseTree);
  //Get the number used bintrees
  int GetNumTrees(void) { return numTrees; }
  //Get the maximum simultaneously used bintrees
  int GetTopUsedTrees(void) { return topUsedTrees; }
private:
  int resolution, numTrees, usedTrees, topUsedTrees;
  DynamicBinTree** binTrees; //Array of dynamic bintrees
};

#ifndef BIN_TREE_MANAGER_H_INCLUDED
#endif

BinTreeManager.cpp
#include "math.h"
#include "CommonInclude.h"
#include "BinTreeManager.h"
#include "Helper.h"

DynamicBinTree::DynamicBinTree(int resolution, int id)
{}
  DynamicBinTree::id = id;
  DynamicBinTree::resolution = resolution;
  busy = false;
  patchLink = NULL;
  numRenderElements = 0;
  width = resolution+1;
  //Create render_list and index_list for vertex array rendering
  render_list = new unsigned int[(width-1)*(width-1)*2*3];
  index_list = new unsigned int[(width-1)*(width-1)*2];
  //Create vertex list
  vertex_list = new VertexExt[width*width];
  for(int j=0; j<width; j++) {
    for(int i=0; i<width; i++) {
      vertex_list[i+j*width] = VertexExt(i*1.0,0.0,j*1.0);
    }
  }
  //Calculate treedepth and total number of triangles needed
  treeDepth = (log((resolution)*(resolution))/log(2));
  numAllocatedTris = (2*resolution*resolution-1)*2+2;
  //Allocate storage for bintree structures
  binTree = new (Triangle*[numAllocatedTris]);
  //Create the two root triangles
  binTree[2] = new Triangle((width-1)*width,0,width*width-1,2,&patchLink);
  binTree[3] = new Triangle(width-1,width*width-1,0,3,&patchLink);
RecursiveCreateSplit(binTree[2], 0, treeDepth);
RecursiveCreateSplit(binTree[3], 0, treeDepth);

#ifdef MEMCOUNTER
// Add the total amount of memory allocated to global memory tracking
allocatedMem += sizeof(unsigned int) * (width-1) * (width-1) * 2 * 3;
allocatedMem += sizeof(unsigned int) * (width-1) * (width-1) * 2;
allocatedMem += sizeof(VertexExt) * width * width;
allocatedMem += sizeof(Triangle*) * numAllocatedTris;
allocatedMem += sizeof(Triangle) * 2;
#endif
}

DynamicBinTree::~DynamicBinTree()
{
    // Deallocate all triangles and the bintree structures
    if (binTree != NULL) {
        for (int i = 2; i < numAllocatedTris; i++) {
            if (binTree[i] != NULL) {
                delete binTree[i];
            }
        }
        delete[] binTree;
    }
    // Deallocate vertex_list, render_list, and index_list
    delete[] vertex_list;
    delete[] render_list;
    delete[] index_list;
}

void DynamicBinTree::RecursiveCreateSplit(Triangle *triangle, int thisLevel, int stopLevel)
{
    // Check if we have reached leaf triangle level
    if (thisLevel < stopLevel) {
        // This is not a leaf triangle, create children triangles
        CreateSplit(triangle);
        // Recursively split the children triangles
        RecursiveCreateSplit(binTree[triangle->GetID() * 2], thisLevel + 1, stopLevel);
        RecursiveCreateSplit(binTree[triangle->GetID() * 2 + 1], thisLevel + 1, stopLevel);
    }
}

void DynamicBinTree::CreateSplit(Triangle* triangle)
{
    // Two new triangles should be created, extract the existing vertices indexes
    int top = triangle->GetVertex(0);
    int right = triangle->GetVertex(1);
    int left = triangle->GetVertex(2);
    // Calculate coordinates for midpoint on the base of triangle
    int new_x = (int)((vertex_list[right].GetX() + vertex_list[left].GetX()) / 2);
    int new_z = (int)((vertex_list[right].GetZ() + vertex_list[left].GetZ()) / 2);
    if (new_x < 0) new_x = new_x * (-1);
    if (new_z < 0) new_z = new_z * (-1);
    // Calculate the index of the midpoint vertex
    int center = new_x + new_z * width;
    // Create the two new triangles
    binTree[triangle->GetID() * 2] = new Triangle(center, left, top,
            triangle->GetID() * 2, &patchLink);
    binTree[triangle->GetID() * 2 + 1] = new Triangle(center, top, right,
            triangle->GetID() * 2 + 1, &patchLink);
#endif MEMCOUNTER
    // Add the amount of allocated memory to the global memory tracker
    allocatedMem += sizeof(Triangle) * 2;
#endif
}

BinTreeManager::BinTreeManager(int in_numTrees, int in_resolution)
{
    // Create the given number of dynamic binary trees with given resolution
    // Add the amount of allocated memory to global memory tracker, if defined
    numTrees = in_numTrees;
    resolution = in_resolution;
    #ifdef MEMCOUNTER
    allocatedMem += sizeof(DynamicBinTree) * numTrees;
    #endif
}
binTrees = new DynamicBinTree*[numTrees];
for(int i=0;i<numTrees;i++) {
    binTrees[i] = new DynamicBinTree(resolution,i);
#endif MEMCOUNTER
    allocatedMem += sizeof(DynamicBinTree);
#endif}
usedTrees = 0; //Number of used trees
topUsedTrees = 0; //Maximum number of used trees simultaneously
}
BinTreeManager::~BinTreeManager() {
    //Free all dynamic binary trees
    if(binTrees!=NULL) {
        for(int i=0;i<numTrees;i++) {
            delete binTrees[i];
        }
        delete[] binTrees;
    }
}
DynamicBinTree* BinTreeManager::GetFreeBinTree(void) {
    //Return next free dynamic bintree
    for(int i=0;i<numTrees;i++) {
        if(!binTrees[i]->IsBusy()) {
            binTrees[i]->SetBusy(true);
            usedTrees++;
            if(usedTrees>topUsedTrees) topUsedTrees = usedTrees;
            return binTrees[i];
        }
    }
    //If the total number of dynamic binary trees is not enough,
    //more trees could be created here.
    return NULL;
}
void BinTreeManager::ReleaseBinTree(DynamicBinTree* releaseTree) {
    //A dynamic tree have been released, reset it
    releaseTree->SetBusy(false);
    releaseTree->SetNumRenderElements(0);
    usedTrees--;
}

Camera.h

/****************************************************************************
 * Camera.h                                                                    *
 * Author: Anders Dahlbom                                                     *
 * Last modified: 030603                                                        *
 *                                                                            *
 * Purpose: Class for a camera through which the world can be viewed upon.    *
 *                                                                            *
****************************************************************************/
#ifndef CAMERA_H
#define CAMERA_H
#include <windows.h>
#include <gl\gl.h>
#include "Matrix4f.h"
#include "Vertex.h"

class Camera {
public:
    Camera(GLdouble eyeX, GLdouble eyeY, GLdouble eyeZ, GLfloat xrot, 
        GLfloat yrot, GLfloat zrot);
    ~Camera();
    //Tell openGL to look through this camera
    void LookAt(void);
    //Rotate an amount of degrees around the X axis
    void RotateX(GLfloat degrees);
    //Rotate an amount of degrees around the Y axis
};

85
void RotateY(GLfloat degrees);
//Rotate an amount of degrees around the Z axis
void RotateZ(GLfloat degrees);
//Move forward with a given speed
void MoveForward(GLfloat speed);
//Move backwards with a given speed
void MoveBackward(GLfloat speed);
//Move left with a given speed
void MoveLeft(GLfloat speed);
//Move right with a given speed
void MoveRight(GLfloat speed);
//Move to a new location
void MoveTo(Vertex newEye);
//Rotate camera to look at a specific point
void RotateTo(Vertex newPoint);

//Update the viewing matrix
void UpdateViewMatrix(void);
//Check if a point is inside the frustum
bool PointInFrustum(float x, float y, float z);
//Check if a sphere is inside the frustum, intersecting equals inside
float SphereInFrustum(float x, float y, float z, float radius);
float SphereInFrustum(Vertex point, float radius);
//Intersection with remember flags
unsigned char SphereIntersectFrustumF(Vertex point, float radius);
unsigned char SphereInFrustumSqF(Vertex point, float sqradius, unsigned char &bFlags);
//Intersection with squared radius
int SphereIntersectFrustumSq(Vertex point, float radius);
//Intersection with squared radius and remember flags
unsigned char SphereIntersectFrustumSqF(Vertex point, float sqradius, unsigned char &bFlags);

Camera::Camera(GLdouble eyeX, GLdouble eyeY, GLdouble eyeZ, GLfloat xrot, GLfloat yrot, GLfloat zrot)
{
    eye.p[0] = eyeX;
    eye.p[1] = eyeY;
    eye.p[2] = eyeZ;
    xr = xrot;
    yr = yrot;
    zr = zrot;
}
Camera::~Camera()
{
}

void Camera::LookAt()
{
    // Update the viewing matrix
    UpdateViewMatrix();
    // Multiply our viewing matrix to the current OpenGL matrix
    glMultMatrixf(rm.dm);
    // Translate to eyes position
    glTranslatef(-eye.p[0], -eye.p[1], -eye.p[2]);
    // Extract the frustum
    ExtractFrustum();
    // Get the projection matrix from OpenGL
    glGetFloatv(GL_PROJECTION_MATRIX, pm.dm);
    // Calculate the transpose of the viewing matrix with the projection matrix
    rmt = rmtpm;
    rmt.Multiply(&pm);
    // The corresponding GLU call would be gluLookAt, and it would look like below
    // gluLookAt(eye.d[0], eye.d[1], eye.d[2], center.d[0],
    //           center.d[1], center.d[2], up.d[0], up.d[1], up.d[2]);
}

void Camera::RotateX(GLfloat degrees)
{
    xr += degrees;
}

void Camera::RotateY(GLfloat degrees)
{
    yr += degrees;
}

void Camera::RotateZ(GLfloat degrees)
{
    zr += degrees;
}

void Camera::MoveForward(GLfloat speed)
{
    eye = eye + relativecenter*speed;
}

void Camera::MoveBackward(GLfloat speed)
{
    eye = eye - relativecenter*speed;
}

void Camera::MoveLeft(GLfloat speed)
{
    // Calculate a vector to the left, dont move in Y
    Vertex left = Vertex(0.0, 0.0, 0.0);
    GLfloat xrot_xz = cos((xr)*PIOVER180) * 1.0;
    left.p[0] = sin((yr-90)*PIOVER180) * xrot_xz;
    left.p[1] = 0.0;
    // Move us
    eye = eye + left*speed;
}

void Camera::MoveRight(GLfloat speed)
{
    // Calculate a vector to the right, dont move in Y
    Vertex right = Vertex(0.0, 0.0, 0.0);
    GLfloat xrot_xz = cos((xr)*PIOVER180) * 1.0;
    right.p[0] = sin((yr+90)*PIOVER180) * xrot_xz;
    right.p[1] = 0.0;
    right.p[2] = cos((yr+90)*PIOVER180) * xrot_xz;
    // Move us
    eye = eye + right*speed;
}

void Camera::MoveTo(Vertex newEye)
{
    eye = newEye;
}
void Camera::RotateTo(Vertex newPoint)
{
    if (newPoint == eye) return;
    Vertex nVector = newPoint - eye;
    Vertex oldCenter = relativecenter;
    // to retrieve the rotation around y, kill the y component
    Vertex kyl = Vertex(oldCenter.p[0], 0.0, oldCenter.p[2]);
    Vertex ky2 = Vertex(oldCenter.p[0], 0.0, oldCenter.p[2]);
    float dy = (kyl.LengthOf() * ky2.LengthOf());
    float cosphi;
    if (dy != 0) cosphi = kyl.DotProduct(ky2) / dy;
    else cosphi = 1;
    Vertex oNormal = oldCenter.CrossProduct(up);
    float direction = 1.0;
    if (newCenter.DotProduct(oNormal) < 0)
        direction = -1.0;
    if (cosphi > 1.0)
        cosphi = 1.0;
    else if (cosphi < -1.0)
        cosphi = -1.0;
    float dyr = direction * acos(cosphi) / PIOVER180;
    // to retrieve the rotation around x, kill the x component
    Vertex kxl = Vertex(0.0, newCenter.p[1], 1.0);
    Vertex kx2 = Vertex(0.0, oldCenter.p[1], 1.0);
    float dx = (kxl.LengthOf() * kx2.LengthOf());
    if (dx != 0) {
        cosphi = kxl.DotProduct(kx2) / dx;
        direction = 1.0;
        if (newCenter.DotProduct(up) < 0) {
            direction = -1.0;
        }
    }
    else if (cosphi > 1.0) {
        cosphi = 1.0;
    } else if (cosphi < -1.0) {
        cosphi = -1.0;
    }
    dxr = direction * acos(cosphi) / PIOVER180;
    RotateY(dyr);  // Update the rotation values
    RotateX(dxr);
}

void Camera::UpdateViewMatrix(void)
{
    // Right-handed system
    // Keep the rotation values within specified ranges
    if (xr < -90.0f) xr = -90.0f;
    if (xr > 90.0f) xr = 90.0f;
    if (yr < 0.0f) yr = 360.0f + yr;
    if (yr > 360.0f) yr = yr - 360.0f;
    // Calculate at what relative point we are looking at
    GLdouble xrot_xz = cos(xr * PIOVER180) * 1.0;
    relativecenter.p[0] = - sin(yr * PIOVER180) * xrot_xz;
    relativecenter.p[1] = sin(xr * PIOVER180) * 1.0;
    // Create the world point we are looking at
    center.p[0] = relativecenter.p[0] + eye.p[0];
    // Create our up vector
    xrot_xz = sin((xr) * PIOVER180) * 1.0;
    up.p[0] = sin((yr) * PIOVER180) * xrot_xz;
    up.p[1] = cos((xr) * PIOVER180) * 1.0;
    // Normalize the relative center point, and store in f
    Vertex f = relativecenter / relativecenter.LengthOf();
    // Normalize the up vector
    Vertex upn = up / up.LengthOf();
    // Calculate s elements of the matrix
    Vertex s = f.CrossProduct(upn);
    // Normalize
s = s/s.Length();
//Calculate the u elements of the matrix
Vertex u = s.CrossProduct(f);

//Load identity and create the viewing matrix
rm.LoadIdentity();
rm.SetData(0,0,s.p[0]);
rm.SetData(0,1,u.p[0]);
rm.SetData(0,2,-f.p[0]);
rm.SetData(1,0,s.p[1]);
rm.SetData(1,1,u.p[1]);
rm.SetData(1,2,-f.p[1]);
rm.SetData(2,0,s.p[2]);
rm.SetData(2,1,u.p[2]);
rm.SetData(2,2,-f.p[2]);

//Create the transpose of the viewing matrix
rmt = rm.Transpose();

} void Camera::ExtractFrustum(void)
{
    //This functionality is adapted from Mark Morley
    //http://www.markmorley.com/opengl/frustumculling.html
    float t;
    Matrix4f clip,modl;

    //Get the current projection matrix from OpenGL
    glGetFloatv( GL_PROJECTION_MATRIX, clip.dm );
    //Get the current modelview matrix from OpenGL
    glGetFloatv( GL_MODELVIEW_MATRIX, modl.dm );
    //Multiply the projection matrix with the modelview matrix to
    //get a clipping matrix
    clip.Multiply(&modl);
    //Extract the right plane
    frustum[0][0] = clip.dm[ 3] - clip.dm[ 0];
    frustum[0][1] = clip.dm[ 7] - clip.dm[ 4];
    frustum[0][3] = clip.dm[15] - clip.dm[12];
    //Normalize
    t = sqrt(frustum[0][0] * frustum[0][0] + frustum[0][1] * frustum[0][1] +
             frustum[0][2] * frustum[0][2]);
    frustum[0][0] /= t;
    frustum[0][1] /= t;
    frustum[0][2] /= t;
    frustum[0][3] /= t;

    //Extract left plane
    frustum[1][0] = clip.dm[ 3] + clip.dm[ 0];
    frustum[1][1] = clip.dm[ 7] + clip.dm[ 4];
    //Normalize
    t = sqrt(frustum[1][0] * frustum[1][0] + frustum[1][1] * frustum[1][1] +
             frustum[1][2] * frustum[1][2]);
    frustum[1][0] /= t;
    frustum[1][1] /= t;
    frustum[1][2] /= t;
    frustum[1][3] /= t;

    //Extract the bottom plane
    frustum[2][0] = clip.dm[ 3] + clip.dm[ 1];
    //Normalize
    t = sqrt(frustum[2][0] * frustum[2][0] + frustum[2][1] * frustum[2][1] +
             frustum[2][2] * frustum[2][2]);
    frustum[2][0] /= t;
    frustum[2][1] /= t;
    frustum[2][2] /= t;
frustum[2][3] /= t;

// Extract the top plane
frustum[3][0] = clip.dm[3] - clip.dm[1];
frustum[3][1] = clip.dm[7] - clip.dm[5];

// Normalize
f = sqrt(frustum[3][0] * frustum[3][0] + frustum[3][1] * frustum[3][1] + frustum[3][2] * frustum[3][2]);
frustum[3][0] /= f;
frustum[3][1] /= f;
frustum[3][2] /= f;
frustum[3][3] /= f;

// Extract the far plane

// Normalize
frustum[4][0] /= f;
frustum[4][1] /= f;
frustum[4][2] /= f;
frustum[4][3] /= f;

// Extract the near plane
frustum[5][0] = clip.dm[3] + clip.dm[2];
frustum[5][1] = clip.dm[7] + clip.dm[6];

// Normalize
f = sqrt(frustum[5][0] * frustum[5][0] + frustum[5][1] * frustum[5][1] + frustum[5][2] * frustum[5][2]);
frustum[5][0] /= f;
frustum[5][1] /= f;
frustum[5][2] /= f;
frustum[5][3] /= f;

bool Camera::PointInFrustum(float x, float y, float z)
{
    // Is a point in the frustum test
    int p;
    for( p = 0; p < 6; p++ )
        if( frustum[p][0]*x + frustum[p][1]*y + frustum[p][2]*z + frustum[p][3] <= 0 )
            return false;
    return true;
}

float Camera::SphereInFrustum(float x, float y, float z, float radius)
{
    // Is a sphere in the frustum test, or intersecting it.
    // Returns 0 if out, otherwise distance to the near plane
    int p;
    float d;
    for( p = 0; p < 6; p++ )
        if( d = frustum[p][0]*x + frustum[p][1]*y + frustum[p][2]*z + frustum[p][3] <= 0 )
            return 0;
    return d + radius;
}

float Camera::SphereInFrustum(Vertex point, float radius)
{
    // Same as before but with a vertex as in data
    int p;
    float d;
}
for ( p = 0; p < 6; p++ )
{
    d = frustum[p][0]*point.p[0] + frustum[p][1]*point.p[1] +
        frustum[p][2]*point.p[2] + frustum[p][3];
    if ( d <= -radius )
        return 0;
} return d + radius;

int Camera::SphereIntersectFrustum(Vertex point, float radius)
{
    //Does a sphere intersect the frustum, return 0 if totally out
    //1 if intersection at the boundary, 2 if totally in
    int p;
    int c = 0;
    float d;
    for(p=0; p<6; p++) {
        d = frustum[p][0]*point.p[0] + frustum[p][1]*point.p[1] +
            frustum[p][2]*point.p[2] + frustum[p][3];
        if (d <= -radius)
            return 0;
        if (d > radius)
            c++;
    }
    return (c==6) ? 2 : 1;
}

unsigned char Camera::SphereIntersectFrustumF(Vertex point, float radius,
    unsigned char &bFlags)
{
    //Does a sphere intersect the frustum, with remember flags
    //The bit mask says which planes should be tested
    //If a parent sphere in the BVH, was found intersecting, then we only
    //need to test against those planes that was intersected
    int p;
    float d;
    for(p=0; p<6; p++) {
        if (~bFlags & pIMask[p]) {
            d = frustum[p][0]*point.p[0] + frustum[p][1]*point.p[1] +
                frustum[p][2]*point.p[2] + frustum[p][3];
            if (d <= -radius)
                return ALL_OUT;
            if (d > radius)
                bFlags |= pIMask[p];
        }
    }
    return (bFlags == ALL_CHECKEDIN) ? ALL_IN : INTERSECT;
}

int Camera::SphereIntersectFrustumSq(Vertex point, float squradius)
{
    //Does a sphere intersect the frustum, using squared radius
    int p;
    int c = 0;
    float d;
    for(p=0; p<6; p++) {
        d = frustum[p][0]*point.p[0] + frustum[p][1]*point.p[1] +
            frustum[p][2]*point.p[2] + frustum[p][3];
        if (d < 0) {
            if (d*d > squradius)
                return 0;
        }
    }
    return (c==6) ? 2 : 1;
}

unsigned char Camera::SphereIntersectFrustumSqF(Vertex point, float squradius,
    unsigned char &bFlags)
{
    //Does a sphere intersect the frustum, with remember flags and squared radius
    int p;
    float d;
    for(p=0; p<6; p++) {
        if (~bFlags & pIMask[p]) {
            d = frustum[p][0]*point.p[0] + frustum[p][1]*point.p[1] +
                frustum[p][2]*point.p[2] + frustum[p][3];
            if (d <= -radius)
                return ALL_OUT;
            if (d > radius)
                bFlags |= pIMask[p];
        }
    }
    return (bFlags == ALL_CHECKEDIN) ? ALL_IN : INTERSECT;
}
d = frustum[p][0]*point.p[0] + frustum[p][1]*point.p[1] +
frustum[p][2]*point.p[2] + frustum[p][3];
if (d < 0) {
    if (d*d > sqradius)
        return ALL_OUT;
} else {
    if (d*d > sqradius)
        bFlags |= pIMask[p];
}
return (bFlags == ALL_CHECKEDIN) ? ALL_IN : INTERSECT;

CommonInclude.h

/**
 * CommonInclude.h
 * Author: Anders Dahlbom
 * Last modified: 030603
 * Purpose: Header file for capturing some common properties of the terrain rendering system.
 */
#ifndef COMMONINCLUDE_H
#define COMMONINCLUDE_H

#ifndef OUTPUTTEXT
#define OUTPUTTEXT
#endif

#ifndef LOGSTAT
#define LOGSTAT
#endif

#ifndef MEMCOUNTER
#define MEMCOUNTER
#endif

#define G 9.82
// Small value :)
#define EPSILON 0.00001
// PI divided by 180, used for rotation calculations
#define PIOVER180 0.017453292519943295769236907684886

#define NULLFLOAT -999999.0f

#ifdef MEMCOUNTER
extern int allocatedMem;
#endif

#endif

Matrix4f.h

92
Matrix4f.h

```
class Matrix4f {
public:
    Matrix4f();
    ~Matrix4f();

    //Matrix multiplication with another matrix4f
    void Multiply(Matrix4f* mmatrix);
    //Set a vector at a specified position x:column, y:row
    void SetData(int x, int y, float newdata);
    //Set this matrix to the identity matrix
    voidLoadIdentity(void);
    //Set this matrix to the transpose of itself
    Matrix4f Transpose(void);

    float dm[16]; //The elements of the matrix
};
```

Matrix4f.cpp

```
#include "CommonInclude.h"
#include "Matrix4f.h"
#include "Helper.h"

Matrix4f::Matrix4f()
{
    //Initialize as the identity matrix
    LoadIdentity();
}

Matrix4f::~Matrix4f()
{
}

void Matrix4f::Multiply(Matrix4f* mmatrix)
{
    float tmpmatrix[16];
    //Copy matrix to temporary matrix
    for(int i=0; i<16; i++) {
        tmpmatrix[i] = dm[i];
    }
    //Multiply
    for(int i=0; i<4; i++) {
        for(int j=0; j<4; j++) {
            dm[i*4+j] = 0.0;
            for(int k=0; k<4; k++) {
                dm[i*4+j] += tmpmatrix[j+4*k]*mmatrix->dm[i*4+k];
            }
        }
    }
}

void Matrix4f::SetData(int x, int y, float newdata)
{
    if(x>=0 && x<4 && y>=0 && y<4)
        dm[x*4+y] = newdata;
}

void Matrix4f::LoadIdentity(void)
{
    dm[0] = 1.0;
    dm[1] = 0.0;
    dm[2] = 0.0;
}
```
Matrix4f Matrix4f::Transpose(void) {
    // Swap the rows and the columns
    Matrix4f temp; 
    for (int i=0; i<4; i++) {
        for (int j=0; j<4; j++) {
            temp.dm[j*4+i] = dm[i*4+j];
        }
    }
    return temp;
}

Patch.h
 seksi277f Patch.h
 seksi277f																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																								(94,108,887,904)
//Evaluate the consistency of this patch
void EvaluateConsistency(void);
//Check if this patch can be removed without any loss
bool Removable(void);
//Get the triangle associated with the top border of this patch
Triangle* GetTopTriangle(void);
//Get the triangle associated with the bottom border of this patch
Triangle* GetBottomTriangle(void);
//Get the triangle associated with the left border of this patch
Triangle* GetLeftTriangle(void);
//Get the triangle associated with the right border of this patch
Triangle* GetRightTriangle(void);
//Get the left child triangle for a specific triangle
Triangle* GetLeftChildTriangle(int get_id);
//Get the right child triangle for a specific triangle
Triangle* GetRightChildTriangle(int get_id);
//Get the left child triangle, which is in another patch for a specified triangle
Triangle* GetLeftChildTriangle(Triangle* triangle);
//Get the right child triangle, which is in another patch, for a specified triangle
Triangle* GetRightChildTriangle(Triangle* triangle);
//Get the parent triangle for a specified triangle
Triangle* GetParentTriangle(int get_id);

//Convert this patch from transition to normal
void ConvertToRegular(void) { bits.transition = false; }
//Set the update flag for this patch
void SetUpdateFlag(void) { bits.needsUpdate = true; }
//Set the top neighboring patch for this patch
void SetTopNeighbor(Patch* p) { topNeighbor = p; }
//Set the bottom neighboring patch for this patch
void SetBottomNeighbor(Patch* p) { bottomNeighbor = p; }
//Set the left neighboring patch for this patch
void SetLeftNeighbor(Patch* p) { leftNeighbor = p; }
//Set the right neighboring patch for this patch
void SetRightNeighbor(Triangle* p) { rightNeighbor = p; }
//External reference released
void ReleaseRef(void) { --referenceCount; }
//External reference added
void AddRef(void) ++referenceCount;
//Split a triangle (not really just make updates on pointers)
void SplitTriangle(Triangle* triangle, Triangle* leftChild, Triangle* rightChild);
//Merge a triangle (not really just make updates on pointers)
void MergeTriangle(Triangle* triangle, Triangle* leftChild, Triangle* rightChild);

//Check if this patch is initialized
bool IsInitialized(void) { return bits.initialized; }
//Check if this patch is activated
bool IsActivated(void) { return bits.activated; }
//Check if this patch needs to be updated
bool NeedsUpdate(void) { return bits.needsUpdate; }
//Check if this patch is a transition patch
bool IsTransition(void) { return bits.transition; }
//Get the world X offset for this patch
int GetXOffset(void) { return xstart; }
//Get the world Z offset for this patch
int GetZOffset(void) { return zstart; }
//Get the width of this patch
int GetWidth(void) { return width; }
//Get the depth of the triangle tree associated/owned by this patch
int GetDepth(void) { return treeDepth; }
//Get the current frustum status for this patch
unsigned char InFrustum(void) { return bits.inFrustum; }
//Get this patchs left parent triangle
Triangle* GetPatchLeftParent(void) { return leftParent; }
//Get this patchs right parent triangle
Triangle* GetPatchRightParent(void) { return rightParent; }

//Virtual functions, not defined in the superclass
//Deactivate this patch
virtual void DeactivatePatch(void) = 0;
//Activate this patch
virtual bool ActivatePatch(void) = 0;
//Set the variance for a triangle owned by this patch
virtual void SetVariance(int id, float error) = 0;
//Update this patch
virtual bool Update(void) = 0;
//Update the frustum status for this patch
virtual bool UpdateInFrustum(Camera* camera, unsigned char setValue) = 0;
//Render the triangles of this patch by using vertex array rendering

virtual int RenderVarr(int type) = 0;
    // Get the control map for this patch, if it has any
virtual float* GetControlMap(void) = 0;
    // Get the total patch variance
virtual float GetPatchVariance(void) = 0;
    // Get the variance for a triangle owned by this patch
virtual float GetVariance(int id) = 0;
    // Get a vertex from the vertex list associated with the patch
virtual VertexExt* GetVertex(int vID) = 0;
    // Calculate a vertex at the mid hypotenuse for a triangle
virtual void MidHypotenuse(int tId, int v1, int v2, Vertex& v) = 0;
    // Add parent to rendering list, remove children from rendering list
virtual void AddRenderParent(Triangle* parent, Triangle* leftChild,
    Triangle* rightChild) = 0;
    // Add children to rendering list, remove parent
virtual void AddRenderChildren(Triangle* parent, Triangle* leftChild,
    Triangle* rightChild) = 0;

protected:
    Patch* topNeighbor;    // Neighboring patch connections
    Patch* bottomNeighbor;
    Patch* leftNeighbor;
    Patch* rightNeighbor;
    Patch* pHere;    // Link to this patch
    Triangle* leftParent;    // If this patch are associated with parent triangles
    Triangle* rightParent;
    Triangle** BinTree;    // The current tree of triangles for this patch
    int treeDepth;    // The depth of the triangle tree
    int numAllocatedTris;    // The number of allocated triangles in the tree
    int xstart;    // World x starting point for this patch
    int zstart;    // World z starting point for this patch
    int width;    // The width of this patch
    float tscale;    // The texture scale to use when rendering patch
    PBITFIELD bits;    // State bits for this patch
    int referenceCount;    // Number of external references made to this patch
};

#endif

Patch.cpp

#include <math.h>
#include "CommonInclude.h"
#include "Patch.h"
#include "Matrix4f.h"
#include "Helper.h"

Patch::Patch(int x, int z, int w, int in_id, bool in_transition)
{
    xstart = x;    // sets the starting x position for patch in world space
    zstart = z;    // sets the starting z position for patch in world space
    width = w;    // sets the width of this patch
    bits.transition = in_transition;
    BinTree = NULL;
    topNeighbor = NULL;
    bottomNeighbor = NULL;
    leftNeighbor = NULL;
    rightNeighbor = NULL;
    leftParent = NULL;
    rightParent = NULL;
    bits.inFrustum = 0;
    bits.initialized = false;
    bits.needsUpdate = false;
    bits.consistent = false;
    pHere = this;
    numAllocatedTris = 0;
    // Calculate the tree depth
    treeDepth = (log((width)*(width))/log(2));
}

Patch::~Patch()
{
    if(bits.initialized) Uninitialize();
}
void Patch::Initialize(Patch* top, Patch* bottom, Patch* left, Patch* right, Triangle* leftP, Triangle* rightP)
{
    leftParent = leftP;
    rightParent = rightP;
    topNeighbor = top;
    bottomNeighbor = bottom;
    leftNeighbor = left;
    rightNeighbor = right;
    // Set the parent triangles to point at us
    if(leftParent!=NULL) leftParent->SetChildPatch(&pHere);
    if(rightParent!=NULL) rightParent->SetChildPatch(&pHere);
    bits.initialized = true;
}

void Patch::Initialize(Patch* top, Patch* bottom, Patch* left, Patch* right)
{
    Initialize(top,bottom,left,right,NULL,NULL);
}

void Patch::Uninitialize()
{
    bits.initialized = false;
}

void Patch::EvaluateConsistency()
{
    bits.consistent = true;
    // The purpose of this function is to validate if a patch is consistent
    // with concern about its neighboring patches.
    // This can be good to know if extending with dynamic patches, in order
    // to ensure that each dynamic patch, is surrounded by transition patches
    // The functionality of this function has not been implemented, so
    // each patch is considered to be consistent, and the consistency needs
    // to be ensured from the context which creates the patches
}

bool Patch::Removable()
{
    if(!bits.transition) {
        // Convert patch to transition if the variance is zero
        if(GetPatchVariance()>=0.0) return false;
        bits.transition = true;
    }
    // If all surrounding patches are transitions, then we can be removed
    if(leftNeighbor!=NULL)
        if(!leftNeighbor->IsTransition()) return false;
    if(rightNeighbor!=NULL)
        if(!rightNeighbor->IsTransition()) return false;
    if(topNeighbor!=NULL)
        if(!topNeighbor->IsTransition()) return false;
    if(bottomNeighbor!=NULL)
        if(!bottomNeighbor->IsTransition()) return false;
    return true;
}

Triangle* Patch::GetTopTriangle()
{
    if(bits.activated) return BinTree[3];
    return NULL;
}

Triangle* Patch::GetBottomTriangle()
{
    if(bits.activated) return BinTree[2];
    return NULL;
}

Triangle* Patch::GetLeftTriangle()
{
    if(bits.activated) return BinTree[2];
    return NULL;
}

Triangle* Patch::GetRightTriangle()
{
    if(bits.activated) return BinTree[3];
    return NULL;
}
return NULL;
}

Triangle* Patch::GetLeftChildT(int get_id) {
if(!bits.activated) return NULL;
int newID = get_id*2;
if(newID<numAllocatedTris && get_id>1) {
    //The child triangle requested is contained in this patch, return it
    return BinTree[newID];
} else {
    //The calculate children is out of index range, is there any
    //child patches present for this triangle
    Patch* child = BinTree[get_id]->GetChildPatch();
    if(child != NULL) {
        if(child->IsActivated()) {
            //A child patch was found and it is activated, request
            //the child triangle from the child patch
            return child->GetLeftChildT2(BinTree[get_id]);
        }
    }
    //No child triangle found
    return NULL;
}

Triangle* Patch::GetRightChildT(int get_id) {
if(!bits.activated) return NULL;
int newID = get_id*2+1;
if(newID<numAllocatedTris && get_id>1) {
    //Found requested triangle within this patch, return it
    return BinTree[newID];
} else {
    //The requested child index was out of index range,
    //check if there is any child patches present
    Patch* child = BinTree[get_id]->GetChildPatch();
    if(child != NULL) {
        if(child->IsActivated()) {
            //Request child triangle from the child patch
            return child->GetRightChildT2(BinTree[get_id]);
        }
    }
    //No child triangle found
    return NULL;
}

Triangle* Patch::GetLeftChildT2(Triangle* triangle) {
if(!bits.activated) return NULL;
//The child of an external triangle is requested
//The child exists in this patch, but the parent is not
//Find the correct child
if(triangle==leftParent) {
    return BinTree[4];
} else if(triangle==rightParent) {
    return BinTree[6];
}
//No match
return NULL;
}

Triangle* Patch::GetRightChildT2(Triangle* triangle) {
if(!bits.activated) return NULL;
//The child of an external triangle is requested, the child should exist
//In this patch, check if we are attached to triangle, and return matching
//child triangle
if(triangle==leftParent) {
    return BinTree[5];
} else if(triangle==rightParent) {
    return BinTree[7];
}
//No match
return NULL;
}
Triangle* Patch::GetParentT(int get_id) {
    if(!bits.activated) return NULL;
    //Calculate the id for the requested parent
    int newID = (int) get_id/2;
    if(get_id<numAllocatedTris && newID>1) {
        //If this patch has parent triangles, return them if they are requested
        if(newID==2 && leftParent!=NULL) {
            return leftParent;
        } else if(newID==3 && rightParent!=NULL) {
            return rightParent;
        }
        //The requested parent triangle exists in this patch
        return BinTree[newID];
    }
    //No parent triangle found
    return NULL;
}

void Patch::SplitTriangle(Triangle* triangle, Triangle* leftChild, Triangle* rightChild) {
    //This function does not really split a triangle, it only
    //updates the neighboring connections of the triangles in focus.
    //Split functionality adapted from Seumas McNally
    //http://www.longbowdigitalarts.com/seumas/
    leftChild->leftNeighbor = rightChild;
    rightChild->rightNeighbor = leftChild;
    leftChild->baseNeighbor = triangle->leftNeighbor;
    //Update the left neighbor of the parent triangle
    if(triangle->leftNeighbor!=NULL) {
        if(triangle->leftNeighbor->baseNeighbor == triangle) {
            triangle->leftNeighbor->baseNeighbor = leftChild;
        } else {
            if(triangle->leftNeighbor->leftNeighbor == triangle) {
                triangle->leftNeighbor->leftNeighbor = leftChild;
            } else {
                triangle->leftNeighbor->rightNeighbor = leftChild;
            }
        }
    }
    rightChild->baseNeighbor = triangle->rightNeighbor;
    //Update the right neighbor of the parent triangle
    if(triangle->rightNeighbor!=NULL) {
        if(triangle->rightNeighbor->baseNeighbor == triangle) {
            triangle->rightNeighbor->baseNeighbor = rightChild;
        } else {
            if(triangle->rightNeighbor->rightNeighbor == triangle) {
                triangle->rightNeighbor->rightNeighbor = rightChild;
            } else {
                triangle->rightNeighbor->leftNeighbor = rightChild;
            }
        }
    }
}

void Patch::MergeTriangle(Triangle* triangle, Triangle* leftChild, Triangle* rightChild) {
    //This function does not really merge two triangles, it only updates
    //the neighboring connections for the triangles in focus
    triangle->leftNeighbor = leftChild->baseNeighbor;
    //Update the left child's base neighbor
    if(triangle->leftNeighbor!=NULL) {
        if(triangle->leftNeighbor->leftNeighbor == leftChild) {
            triangle->leftNeighbor->leftNeighbor = triangle;
        } else {
            if(triangle->leftNeighbor->rightNeighbor == leftChild) {
                triangle->leftNeighbor->rightNeighbor = triangle;
            } else {
                triangle->leftNeighbor->baseNeighbor = triangle;
            }
        }
    }
}
triangle->rightNeighbor = rightChild->baseNeighbor;

// Update the right child's base neighbor
if(triangle->rightNeighbor!=NULL) {
    if(triangle->rightNeighbor->rightNeighbor == rightChild) {
        triangle->rightNeighbor->rightNeighbor = triangle;
    } else {
        if(triangle->rightNeighbor->leftNeighbor == rightChild) {
            triangle->rightNeighbor->leftNeighbor = triangle;
        } else {
            triangle->rightNeighbor->baseNeighbor = triangle;
        }
    }
}
}

PatchDynamic.h

/*****************************************************************************/
/* PatchDynamic.h                                                             */
/* Author: Anders Dahlbom                                                     */
/* Last modified: 030603                                                      */
/*                                                                            */
/* Purpose: This class captures the functionality of a dynamic patch. It      */
/* inherits from the superclass Patch, and besides the virtual               */
/* common functions, it also introduces dynamic patch specific              */
/* functionality. A dynamic patch is based on a b-spline surface, and it     */
/* has the knowledge of a dynamic bintree manager. At need a bintree is      */
/* received from the manager and attached to the two parent triangles        */
/* covered by a specific dynamic patch. A vertex list associated with the    */
/* dynamic bintree which have been received, is also received from the        */
/* manager. This vertex list is then filled with information extracted from   */
/* the b-spline.                                                            */
/*****************************************************************************/

#ifndef PATCHDYNAMIC_H
#define PATCHDYNAMIC_H

#include "Patch.h"
#include "Vertex.h"
#include "VertexExt.h"
#include "BinTreeManager.h"
#include "RoamNurb.h"

// Bits for the current state of this dynamic patch
typedef struct pdbitfield {
    unsigned char nurbsetup : 1;
    unsigned char rotation : 2;
    unsigned char numControlPoints : 5;
} PDBITFIELD;

class PatchDynamic : public Patch {
public:
    PatchDynamic(int x, int z, int w, int in_id, bool in_transition);
    ~PatchDynamic();
    // Initialize this patch, which is not associated with parent triangles
    void Initialize(Patch* top, Patch* bottom, Patch* left, Patch* right);
    // Initialize this patch, which is associated with two parent triangles
    void Initialize(Patch* top, Patch* bottom, Patch* left, Patch* right, Triangle* leftP, Triangle* rightP);
    // Setup the nurb, etc. for this dynamic patch
    void SetupNurb(RoamNurbBasis* nurbBasis, Vertex* ulc, Vertex* urc, Vertex* llc, Vertex* lrc, int in_numControlPoints);
    // Uninitialize this patch
    void Uninitialize();
    // Update this patch
    bool Update();
    // Evaluate the consistency for this patch
    void EvaluateConsistency();
    // Activate this patch, or try at least
    bool ActivatePatch();
    // Deactivate this patch, or try at least
    void DeactivatePatch();
    // Get the variance for a specified triangle in this dynamic patch
    float GetVariance(int id);

100
void SetVariance(int id, float error);  

bool UpdateInFrustum(Camera* camera, unsigned char setValue);  

int GetNumControlPoints(void) { return dbits.numControlPoints; }  

float GetPatchVariance(void) { return variance; }  

float* GetControlMap(void) { return ctrl_map; }  

RoamNurb* GetNurb(void) { return ourNurb; }  

void MidHypotenuse(int tId, int v1, int v2, Vertex& v);  

int RenderVarr(int type);  

void AddRenderParent(Triangle* parent, Triangle* leftChild, Triangle* rightChild);  

void AddRenderChildren(Triangle* parent, Triangle* leftChild, Triangle* rightChild);  

void InterpolateControlPoints(void);  

void SetBinTreeManager(BinTreeManager* bManager);  

void SetTextureStepping(float s);  

int numAttachmentsNow;  

private:  

void InterpolateControlPoints(void);  

float* ctrl_map;  

float variance;  

RoamNurb* ourNurb;  

DynamicBinTree* dynamicBinTree;  

PDBITFIELD dbits;  

PatchDynamic.cpp

#include "CommonInclude.h"  
#include "PatchDynamic.h"  
#include "Helper.h"  

PatchDynamic::PatchDynamic()  

static void SetBinTreeManager(BinTreeManager* bManager);  

static void SetTextureStepping(float s);  

static int numAttachmentsNow;  

};  

#endif
TextureStepping = s;

PatchDynamic::PatchDynamic(int x, int z, int w, int in_id, bool in_transition) :
Patch(x,z,w,in_id, in_transition)
{
  dynamicBinTree = NULL;
  bits.needsUpdate = true;
  ctrl_map = NULL;
  bits.rotation = 0;
  bits.activated  = false;
  variance = 0.0f;
  upperLeft = NULL;
  upperRight = NULL;
  lowerLeft = NULL;
  lowerRight = NULL;
  referenceCount = 0;
}

PatchDynamic::~PatchDynamic()
{
  if(bits.initialized) Uninitialize();
}

void PatchDynamic::Initialize(Patch* top, Patch* bottom, Patch* left, Patch* right)
{
  Initialize(top,bottom,left,right,NULL,NULL);
}

void PatchDynamic::Initialize(Patch* top, Patch* bottom, Patch* left, Patch* right,
Triangle* leftP, Triangle* rightP)
{
  Patch::Initialize(top,bottom,left,right,leftP,rightP);
  //Update the neighboring connections for our neighboring patches
  if(topNeighbor!=NULL) {
    top->SetBottomNeighbor(this);
  }
  if(bottomNeighbor!=NULL) {
    bottom->SetTopNeighbor(this);
  }
  if(leftNeighbor!=NULL) {
    left->SetRightNeighbor(this);
  }
  if(rightNeighbor!=NULL) {
    right->SetLeftNeighbor(this);
  }
}

void PatchDynamic::SetupNurb(RoamNurbBasis* nurbBasis, Vertex* ulc, Vertex* urc,
Vertex* llc, Vertex* lrc,
int in_numControlPoints)
{
  dbits.numControlPoints = in_numControlPoints;
  upperLeft = ulc;
  upperRight = urc;
  lowerLeft = llc;
  lowerRight = lrc;
  int numControlPoints = dbits.numControlPoints;
  //Allocate array for the control points
  ctrl_map = new float[numControlPoints*numControlPoints];
  //Set all control points to an artificial NULL value
  for(int i=0; i<numControlPoints; i++) {
    for(int j=0; j<numControlPoints; j++) {
      ctrl_map[i+numControlPoints*j] = NULLFLOAT;
    }
  }
  //Set the corner control points to match our corner vertices
  ctrl_map[0] = upperLeft->p[1];
  ctrl_map[numControlPoints-1] = upperRight->p[1];
  ctrl_map[(numControlPoints-1)*numControlPoints] = lowerLeft->p[1];
  ctrl_map[numControlPoints*numControlPoints-1] = lowerRight->p[1];
  //Match our border control points with neighboring patches
  if(topNeighbor!=NULL) {
    float* tmp_map = topNeighbor->GetControlMap();
    for(int i=1; i<numControlPoints; i++) {
      ctrl_map[i] = tmp_map[i*(numControlPoints-1)*numControlPoints];
    }
  }
  if(bottomNeighbor!=NULL) {

}
float* tmp_map = bottomNeighbor->GetControlMap();
for(int i=1; i<numControlPoints-1; i++) {
    ctrl_map[i+(numControlPoints-1)*numControlPoints] = tmp_map[i];
}
}
if(leftNeighbor!=NULL) {
    float* tmp_map = leftNeighbor->GetControlMap();
    for(int i=1; i<numControlPoints-1; i++) {
        ctrl_map[i*numControlPoints] = tmp_map[numControlPoints – 1 + i*numControlPoints];
    }
}
if(rightNeighbor!=NULL) {
    float* tmp_map = rightNeighbor->GetControlMap();
    for(int i=1; i<numControlPoints-1; i++) {
        ctrl_map[numControlPoints-1+i*numControlPoints] = tmp_map[i*numControlPoints];
    }
}
//Interpolate to calculate the height for all control points which
//have not yet been set
Vertex lside;
Vertex rsdie;
Vertex across;
for(int j=0; j<numControlPoints; j++) {
    float u = (float)j/(float)(numControlPoints-1);
    lside = *ulc*(1-u) + *llc*u;
    rsdie = *urc*(1-u) + *lrc*u;
    for(int i=0; i<numControlPoints; i++) {
        if(ctrl_map[i+j*numControlPoints] == NULLFLOAT) {
            float v = (float)i/(float)(numControlPoints-1);
            across = lside*(1-v) + rsdie*v;
            ctrl_map[i+j*numControlPoints] = across.p[1];
        }
    }
}
//Create a nurb object
ourNurb = new RoamNurb(nurbBasis,ctrl_map);
//Evaluate how this patch matches the parent triangles
Patch* ptpPatch = leftParent->GetParentPatch();
Patch* testVertex = ptpPatch->GetVertex(leftParent->GetVertex(0));
if(testVertex == lowerLeft) { //no rotation , perfect match
dbits.rotation = 0;
} else if(testVertex == upperLeft) { //bintree needs to be rotated 1 step
    dbits.rotation = 1;
} else if(testVertex == upperRight) { //bintree needs to be rotated 2 steps
    dbits.rotation = 2;
} else if(testVertex == lowerRight) { //bintree needs to be rotated 3 steps
    dbits.rotation = 3;
} dbits.nurbsetup = true;
//Check if we are currently needed
if(leftParent!=NULL && rightParent!=NULL) {
    if(!leftParent->IsDirtySplit()) ActivatePatch();
    else if(!rightParent->IsDirtySplit()) ActivatePatch();
} bits.needsUpdate = true;

bool PatchDynamic::Update(void){
    if(!bits.needsUpdate) return false;
    bits.needsUpdate = false;
    //Update the control points
    InterpolateControlPoints();
    if(bits.transition)
        variance = 0.0;
    else
        variance = ourNurb->GetMaxError();
    if(bits.activated) {
        //If patch is activated, then re-extract information from the nurb
        VertexExt* tmp_vertex_list = dynamicBinTree->GetVertexList();
        ourNurb->FillVertexList(tmp_vertex_list,dbits.rotation,xstart,
                               zstart,width,TextureStepping);
void PatchDynamic::Uninitialize(void)
{
    bits.initialized = false;
    Patch::Uninitialize();
    if (bits.nurbsetup) {
        // Notify our neighbors that we do not exist any more
        if (topNeighbor != NULL) {
            topNeighbor->SetBottomNeighbor(NULL);
        }
        if (bottomNeighbor != NULL) {
            bottomNeighbor->SetTopNeighbor(NULL);
        }
        if (leftNeighbor != NULL) {
            leftNeighbor->SetRightNeighbor(NULL);
        }
        if (rightNeighbor != NULL) {
            rightNeighbor->SetLeftNeighbor(NULL);
        }
        delete[] ctrl_map;
    }
}

void PatchDynamic::EvaluateConsistency(void)
{
    if (binTreeManager == NULL || bits.nurbsetup == false) bits.consistent = false;
    else Patch::EvaluateConsistency();
}

bool PatchDynamic::ActivatePatch(void)
{
    if (!bits.consistent) EvaluateConsistency();
    // If this patch is not yet activated, and there is someone
    // which have been referencing us, try to activate us
    if (bits.consistent && !bits.activated && referenceCount > 0) {
        dynamicBinTree = binTreeManager->GetFreeBinTree();
        if (dynamicBinTree == NULL) {
            // There was no free dynamic bintrees
            return false;
        }
        // Make the triangles of the dynamic bintree have us for parent patch
        dynamicBinTree->SetPatch(pHere);
        // Receive the actual bintree
        BinTree = dynamicBinTree->GetBinTree();
        // Get the list of vertices from the dynamic bintree
        VertexExt* tmp_vertex_list = dynamicBinTree->GetVertexList();
        // Receive the tree depth
        treeDepth = dynamicBinTree->GetTreeDepth();
        // Get the number of triangles present in the bintree (-2)
        numAllocatedTris = dynamicBinTree->GetNumAllocatedTris();
        // Fill in the associated vertex list with information from our nurb object
        ourNurb->FillVertexList(tmp_vertex_list, dbits.rotation, xstart, zstart, width, TextureStepping);
        bits.activated = true;
        // Increase the global patch dynamic currently attachment counter
        numAttachmentsNow++;
    }
    return true;
}

void PatchDynamic::DeactivatePatch(void)
{
    // If we are activated, and the number of external references to us is zero
    if (bits.activated && referenceCount <= 0) {
        // Then deactivate us
        BinTree = NULL;
        numAllocatedTris = 0;
        treeDepth = 0;
        // Notify the bintree manager that we are releasing a dynamic bintree
        binTreeManager->ReleaseBinTree(dynamicBinTree);
        dynamicBinTree = NULL;
        bits.activated = false;
        referenceCount = 0;
    }
}
float PatchDynamic::GetVariance(int id)
{
    //Calculate an artificial variance for a specified triangle
    //The variance is dependant on the level at which the triangle resides in the bintree, vTri = vTotal / (1Tri^2)
    static int level;
    if(bits.transition) return 0.0;
    if(bits.activated) {
        if(id>2 && id<(numAllocatedTris/2)) {
            level = 0;
            while(id>3) {
                id >>= 1;
                level++;
            }
            if(level==0) level=1;
            return variance/(level*level);
        }
    }
    return 0.0;
}

void PatchDynamic::SetVariance(int id, float error)
{
}

void PatchDynamic::InterpolateControlPoints(void)
{
    int numControlPoints = dbits.numControlPoints;
    int jbegin = 0;
    int jend = numControlPoints;
    int ibegin = 0;
    int iend = numControlPoints;
    float* tmp_map;
    int i,j;
    float v;

    //Set corner points so that our corner control heights match the static vertices
    ctrl_map[0] = upperLeft->p[1];
    ctrl_map[numControlPoints-1] = upperRight->p[1];
    ctrl_map[numControlPoints*(numControlPoints-1)] = lowerLeft->p[1];
    ctrl_map[numControlPoints*numControlPoints-1] = lowerRight->p[1];

    //match neighbors
    if(topNeighbor!=NULL) {
        if(!topNeighbor->IsTransition()) {
            tmp_map = topNeighbor->GetControlMap();
            for(i=1; i<numControlPoints-1; i++) {
                ctrl_map[i] = tmp_map[i+(numControlPoints-1)*numControlPoints];
            }
            jbegin++;
        } else {
            if(!bits.transition)
                topNeighbor->SetUpdateFlag();
        }
    }
    if(bottomNeighbor!=NULL) {
        if(!bottomNeighbor->IsTransition()) {
            tmp_map = bottomNeighbor->GetControlMap();
            for(i=1; i<numControlPoints-1; i++) {
                ctrl_map[i+(numControlPoints-1)*numControlPoints] = tmp_map[i];
            }
            jend--;
        } else {
            if(!bits.transition)
                bottomNeighbor->SetUpdateFlag();
        }
    }
    if(leftNeighbor!=NULL) {
        if(!leftNeighbor->IsTransition()) {
            tmp_map = leftNeighbor->GetControlMap();
            for(i=1; i<numControlPoints-1; i++) {
                ctrl_map[i*numControlPoints] = tmp_map[numControlPoints-1+i*numControlPoints];
            }
            ibegin++;
        } else {
            if(!bits.transition)
leftNeighbor->SetUpdateFlag();
}
if(rightNeighbor!=NULL) {
  if(!rightNeighbor->IsTransition()) {
    tmp_map = rightNeighbor->GetControlMap();
    for(i=1; i<numControlPoints-1; i++) {
      ctrl_map[numControlPoints-1+i*numControlPoints] =
        tmp_map[i*numControlPoints];
    } iend--;
  } else {
    if(!bits.transition) rightNeighbor->SetUpdateFlag();
  }
}
if(bits.transition) {
  //This is a transition patch, reinterpolate the control points in need
  float lsidey,rsidey;
  for(j=jbegin; j<jend; j++) {
    float u = (float)j/(float)(numControlPoints-1);
    lsidey = upperLeft->p[1]*(1-u) + lowerLeft->p[1]*u;
    rsidey = upperRight->p[1]*(1-u) + lowerRight->p[1]*u;
    for(i=ibegin; i<iend; i++) {
      if((i>0 && i<numControlPoints-1) ||(j>0 && j<numControlPoints-1)) {
        v = (float)i/(float)(numControlPoints-1);
        ctrl_map[i+j*numControlPoints] = lsidey*(1-v) + rsidey*v;
      }
    }
  }
}
bool PatchDynamic::UpdateInFrustum(Camera* camera, unsigned char setValue) {
  //Set the frustum status for this patch
  if(setValue==0) bits.inFrustum = setValue;
  else bits.inFrustum = 2;
  return !(bits.inFrustum==0);
}
VertexExt* PatchDynamic::GetVertex(int vID) {
  if(bits.activated) return dynamicBinTree->GetVertex(vID);
  return NULL;
}
void PatchDynamic::MidHypotenuse(int tId, int v1, int v2, Vertex& v) {
  globalVertexList = dynamicBinTree->GetVertexList();
  //If this is not a leaf triangle, then use the top vertex of one of the
  //children, as mid hypotenuse, otherwise, calculate the vertex
  if((tId<<1)<numAllocatedTris) v = globalVertexList[BinTree[tId<<1]->GetVertex(0)];
  else v = (globalVertexList[v1] + globalVertexList[v2])*0.5;
}
int PatchDynamic::RenderVarr(int type) {
  if(!bits.activated) {
    //If we are not active, then we can not be rendered
    return 0;
  }
  //Get the vertex_list, render_list, and number of elements to render
  VertexExt* vertex_list = dynamicBinTree->GetVertexList();
  unsigned int* r_list = dynamicBinTree->GetRenderList();
  unsigned int numRenderElements = dynamicBinTree->GetNumRenderElements();
  //Tell OpenGL to use the specified vertex list
  glVertexPointer(3, GL_FLOAT, sizeof(VertexExt), &vertex_list[0].p);
  if(type==2 || type==3) {
    glColor3f(1.0f,1.0f,1.0f);
    //Tell OpenGL to use the specified texture coordinates
    glTexCoordPointer(2, GL_FLOAT, sizeof(VertexExt), &vertex_list[0].uv);
    //If lighting is turned on, tell OpenGL to use the specified
    //list of normal vectors
}
if (type == 3)
    {glNormalPointer(GL_FLOAT, sizeof(VertexExt), &vertex_list[0].n);
} else {glColor3f(1.0f, 0.85f, 0.85f);

// Tell OpenGL to render the elements in the render_list, using the
// specified vertex_list, and possibly texture coordinates and normal vectors
glDrawElements(GL_TRIANGLES, numRenderElements*3, GL_UNSIGNED_INT, &r_list[0]);
return numRenderElements;
}

void PatchDynamic::AddRenderParent(Triangle* parent, Triangle* leftChild, Triangle* rightChild)
{
    if (leftChild == NULL || rightChild == NULL) return;

    // Fetch the render_list, index_list, and number of elements
    unsigned int* render_list = dynamicBinTree->GetRenderList();
    unsigned int* index_list = dynamicBinTree->GetIndexList();
    unsigned int numRenderElements = dynamicBinTree->GetNumRenderElements();
    if (parent == NULL)
        { // Just remove the children
            unsigned int li = leftChild->GetRenderIndex();
            if ([li*3] != [numRenderElements-1]*3)
                remove (render_list, [li*3], [numRenderElements-1]*3);
            render_list [li*3+1] = render_list [(numRenderElements-1)*3+1];
            render_list [li*3+2] = render_list [(numRenderElements-1)*3+2];
            index_list [li] = index_list [numRenderElements-1];
            BinTree [index_list [li]]->SetRenderIndex (li);
        }
    numRenderElements --;
    unsigned int ri = rightChild->GetRenderIndex();
    if ([ri*3] != [numRenderElements-1]*3)
        remove (render_list, [ri*3], [numRenderElements-1]*3);
    render_list [ri*3+1] = render_list [(numRenderElements-1)*3+1];
    render_list [ri*3+2] = render_list [(numRenderElements-1)*3+2];
    index_list [ri] = index_list [numRenderElements-1];
    BinTree [index_list [ri]]->SetRenderIndex (ri);
    numRenderElements --;
    }
    else
        { // Remove children
            unsigned int li = leftChild->GetRenderIndex();
            if ([li*3] != [numRenderElements-1]*3)
                remove (render_list, [li*3], [numRenderElements-1]*3);
            render_list [li*3+1] = render_list [(numRenderElements-1)*3+1];
            render_list [li*3+2] = render_list [(numRenderElements-1)*3+2];
            index_list [li] = index_list [numRenderElements-1];
            BinTree [index_list [li]]->SetRenderIndex (li);
        }
    numRenderElements --;
    unsigned int ri = rightChild->GetRenderIndex();
    if ([ri*3] != [numRenderElements-1]*3)
        remove (render_list, [ri*3], [numRenderElements-1]*3);
    render_list [ri*3+1] = render_list [(numRenderElements-1)*3+1];
    render_list [ri*3+2] = render_list [(numRenderElements-1)*3+2];
    index_list [ri] = index_list [numRenderElements-1];
    BinTree [index_list [ri]]->SetRenderIndex (ri);
    numRenderElements --;
    }
    else
        { // Add parent
            render_list [numRenderElements*3] = parent->GetVertex(0);
            render_list [numRenderElements*3+1] = parent->GetVertex(2);
            render_list [numRenderElements*3+2] = parent->GetVertex(1);
            index_list [numRenderElements] = parent->GetID();
            parent->SetRenderIndex (numRenderElements);
            numRenderElements ++;
        }
    MergeTriangle (parent, leftChild, rightChild);
    dynamicBinTree->SetNumRenderElements (numRenderElements);
}

void PatchDynamic::AddRenderChildren(Triangle* parent, Triangle* leftChild, Triangle* rightChild)
{
    if (leftChild == NULL || rightChild == NULL) return;
    unsigned int* render_list = dynamicBinTree->GetRenderList();
    unsigned int* index_list = dynamicBinTree->GetIndexList();
unsigned int numRenderElements = dynamicBinTree->GetNumRenderElements();
if (parent == NULL) {
    // Just add the children
    render_list[numRenderElements*3] = leftChild->GetVertex(0);
    render_list[numRenderElements*3+1] = leftChild->GetVertex(2);
    render_list[numRenderElements*3+2] = leftChild->GetVertex(1);
    index_list[numRenderElements] = leftChild->GetID();
    leftChild->SetRenderIndex(numRenderElements);
    numRenderElements++;
    render_list[numRenderElements*3] = rightChild->GetVertex(0);
    render_list[numRenderElements*3+1] = rightChild->GetVertex(2);
    render_list[numRenderElements*3+2] = rightChild->GetVertex(1);
    index_list[numRenderElements] = rightChild->GetID();
    rightChild->SetRenderIndex(numRenderElements);
    numRenderElements++;
} else {
    // remove parent, add children, put left child at parents position
    unsigned int pi = parent->GetRenderIndex();
    render_list[pi*3] = leftChild->GetVertex(0);
    render_list[pi*3+1] = leftChild->GetVertex(2);
    render_list[pi*3+2] = leftChild->GetVertex(1);
    index_list[pi] = leftChild->GetID();
    leftChild->SetRenderIndex(pi);
    // Add the right child to the end
    render_list[numRenderElements*3] = rightChild->GetVertex(0);
    render_list[numRenderElements*3+1] = rightChild->GetVertex(2);
    render_list[numRenderElements*3+2] = rightChild->GetVertex(1);
    index_list[numRenderElements] = rightChild->GetID();
    rightChild->SetRenderIndex(numRenderElements);
    numRenderElements++;
    // Split triangle
    SplitTriangle(parent, leftChild, rightChild);
}
void DeactivatePatch(void);
// Set the variance of a specific triangle in this patch
void SetVariance(int id, float error);
// Add a parent triangle to the rendering list, and remove its children
void AddRenderParent(Triangle* parent, Triangle* leftChild, Triangle* rightChild);
// Add the children of a triangle to the rendering list, and remove the triangle
void AddRenderChildren(Triangle* parent, Triangle* leftChild, Triangle* rightChild);
// Reset the current rendering list
void ResetRenderList(void) {
    numRenderElements = 0;
}
// Calculate the vertex at the mid of the hypotenuse of a triangle
void MidHypotenuse(int tId, int v1, int v2, Vertex& v);
// Activate this patch
bool ActivatePatch(void);
// Update this patch
bool Update(void);
// Find the two leaf triangles bound in the area specified
bool FindLeafs(float x1, float z1, float x2, float z2, Triangle** t1, Triangle** t2);
// Update this triangles frustum status
bool UpdateInFrustum(Camera* camera, unsigned char setValue);
// Render this patch by using vertex arrays
int RenderVarr(int type);
// Get the height at the center of this patch
float GetCenterHeight(void);
// Get the total patch variance
float GetPatchVariance(void);
// Get the variance of a specified triangle
float GetVariance(int id) {
    return VarianceTree[id];
}
// Get the control map for this patch, which is always NULL
float* GetControlMap(void);
// Get a vertex specified by its index
VertexExt* GetVertex(int vID);
// Static function to set the amount vertices in side for all Static patches
static void SetVertexRowCount(int inCount);

private:
    // Create two child triangles of a specific triangle
    void CreateSplit(Triangle* triangle);
    // Recursively calculate the frustum status for triangles
    void RecursiveTriangleInFrustum(int in_id, unsigned char parentIn);
    // Recursively set the frustum flag for triangles
    void RecursiveSetFrustumFlag(int in_id, unsigned char parentIn);
    // Recursively recalculate the variance of triangles affected by a change
    bool RecursiveRecalculateVariance(int tri_id, int thisLevel, int stopLevel);
    // Recursively create child triangles for a triangle
    int RecursiveCreateSplit(Triangle* triangle, int thisLevel, int stopLevel);
    // Recursively calculate the variance of the triangles contained in this patch
    float RecursiveCalculateVariance(Triangle* triangle, int thisLevel, int stopLevel);
    // Recursively find a triangle matching the area given
    Triangle* FindLeaf2(Triangle* triangle, float x1, float z1, float x2, float z2,
                        Triangle* notThis, int level);

    float* VarianceTree;    // An array of variances for the triangles
    float hscale, vscale;    // Horizontal and vertical multiplier
    int numRenderPatch;     // Number of child patches to be rendered
    unsigned int* render_list; // A array of triplets of vertices to be rendered

    // In the rendering list
    unsigned int* index_list;
    unsigned int numRenderElements; // Number of triplets to be rendered
    VertexExt* vertex_list; // Pointer to a list of vertices used by the rendering list
    Patch** render_patch_list; // A list of child patches to be rendered

    // Static variables used by all static patches
    static int VRowCount;    // The number of vertices in each row of vertex list
    static Camera* globalCamera; // The current camera in use
};

#endif

PatchStatic.cpp
```cpp
#include "CommonInclude.h"
#include "PatchStatic.h"
#include "Helper.h"

// The static variables shared amongst all instances of the patch static class
// are initialize here
Camera* PatchStatic::globalCamera = NULL;
int PatchStatic::vRowCount = 0;

// To globally set the amount of vertices in each row of the global vertex list
void PatchStatic::SetVertexRowCount(int inCount)
{
    vRowCount = inCount;
}

PatchStatic::PatchStatic(int x, int z, int w, int in_id, bool in_transition, float in_hscale, float in_vscale, VertexExt* pVertices) : Patch(x,z,w,in_id, in_transition)
{
    vertex_list = pVertices;
    hscale = in_hscale;
    vscale = in_vscale;
    numRenderElements = 0;
    numRenderPatch = 0;

    // Allocate an array used for rendering child patches which is connected
    render_patch_list = new (Patch*)[width*width];

    // Calculate how many triangles will be created in this static patch
    numAllocatedTris = (2*width*width-1)*2+2;

    // Allocate storage for an array of triangles, and an array of variance values
    BinTree = new (Triangle*)[numAllocatedTris];
    VarianceTree = new (float)[numAllocatedTris];

    // Allocate a rendering list, and a index list for vertex array rendering
    render_list = new unsigned int[width*width*2*3];
    index_list = new unsigned int[width*width*2];

    // Create the two root triangles in this static patch
    BinTree[2] = new Triangle(xstart+(zstart+width)*vRowCount,xstart+zstart*vRowCount,
                               xstart+width+(zstart+width)*vRowCount,2,&pHere);
    BinTree[3] = new Triangle(xstart+width+zstart*vRowCount, xstart+width+(zstart+width)*vRowCount,
                               xstart+zstart*vRowCount,3,&pHere);

    // Recursively create triangles until the give tree depth is reached
    RecursiveCreateSplit(BinTree[2],0,treeDepth);
    RecursiveCreateSplit(BinTree[3],0,treeDepth);

    // Recursively calculate the variance for all triangles created
    RecursiveCalculateVariance(BinTree[2],0,treeDepth);
    RecursiveCalculateVariance(BinTree[3],0,treeDepth);

    // Set the activation flag for this patch, as it is static and always present
    bits.activated = true;

    #ifdef MEMCOUNTER
    // Add the amount of memory allocated to the global memory tracker
    allocatedMem += sizeof(Patch*)*width*width;
    allocatedMem += sizeof(int)*width*width;
    allocatedMem += sizeof(Triangle*)*numAllocatedTris;
    allocatedMem += sizeof(float)*numAllocatedTris;
    allocatedMem += sizeof(unsigned int)*width*width*2*3;
    allocatedMem += sizeof(Triangle)*2;
    #endif
}

PatchStatic::~PatchStatic()
{
    if(bits.initialized) Uninitialize();
}

void PatchStatic::Initialize(Patch* top, Patch* bottom, Patch* left, Patch* right)
{
    Initialize(top,bottom,left,right,NULL,NULL);
}
void PatchStatic::Initialize(Patch* top, Patch* bottom, Patch* left, Patch* right,
Triangle* leftP, Triangle* rightP)
{
    //Call the initialize in the superclass
    Patch::Initialize(top,bottom,left,right,leftP,rightP);
    //Setup the neighboring connections for the two root triangles
    if(bottomNeighbor!=NULL) {
        BinTree[2]->leftNeighbor = bottomNeighbor->GetTopTriangle();
    }
    if(leftNeighbor!=NULL) {
        BinTree[2]->rightNeighbor = leftNeighbor->GetRightTriangle();
    }
    if(topNeighbor!=NULL) {
        BinTree[3]->leftNeighbor = topNeighbor->GetBottomTriangle();
    }
    if(rightNeighbor!=NULL) {
        BinTree[3]->rightNeighbor = rightNeighbor->GetLeftTriangle();
    }
    BinTree[2]->baseNeighbor = BinTree[3];
    BinTree[3]->baseNeighbor = BinTree[2];
}

void PatchStatic::Uninitialize()
{
    //Release all allocated memroy
    bits.initialized = false;
    delete[] VarianceTree;
    if(BinTree!=NULL) {
        for(int i=2;i<numAllocatedTris;i++) {
            if(BinTree[i]!=NULL) {
                delete BinTree[i];
            }
        }
    }
    delete[] BinTree;
    delete[] render_list;
    delete[] index_list;
    delete[] render_patch_list;
    Patch::Uninitialize();
}

void PatchStatic::EvaluateConsistency()
{
    Patch::EvaluateConsistency();
    if(bits.consistent) bits.activated = true;
}

void PatchStatic::DeactivatePatch()
{
    //cannot deactivate this kind of patch
}

bool PatchStatic::Update()
{
    if(!bits.needsUpdate || bits.transition) return false;
    //Recursively recalculate the variance for all affected triangles
    RecursiveRecalculateVariance(2,0,treeDepth);
    RecursiveRecalculateVariance(3,0,treeDepth);
    bits.needsUpdate = false;
    return true;
}

int PatchStatic::RecursiveCreateSplit(Triangle *triangle, int thisLevel,
int stopLevel)
{
    int numTris;
    if(thisLevel<=stopLevel) {
        //Create the two child triangles for this triangle
        CreateSplit(triangle);
        int id = triangle->GetID();
        numTris = 2;
        //Recursively split the two newly created child triangles
        numTris+=RecursiveCreateSplit(BinTree[id*2],thisLevel+1,stopLevel);
        numTris+=RecursiveCreateSplit(BinTree[id*2+1],thisLevel+1,stopLevel);
    } else {
numTris = 0;
}
return numTris;

void PatchStatic::CreateSplit(Triangle* triangle)
{
    // Fetch the indexes for the vertices in the triangle to be split
    int top = triangle->GetVertex(0);
    int right = triangle->GetVertex(1);
    int left = triangle->GetVertex(2);
    // Calculate the midpoint to be used as top corner in the two child triangles
    int new_x = (int)((vertex_list[right].GetX()+vertex_list[left].GetX()) / 2) / hscale;
    int new_z = (int)((vertex_list[right].GetZ()+vertex_list[left].GetZ()) / 2) / hscale;
    if (new_x<0) new_x=new_x*(-1);
    if (new_z<0) new_z=new_z*(-1);
    // Create the index from the newly created coordinates
    int center = new_x+new_z*vRowCount;
    // Create the two child triangles
    BinTree[triangle->GetID()*2] = new Triangle(center,left,top,
        triangle->GetID()*2,&pHere);
    BinTree[triangle->GetID()*2+1] = new Triangle(center,top,right,
        triangle->GetID()*2+1,&pHere);
}

float PatchStatic::RecursiveCalculateVariance(Triangle *triangle, int thisLevel,
    int stopLevel)
{
    if (thisLevel>stopLevel) {
        // This is a leaf triangle which should have zero variance
        triangle->SetVariance(0.0);
    } else if (!bits.transition) {
        // Calculate the variance of this triangle, based on the difference in
        // Height of the actual midpoint on the hypotenuse, and an interpolated
        // Height of the midpoint of the hypotenuse. Then add the variance of the
        // Child triangle whose variance is the highest
        float v = ((vertex_list[triangle->GetVertex(1)].GetY() +
            vertex_list[triangle->GetVertex(2)].GetY()) / 2.0);
        v = v - vertex_list[BinTree[triangle->GetID()*2]->GetVertex(0)].GetY();
        if (v<0) v = v*(-1.0);
        int id = triangle->GetID();
        // Recursively calculate the variance for all child triangles
        float tmp1 = RecursiveCalculateVariance(BinTree[id*2],thisLevel+1, stopLevel);
        float tmp2 = RecursiveCalculateVariance(BinTree[id*2+1],thisLevel+1, stopLevel);
        triangle->SetVariance(v + MaxOf(tmp1,tmp2));
    } else {
        // Transition patches should also have zero variance
        triangle->SetVariance(0.0);
    }
    return triangle->GetVariance();
}

float PatchStatic::GetCenterHeight(void)
{
    return vertex_list[BinTree[4]->GetVertex(0)].GetY();
}

float PatchStatic::GetPatchVariance(void)
{
    return MaxOf(BinTree[2]->GetVariance(),BinTree[3]->GetVariance());
}

bool PatchStatic::FindLeafs(float x1, float z1, float x2, float z2, 
    Triangle** t1, Triangle** t2)
{
    // Find the two leaf triangles that fit x1,z1 -> x2,z2 store the result in t1 and t2
    *t1=NULL;
    // Find the first leaf
    *t1=FindLeaf2(BinTree[2],x1,z1,x2,z2,NULL,0);
    if (*t1==NULL) { *t1 = FindLeaf2(BinTree[3],x1,z1,x2,z2,NULL,0); *t2=NULL;
    return true;
}
// Find the second leaf
*t2=FindLeaf2(BinTree[2], x1, z1, x2, z2, *t1, 0);
if(*t2==NULL) *t2 = FindLeaf2(BinTree[3], x1, z1, x2, z2, *t1, 0);
if(*t1==NULL || *t2==NULL) return false;
return true;
}

Triangle* PatchStatic::FindLeaf2(Triangle* triangle, float x1, float z1, float x2, float z2, Triangle* notThis, int level)
{
    bool insideArea = false;
    int found = 0;
    Triangle* retTriangle = NULL;
    // Check if this triangle is inside the given rectangle
    if (x1 >= vertex_list[triangle->GetVertex(0)].p[0] ||
        x1 >= vertex_list[triangle->GetVertex(1)].p[0] ||
        x1 >= vertex_list[triangle->GetVertex(2)].p[0])
    {
        if (z1 >= vertex_list[triangle->GetVertex(0)].p[2] ||
            z1 >= vertex_list[triangle->GetVertex(1)].p[2] ||
            z1 >= vertex_list[triangle->GetVertex(2)].p[2])
        {
            if (x2 <= vertex_list[triangle->GetVertex(0)].p[0] ||
                x2 <= vertex_list[triangle->GetVertex(1)].p[0] ||
                x2 <= vertex_list[triangle->GetVertex(2)].p[0])
            {
                if (z2 <= vertex_list[triangle->GetVertex(0)].p[2] ||
                    z2 <= vertex_list[triangle->GetVertex(1)].p[2] ||
                    z2 <= vertex_list[triangle->GetVertex(2)].p[2])
                {
                    insideArea = true;
                }
            }
        }
    }
    if (level>treeDepth) {
        // This is a leaf triangle, investigate if it matches the criterion completely
        if (insideArea) {
            if (vertex_list[triangle->GetVertex(0)].p[0]>=x1 &&
                vertex_list[triangle->GetVertex(0)].p[0]<=x2)
            {
                found++;
            }
            if (vertex_list[triangle->GetVertex(1)].p[0]>=x1 &&
                vertex_list[triangle->GetVertex(1)].p[0]<=x2)
            {
                found++;
            }
            if (vertex_list[triangle->GetVertex(2)].p[0]>=x1 &&
                vertex_list[triangle->GetVertex(2)].p[0]<=x2)
            {
                found++;
            }
            if (vertex_list[triangle->GetVertex(0)].p[2]>=z1 &&
                vertex_list[triangle->GetVertex(0)].p[2]<=z2)
            {
                found++;
            }
            if (vertex_list[triangle->GetVertex(1)].p[2]>=z1 &&
                vertex_list[triangle->GetVertex(1)].p[2]<=z2)
            {
                found++;
            }
            if (vertex_list[triangle->GetVertex(2)].p[2]>=z1 &&
                vertex_list[triangle->GetVertex(2)].p[2]<=z2)
            {
                found++;
            }
            if (insideArea && found==6) {
                // Triangle is a perfect match, assure it is a legal finding
                if (triangle!=notThis) {
                    return triangle;
                }
            }
        }
    }
    else {
        // This is not a leaf triangle
        if (insideArea) {
            // Further investigation...
        }
    }
}

113
// The triangle intersects the rectangle, investigate its children
retTriangle = FindLeaf2(triangle->GetLeftChild(), x1, z1, x2, z2, notThis, level + 1);
// The triangle intersects the rectangle, investigate its children
if (retTriangle == NULL)
    retTriangle = FindLeaf2(triangle->GetRightChild(), x1, z1, x2, z2, notThis, level + 1);
}
return retTriangle;

bool PatchStatic::ActivatePatch(void)
{
    if (bits.consistent && !bits.activated)
        bits.activated = true;
    return false;
}

void PatchStatic::SetVariance(int id, float error)
{
    VarianceTree[id] = error;
}

float* PatchStatic::GetControlMap(void)
{
    return NULL;
}

bool PatchStatic::UpdateInFrustum(Camera* camera, unsigned char setValue)
{
    // Update the frustum status for this patch
    bits.inFrustum = 0;
    if (bits.activated)
    {
        // Create a sphere around the patch
        float v = GetPatchVariance();
        float c = GetCenterHeight();
        Vertex vC = Vertex((xstart + width / 2.0) * hscale, c, (zstart + width / 2.0) * hscale);
        Vertex c1 = Vertex(xstart * hscale, c + v / 2.0, zstart * hscale);
        Vertex c2 = Vertex((xstart + width) * hscale, c - v / 2.0, (zstart + width) * hscale);
        // Check if the sphere intersects the frustum
        bits.inFrustum = camera->SphereIntersectFrustum(vC, Distance(c1, c2) / 2.0);
        globalCamera = camera;
        // Recursively update the frustum status for all triangles in this patch
        RecursiveTriangleInFrustum(2, bits.inFrustum);
        RecursiveTriangleInFrustum(3, bits.inFrustum);
    }
    return !(bits.inFrustum == 0);
}

void PatchStatic::RecursiveTriangleInFrustum(int in_id, unsigned char parentIn)
{
    static Vertex vC;
    static Vertex c1;
    // Put a bit mask on the parentIn status, if it intersects the frustum, that is
    // not completely in or completely out, then test this triangle against the frustum
    if (parentIn & INTERSECT)
    {
        // Fetch the centerpoint in the sphere to check against the frustum
        vC = vertex_list[BinTree[in_id<<1]->GetVertex(0)];
        // Use one of the corner vertices of the triangle, add the variance, and use
        // the distance between the center point and the calculated point, as radius
        { case 0:
            // The triangle completely out, propagate this to the children
            RecursiveSetFrustumFlag(in_id, 0);
            return;
        case 2:
            // The triangle completely in, propagate this to the children
            RecursiveSetFrustumFlag(in_id, 2);
            return;
        }
        // This triangle lays on the frustum boundary, investigate if the BVH should
        // be checked further down the tree. Stop 3 levels above the leaf triangle
    }
//level, replacing 3 with 2, would stop 2 level up, etc.
if((in_id<<3)<numAllocatedTris) {
    //Frustum checking should be performed further down the tree.
    //If the triangle was not previously on the boundary, set its changed flag
    //so that its priority gets immediate updated.
    if(BinTree[in_id]->InFrustum() != 1)
        BinTree[in_id]->SetChanged();
    BinTree[in_id]->SetInFrustum(1);
    in_id <<= 1;
    //Recursively check the child triangles
    RecursiveTriangleInFrustum(in_id,parentIn);
    RecursiveTriangleInFrustum(in_id+1,parentIn);
} else {
    //No further frustum checks down the tree, only propagate the status
    RecursiveSetFrustumFlag(in_id,parentIn & STOREMASK);
    RecursiveSetFrustumFlag(in_id+1,parentIn & STOREMASK);
} else {
    //Triangle completely in our out, propagate this status down the tree
    RecursiveSetFrustumFlag(in_id, parentIn);
}

void PatchStatic::RecursiveSetFrustumFlag(int in_id, unsigned char parentIn) {
    //Check if the frustum state for this triangle have changes since the previous
    //frame, if true, update the status, and propagate further down the tree
    if(parentIn!=BinTree[in_id]->InFrustum()) {
        if(BinTree[in_id]->InFrustum()!=parentIn)
            BinTree[in_id]->SetChanged();
        BinTree[in_id]->SetInFrustum(parentIn);
        if((in_id<<1) < numAllocatedTris) {
            in_id = in_id<<1;
            RecursiveSetFrustumFlag(in_id, parentIn);
            RecursiveSetFrustumFlag(in_id+1, parentIn);
        } else {
            //This is a leaf triangle, check if it has a dynamic patch beneath it
            if(BinTree[in_id]->GetChildPatch()!=NULL) {
                //Dynamic patch found, update the frustum status for it.
                BinTree[in_id]->GetChildPatch()->UpdateInFrustum(globalCamera,parentIn);
            }
        }
    }
}

VertexExt* PatchStatic::GetVertex(int vID) {
    return &vertex_list[vID];
}

void PatchStatic::MidHypotenuse(int tId, int v1, int v2, Vertex& v) {
    //Calculate a midpoint on the hypotenuse of the given triangle.
    //If this triangle, has child triangles, then use the top vertex
    //in one of the children as mid point, otherwise calculate the midpoint.
    if((tId<<1)<numAllocatedTris)
        v = vertex_list[BinTree[tId<<1]->GetVertex(0)];
    else
        v = (vertex_list[v1] + vertex_list[v2])*0.5;
}

int PatchStatic::RenderVarr(int type) {
    int numTris = numRenderElements;
    //Tell OpenGL to use the specified vertex list, when rendering.
    glVertexPointer(3, GL_FLOAT, sizeof(VertexExt), &vertex_list[0].p);
    if(type==2 || type==3) {
        glTexCoordPointer(2, GL_FLOAT, sizeof(VertexExt), &vertex_list[0].uv);
        if(type==3) {
            //Tell OpenGL to use the specified normal vectors, when rendering
            glNormalPointer(GL_FLOAT, sizeof(VertexExt), &vertex_list[0].n);
        } else {
            glColor3f(1.0f,1.0f,1.0f);
        }
    } else {
        glColor3f(1.0f,1.0f,1.0f);
    }
    //Tell OpenGL to use the specified rendering list, when rendering with
    //vertex arrays. The vertices in the above given vertex list will be used
eglDrawElements(GL_TRIANGLES, numRenderElements*3, GL_UNSIGNED_INT, &render_list[0]);

//Initiate rendering for all active dynamic patches covered by this static patch
for(int i=0; i<numRenderPatch; i++) {
    if(render_patch_list[i]->IsActivated()) {
        //Dynamic patch found to be active, render it
        numTris += render_patch_list[i]->RenderVarr(type);
    } else {
        //Child patch is not active, remove it from the list of dynamic
        //patches to be rendered.
        render_patch_list[i] = render_patch_list[--numRenderPatch];
        i--;
    }
}

return numTris;

void PatchStatic::AddRenderParent(Triangle* parent, Triangle* leftChild, Triangle* rightChild) {
    //This function will add the given parent to the rendering list,
    //and remove its children
    if(leftChild == NULL || rightChild == NULL) {
        //root element needs to be inserted, add the vertices of it to render list
        render_list[numRenderElements*3] = parent->GetVertex(0);
        render_list[numRenderElements*3+1] = parent->GetVertex(1);
        render_list[numRenderElements*3+2] = parent->GetVertex(2);
        //Store which triangle is located at the index
        index_list[numRenderElements] = parent->GetID();
        //Tell the triangle, at what index in the rendering list, it is located
        parent->SetRenderIndex(numRenderElements);
        numRenderElements++;
        return;
    }

    if(leftChild->GetParentPatch() != pHere) {
        //special treatment, the children belongs to another patch, add the parent
        //to this rendering list, then initiate a removal of the children from
        //the other patches rendering list.
        render_list[numRenderElements*3] = parent->GetVertex(0);
        render_list[numRenderElements*3+1] = parent->GetVertex(2);
        render_list[numRenderElements*3+2] = parent->GetVertex(1);
        index_list[numRenderElements] = parent->GetID();
        parent->SetRenderIndex(numRenderElements);
        numRenderElements++;
        leftChild->GetParentPatch()->AddRenderParent(NULL, leftChild, rightChild);
    } else {
        //Remove the left child
        unsigned int li = leftChild->GetRenderIndex();
        if(li < numRenderElements-1) {
            render_list[li*3] = render_list[(numRenderElements-1)*3];
            render_list[li*3+1] = render_list[(numRenderElements-1)*3+1];
            render_list[li*3+2] = render_list[(numRenderElements-1)*3+2];
            index_list[li] = index_list[numRenderElements-1];
            BinTree[index_list[li]]->SetRenderIndex(li);
        }
        numRenderElements--;

        //Remove the right child
        unsigned int ri = rightChild->GetRenderIndex();
        if(ri < numRenderElements-1) {
            render_list[ri*3] = render_list[(numRenderElements-1)*3];
            render_list[ri*3+1] = render_list[(numRenderElements-1)*3+1];
            render_list[ri*3+2] = render_list[(numRenderElements-1)*3+2];
            index_list[ri] = index_list[numRenderElements-1];
            BinTree[index_list[ri]]->SetRenderIndex(ri);
        }
        numRenderElements--;

        //Add the parent
        render_list[numRenderElements*3] = parent->GetVertex(0);
        render_list[numRenderElements*3+1] = parent->GetVertex(2);
        render_list[numRenderElements*3+2] = parent->GetVertex(1);
        index_list[numRenderElements] = parent->GetID();
        parent->SetRenderIndex(numRenderElements);
        numRenderElements++;
    }

    //Merge triangle functionality
    MergeTriangle(parent, leftChild, rightChild);
}
void PatchStatic::AddRenderChildren(Triangle* parent, Triangle* leftChild, Triangle* rightChild) {
    unsigned int pi;
    if (parent == NULL || leftChild == NULL || rightChild == NULL) return;
    //special treatment, the child triangles are located in another patch
    //Remove the parent from this rendering list
    pi = parent->GetRenderIndex();
    if (pi != numRenderElements - 1) {
        //Insert the vertices of the triangle, to the rendering list.
        render_list[pi*3] = render_list[(numRenderElements-1)*3];
        render_list[pi*3+1] = render_list[(numRenderElements-1)*3+1];
        render_list[pi*3+2] = render_list[(numRenderElements-1)*3+2];
        //Add a reference, so it can be known which triangle resides at what
        //position in the rendering list.
        index_list[pi] = index_list[numRenderElements-1];
        //Tell the triangle, at what index in the rendering list, it is located
        BinTree[index_list[pi]]->SetRenderIndex(pi);
    }
    numRenderElements--;
    //Call the function in the other patch, to insert the children into its
    //rendering list.
    leftChild->GetParentPatch()->AddRenderChildren(NULL, leftChild, rightChild);
    //If the parent matches the other patches left triangle, then add the
    //patch containing the child triangles, to the list of patches to be
    //rendered. The right parent could be used as well, but if both are used,
    //Then the same patch would be rendered twice in the same frame.
    if (parent == leftChild->GetParentPatch()->GetPatchLeftParent())
        render_patch_list[numRenderPatch++] = leftChild->GetParentPatch();
    else {
        //Remove the parent, by inserting the left child at its position
        pi = parent->GetRenderIndex();
        render_list[pi*3] = leftChild->GetVertex(0);
        render_list[pi*3+1] = leftChild->GetVertex(2);
        render_list[pi*3+2] = leftChild->GetVertex(1);
        index_list[pi] = leftChild->GetID();
        leftChild->SetRenderIndex(pi);
        //Add the right child to the end of the rendering list
        render_list[numRenderElements*3] = rightChild->GetVertex(0);
        render_list[numRenderElements*3+1] = rightChild->GetVertex(2);
        render_list[numRenderElements*3+2] = rightChild->GetVertex(1);
        index_list[numRenderElements] = rightChild->GetID();
        rightChild->SetRenderIndex(numRenderElements);
        numRenderElements++;
    }
    //Split triangle
    SplitTriangle(parent, leftChild, rightChild);
}

bool PatchStatic::RecursiveRecalculateVariance(int tri_id, int thisLevel, int stopLevel) {
    //Recursively recompute the variance for triangles in need of a variance update
    bool hasChanged = false;
    if (thisLevel > stopLevel) {
        //This is a leaf triangle, check if its vertices have been modified
        //must check all three vertices, so that changed vertices get reset
        hasChanged |= vertex_list[BinTree[tri_id]->GetVertex(0)].HasChanged();
        hasChanged |= vertex_list[BinTree[tri_id]->GetVertex(1)].HasChanged();
        hasChanged |= vertex_list[BinTree[tri_id]->GetVertex(2)].HasChanged();
        //Is there any high resolution patches attached to us?
        if (BinTree[tri_id]->GetChildPatch() != NULL) {
            //Are they in need of an update, then update them.
            if ((BinTree[tri_id]->GetChildPatch())->NeedsUpdate()) {
                (BinTree[tri_id]->GetChildPatch())->Update();
                //Set the variance for this triangle to the variance of the child patch
                VarianceTree[tri_id] = BinTree[tri_id]->GetChildPatch()->GetPatchVariance();
                return true;
            }
        }
        return hasChanged;
    } else {
        ++thisLevel;
        //Recursively investigate the child triangles.
        hasChanged |= RecursiveRecalculateVariance(tri_id<<1, thisLevel, stopLevel);
        hasChanged |= RecursiveRecalculateVariance((tri_id<<1)+1, thisLevel, stopLevel);
        return hasChanged;
    }
}
If any of the leaf triangles descending from this triangle, have vertices that have been modified, then recalculate our variance.

```c++
if (hasChanged) {
    VarianceTree[tri_id] = ((vertex_list[BinTree[tri_id]->GetVertex(1)].p[1] + vertex_list[BinTree[tri_id]->GetVertex(2)].p[1]) / 2.0f);
    VarianceTree[tri_id] -= vertex_list[BinTree[tri_id<<1]->GetVertex(0)].p[1];
    if (VarianceTree[tri_id] < 0) VarianceTree[tri_id] *= (-1.0f);
    if (VarianceTree[tri_id<<1] > VarianceTree[(tri_id<<1)+1])
        VarianceTree[tri_id] += VarianceTree[tri_id<<1];
    else
        VarianceTree[tri_id] += VarianceTree[(tri_id<<1)+1];
    return true;
} else
    return false;
```
RoamNurb.h

#ifndef ROAMNURB_H
#define ROAMNURB_H

#include "RoamNurbBasis.h"
#include "Vertex.h"

class RoamNurb {
    public:
    // The nurb will create and be the owner of a new nurb basis
    RoamNurb(int in_numUPoints, int in_numVPoints, int in_surfaceOrderU,
              int in_surfaceOrderV, int in_width, float* in_controlPoints);
    // The nurb will use an already existing nurb basis
    RoamNurb(RoamNurbBasis* nBasis, float* in_controlPoints);

    // Fill in a specified array of vertices with information from the nurb basis
    void FillVertexList(VertexExt* vertex_list, int rotation, float xoff,
                        float zoff, float xz_w, float tex_step);

    // Get the number of vertices as a floating point value
    float GetNumVertices(void) { return (float)width; }

    // Get the number of control points (assuming same amount in u and v direction)
    int GetNumControlPointsUV(void);

    // Get the maximum error contained in the nurb object
    float GetMaxError(void);

    private:
    // A nurb basis object, several nurb objects can share the same basis
    RoamNurbBasis* nurbBasis;
    bool ownerOfBasis; // Is this nurb an owner of the basis
    float* controlMap; // Array of evenly spaced control points

    // Width of nurb, given in the number of maximum extractable vertices in side
    int width;
};

#endif

RoamNurb.cpp

#include "CommonInclude.h"
#include "RoamNurb.h"
#include "Helper.h"

RoamNurb::RoamNurb(int in_numUPoints, int in_numVPoints, int in_surfaceOrderU,
                    int in_surfaceOrderV, int in_width, float* in_controlPoints)
{
    width = in_width;
    ownerOfBasis = true;

    // Create a nurb basis
    nurbBasis = new RoamNurbBasis(in_numUPoints, in_numVPoints, in_surfaceOrderU,
                                  in_surfaceOrderV, in_width);

    controlMap = in_controlPoints;
}

RoamNurb::RoamNurb(RoamNurbBasis* nBasis, float* in_controlPoints)
{
    nurbBasis = nBasis;
    ownerOfBasis = false;

    width = nurbBasis->GetNumVertices();
    controlMap = in_controlPoints;
}

RoamNurb::~RoamNurb()
if(ownerOfBasis) delete nurbBasis;
}

void RoamNurb::FillVertexList(VertexExt* vertex_list, int rotation, float xoff, float zoff, float xz_w, float tex_step)
{
    nurbBasis->FillVertexBuffer(vertex_list, controlMap, rotation, xoff, zoff, xz_w, tex_step);
}

int RoamNurb::GetNumControlPointsUV(void)
{
    //using fixed cpoints u and v the same
    return nurbBasis->GetNumUPoints();
}

float RoamNurb::GetMaxError(void)
{
    static int N;
    static float totalerror;
    static float tmperror;
    static float u;
    static float v;
    static int i;
    static int j;
    static float Ay;
    static float By;
    static float Cy;
    static float Dy;
    static float lsidey;
    static float rsidey;
    static float NI;

    N = nurbBasis->GetNumUPoints();
    Ay = controlMap[0];
    By = controlMap[N-1];
    Cy = controlMap[N*(N-1)];
    Dy = controlMap[N*N-1];
    totalerror = 0.0;
    NI = (float)1.0f/(float)(N-1);
    //Interpolate across the corner points, subtract the actual control point
    //locations, sum up, and the error is calculated
    for(j=0; j<N; j++) {
        u = j*NI;
        lsidey = Ay*(1-u) + Cy*u;
        rsidey = By*(1-u) + Dy*u;
        for(i=0; i<N; i++) {
            v = i*NI;
            tmperror = controlMap[i+N*j];
            tmperror -= lsidey*(1-v);
            tmperror -= rsidey*v;
            if(tmperror>totalerror)
                totalerror = totalerror+tmperror;
        }
    }
    return totalerror;
}

RoamNurbBasis.h

//*****************************************************************************
/* RoamNurbBasis.h                                                            */
/* Author: Anders Dahlbom                                                     */
/* Last modified: 030603                                                      */
/*                                                                            */
/* Purpose: Class for capturing the basis functions for a nurb object.        */
/* The nurb basis is turned into a b-spline basis, as constant                */
/* weighting factors of 1.0 is used                                           */
/* The control points are assumed to be evenly spaced, hence                  */
/* the x, and z components of each control point is calculated.              */
/*                                                                            */
/* This class is based on the functionality described in the book            */
/* Focus on curves and surfaces.                                             */
/*******************************************************************************/
#ifndef ROAMNURBBASIS_H
#define ROAMNURBBASIS_H

#include "VertexExt.h"

class RoamNurbBasis {
public:
  RoamNurbBasis(int in_numUPoints, int in_numVPoints, int in_surfaceOrderU, int in_surfaceOrderV, int in_numVertices);
  ~RoamNurbBasis();

  //Set the knot vectors for this nurb basis
  void SetKnotVectors(void);

  //Define the basis function for this nurb basis
  void DefineBasisFunctions(void);

  //Fill a specified array of vertices with information from this basis
  //together with a set of control points
  void FillVertexBuffer(VertexExt* pVertices, float* cVertices, int rotation, float xoff, float zoff, float xz_w, float tex_step);

  //Get the number of control points in the U direction
  int GetNumUPoints(void) { return numUPoints; }

  //Get the number of control points in the V direction
  int GetNumVPoints(void) { return numVPoints; }

  //Get the surface order in the U direction
  int GetSurfaceOrderU(void) { return surfaceOrderU; }

  //Get the surface order in the V direction
  int GetSurfaceOrderV(void) { return surfaceOrderV; }

  //Get the maximum number of vertices that can be extracted in side
  int GetNumVertices(void) { return numVertices; }

  //Get the number of knot values in the U direction
  int GetNumUKnots(void) { return numUPoints+surfaceOrderU; }

  //Get the number of knot values in the V direction
  int GetNumVKnots(void) { return numVPoints+surfaceOrderV; }

  //Get the knot vector for the U direction
  float* GetKnotVectorU(void) { return knotVectorU; }

  //Get the knot vector for the V direction
  float* GetKnotVectorV(void) { return knotVectorV; }

private:
  int numUPoints; //The number of control points in the U direction
  int numVPoints; //The number of control points in the V direction
  int surfaceOrderU; //The surface order in the U direction
  int surfaceOrderV; //The surface order in the V direction
  float* knotVectorU; //The knot vector for the U direction
  float* knotVectorV; //The knot vector for the V direction
  int numVertices; //The maximum extractable vertices in side
  float* basisFunctionsU; //The basis functions in the U direction
  float* derivativeBasisU; //The derivative basis functions in the U direction
  float* basisFunctionsV; //The basis functions in the V direction
  float* derivativeBasisV; //The derivative basis functions in the V direction
  bool allocatedKnots; //Have the knot vector been allocated
  bool allocatedBasis; //Have the basis function values been allocated
};

#endif

RoamNurbBasis.cpp

#include "math.h"

#include "CommonInclude.h"
#include "RoamNurbBasis.h"
#include "Helper.h"

RoamNurbBasis::RoamNurbBasis(int in_numUPoints, int in_numVPoints, int in_surfaceOrderU, int in_surfaceOrderV, int in_numVertices)
{
  numUPoints = in_numUPoints;
  numVPoints = in_numVPoints;
  surfaceOrderU = in_surfaceOrderU;
  surfaceOrderV = in_surfaceOrderV;
  numVertices = in_numVertices;

  //Allocate storage for knot vectors, and set them up
  knotVectorU = new float[numUPoints+surfaceOrderU];

  //Allocate storage for basis functions, and set them up
  basisFunctionsU = new float[numUPoints+surfaceOrderU];
knotVectorV = new float[numVPoints+surfaceOrderV];
allocatedKnots = true;
SetKnotVectors();

// Allocate storage for basis functions, and calculate them
basisFunctionsU = new float[(int)numVertices*numUPoints*surfaceOrderU];
derivativeBasisU = new float[(int)numVertices*numUPoints*surfaceOrderU];
derivativeBasisV = new float[(int)numVertices*numUPoints*surfaceOrderV];
basisFunctionsV = new float[(int)numVertices*numVPoints*surfaceOrderV];
allocatedBasis = true;
DefineBasisFunctions();
#endif

void RoamNurbBasis::SetKnotVectors(void) {
    //Calculate the knot vectors
    //A knot vector must be N + k elements, where N is the number of
    //control points, and k the order
    int knotValue = 0;
    for(int i=0; i<numUPoints+surfaceOrderU; i++) {
        if(i>=surfaceOrderU && i<=numUPoints)
            knotValue++;
        knotVectorU[i] = (float)knotValue / (float)(numUPoints-surfaceOrderU+1);
    }
    knotValue = 0;
    for(int i=0; i<numVPoints+surfaceOrderV;i++) {
        if(i>=surfaceOrderV && i<=numVPoints)
            knotValue++;
        knotVectorV[i] = (float)knotValue / (float)(numVPoints-surfaceOrderV+1);
    }
}

void RoamNurbBasis::DefineBasisFunctions(void) {
    //Define basis and derivative basis functions for det u direction
    memset(derivativeBasisU, 0, numVertices*numUPoints*surfaceOrderU*sizeof(float));
    //Define the first order basis according to the Cox-de Boor recursive formula
    for(int iVertex=0; iVertex<numVertices; iVertex++) {
        float t = (float)iVertex / (float)numVertices;
        if (t >= 1.0f) t = 1.0f - EPSILON;
        for(int cPoint=0; cPoint<numUPoints; cPoint++) {
            if(t >= knotVectorU[cPoint] && t < knotVectorU[cPoint+1])
                basisFunctionsU[iVertex*(numUPoints*surfaceOrderU)+cPoint*(surfaceOrderU)] = 1.0f;
            else
                basisFunctionsU[iVertex*(numUPoints*surfaceOrderU)+cPoint*(surfaceOrderU)] = 0.0f;
        }
    }
    //Define the second, third, etc. orders according to the Cox-de Bor
    //recursive formulas
    for(int order=1; order<=surfaceOrderU; order++) {
        for(int cPoint=0; cPoint<numUPoints; cPoint++) {
            int cP0o = cPoint*surfaceOrderU;
            int cP0ls0 = (cPoint+1)*surfaceOrderU;
            for(int iVertex=0; iVertex<numVertices; iVertex++) {
                // Code continues here...
int inUsO = iVertex*numUPoints*surfaceOrderU;

float t = (float)iVertex / (float)(numVertices-1);

float Nikm1 = basisFunctionsU[inUsO + cpsO + order - 1];
float Nip1km1 = basisFunctionsU[inUsO + cpp1sO + order - 1];
float Dikm1 = derivativeBasisU[inUsO + cpsO + order - 1];
float Dip1km1 = derivativeBasisU[inUsO + cpp1sO + order - 1];
float xi = knotVectorU[cPoint];
float xikm1 = knotVectorU[cPoint + order - 1 + 1];
float xik = knotVectorU[cPoint + order + 1];
float xip1 = knotVectorU[cPoint + 1];

float FirstTermBasis;
float FirstTermDerivative;
if(fabs(xikm1 - xi) < EPSILON) {
    FirstTermBasis = 0.0f;
    FirstTermDerivative = 0.0f;
} else {
    FirstTermBasis = ((t - xi) * Nikm1) / (xikm1 - xi);
    FirstTermDerivative = (Nikm1 + ((t - xi) * Dikm1)) / (xikm1 - xi);
}

float SecondTermBasis;
float SecondTermDerivative;
if(fabs(xik - xip1) < EPSILON) {
    SecondTermBasis = 0.0f;
    SecondTermDerivative = 0.0f;
} else {
    SecondTermBasis = ((xik - t) * Nip1km1) / (xik - xip1);
    SecondTermDerivative = (((xik - t)*Dip1km1) - Nip1km1) / (xik - xip1);
}
basisFunctionsU[inUsO+cpsO+order] = FirstTermBasis + SecondTermBasis;
derivativeBasisU[inUsO+cpsO+order] =
    FirstTermDerivative + SecondTermDerivative;
}

//Define basis and derivative basis functions for det v direction,
//again, using the Cox-de Boor recursive formulas.
memset(derivativeBasisV, 0, numVertices*numVPoints*surfaceOrderV*sizeof(float));

//Define the first order basis
for(int iVertex=0; iVertex<numVertices; iVertex++) {
    float t = (float)iVertex / (float)numVertices;
    if(t == 1.0f) t = 1.0f - EPSILON;
    for(int cPoint=0; cPoint<numVPoints; cPoint++) {
        int currentIndexCp0 = iVertex*(numVPoints*surfaceOrderV) + cPoint*(surfaceOrderV);
        if(t >= knotVectorV[cPoint] && t < knotVectorV[cPoint+1])
            basisFunctionsV[iVertex*(numVPoints*surfaceOrderV)+cPoint*(surfaceOrderV)]= 1.0f;
        else
            basisFunctionsV[iVertex*(numVPoints*surfaceOrderV)+cPoint*(surfaceOrderV)]= 0.0f;
    }
}
//Define the other orders
for(int order=1; order<surfaceOrderV; order++) {
    for(int cPoint=0; cPoint<numVPoints; cPoint++) {
        int Cp0 = cPoint*surfaceOrderV;
        int Cp1 = (cPoint+1)*surfaceOrderV;
        for(int iVertex=0; iVertex<numVertices; iVertex++) {
            float t = (float)iVertex / (float)(numVertices-1);
            int currentIndexCp0 = iVertex*(numVPoints*surfaceOrderV);
            int currentIndexCp1 = currentIndexCp0 + Cp1;
            int currentIndexCp0p0 = currentIndexCp0 + Cp0;
            float Nikm1 = basisFunctionsV[currentIndexCp0p0 + order - 1];
            float Nip1km1 = basisFunctionsV[currentIndexCp1 + order - 1];
            float Dikm1 = derivativeBasisV[currentIndexCp0p0 + order - 1];
            float Dip1km1 = derivativeBasisV[currentIndexCp1 + order - 1];
            float xi = knotVectorV[cPoint];
            float xikm1 = knotVectorV[cPoint + order - 1 + 1];
            float xik = knotVectorV[cPoint + order + 1];
            float xip1 = knotVectorV[cPoint + 1];
            float FirstTermBasis;
            float FirstTermDerivative;
            if(fabs(xikm1 - xi) < EPSILON) {
                FirstTermBasis = ((t - xi) * Nikm1) / (xikm1 - xi);
                FirstTermDerivative = (Nikm1 + ((t - xi) * Dikm1)) / (xikm1 - xi);
            } else {
                FirstTermBasis = 0.0f;
                FirstTermDerivative = 0.0f;
            }
        }
    }
}
void RoamNurbBasis::FillVertexBuffer(VertexExt* pVertices, float* cVertices, int rotation, float xoff, float zoff, float xz_w, float tex_step)
{
    float NumeratorMultiplier, Denominator, CurrentU, CurrentV, xmul, zmul;
    int newU, newV, currentVertex, ControlIndex, U, V, UStep, VStep;
    float DenominatorDU, DenominatorDV, nlen;
    Vertex dPdU, dPdV, NormalV;
    float UConst, VConst;
    //Calculate all vertices in the 2d matrix of vertices
    for(U=0; U<numVertices; U++) {
        for(V=0; V<numVertices; V++) {
            NumeratorMultiplier = 0.0f;
            Denominator = 0.0f;
            DenominatorDU = 0.0f;
            DenominatorDV = 0.0f;
            CurrentU = (float)U / (float)(numVertices-1);
            CurrentV = (float)V / (float)(numVertices-1);
            //Calculate the rotation
            newU = U;
            newV = V;
            if(rotation == 1) {
                newU = V;
                newV = numVertices-1-U;
            } else if(rotation == 2) {
                newU = numVertices-1-U;
                newV = numVertices-1-V;
            } else if(rotation == 3) {
                newU = numVertices-1-V;
                newV = U;
            }
            currentVertex = newU+newV*numVertices;
            //Reset the vertex
            pVertices[currentVertex].p[0] = 0.0f;
            pVertices[currentVertex].p[1] = 0.0f;
            pVertices[currentVertex].p[2] = 0.0f;
            //Reset the tangents
            dPdU.p[0] = 0.0f;
            dPdU.p[1] = 0.0f;
            dPdU.p[2] = 0.0f;
            dPdV.p[0] = 0.0f;
            dPdV.p[1] = 0.0f;
            dPdV.p[2] = 0.0f;
            //Calculate step sizes for horizontal plane
            UConst = xz_w / (float)(numUPoints-1);
            VConst = xz_w / (float)(numVPoints-1);
            xmul = xoff;
            //Calculate the vertex, by summing together the effect of each control point
            for(UStep = 0; UStep < numUPoints; UStep++) {
                zmul = zoff;
                for(VStep = 0; VStep < numVPoints; VStep++) {
                    //Check if the current control points affects the current vertex
                    if(knotVectorU[UStep] <= CurrentU && CurrentU <=
                        knotVectorU[UStep + surfaceOrderU] &&
                        knotVectorV[VStep] <= CurrentV && CurrentV <=
                        knotVectorV[VStep + surfaceOrderV]) {
                        //Calculate the basis functions and derivatives
                        float FirstTermBasis = 0.0f;
                        float FirstTermDerivative = 0.0f;
                        else {
                            FirstTermBasis = ((t - xi) * Nikm1) / (xikm1 - xi);
                            FirstTermDerivative = (Nikm1 + ((t - xi) * Dikm1)) / (xikm1 - xi);
                        }
                        float SecondTermBasis;
                        float SecondTermDerivative;
                        if(fabs(xik - xip1) < EPSILON) {
                            SecondTermBasis = 0.0f;
                            SecondTermDerivative = 0.0f;
                        } else {
                            SecondTermBasis = ((xik - t) * Nip1km1) / (xik - xip1);
                            SecondTermDerivative = (((xik - t)*Dip1km1) - Nip1km1) / (xik - xip1);
                        }
                        basisFunctionsV[currentIndexCp0+order] = FirstTermBasis + SecondTermBasis;
                        derivativeBasisV[currentIndexCp0+order] =
                            FirstTermDerivative + SecondTermDerivative;
                    }
                }
            }
        }
    }
}
knotVectorV[VStep + surfaceOrderV])
{
    //Calculate which control points we are dealing with
    ControlIndex = UStep + VStep * numVPoints;
    //Multiply the basis values for the U, and V, and add to the
    //overall divisor
    Denominator += 1.0 * basisFunctionsU[U*numUPoints*surfaceOrderU +
    UStep*surfaceOrderU + surfaceOrderU - 1] *
    basisFunctionsV[V*numVPoints*surfaceOrderV +
    VStep*surfaceOrderV + surfaceOrderV - 1];

    //Equally for the derivative divisors
    DenominatorDU += 1.0 * derivativeBasisU[U*numUPoints*surfaceOrderU +
    UStep*surfaceOrderU + surfaceOrderU - 1] *
    basisFunctionsV[V*numVPoints*surfaceOrderV +
    VStep*surfaceOrderV + surfaceOrderV - 1];
    DenominatorDV += 1.0 * basisFunctionsU[U*numUPoints*surfaceOrderU +
    UStep*surfaceOrderU + surfaceOrderU - 1] *
    derivativeBasisV[V*numVPoints*surfaceOrderV +
    VStep*surfaceOrderV + surfaceOrderV - 1];

    //Calculate the numerator for the current control point
    NumeratorMultiplier = 1.0 *
    basisFunctionsU[U*numUPoints*surfaceOrderU +
    UStep*surfaceOrderU + surfaceOrderU - 1] *
    basisFunctionsV[V*numVPoints*surfaceOrderV +
    VStep*surfaceOrderV + surfaceOrderV - 1];

    //Add the numerator times the current control point position,
    //to the vertex
    pVertices[currentVertex].p[0] += NumeratorMultiplier * xmul;
    pVertices[currentVertex].p[1] += NumeratorMultiplier *
    cVertices[ControlIndex];

    //Similar behaviour for the derivatives
    NumeratorMultiplier = 1.0 *
    derivativeBasisU[U*numUPoints*surfaceOrderU +
    UStep*surfaceOrderU + surfaceOrderU - 1] *
    basisFunctionsV[V*numVPoints*surfaceOrderV +
    VStep*surfaceOrderV + surfaceOrderV - 1];
    dPdU.p[0] += NumeratorMultiplier * xmul;
    dPdU.p[1] += NumeratorMultiplier * cVertices[ControlIndex];
    NumeratorMultiplier = 1.0 *
    basisFunctionsU[U*numUPoints*surfaceOrderU +
    UStep*surfaceOrderU + surfaceOrderU - 1] *
    derivativeBasisV[V*numVPoints*surfaceOrderV +
    VStep*surfaceOrderV + surfaceOrderV - 1];
    dPdV.p[0] += NumeratorMultiplier * xmul;
    dPdV.p[1] += NumeratorMultiplier * cVertices[ControlIndex];
    dPdV.p[2] += NumeratorMultiplier * zmul;
}

}  //End of the function

//Divide the summed numerator values with the summed denominator values
//for the tangents
dPdU.p[0] = (dPdU.p[0] / Denominator) - (pVertices[currentVertex].p[0] *
    DenominatorDU / (Denominator * Denominator));
dPdV.p[0] = (dPdV.p[0] / Denominator) - (pVertices[currentVertex].p[0] *
    DenominatorDV / (Denominator * Denominator));
    DenominatorDU / (Denominator * Denominator));
    DenominatorDV / (Denominator * Denominator));
    DenominatorDU / (Denominator * Denominator));
    DenominatorDV / (Denominator * Denominator));

zmul += VConst;
}

xmul += UConst;

}  //End of the function

}  //End of the class
//Divide the vertex position by the summed denominator value
pVertices[currentVertex].p[0] /= Denominator;
pVertices[currentVertex].uv[0] = pVertices[currentVertex].p[0]*tex_step;

//create the normals from dPdU,dPdV, by taking cross product
NormalV = dPdV.CrossProduct(dPdU);
nlen = NormalV.LengthOf();
if(nlen!=0) NormalV /= nlen;
pVertices[currentVertex].n[0] = NormalV.p[0];
pVertices[currentVertex].n[1] = NormalV.p[1];
} }
}
}

Terrain.h

/***************************************************************************/
/* Terrain.h                                                                */
/* Author: Anders Dahlbom                                                   */
/* Last modified: 030603                                                     */
/*                                                                            */
/* Purpose: This class is the glue of the terrain rendering system.          */
/* It creates all static patches, and their corresponding vertices.          */
/* It also manages all dynamic patches.                                     */
/*                                                                            */
/* During run-time, this class is used to triangulate the terrain            */
/* according to an updated view position and rotation.                      */
/* After the terrain have been triangulated, it should be rendered.         */
/* This class call the rendering functions in all the static patches,       */
/* which in turn calls the rendering functions of all dynamic patches       */
/* that should be rendered.                                                 */
/* This class creates and binds the texture for each static patch.           */
/*                                                                            */
/***************************************************************************/

#ifndef TERRAIN_H
#define TERRAIN_H
#include "CommonInclude.h"
#include "Triangle.h"
#include "Vertex.h"
#include "VertexExt.h"
#include "TriangleQue.h"
#include "QueItem.h"
#include "Camera.h"
#include "PatchStatic.h"
#include "PatchDynamic.h"
#ifdef LOGSTAT
//If logging of statistics is used, the statistics functionallity is included
#include "RStatistics.h"
#endif

//Bitfield with the current settings for rendering
typedef struct settings {
  unsigned char lighting : 1;
} SETTINGS;

class Terrain {
public:
  Terrain();
  ~Terrain();
  //Initialize the terrain, given a height map and texture name
  int Initialize(float* hmap, int in_width, int in_patchWidth, float in_hscale,
                 float in_vscale, char* textureName);
  //Render the terrain, given a rendering type
  int RenderTerrain(int type);
  //Triangulate the terrain, given a camera and an accuracy level
  void TriangulateTerrain(Camera* camera, float acc);
  //Is the terrain correctly initialized?
  bool IsInitialized();
};
void Uninitialize();

void Deform(Vertex p, float radius, float effect);

float GetHeight(float xp, float yp);

void SetNoExtension();

void SetLightFlag(bool f);

void OutputStatistics(char* filename);

void PrintStatus(char* filename);

private:

void Triangulate();

void MergeLoop();

void SplitLoop();

void RecursiveSplit(Triangle* triangle, bool& failed);

TriangleQue* RecalculateSplitQue(TriangleQue* que);

TriangleQue* RecalculateMergeQue(TriangleQue* que);

void InDiamondAdd(Triangle* triangle);

void InsertToSplitQue(Triangle* tri1, Triangle* tri2, float priority, bool spliting);

void InsertToMergeQue(Triangle* tri1, Triangle* tri2, float priority);

void RemoveFromSplitQue(Triangle* tri, bool spliting);

void RemoveFromMergeQue(Triangle* tri);

void CreateDynamicPatch(int mx, int mz, int x, int z, int w);

bool DisplaceVertex(Vertex* pTarget, Vertex Source, float radius, float effect);

AddToActivationList(Patch* p);

AddToDeactivationList(Patch* p);

ProcessActivationList(void);

ProcessDeactivationList(void);

float CalcPriority(Triangle* triangle, Camera* camera);

float CalcPrioritySimple(Triangle* triangle, Camera* camera);

float CalcPriorityByDistance(Triangle* triangle, Camera* camera);

int width;  // The width of the terrain

bool initialized;  // Have the terrain been initialized

long framenumber;  // The current frame number

PatchDynamic** nurbPatches;  // Array to reach all dynamic patches

int numNurbPatches;

int numCreatedDyn;  // Number of currently created dynamic patches

Patch** activationList;}
int numWaitingActivation; //Number of dynamic patches waiting activation
int maxWaitingActivation; //Maximum number of patches in the activation list
int maxAttachments; //The max number of attachments allowed in each frame
Patch** deactivationList; //List of dynamic patches waiting deactivation
int numWaitingDeactivation; //Number of dynamic patches waiting deactivation
int maxWaitingDeactivation; //Maximum number of items in the deactivation list
PatchStatic** patchlist; //List of static patches
int numPatches; //Number of static patches
int numPatchesPerSide; //Number of static patches in each direction
int patchWidth; //The width of each static patch
Camera* worldCamera; //The current camera to triangulate according to
float accuracy; //The current accuracy level
TriangleQue* mergeQue; //The merge queue
TriangleQue* splitQue; //The split queue
GLuint* texture; //Array of named textures
float hscale; //Horizontal multiplicator
float vscale; //Vertical multiplicator

//An nurb (b-spline) basis object, using 5 control points, maximum
//17x17 vertices can be extracted from it
RoamNurbBasis* nurbBase_5x17;

//Bintree manager for the dynamic patches, the resolution of each dynamic
//bintree is 16, resulting in 2*16*16 leaf triangles
BinTreeManager* binTreeManager16;

float squaredThreshold; //The accuracy squared
bool noExtension; //Should dynamic patches be introduced if needed
SETTINGS setup; //The current rendering settings

#endif

Terrain.cpp

#include <math.h>

#ifdef LOGSTAT
#include <stdio.h>
#endif

#include "Terrain.h"
#include "Camera.h"
#include "Vertex.h"
#include "Helper.h"

#define MAX_SPLITS 2000 //Maximum number of split per frame
#define MAX_MERGES 2000 //Maximum number of merges per frame
#define MAX_LOOPS 2000 //Maximum number of split-, and merge loops per frame

Terrain::Terrain()
{
    //Reset all variables
    width = 0;
    initialized = false;
    framenumber = 0;
    nurbPatches = NULL;
    numNurbPatches = 0;
}
patchlist = NULL;
numPatches = 0;
numPatchesPerSide = 0;
patchWidth = 0;
worldCamera = NULL;
accuracy = 0.0f;
mergeQue = NULL;
splitQue = NULL;
texture = NULL;
hscale = 1.0f;
vscale = 1.0f;
nurbBase_5x17 = NULL;
binTreeManager16 = NULL;
activationList = NULL;
numWaitingActivation = 0;
maxWaitingActivation = 0;
deactivationList = NULL;
umWaitingDeactivation = 0;
maxWaitingDeactivation = 0;
maxAttachments = 20;
noExtension = false;
setup.lighting = false;

#ifdef LOGSTAT
   //If logging statistics, reset those variables too
   outputStatistics = false;
   statistics = NULL;
   rtime=0;    ttime=0;    dtime=0;    wetime = 0;
ttris=0;    rtris=0;    numdeform=0;
#endif
}

Terrain::~Terrain()
{
#ifdef LOGSTAT
   //If we are logging statistics, remove the log object
   if(outputStatistics) {
      statistics->PrintOut();
      delete statistics;
   }
#endif
   if(initialized) Uninitialize();
}

int Terrain::Initialize(float* hmap, int in_width, int in_patchWidth, float in_hscale, float in_vscale, char* textureName)
{
#ifdef LOGSTAT
   //Track time
   QueryPerformanceCounter(&time1);
#endif
   width = in_width;
   hscale = in_hscale;
   vscale = in_vscale;
   patchWidth = in_patchWidth;
   framenumber = 0;
   if(hmap==NULL || in_width==0 || in_patchWidth==0) {
      return 0;
   }

   //Create all static vertices
   vertex_list = new VertexExt[width*width];

   //Setup all static vertices
   Vertex v1, v2, vt1, vt2 NormalV;
   for(int j=0; j<width; j++) {
      for(int i=0; i<width; i++) {
         //Set the position in world space
         vertex_list[i+j*width].Set(i*hscale,hmap[i+j*width]*vscale,j*hscale);
         //Set the texture coordinates
         vertex_list[i+j*width].uv[0] = ((float)i)*1.0f/(float)patchWidth;
         vertex_list[i+j*width].uv[1] = ((float)j)*1.0f/(float)patchWidth;
         //Calculate the normal
         if(i==0 || j==0) {
            NormalV.Set(0.0f,1.0f,0.0f);
         } else {
            //Rest of the code...
         }
      }
   }

   //Setup lighting and rendering...
   //...
vt1 = (Vertex)vertex_list[i+j*width];
vt2 = (Vertex)vertex_list[i-1+j*width];
v1 = vt1 - vt2;
vt2 = (Vertex)vertex_list[i+(j-1)*width];
v2 = vt1 - vt2;
NormalV = CreateNormal(v1,v2);
}
//Set the normal
vertex_list[i+j*width].n[0] = NormalV.p[0];
vertex_list[i+j*width].n[1] = NormalV.p[1];
vertex_list[i+j*width].n[2] = NormalV.p[2];
}
//Calculate how many static patches there will be
numPatchesPerSide = (width-1)/patchWidth;
numPatches = numPatchesPerSide*numPatchesPerSide;
if(numPatchesPerSide==0 || numPatches==0) {
  Uninitialize();
  return 0;
}
//Create all static patches
PatchStatic::SetVertexRowCount(width);
patchlist = new PatchStatic*[numPatches];
for(int i=0; i<numPatchesPerSide; i++) {
  for(int j=0; j<numPatchesPerSide; j++) {
    patchlist[j*numPatchesPerSide+i] = new PatchStatic(j*(patchWidth),
i*(patchWidth),
patchWidth,
i*numPatchesPerSide+j,
false,
hscale,
vscale,
vertex_list);
  }
}
//Setup their neighboring connections
for(int i=0; i<numPatchesPerSide; i++) {
  for(int j=0; j<numPatchesPerSide; j++) {
    Patch *tp=NULL, *bp=NULL, *lp=NULL, *rp=NULL;
    if(i>0) {
      tp = patchlist[(i-1)*numPatchesPerSide+j];
    }
    if(i<numPatchesPerSide-1) {
      bp = patchlist[(i+1)*numPatchesPerSide+j];
    }
    if(j>0) {
      lp = patchlist[i*numPatchesPerSide+j-1];
    }
    if(j<numPatchesPerSide-1) {
      rp = patchlist[i*numPatchesPerSide+j+1];
    }
    patchlist[i*numPatchesPerSide+j]->Initialize(tp,bp,lp,rp);
  }
}
//Evaluate the static patches. (not useful at this stage)
for(int i=0; i<numPatches; i++) {
  patchlist[i]->EvaluateConsistency();
}
//Generate and load textures
texture = new GLuint[numPatches];
glGenTextures(numPatches, texture);
if(!LoadTexture(textureName,texture,numPatches)) {
  Uninitialize();
  return 0;
}
//Create an array to contain all dynamic patches (nurb (b-spline) patches)
nurbPatches = new PatchDynamic*[numNurbPatches];
//Reset all positions in the array
for(int i=0; i<numNurbPatches; i++)
  nurbPatches[i] = NULL;
// Create a split and a merge queue
splitQue = new TriangleQue();
mergeQue = new TriangleQue();

#ifdef LOGSTAT
    ttris = 0;
#endif

// Insert root triangles to split queue, and rendering lists
for (int i = 0; i < numPatches; i++) {
    InsertToSplitQue(patchlist[i]->GetLeftTriangle(), NULL, 0.0, false);
    InsertToSplitQue(patchlist[i]->GetRightTriangle(), NULL, 0.0, false);
    patchlist[i]->AddRenderParent(patchlist[i]->GetLeftTriangle(), NULL, NULL);
    patchlist[i]->AddRenderParent(patchlist[i]->GetRightTriangle(), NULL, NULL);
#ifdef LOGSTAT
    ttris+=2;
#endif
}

// Create a nurb (b-spline) nurb basis, which is based on 5 control points
// in each parametric direction, and who's maximum surface order is 4 in each
// parametric direction, A maximum of 17x17 vertices can be extracted from
// the basis functions
nurbBase_5x17 = new RoamNurbBasis(5, 5, 4, 4, 17);

// Create a bintree manager which creates 500 bintrees of 2*16*16 leaf triangles
binTreeManager16 = new BinTreeManager(500, 16);

// Set all dynamic patches to use the newly created bintree manager
PatchDynamic::SetBinTreeManager(binTreeManager16);

// Calculate the texture stepping to be used in all dynamic patches
PatchDynamic::SetTextureStepping(1.0f / ((float)patchWidth * (float)hscale));

#ifdef MEMCOUNTER
    // Add the amount of memory consumed, to the global memory tracker
    allocatedMem += sizeof(PatchDynamic*) * numNurbPatches;
    allocatedMem += sizeof(TriangleQue)*2;
    allocatedMem += sizeof(BinTreeManager);
    allocatedMem += sizeof(RoamNurbBasis);
    allocatedMem += sizeof(GLuint)*numPatches;
    allocatedMem += sizeof(VertexExt)*width*width;
    allocatedMem += sizeof(PatchStatic*)*numPatches;
    allocatedMem += sizeof(PatchStatic)*numPatches;
#endif

initialized = true;  // We are initialized and ready
return (2*(2*(width-1)*(width-1)-1));  // Return the total number of triangles
}

void Terrain::Uninitialize(void) {
    // Deallocate all allocated memory
    if (splitQue != NULL) delete splitQue;
    if (mergeQue != NULL) delete mergeQue;
    if (texture != NULL) delete[] texture;
    if (activationList != NULL) delete[] activationList;
    if (deactivationList != NULL) delete[] deactivationList;
    if (nurbPatches != NULL) {
        for (int i = 0; i < numNurbPatches; i++) {
            if (nurbPatches[i] != NULL) {
                nurbPatches[i]->Uninitialize();
                delete nurbPatches[i];
            }
        }
        delete[] nurbPatches;
    }
    delete nurbBase_5x17;
    delete binTreeManager16;
    if (patchlist != NULL) {
        for (int i = 0; i < numPatches; i++) {
            if (patchlist[i] != NULL) {
                patchlist[i]->Uninitialize();
                delete patchlist[i];
            }
        }
        delete[] patchlist;
    }
    delete[] vertex_list;
TriangleQue* Terrain::RecalculateSplitQue(TriangleQue* que)
{
    // Run through all elements in que and recalculate error
    bool reinsert;
    Triangle* tmpTriangle;
    float t = que->GetThreshold();
    float c = (float)t/(float)10.0;
    if(que!=NULL) {
        TriangleQue* newQue = new TriangleQue();
        newQue->SetThreshold(t);
        QueItem* nextItem = que->RemoveBack();
        while(nextItem!=NULL) {
            reinsert = false;
            tmpTriangle = nextItem->data1;
            if(tmpTriangle!=NULL)
                if(!tmpTriangle->IsDirtySplit())
                    reinsert = true;
            if(reinsert) {
                // Add aging priority, or if immediate recalculation is needed, set
                // value above recalculation limit
                if(tmpTriangle->HasChanged()) {
                    nextItem->itemPriority = t;
                    tmpTriangle->SetUnchanged();
                } else
                    nextItem->itemPriority += c;
                // If priority above recalculation limit, recalculate
                if(nextItem->itemPriority >= t)
                    nextItem->itemPriority = CalcPriority(tmpTriangle,worldCamera);
            } else {
                delete nextItem;
            }
            nextItem = que->RemoveBack();
        }
        delete que;
        return newQue;
    }
    return NULL;
}

TriangleQue* Terrain::RecalculateMergeQue(TriangleQue* que)
{
    // Run through all elements in que and recalculate error
    // Not using aging priority in the merge queue, as there is normally not
    // very many elements in it, and we would also like to merge diamonds
    // as soon as possible, when needed.
    bool reinsert;
    Triangle* tmpTriangle;
    Triangle* tmpTriangle2;
    float priol,prio2;
    float t = que->GetThreshold();
    if(que!=NULL) {
        TriangleQue* newQue = new TriangleQue();
        newQue->SetThreshold(t);
        QueItem* nextItem = que->RemoveBack();
        while(nextItem!=NULL) {
            reinsert = false;
            tmpTriangle = nextItem->data1;
            tmpTriangle2 = nextItem->data2;
            if(tmpTriangle!=NULL) {
                if(!tmpTriangle->IsDirtyMerge())
                    reinsert = true;
            } else {
                delete nextItem;
            }
            nextItem = que->RemoveBack();
        }
        delete que;
        return newQue;
    }
    return NULL;
}
if(tmpTriangle2!=NULL) {
    prio2 = CalcPriority(tmpTriangle2,worldCamera);
} else {
    delete nextItem;
    nextItem = que->RemoveBack();
}
delete que;
return newQue;

} else {
    delete nextItem;
    nextItem = que->RemoveBack();
}
delete que;
return NULL;

} else {
    delete nextItem;
    nextItem = que->RemoveBack();
}
delete que;
return NULL;

void Terrain::Triangulate(void)
{
    if(initialized) {
        //Reset the counter for the number of attachments made this frame
        PatchDynamic::numAttachmentsNow = 0;
        //If there are dynamic patches waiting to be attached, try now
        if(numWaitingActivation>0)
            ProcessActivationList();
    #ifdef LOGSTAT
        //Track the time
        LARGE_INTEGER weutime1,weutime2;
        QueryPerformanceCounter(&weutime1);
    #endif
        //Update the frustum status for all static patches, and update the patches
        //which are in the frustum, if they need it.
        for(int i=0; i<numPatches; i++) {
            if(patchlist[i]->UpdateInFrustum(worldCamera,0)) {
                patchlist[i]->Update();
            }
        }
    #ifdef LOGSTAT
        //Track the time
        QueryPerformanceCounter(&weutime2);
        wetime = ElapsedTime(weutime1,weutime2);
    #endif
        if(splitQue==NULL) {
            //If there is no split queue, create new queues now, and insert
            //root elements to them
            if(mergeQue!=NULL) delete mergeQue;
            splitQue = new TriangleQue();
            mergeQue = new TriangleQue();
        #ifdef LOGSTAT
            ttris = 0;
        #endif
            for(int i=0; i<numPatches; i++) {
                InsertToSplitQue(patchlist[i]->GetLeftTriangle(), NULL,
                    CalcPriority(patchlist[i]->GetLeftTriangle(),worldCamera),
                    false);
                InsertToSplitQue(patchlist[i]->GetRightTriangle(), NULL,
                    CalcPriority(patchlist[i]->GetRightTriangle(),worldCamera),false);
                patchlist[i]->ResetRenderList();
                patchlist[i]->AddRenderParent(patchlist[i]->GetLeftTriangle(),NULL,NULL);
                patchlist[i]->AddRenderParent(patchlist[i]->GetRightTriangle(),NULL,NULL);
            }
        #ifdef LOGSTAT
            ttris += 2;
        #endif
            ttris -= merges + splits;
        #endif
    }
} else {
    //update priorities in split and merge queues
    splitQue->SetThreshold(accuracy);
    mergeQue->SetThreshold(accuracy);
    splitQue = RecalculateSplitQue(splitQue);
    mergeQue = RecalculateMergeQueue(mergeQue);
}

//Process merge queue
int merges = MergeLoop();

//Process split queue
int splits = SplitLoop();
# ifdef LOGSTAT
    ttris -= ttris - merges + splits;
# endif

// Process deactivation list
if (numWaitingDeactivation>0)
    ProcessDeactivationList();
}

int Terrain::MergeLoop(void)
{
    bool proceed = false;
    bool failed;
    int mergecounter = 0;
    int loopcounter = 0;
    Triangle* tril;
    Triangle* tri2;
    bool oktocheck;
    float prio;
    QueItem* nextItem = mergeQue->GetBack();
    if (nextItem==NULL) proceed = true;
    // Loop through the merge loop until no more mergable diamonds, or to many
    // merges, or passes in the loop
    while (!proceed) {
        // Check for a dirty merge flag in triangles of diamond
        tril = nextItem->data1;
        tri2 = nextItem->data2;
        oktocheck = true;
        if (tril!=NULL)
            if (tril->IsDirtyMerge())
                oktocheck = false;
        if (tri2!=NULL)
            if (tri2->IsDirtyMerge())
                oktocheck = false;
        if (oktocheck) {
            // It was not dirty, investigate its priority
            prio = nextItem->itemPriority;
            if (prio<accuracy) {
                failed = false;
                // Below, merge it
                mergecounter+=RecursiveMerge(tri1,tri2,failed);
            } else {
                proceed = true;
            }
            if (mergecounter>MAX_MERGES) {
                proceed = true;
            }
            loopcounter++;
            if (loopcounter>MAX_LOOPS) {
                proceed = true;
            }
        } else {
            delete mergeQue->RemoveBack();
        }
        if (!proceed) nextItem = mergeQue->GetBack();
    }
    return mergecounter;
}

int Terrain::SplitLoop(void)
{
    bool proceed = false;
    bool failed;
    int splitcounter = 0;
    int loopcounter = 0;
    Triangle* tri;
    bool oktocheck;
    QueItem* nextItem = splitQue->RemoveFront();
    if (nextItem==NULL) proceed = true;
    // Continue to loop until no more triangles above the limit, or until
    // the maximum number of splits or loops have been performed
    while (!proceed) {
        // Check for a dirty split flag
        tri = nextItem->data1;
        oktocheck = true;
        if (tri!=NULL) {
            if (tri->IsDirtySplit()) {
                oktocheck = false;
            }
        }
    }
if(oktocheck) {
    //Triangle is not dirty, check its priority
    if(nextItem->itemPriority >= accuracy) {
        failed = false;
        //Try to split it
        splitcounter+=RecursiveSplit(tri, failed);
        if(tri->IsDirtySplit()) {
            //Split conducted, delete the queue item
            delete nextItem;
        } else {
            //Split was not correctly performed, reinsert item to queue
            //This can happen if high resolution was not attached consistently
            splitQueue->InsertMiddle(nextItem);
        }
    } else {
        //Item below accuracy, abort and reinsert item to queue
        proceed = true;
        splitQueue->InsertMiddle(nextItem);
    }
    delete nextItem;
}

if(splitcounter>MAX_SPLITS) {
    proceed = true;
}

loopcounter++; if(loopcounter>MAX_LOOPS) {
    proceed = true;
}
if(!proceed) {
    nextItem = splitQueue->RemoveFront();
    if(nextItem==NULL) {
        proceed = true;
    }
}
return splitcounter;

void Terrain::InDiamondAdd(Triangle* triangle) {
    //Look for new mergable diamonds
    static float prio1, prio2;
    //Check if we are in new diamond
    if(triangle->leftNeighbor != NULL && triangle->rightNeighbor != NULL) {
        if(triangle->leftNeighbor->leftNeighbor != NULL) {
            triangle->rightNeighbor->leftNeighbor = triangle->leftNeighbor;
            triangle->leftNeighbor->rightNeighbor = triangle->rightNeighbor;
            //There is a new diamond add the triangles of it.
            Triangle* parentTri1 = triangle->GetParent();
            Triangle* parentTri2;
            if(triangle->leftNeighbor->leftNeighbor->GetParent() == parentTri1) {
                parentTri2 = triangle->rightNeighbor->GetParent();
            } else if(triangle->rightNeighbor->GetParent() == parentTri1) {
                parentTri2 = triangle->leftNeighbor->GetParent();
            }
            parentTri1->baseNeighbor = parentTri2;
            parentTri2->baseNeighbor = parentTri1;
            prio1 = CalcPriority(parentTri1,worldCamera);
            prio2 = CalcPriority(parentTri2,worldCamera);
            InsertToMergeQueue(parentTri1,parentTri2,MaxOf(prio1,prio2));
        } else {
            //Check for half diamonds
            if(triangle->rightNeighbor == NULL && triangle->leftNeighbor != NULL) {
                if(triangle->leftNeighbor->rightNeighbor == triangle) {
                    if(triangle->leftNeighbor->leftNeighbor == NULL) {
                        if(triangle->GetParent() == triangle->leftNeighbor->GetParent()) {
                            Triangle* parentTri = triangle->GetParent();
                            parentTri->baseNeighbor = NULL;
                            prio1 = CalcPriority(parentTri,worldCamera);
                            InsertToMergeQueue(parentTri,NULL,prio1);
                        }
                    }
                }
            }
        }
    }
}
else if(triangle->leftNeighbor == NULL && triangle->rightNeighbor != NULL) {
    if(triangle->rightNeighbor->leftNeighbor == triangle) {
        if(triangle->rightNeighbor->rightNeighbor == triangle->GetParent()) {
            // half diamond on edge formed
            Triangle* parentTri = triangle->GetParent();
            parentTri->baseNeighbor = NULL;
            prio1 = CalcPriority(parentTri, worldCamera);
            InsertToMergeQue(parentTri, NULL, prio1);
        }
    }
}
}

int Terrain::RecursiveSplit(Triangle* triangle, bool &failed) {
    int rval = 0;
    Triangle* leftChild = triangle->GetLeftChild();
    Triangle* rightChild = triangle->GetRightChild();
    // Check if we have children, in that case, we can be split
    if(leftChild != NULL)
    {
        // Check if we are part of a diamond.
        if(triangle->baseNeighbor != NULL) {
            rval += RecursiveSplit(triangle->baseNeighbor, failed);
        }
        Triangle* bleftChild = triangle->baseNeighbor->GetLeftChild();
        Triangle* brightChild = triangle->baseNeighbor->GetRightChild();
        // Make sure our base neighbor have children
        if(bleftChild == NULL || brightChild == NULL)
        {
            failed = true;
            if(failed)
            {
                return rval;
            }
            rval += 2;
            // Add the children to the renderin list, and also update their
            // neighboring connections
            triangle->GetParentPatch()->AddRenderChildren(triangle, leftChild, rightChild);
            triangle->baseNeighbor->GetParentPatch()->AddRenderChildren(triangle->baseNeighbor, bleftChild, brightChild);

            // Remove the parents from the split queue
            RemoveFromSplitQue(triangle, true);
            RemoveFromSplitQue(triangle->baseNeighbor, true);

            // Insert the children to the split queue
            InsertToSplitQue(leftChild, NULL, CalcPriority(leftChild, worldCamera), true);
            InsertToSplitQue(rightChild, NULL, CalcPriority(rightChild, worldCamera), true);
            InsertToSplitQue(bleftChild, NULL, CalcPriority(bleftChild, worldCamera), true);
            InsertToSplitQue(brightChild, NULL, CalcPriority(brightChild, worldCamera), true);

            // Insert parents to merge queue
            float pri01 = CalcPriority(triangle, worldCamera);
            float pri02 = CalcPriority(triangle->baseNeighbor, worldCamera);
            InsertToMergeQue(triangle, triangle->baseNeighbor, MaxOf(pri01, pri02));

            // Remove items from merge queue, which no longer be merged
            if(triangle->GetParent() != NULL) {
                RemoveFromMergeQueue(triangle->GetParent());
                if(triangle->GetParent()->baseNeighbor != NULL) {
                    RemoveFromMergeQueue(triangle->GetParent()->baseNeighbor);
                }
                RemoveFromMergeQueue(triangle->baseNeighbor->GetParent());
                if(triangle->baseNeighbor->GetParent() != NULL) {
                    RemoveFromMergeQueue(triangle->baseNeighbor->GetParent()->baseNeighbor);
                }
            }

            // Update the neighboring connections amongst the children
            leftChild->rightNeighbor = brightChild;
            rightChild->leftNeighbor = bleftChild;
            bleftChild->rightNeighbor = rightChild;
            brightChild->leftNeighbor = leftChild;
        }
    }
    else if(triangle->leftNeighbor == NULL && triangle->rightNeighbor != NULL) {
        if(triangle->rightNeighbor->leftNeighbor == triangle) {
            if(triangle->rightNeighbor->rightNeighbor == triangle->GetParent()) {
                // half diamond on edge formed
                Triangle* parentTri = triangle->GetParent();
                parentTri->baseNeighbor = NULL;
                prio1 = CalcPriority(parentTri, worldCamera);
                InsertToMergeQue(parentTri, NULL, prio1);
            }
        }
    }
    return rval;
}
//We are not part of a diamond, half diamond split
rval = 1;
//Add the children to the rendering list, and update neighboring connections
triangle->GetParentPatch()->AddRenderChildren(triangle, leftChild, rightChild);

//Remove parent from split queue
RemoveFromSplitQueue(triangle, true);

//Insert children to split queue
InsertToSplitQueue(leftChild, NULL, CalcPriority(leftChild, worldCamera), true);
InsertToSplitQueue(rightChild, NULL, CalcPriority(rightChild, worldCamera), true);

//Insert parent to merge queue
float prio = CalcPriority(triangle, worldCamera);
InsertToMergeQueue(triangle, NULL, prio);

//Remove diamonds, that can no longer be merged, from the merge queue
if (triangle->GetParent() != NULL) {
    RemoveFromMergeQueue(triangle->GetParent());
    if (triangle->GetParent()->baseNeighbor != NULL) {
        RemoveFromMergeQueue(triangle->GetParent()->baseNeighbor);
    }
    leftChild->rightNeighbor = NULL;
    rightChild->leftNeighbor = NULL;
}
return rval;

int Terrain::RecursiveMerge(Triangle* triangle, Triangle* ttbnb, bool &failed) {
    Triangle* leftChild = triangle->GetLeftChild();
    Triangle* rightChild = triangle->GetRightChild();
    int rval = 0;
    bool doneMerge = false;
    if (ttbnb != NULL) {
        if (triangle->baseNeighbor == ttbnb && ttbnb->baseNeighbor == triangle) {
            //A diamond is formed, merge
            Triangle* bleftChild = ttbnb->GetLeftChild();
            Triangle* brightChild = ttbnb->GetRightChild();
            rval = 2;
            //Add parents to rendering list, remove children, and update neighboring
            //connections
            triangle->GetParentPatch()->AddRenderParent(triangle, leftChild, rightChild);
            ttbnb->GetParentPatch()->AddRenderParent(ttbnb, bleftChild, brightChild);
            //Update the queues
            RemoveFromSplitQueue(leftChild, false);
            RemoveFromSplitQueue(rightChild, false);
            RemoveFromMergeQueue(triangle);
            InsertToSplitQueue(triangle, NULL, CalcPriority(triangle, worldCamera), false);
            RemoveFromSplitQueue(bleftChild, false);
            RemoveFromSplitQueue(brightChild, false);
            RemoveFromMergeQueue(ttbnb);
            InsertToSplitQueue(ttbnb, NULL, CalcPriority(ttbnb, worldCamera), false);
            doneMerge = true;
        }
    } else {
        //We have no baseneighbor, only merge this tri.
        rval = 1;
        triangle->GetParentPatch()->AddRenderParent(triangle, leftChild, rightChild);
        //Update queues
        RemoveFromSplitQueue(leftChild, false);
        RemoveFromSplitQueue(rightChild, false);
        RemoveFromMergeQueue(triangle);
        InsertToSplitQueue(triangle, NULL, CalcPriority(triangle, worldCamera), false);
        doneMerge = true;
    }
    //If a merge have been performed, and the new triangle has parents,
    //then look for new mergable diamonds
    if (triangle->GetParent() != NULL && doneMerge) {
        InDiamondAdd(triangle);
        //Check for the base neighbor as well
        if (ttbnb != NULL)
            InDiamondAdd(ttbnb);
    }
}
return rval;
}

void Terrain::InsertToSplitQue(Triangle* tri1, Triangle* tri2, float priority, bool splitting)
{
    if (tri1 != NULL) {
        // This triangle is no longer dirty in the split queue
        tri1->SetDirtySplit(false);
        if (splitting) {
            // If called from splitting instance
            if (tri1->GetChildPatch() != NULL) {
                // And there is a child patch connected
                if (PatchDynamic::numAttachmentsNow > maxAttachments) {
                    // No more attachments allowed in this frame, increase the reference
                    // count for the child patch, and add it to the activation list
                    tri1->GetChildPatch()->AddRef();
                    AddToActivationList(tri1->GetChildPatch());
                }
                // Increase the reference count for the child patch, and then try
                // to activate it.
                tri1->GetChildPatch()->AddRef();
                if (!tri1->GetChildPatch()->ActivatePatch()) {
                    // Activation failed, add it to activation list
                    AddToActivationList(tri1->GetChildPatch());
                }
            }
        }
    }
    if (tri2 != NULL)
        tri2->SetDirtySplit(false);
    // Really insert the triangle
    splitQue->InsertMiddle(tri1, tri2, priority);
}

void Terrain::InsertToMergeQue(Triangle* tri1, Triangle* tri2, float priority)
{
    // Update the dirtyMerge flags
    if (tri1 != NULL)
        tri1->SetDirtyMerge(false);
    if (tri2 != NULL)
        tri2->SetDirtyMerge(false);
    // Insert elements to merge queue
    mergeQue->InsertMiddle(tri1, tri2, priority);
}

void Terrain::RemoveFromSplitQue(Triangle* tri, bool splitting)
{
    // Do not really remove it, just set it dirty
    if (tri != NULL) {
        tri->SetDirtySplit(true);
        if (!splitting) {
            // We have a child patch, add it to the deactivation list, and
            // decrease the reference count for it.
            AddToDeactivationList(tri->GetChildPatch());
            tri->GetChildPatch()->ReleaseRef();
        }
    }
}

void Terrain::RemoveFromMergeQue(Triangle* tri)
{
    // Do not really remove it, just set it dirty
    if (tri != NULL) {
        tri->SetDirtyMerge(true);
    }
}

int Terrain::RenderTerrain(int type)
{
    int numTris = 0;
    // Tell OpenGL to cull all backfaces, and to enable culling
    glCullFace(GL_BACK);
    glEnable(GL_CULL_FACE);
    // Tell OpenGL to normalize all normal vectors
glEnable(GL_NORMALIZE);
//Tell OpenGL to use color material on front faces, with lit amb and diffuse
setColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
//Enable Vertex Array rendering
if (type == 0) {
    //Render flat shaded triangles
    if (setup.lighting) {
        //Enable lighting only when using textures
        glEnable(GL_LIGHTING);
        //Enable the usage of an array of normal vectors, when using vertex arrays
        glEnableClientState(GL_NORMAL_ARRAY);
        //Set the type to both lighting and textures
        type = 3;
    } else {
        //Disable lighting
        glDisable(GL_LIGHTING);
        //Disable the usage of an array of normal vectors, for vertex arrays
        glDisableClientState(GL_NORMAL_ARRAY);
    }
} else if (type == 1) {
    //Render wireframe
    //Disable the usage of textures
    glDisable(GL_TEXTURE_2D);
    //Disable the usage of a array of texture coordinates
    glDisableClientState(GL_TEXTURE_COORD_ARRAY);
    //Tell OpenGL to draw outlined primitives when rendering
    glPolygonMode(GL_FRONT, GL_LINE);
} else if (type == 2) {
    //Render with textures
    //Enable texturing
    glEnable(GL_TEXTURE_2D);
    //Enable the usage of an array of texture coordinates for vertex arrays
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);
    //Tell OpenGL to fill all primitives rendered
    glPolygonMode(GL_FRONT, GL_FILL);
}
if (setup.lighting) {
    //If lighting is enables
    if (type == 2) {
        //Enable lighting only when using textures
        glEnable(GL_LIGHTING);
        //Enable the usage of an array of normal vectors, when using vertex arrays
        glEnableClientState(GL_NORMAL_ARRAY);
        //Set the type to both lighting and textures
        type = 3;
    } else {
        //Disable lighting
        glDisable(GL_LIGHTING);
        //Disable the usage of an array of normal vectors, for vertex arrays
        glDisableClientState(GL_NORMAL_ARRAY);
    }
} else {
    //Now render
    for (int i = 0; i < numPatches; i++) {
        if (type == 2 || type == 3) {
            //If texturing active, bind texture for next static patch
            glBindTexture(GL_TEXTURE_2D, texture[i]);
        }
        //Call the render for the static patch
        numTris += patchlist[i]->RenderVar(type);
    }
}
#ifdef LOGSTAT
    //If logging statistics
    if (outputStatistics) {
        //Add statistics is currently acive
        //Track the time
        QueryPerformanceCounter(&time5);
        //Calculate elapsed times in this frame
        rtime = ElapsedTime(time4, time5);
        ttime = ElapsedTime(time3, time4);
        tottime = ElapsedTime(time1, time5);
        rtris = numTris;
        //Add a line to the statistics object, which will output to file when its
        //buffer is full
        statistics->AddLine(framenumber, tottime, rtime, ttime, dtime, rtris,
            ttris, numdeform, wetime);
        dtime = 0.0;
        wetime = 0.0;
        numdeform = 0;
        //Track time to measure total elapsed time between consecutive frames
        QueryPerformanceCounter(&time1);
    }
#endif
    return numTris;
void Terrain::TriangulateTerrain(Camera* camera, float acc)
{
  ++framenumber;
  //Set the global terrain camera
  worldCamera = camera;
  //Set the global terrain accuracy, and also calculate a squared accuracy
  //so that the square root can be skipped when calculating priorities
  accuracy = acc;
  squaredThreshold = acc*acc*0.01*0.01;
  #ifdef LOGSTAT
  //If logging statistics, then track time
  if(outputStatistics)
    QueryPerformanceCounter(&time3);
  #endif
  //Actually perform the triangulating of the terrain
  Triangulate();
  #ifdef LOGSTAT
  //Track time if active
  if(outputStatistics)
    QueryPerformanceCounter(&time4);
  #endif
}

bool Terrain::IsInitialized()
{
  if(!initialized) return false;
  for(int i=0; i<numPatches; i++) {
    //Check so that all static patches are initialized
    if(!patchlist[i]->IsInitialized()) return false;
  }
  return true;
}

float Terrain::TestCollision(Vertex center, float radius)
{
  //Distance from a bounding sphere to ground, negative if impact
  //not considering collision at the sides.
  float x1 = floor(center.p[0]/hscale);
  float z1 = floor(center.p[2]/hscale);
  float x2 = ceil(center.p[0]/hscale);
  float z2 = floor(center.p[2]/hscale);
  float x3 = floor(center.p[0]/hscale);
  float z3 = ceil(center.p[2]/hscale);
  float dx = center.p[0] - x1;
  float dz = center.p[2] - z1;
  if(dx>0.5 && dz>0.5) {
    x1 = ceil(center.p[0]/hscale);
    z1 = ceil(center.p[2]/hscale);
  }
  if(x1<0 || x2<0 || x3<0 || z1<0 || z2<0 || z3<0) return 0.0;
  if(x1>=width || x2>=width || x3>=width || z1>=width || z2>=width || z3>=width)
    return 0.0;
  //create a plane, distance from point to plane
  Vertex p1 = vertex_list[(int)x1+(int)z1*width];
  Vertex p2 = vertex_list[(int)x2+(int)z2*width];
  Vertex p3 = vertex_list[(int)x3+(int)z3*width];
  Vertex N = CreateNormal(p1, p2, p3);
  float A = N.p[0];
  float B = N.p[1];
  float C = N.p[2];
  float D = -A*p1.p[0]-B*p1.p[1]-C*p1.p[2];
  return -d/radius;
}

void Terrain::CreateDynamicPatch(int mx, int mz, int x, int z, int w)
{
  //Creates a dynamic patch, and adds transition patches surrounding it
  //If there is a transition patch active at location, then convert
  //it to a regular patch
  bool created = false;
  int location = mx + mz*(width-1);
  if(mx>2 && mz>2 && mx<(width-4) && mz<(width-4)) {
    //
if(nurbPatches[location]!=NULL) {
    if(nurbPatches[location]->IsTransition()) {
        //Convert to regular patch
        nurbPatches[location]->ConvertToRegular();
        created = true;
    } else {
        //illegal, there already is a patch there
        return;
    }
} else {
    int px = (int)((float)mx/(float)patchWidth);
    int pz = (int)((float)mz/(float)patchWidth);
    int id = location;
    bool transition = false;
    Triangle* leftParent;
    Triangle* rightParent;
    patchlist[px+pz*numPatchesPerSide]->FindLeafs(x,z,x+w,z+w,
                                                  &leftParent,&rightParent);
    numCreatedDyn++;
    PatchDynamic* newPatch = new PatchDynamic(x,z,w,id,transition);
    //Initialize the dynamic patch
    newPatch->Initialize(nurbPatches[location-(width-1)],
                         nurbPatches[location+(width-1)],
                         nurbPatches[location-1],
                         nurbPatches[location+1],
                         leftParent,
                         rightParent);
    //Setup the nurb (b-spline) for the patch
    newPatch->SetupNurb(nurbBase_5x17,
                        &vertex_list[mx+mz*width],
                        &vertex_list[mx+1+mz*width],
                        &vertex_list[mx+(mz+1)*width],
                        &vertex_list[mx+1+(mz+1)*width],5);
    nurbPatches[location] = newPatch;
    created = true;
}

if(created) {
    //If a patch have been created, then create transitions around it
    for(int i=-1; i<2; i++) {
        for(int j=-1; j<2; j++) {
            if(nurbPatches[(mx+i)+(mz+j)*(width-1)] == NULL) {
                if((mx+i)>1 && (mx+i)<(width-3) && (mz+j)>1 && (mz+j)<(width-3)) {
                    int px = (int)((float)(mx+i)/(float)patchWidth);
                    int id = (mx+i)+(mz+j)*(width-1);
                    bool transition = true;
                    Triangle* leftParent;
                    Triangle* rightParent;
                    int nx = x + i*hscale;
                    int nz = z + j*hscale;
                    patchlist[px+pz*numPatchesPerSide]->FindLeafs(nx, nz, nx+w, nz+w,
                                                                  &leftParent,&rightParent);
                    numCreatedDyn++;
                    PatchDynamic* newPatch = new PatchDynamic(nx,nz,w,id,transition);
                    newPatch->Initialize(nurbPatches[mx+i+(mz+j-1)*width-1],
                                         nurbPatches[mx+i+(mz+j+1)*width-1],
                                         nurbPatches[mx+i+1+(mz+j)*width-1],
                                         nurbPatches[mx+i+1+(mz+j+1)*width-1],
                                         leftParent,
                                         rightParent);
                    newPatch->SetupNurb(nurbBase_5x17,
                                        &vertex_list[mx+i+(mz+j)*width],
                                        &vertex_list[mx+i+1+(mz+j)*width],
                                        &vertex_list[mx+i+(mz+j+1)*width],
                                        &vertex_list[mx+i+1+(mz+j+1)*width],5);
                    nurbPatches[(mx+i)+(mz+j)*(width-1)] = newPatch;
                }
            }
        }
    }
}
}

void Terrain::Deform(Vertex pSource, float radius, float effect) {
#ifdef LOGSTAT
    //Track the deformation time if requested
    LARGE_INTEGER dtid1, dtid2;
    QueryPerformanceCounter(&dtid1);
    numDeform++;
    #endif

    //Our calculations must stretch a little bit beyond the range
    //Calculate the range of deformation
    int xstart = floor(pSource.p[0] - radius)/hscale - 1;
    int zstart = floor(pSource.p[2] - radius)/hscale - 1;
    int xstop = ceil(pSource.p[0] + radius)/hscale + 1;
    int zstop = ceil(pSource.p[2] + radius)/hscale + 1;
    //Ensure deformation takes place within terrain, and not on border
    if(xstart<=2) xstart = 2;
    if(zstart<=2) zstart = 2;
    if(xstop>=width-2) xstop = width-3;
    if(zstop>=width-2) zstop = width-3;

    //How many temporary storage locations do we need
    int nVertices = (xstop-xstart+1)*(zstop-zstart+1);
    int nRow = xstop-xstart+1;

    //Allocate memory for temporary deformation variables
    Vertex* Ftotal = new Vertex[nVertices];
    float* PrelaxE = new float[nVertices];
    float* PrelaxS = new float[nVertices];
    float* PrelaxSE = new float[nVertices];
    Vertex* cPoint = new Vertex(0.0,0.0,0.0);

    //Some more temporary variables
    int pxl, pzl, px, pz, cpx, cpz;
    int currentI, currentV, currentVE, currentVS, currentVSE;
    float distance;
    bool smallExplosion = false;

    //Check if we should add high resolution patches at deformation
    if(radius <= hscale*2) {
        smallExplosion = true;
    }
    if(noExtension) smallExplosion = false;

    //Displace all vertices within range
    for(px=zstart; px<zstop; px++) {
        for(pz=xstart; pz<xstop; pz++) {
            pxl = px - xstart;
            pzl = pz - zstart;
            currentI = pxl + pzl*nRow;
            currentV = px + pz*width;
            currentVE = currentV + 1;
            currentVS = currentV + width;
            currentVSE = currentVS + 1;
            Ftotal[currentI] = Vertex(0.0,0.0,0.0);
            PrelaxE[currentI] = sqrt((vertex_list[currentVE].p[0] -
                                      vertex_list[currentV].p[0]) * (vertex_list[currentVE].p[0] -
                                      vertex_list[currentV].p[0]) + (vertex_list[currentVE].p[1] -
                                      vertex_list[currentV].p[1]) * (vertex_list[currentVE].p[1] -
                                      vertex_list[currentV].p[1]) + (vertex_list[currentVE].p[2] -
                                      vertex_list[currentV].p[2]) * (vertex_list[currentVE].p[2] -
                                      vertex_list[currentV].p[2]));
            PrelaxS[currentI] = sqrt((vertex_list[currentVS].p[0] -
                                      vertex_list[currentV].p[0]) * (vertex_list[currentVS].p[0] -
                                      vertex_list[currentV].p[0]) + (vertex_list[currentVS].p[1] -
                                      vertex_list[currentV].p[1]) * (vertex_list[currentVS].p[1] -
                                      vertex_list[currentV].p[1]) + (vertex_list[currentVS].p[2] -
                                      vertex_list[currentV].p[2]) * (vertex_list[currentVS].p[2] -
                                      vertex_list[currentV].p[2]));
            PrelaxSE[currentI] = sqrt((vertex_list[currentVSE].p[0] -
                                      vertex_list[currentV].p[0]) * (vertex_list[currentVSE].p[0] -
                                      vertex_list[currentV].p[0]) + (vertex_list[currentVSE].p[1] -
                                      vertex_list[currentV].p[1]) * (vertex_list[currentVSE].p[1] -
                                      vertex_list[currentV].p[1]) + (vertex_list[currentVSE].p[2] -
                                      vertex_list[currentV].p[2]) * (vertex_list[currentVSE].p[2] -
                                      vertex_list[currentV].p[2]));

            if(smallExplosion) {
                //Add high resolution if not present
                if(nurbPatches[px+pz*(width-1)] == NULL) {
                    CreateDynamicPatch(px, pz, vertex_list[currentV].p[0],
                                        vertex_list[currentVE].p[0],
                                        vertex_list[currentVS].p[0],
                                        vertex_list[currentVSE].p[0]);
                }
            }
        }
    }

    //Add remaining patches if required
    if(noExtension) {
        for(px=0; px<width; px++) {
            for(pz=0; pz<width; pz++) {
                pxl = px - xstart;
                pzl = pz - zstart;
                currentI = pxl + pzl*nRow;
                currentV = px + pz*width;
                currentVE = currentV + 1;
                currentVS = currentV + width;
                currentVSE = currentVS + 1;
                Ftotal[currentI] = Vertex(0.0,0.0,0.0);
                PrelaxE[currentI] = sqrt((vertex_list[currentVE].p[0] -
                                          vertex_list[currentV].p[0]) * (vertex_list[currentVE].p[0] -
                                          vertex_list[currentV].p[0]) + (vertex_list[currentVE].p[1] -
                                          vertex_list[currentV].p[1]) * (vertex_list[currentVE].p[1] -
                                          vertex_list[currentV].p[1]) + (vertex_list[currentVE].p[2] -
                                          vertex_list[currentV].p[2]) * (vertex_list[currentVE].p[2] -
                                          vertex_list[currentV].p[2]));
                PrelaxS[currentI] = sqrt((vertex_list[currentVS].p[0] -
                                          vertex_list[currentV].p[0]) * (vertex_list[currentVS].p[0] -
                                          vertex_list[currentV].p[0]) + (vertex_list[currentVS].p[1] -
                                          vertex_list[currentV].p[1]) * (vertex_list[currentVS].p[1] -
                                          vertex_list[currentV].p[1]) + (vertex_list[currentVS].p[2] -
                                          vertex_list[currentV].p[2]) * (vertex_list[currentVS].p[2] -
                                          vertex_list[currentV].p[2]));
                PrelaxSE[currentI] = sqrt((vertex_list[currentVSE].p[0] -
                                           vertex_list[currentV].p[0]) * (vertex_list[currentVSE].p[0] -
                                           vertex_list[currentV].p[0]) + (vertex_list[currentVSE].p[1] -
                                           vertex_list[currentV].p[1]) * (vertex_list[currentVSE].p[1] -
                                           vertex_list[currentV].p[1]) + (vertex_list[currentVSE].p[2] -
                                           vertex_list[currentV].p[2]) * (vertex_list[currentVSE].p[2] -
                                           vertex_list[currentV].p[2]));

                if(smallExplosion) {
                    //Add high resolution if not present
                    if(nurbPatches[px+pz*(width-1)] == NULL) {
                        CreateDynamicPatch(px, pz, vertex_list[currentV].p[0],
                                            vertex_list[currentVE].p[0],
                                            vertex_list[currentVS].p[0],
                                            vertex_list[currentVSE].p[0]);
                    }
                }
            }
        }
    }

    //Clear temporary storage variables
    delete[] Ftotal;
    delete[] PrelaxE;
    delete[] PrelaxS;
    delete[] PrelaxSE;

if (nurbPatches[px+pz*(width-1)] != NULL) {
    // If it is not a transition patch, then
    if (!nurbPatches[px+pz*(width-1)]->IsTransition()) {
        // Displace the control points of the high resolution patch
        float* controlPoints = nurbPatches[px+pz*(width-1)]->GetControlMap();
        int nControlPoints = nurbPatches[px+pz*(width-1)]->GetNumControlPoints();
        float xoffset = (float)nurbPatches[px+pz*(width-1)]->GetXOffset();
        float zoffset = (float)nurbPatches[px+pz*(width-1)]->GetZOffset();
        float xzwidth = (float)nurbPatches[px+pz*(width-1)]->GetWidth();
        xzwidth = xzwidth/(float)(nControlPoints-1);
        for (cpz=0; cpz<nControlPoints; cpz++) {
            for (cpx=0; cpx<nControlPoints; cpx++) {
                cPoint->p[0] = xoffset + cpx*xzwidth;
                cPoint->p[1] = controlPoints[cpx+cpz*nControlPoints];
                cPoint->p[2] = zoffset + cpz*xzwidth;
                DisplaceVertex(cPoint, pSource, radius, effect);
                controlPoints[cpx+cpz*nControlPoints] = cPoint->p[1];
            }
        }
    }
}

//Some more temporary variables :) :
Vertex De, Da, Dse;
Vertex Fe, Fs, Fse;
Vertex v1,v2,vt1,vt2,NormalV;
float Del,Del1,Del2,Delc,Dselc,Dselc;
float elasticity = 1.0f;
float damping = 0.5f;
//Revert all displaced vertices, as they are pulled back towards each other
for(px=xstart; px<xstop; px++) {
    pxl = px - xstart;
    for(px=xstart; px<xstop; px++) {
        px1 = px - xstart;
        currentI = px1 + px1*nRow;
        currentV = px + px*width;

        //Process the east direction
        De = vertex_list[px1+px*width] - vertex_list[currentV];
        Del1 = sqrt(De.p[0]*De.p[0]+De.p[1]*De.p[1]+De.p[2]*De.p[2]);
        Delc = Del1 - PrelaxE[currentI];
        De = De*(Delc/Del1);
        Pe = De*elasticity*0.5;
        Ftotal[currentI] += Pe;
        Ftotal[px1+1+px1*nRow] -= Pe;
    }
}

} //Is there any high resolution patch present
if (nurbPatches[px+pz*(width-1)] != NULL) {
    if (!DisplaceVertex(vertex_list[currentV], pSource, radius, effect)) {
        //Reset the reverting factors, as displaced vertex was above the center
        //of deformation, this to ensure spikes is not introduced in the terrain
        PrelaxE[currentI] = Distance(vertex_list[currentV],
                                     vertex_list[currentVE]);
        PrelaxS[currentI] = Distance(vertex_list[currentV],
                                     vertex_list[currentVS]);
        PrelaxSE[currentI] = Distance(vertex_list[currentV],
                                     vertex_list[currentVSE]);

        if (pxl > 0) {
            PrelaxS[pxl-1+pxl*nRow] = Distance(vertex_list[currentV],
                                               vertex_list[currentVE]);
        }
        if (pzl > 0) {
            PrelaxS[pxl+(pzl-1)*nRow] = Distance(vertex_list[currentV],
                                               vertex_list[currentVS]);
        }
        if (pxl > 0 && pzl > 0) {
            PrelaxS[pxl-1+(pzl-1)*nRow] = Distance(vertex_list[currentV],
                                               vertex_list[currentVSE]);
        }
    }
    //Compress the vertex
    if (DisplaceVertex(vertex_list[currentV], pSource, radius, effect)) {
        CreateDynamicPatch(px, pz, vertex_list[currentV].p[0],
                            vertex_list[currentV].p[2], hscale);
    }
    //Is the vertex
    else if (nurbPatches[px+pz*(width-1)]->IsTransition()) {
        CreateDynamicPatch(px, pz, vertex_list[currentV].p[0],
                            vertex_list[currentV].p[2], hscale);
    }
}
// Process the south direction
Ds = vertex_list[px+(pz+1)*width] - vertex_list[currentV];
Ds1 = sqrt(Ds.p[0]*Ds.p[0]+Ds.p[1]*Ds.p[1]+Ds.p[2]*Ds.p[2]);
Ds1c = Ds1 - PrelaxS[currentI];
Ds = Ds*(Ds1c/Ds1);
Fs = Ds*elasticity*0.5;
Ftotal[currentI] += Fs;
Ftotal[pxl+(pzl+1)*nRow] -= Fs;

// Process the south-east direction
Dse = vertex_list[px+1+(pz+1)*width] - vertex_list[currentV];
Dse1 = sqrt(Dse.p[0]*Dse.p[0]+Dse.p[1]*Dse.p[1]+Dse.p[2]*Dse.p[2]);
Dse1c = Dse1 - PrelaxSE[currentI];
Dse = Dse*(Dse1c/Dse1);
Fse = Dse*elasticity*0.5;
Ftotal[currentI] += Fse;
Ftotal[pxl+1+(pzl+1)*nRow] -= Fse;

// Should also process one more direction, but skipped for now, and
// that is either southwest or north-west

// Calculate the total reverting factor
Ftotal[currentI] = Ftotal[currentI]*damping;

// Revert the vertex

// Recalculate the normal vector
vt1 = (Vertex)vertex_list[currentV];
vt2 = (Vertex)vertex_list[px -1 + pz*width];
v1 = vt1 - vt2;
vt2 = (Vertex)vertex_list[px + (pz-1)*width];
v2 = vt1 - vt2;
NormalV = CreateNormal(v1,v2);
vertex_list[currentV].n[0] = NormalV.p[0];
vertex_list[currentV].n[1] = NormalV.p[1];

// How many triangles can reference this vertex
int numEffectedTris = 3;
if (px==0 || pz==0 || px==width-1 || pz==width-1) {
    numEffectedTris--;
}

// Set the changed flag for the vertex
vertex_list[currentV].SetChanged(numEffectedTris);

// Revert the control points, if a high resolution patch is present
if (nurbPatches[px+pz*(width-1)] != NULL) {
    Vertex A,B,C,D;
    if (!nurbPatches[px+pz*(width-1)]->IsTransition()) {
        // Interpolate over the four corner reverting sums
        A = Ftotal[currentI];
        B = Ftotal[pxl+1+pzl*nRow];
        C = Ftotal[pxl+(pzl+1)*nRow];
        D = Ftotal[pxl+1+(pzl+1)*nRow];

        Vertex lside;
        Vertex rside;
        Vertex across;

        float* controlPoints = nurbPatches[px+pz*(width-1)]->GetControlMap();
        int nControlPoints = nurbPatches[px+pz*(width-1)] ->
            GetNumControlPoints();
        float u, v;
        for (int cpz=0; cpz<nControlPoints; cpz++) {
            u = (float)cpz/(float)(nControlPoints-1);
            lside = A*(1-u) + C*u;
            rside = B*(1-u) + D*u;
            for (int cpx=0; cpx<nControlPoints; cpx++) {
                v = (float)cpx/(float)(nControlPoints-1);
                across = lside*(1-v) + rside*v;
                controlPoints[cpx+cpz*nControlPoints] += across.p[1];
            }
        }
    }
    nurbPatches[px+pz*(width-1)]->SetUpdateFlag();
}
/* Delete all temporary storage */
delete[] Ftotal;
delete[] PrelaxE;
delete[] PrelaxS;
delete[] PrelaxSE;
delete cPoint;

/* Calculate which patches need updating */
int spx = (int)floor((float)(xstart-1)/(float)patchWidth);
int spz = (int)floor((float)(zstart-1)/(float)patchWidth);
int epx = (int)ceil((float)(xstop+1)/(float)patchWidth);
int epz = (int)ceil((float)(zstop+1)/(float)patchWidth);
if(spx<0) spx = 0;
if(spx>numPatchesPerSide) spx = numPatchesPerSide;
if(epx<0) spx = 0;
if(epx>numPatchesPerSide) spx = numPatchesPerSide;
for(int j=spz; j<epz; j++) {
for(int i=spx; i<epx; i++) {

// Set the update flag in the static patch
patchlist[i+numPatchesPerSide*j]->SetUpdateFlag();

// Now add some changes to the texture for this patch

// What range does this patch cover
int pxstart = i*patchWidth;
int pzstart = j*patchWidth;
int pxstop = (i+1)*patchWidth;
int pzstop = (j+1)*patchWidth;

// In what range is this patch affected
int clipxstart,clipzstart,clipxstop,clipzstop;
if(xstart<pxstart) clipxstart = pxstart;
else clipxstart = xstart;
if(zstart<pzstart) clipzstart = pzstart;
else clipzstart = zstart;
if(xstop>pxstop) clipxstop = pxstop;
else clipxstop = xstop;
if(zstop>pzstop) clipzstop = pzstop;
else clipzstop = zstop;

// Calculate in what range the texture should be affected
clipxstart = clipxstart - patchWidth*clipxstart = clipxstart - patchWidth*
clipzstart = clipzstart - patchWidth*clipzstart = clipzstart - patchWidth*
_clipxstart = clipxstart - patchWidth*
(clipxstart = clipxstart - patchWidth*
clipxstop = clipxstop - patchWidth*
clipzstop = clipzstop - patchWidth*
(clipzstart = clipzstart - patchWidth*
clipxstart = clipxstart - patchWidth*

// Calculate a texture multiplier, texture size 256 is assumed here,
// but that NEEDS to be changed to work with other sizes of textures :)
float multiplier = (float)256.0f/(float)patchWidth;

// Multiply for texture coordinates
clipxstart *= multiplier;
clipzstart *= multiplier;
clipxstop *= multiplier;
clipzstop *= multiplier;

// Calculate the width of the change
int clipxwidth = clipxstop - clipxstart;
int clipzwidth = clipzstop - clipzstart;

// Allocate temporary storage for the texture to be changed
unsigned char* tmpStore = new unsigned char[clipxwidth*clipzwidth*3];
unsigned char* tmpStore2 = new unsigned char[256*256*3];

// Reset them
memset(tmpStore,0,clipxwidth*clipzwidth*3); // sizeof(unsigned char)
memset(tmpStore2,0,256*256*3);

float r, sg, rgs, rr;
rgs = radius*radius;

// Tell OpenGL to bind the wanted texture
glBindTexture(GL_TEXTURE_2D, texture[i+numPatchesPerSide*j]);

// Fetch the texture from the graphics API
glGetTexImage(GL_TEXTURE_2D, 0, GL_RGB, GL_UNSIGNED_BYTE, tmpStore2);

// Now change the affected positions in the texture
for(int l=0; l<clipzwidth; l++) {
    int tj = l + clipzstart;
    for(int k=0; k<clipxwidth; k++) {
        int ti = k + clipxstart;
        // Read the old values
        tmpStore[(k+l*clipxwidth)*3] = tmpStore2[(ti+tj*256)*3];
        tmpStore[(k+l*clipxwidth)*3+1] = tmpStore2[(ti+tj*256)*3+1];
        tmpStore[(k+l*clipxwidth)*3+2] = tmpStore2[(ti+tj*256)*3+2];
        // Is this texture information in range
        int wx = pxstart*multiplier + (float)ti/(float)multiplier*hscale;
        int wz = pzstart*multiplier + (float)tj/(float)multiplier*hscale;
        rrsq = (wx-pSource.p[0])*(wx-pSource.p[0])+(wz-pSource.p[2])*(wz-pSource.p[2]);
        if(rrsq<rsq) {
            // Yes it is, darken the color by an amount relative to the
            // distance from the center of deformation
            rr = sqrt(rrsq)/radius;
            tmpStore[(k+l*clipxwidth)*3] = tmpStore[(k+l*clipxwidth)*3]*rr;
            tmpStore[(k+l*clipxwidth)*3+1] = tmpStore[(k+l*clipxwidth)*3+1]*rr;
            tmpStore[(k+l*clipxwidth)*3+2] = tmpStore[(k+l*clipxwidth)*3+2]*rr;
        }
    }
}
// Send back the changed information to OpenGL, and the texture in question
glTexSubImage2D(GL_TEXTURE_2D,0,clipxstart,clipzstart,clipxwidth,
clipzwidth,GL_RGB,GL_UNSIGNED_BYTE,tmpStore);
// Delete the temporary storage
delete[] tmpStore;
delete[] tmpStore2;
}
#endif

bool Terrain::DisplaceVertex(Vertex* pTarget, Vertex Source, float radius, float effect) {
    // Returns false if target was above the source
    static float distancesq;
    static float distance;
    static float d2dsq;
    static float rsq;
    d2dsq = (Source.p[0]-pTarget->p[0])*(Source.p[0]-pTarget->p[0])+(Source.p[2]-
    rsq = radius*radius;
    // Check if this vertex is within range (2D)
    if(d2dsq < rsq) {
        if(Source.p[1] > pTarget->p[1]) {
            // This vertex is below the center of deformation
            distancesq = d2dsq+(Source.p[0]-pTarget->p[0])*(Source.p[0]-pTarget->p[0])+(Source.p[2]-
            // It was within 3D range, move it
            distance = sqrt(distancesq);
            float displacementY = (radius-distance)/radius*effect;
            if(displacementY > 0)
                pTarget->p[1] -= displacementY;
            return true;
        } else {
            // Special treatment for vertices above the center of deformation,
            // in order to avoid spikes :)
            d2d = sqrt(d2dsq);
            float displacementY = (radius - d2d)/radius*effect;
            if(displacementY > 0)
                pTarget->p[1] -= displacementY;
        }
    }
    return false;
}
float Terrain::CalcPriority(Triangle* triangle, Camera* camera)
{
    if(triangle->GetParentPatch()->InFrustum() != 0) {
        // The patch is intersecting the frustum
        if(triangle->InFrustum() != 0 || triangle->GetParentPatch()->InFrustum() == 2) {
            // Either the patch is completely within the frustum, or the triangle
            // is either fully or partially inside the frustum
            // Calculate the center of the triangle
            triangle->MidHypotenuse(vCC);
            // Calculate a simple priority
            return CalcPrioritySimple(triangle, camera);
        } else {
            // This triangle is not in the frustum, calculate a low
            // priority based on the distance, so that triangles with
            // really high height faults still get split even if not in
            // the frustum, this to prepare in case they will soon be in frustum
            return CalcPriorityByDistance(triangle, camera);
        }
    }
    return 0.0;
}

float Terrain::CalcPrioritySimple(Triangle* triangle, Camera* camera)
{
    // Get the variance of this triangle
    error = triangle->GetVariance() * 0.5;

    // Some static variables
    static Vertex vC1;
    static Vertex vC2;
    static Vertex eT;
    static float sx1;
    static float sy1;
    static float sx2;
    static float sy2;

    // Translate to camera space
    vCC -= camera->eye;

    // Project the error
    eT.p[0] = error * camera->rmtpm.dm[1];
    eT.p[1] = error * camera->rmtpm.dm[5];
    eT.p[2] = error * camera->rmtpm.dm[9];

    // Project the center point
    vCC.Transform(&camera->rmtpm);
    // Add the projected error
    vC1 = vCC - eT;
    vC2 = vCC + eT;

    if(vC1.p[2]<0.4f && vC2.p[2]<0.4f) {
        // Behind the near clipping plane, assumed this resided at 0.4
        return 0.0f;
    }

    if(vC1.p[2]>0.4f && vC2.p[2]>0.4f) {
        // Calculate screen space coordinates
        sx1 = vC1.p[0]/vC1.p[2];
        sy1 = vC1.p[1]/vC1.p[2];
        sx2 = vC2.p[0]/vC2.p[2];
        sy2 = vC2.p[1]/vC2.p[2];

        // Check if the measured pixel error is above the target accuracy or below it
        // if below, then return a zero priority, if above, then return a 10.0 priority
        return ((sx1-sx2)*(sx1-sx2)+(sy1-sy2)*(sy1-sy2))*0.16 < squaredThreshold) ? 0.0f : 10.0f;
    }

    // Intersecting the near clipping plane, set high priority
    return 10.0f;
}

float Terrain::CalcPriorityByDistance(Triangle* triangle, Camera* camera)
{
    // Calculate a distance based priority
    Vertex vEye = camera->eye;
    return triangle->GetVariance() / Distance(vCC, vEye);
}
float Terrain::CalcPriorityHeavy(Triangle* triangle, Camera* camera)
{
    // This is the original priority computations, but they are not used,
    // as they have been found to be slow. This can of course depend on
    // how they are implemented here.

    // Top or apex vertex of triangle
    Vertex vT = vertex_list[triangle->GetVertex(0)];
    // Left vertex of triangle
    Vertex vL = vertex_list[triangle->GetVertex(1)];
    // Right vertex of triangle
    Vertex vR = vertex_list[triangle->GetVertex(2)];

    // Translate into camera space
    vT -= camera->eye;
    vL -= camera->eye;
    vR -= camera->eye;

    // Apply the rotation transform (transposed) into camera space
    vT.Transform(&camera->rmtpm);
    vL.Transform(&camera->rmtpm);
    vR.Transform(&camera->rmtpm);

    // Wedge thickness vector (a,b,c)
    Vertex eT(0.0,triangle->GetVariance(),0.0);

    // Transform to camera space
    eT.Transform(&camera->rmtpm);

    // Find the minimum r in (p,q,r) to maximize 2/(r^2-c^2)
    float min = fabs(vT.p[1]);
    float tmp = fabs(vL.p[1]);
    if(tmp < min) min = tmp;
    tmp = fabs(vR.p[1]);
    if(tmp < min) min = tmp;
    min = min*min - eT.p[1]*eT.p[1];

    // Find the maximum (ar-cp)^2+(br-cq)^2
    float arcp = eT.p[0]*vT.p[1] - eT.p[1]*vT.p[0];
    float max = arcp*arcp + brcq*brcq;
    if(max > tmp) max = tmp;
    arcp = eT.p[0]*vL.p[1] - eT.p[1]*vL.p[0];
    tmp = arcp*arcp + brcq*brcq;
    if(tmp > max) max = tmp;

    // Take the square root and return.
    float result = 20.0;
    if(min > 0.1) result = (2.0/min)*sqrt(max);
    return result;
}

#ifdef LOGSTAT
void Terrain::PrintStatus(char* filename)
{
    // Print some information about the terrain system
    char buffer[256];
    FILE* wfile;
    wfile = fopen(filename, "a+");
    if(wfile) {
        // Some information about the terrain system
        sprintf(buffer,"Num allocated dynamic bintrees : %ld\n",
                binTreeManager16->GetNumTrees());
        fputs(buffer, wfile);
        sprintf(buffer,"Max num used dynamic bintrees : %ld\n",
                binTreeManager16->GetTopUsedTrees());
        fputs(buffer, wfile);
        sprintf(buffer,"Allocated memory by application : %ld, allocatedMem\n",
                allocatedMem);
        fputs(buffer, wfile);
        sprintf(buffer,"Number of allocated dynamic cells is %ld, numCreatedDyn\n",
                numCreatedDyn);
        fputs(buffer, wfile);
        fputs("------------------------------------------\n", wfile);
    }
    fclose(wfile);
}
#endif
float Terrain::GetHeight(float xp, float zp)
{
    // Return the height at a x and z position
    int x = ((float)xp/hscale);
    int z = ((float)zp/hscale);
    if(x>=0 && z>=0 && x<width && z<width) {
        return vertex_list[x+z*width].p[1];
    }
    return 0.0;
}

void Terrain::AddToActivationList(Patch* p)
{
    // Add a patch to the activation list
    if(numWaitingActivation>=maxWaitingActivation) {
        // Allocate more space in the activation list
        maxWaitingActivation = maxWaitingActivation*2+1;
        Patch** tmp = new Patch*[maxWaitingActivation];
        memcpy(tmp,activationList,numWaitingActivation*sizeof(Patch*));
        delete[] activationList;
        activationList = tmp;
    }
    // Add the patch
    activationList[numWaitingActivation++] = p;
}

void Terrain::AddToDeactivationList(Patch* p)
{
    // Add a patch to the deactivation list
    if(numWaitingDeactivation>=maxWaitingDeactivation) {
        // Allocate more space in the deactivation list
        maxWaitingDeactivation = maxWaitingDeactivation*2+1;
        Patch** tmp = new Patch*[maxWaitingDeactivation];
        memcpy(tmp,deactivationList,numWaitingDeactivation*sizeof(Patch*));
        delete[] deactivationList;
        deactivationList = tmp;
    }
    // Add the patch
    deactivationList[numWaitingDeactivation++] = p;
}

void Terrain::ProcessActivationList(void)
{
    // Try to activate patches in the activation list
    for(int i=0; i<numWaitingActivation; i++) {
        if(activationList[i]->ActivatePatch()) {
            // Ok, patch activated, remove it from activation list
            if(i!=numWaitingActivation-1) {
                activationList[i] = activationList[numWaitingActivation-1];
                i--;
            }
            numWaitingActivation--;
        } else {
            // Could not activate patch
            return;
        }
    }
    // If too many activations in this frame, then abort
    if(PatchDynamic::numAttachmentsNow>maxAttachments)
    
        return;
    numWaitingActivation = 0;
}

void Terrain::ProcessDeactivationList(void)
{
    // Deactivate patches in the deactivation list
    for(int i=0; i<numWaitingDeactivation; i++) {
        deactivationList[i]->DeactivatePatch();
        deactivationList[i] = NULL;
    }
    numWaitingDeactivation = 0;
}

#ifdef LOGSTAT
void Terrain::OutputStatistics(char* filename)
{
    // We have been told to use a statistics object, create it and
/reset all timers.
if(outputStatistics) {
    statistics->PrintOut();
    delete statistics;
} else {
    statistics = new RStatistics(100, filename);
    allocatedMem += sizeof(RStatistics);
    outputStatistics = true;
    QueryPerformanceCounter(&time1);
    QueryPerformanceCounter(&time2);
    QueryPerformanceCounter(&time3);
    QueryPerformanceCounter(&time4);
    QueryPerformanceCounter(&time5);
}
#endif

Triangle.h

irectory Triangle.h
*****************************************************************************
* Triangle.h                                                                 *
* Author: Anders Dahlbom                                                     *
* Last modified: 030603                                                      *
*                                                                            *
* Purpose: Class for capturing the details of bintree triangle.              *
*                                                                            *
*****************************************************************************

#ifndef TRIANGLE_H
#define TRIANGLE_H
#include "Vertex.h"
#include "VertexExt.h"
#include "Patch.h"

class Patch;

//Bitfield for triangle state
typedef struct bitfield {
    unsigned char dirtymerge : 1; //Triangle is not active in merge queue
    unsigned char dirtysplit : 1; //Triangle is not active in split queue
    unsigned char infrustum : 2; //Frustum flag, 0 = out, 1 = intersect, 2 = in
    unsigned char changed : 1; //Has the frustum flag recently been changed
} BITFIELD;

class Triangle {
public:
    Triangle(int top, int left, int right, int in_id, Patch** pPatch);
    ~Triangle();
    //Set the vertex index for a specified triangle vertex
    void SetVertex(int nr, int vertex);
    //Set the variance for this triangle
    void SetVariance(float error);
    //Get a vertex at the mid point of the hypotenuse, for this triangle
    void MidHypotenuse(Vertex& v);
    //Get the variance for this triangle
    float GetVariance(void);
    //Get this triangles left child
    Triangle* GetLeftChild(void);
    //Get this triangles right child
    Triangle* GetRightChild(void);
    //Get the parent triangle for this triangle
    Triangle* GetParent(void);
    //Get the child patch, connected to this triangle
    Patch* GetChildPatch(void);
    //Get the parent patch, to which this triangle belongs
    Patch* GetParentPatch(void);

    //Set that this triangles frustum flag has not recently been changed
    void SetUnchanged(void) { bits.changed = false; }
    //Set that this triangles frustum flag has recently been changed
    void SetChanged(void) { bits.changed = true; }
    //Set the dirty split flag for this triangle
    void SetDirtySplit(bool sb) { bits.dirtysplit = sb; }
    //Set the dirty merge flag for this triangle
    void SetDirtyMerge(bool sb) { bits.dirtymerge = sb; }

private:
    //Bitfield for triangle state
    BITFIELD bits;
};
void SetInFrustum(unsigned char sb)    { bits.infrustum = sb; }
void SetChildPatch(Patch** cPatch)     { childPatch = cPatch; }
void SetParentPatch(Patch** pPatch)    { parentPatch = pPatch; }
void GetRenderIndex(unsigned int ri)   { render_index = ri; }
bool HasChanged(void)                  { return bits.changed; }
bool IsDirtySplit(void)                { return bits.dirtysplit; }
bool IsDirtyMerge(void)                { return bits.dirtymerge; }
int GetVertex(int nr)                  { return vertices[nr]; }
int GetID(void)                        { return id; }
unsigned char InFrustum(void)         { return bits.infrustum; }
unsigned int GetRenderIndex(void)     { return render_index; }

Triangle *baseNeighbor;
Triangle *leftNeighbor;
Triangle *rightNeighbor;

private:
    int vertices[3];             //Indexes for this triangles vertices
    int id;                      //This triangels id
    unsigned int render_index;   //Position in rendering list
    Patch** parentPatch;        //Patch which this triangle belongs to
    Patch** childPatch;         //Child patch connected beneath this triangle
    BITFIELD bits;              //State variables for this triangle
};
#endif

Triangle.cpp

#include "CommonInclude.h"
#include "Triangle.h"
#include "Helper.h"

Triangle::Triangle(int top, int left, int right, int in_id, Patch** pPatch)
{
    vertices[0] = top;
    vertices[1] = left;
    vertices[2] = right;
    id = in_id;
    parentPatch = pPatch;

    baseNeighbor = NULL;
    leftNeighbor = NULL;
    rightNeighbor = NULL;
    childPatch = NULL;
    bits.dirtymerge = true;
    bits.dirtysplit = true;
    bits.infrustum = true;
    bits.changed = false;
    render_index = 0;
}

Triangle::~Triangle()
{
}

void Triangle::SetVertex(int nr, int vertex)
{
    if(nr>=0 && nr<2) {
        vertices[nr] = vertex;
    }
}

void Triangle::SetVariance(float error)
if (*parentPatch == NULL) return;
(*parentPatch)->SetVariance(id, error);
}

void Triangle::MidHypotenuse(Vertex& v)
{
(*parentPatch)->MidHypotenuse(id, vertices[1], vertices[2], v);
}

float Triangle::GetVariance(void)
{
if (*parentPatch == NULL) return 0.0;
return (*parentPatch)->GetVariance(id);
}

Triangle* Triangle::GetLeftChild(void)
{
if (*parentPatch == NULL) return NULL;
return (*parentPatch)->GetLeftChildT(id);
}

Triangle* Triangle::GetRightChild(void)
{
if (*parentPatch == NULL) return NULL;
return (*parentPatch)->GetRightChildT(id);
}

Triangle* Triangle::GetParent(void)
{
if (*parentPatch == NULL) return NULL;
return (*parentPatch)->GetParentT(id);
}

Patch* Triangle::GetChildPatch(void)
{
if (childPatch == NULL) return NULL;
return *childPatch;
}

Patch* Triangle::GetParentPatch(void)
{
return *parentPatch;
}

TriangleQue.h
*****************************************************************************/
#ifndef TRIANGLEQUE_H
#define TRIANGLEQUE_H
#include "QueItem.h"
#include "Triangle.h"
class Triangle;
typedef Triangle* QueData;
class QueItem;
class TriangleQue {
public:
    TriangleQue();
    ~TriangleQue();
    //Create and insert a new queue item, either at back of upper queue,
// or at front of lower queue
void InsertMiddle(QueData newData1, QueData newData2, float priority);
// Insert an existing queue item
void InsertMiddle(QueItem* newItem);
// Remove an item from the triangle queue
float Remove(QueData removeData);
// Get the front of the queue
QueItem* GetFront();
// Get the back of the queue
QueItem* GetBack();
// Get and remove the front of the queue
QueItem* RemoveFront();
// Get and remove the back of the queue
QueItem* RemoveBack();

// Set the current threshold value for this queue
void SetThreshold(float newThreshold) { threshold = newThreshold; }
// Get the current threshold value for this queue
float GetThreshold(void) { return threshold; }

private:
float threshold; // Current threshold for this queue

// Pointers to the front and back items of the two queues
QueItem* upperFront;
QueItem* lowerFront;
QueItem* upperBack;
QueItem* lowerBack;
QueItem* tmpItem;
};

TriangleQue.cpp

#include "CommonInclude.h"
#include "TriangleQue.h"
#include "Helper.h"

TriangleQue::TriangleQue()
{
    upperFront = NULL;
    upperBack = NULL;
    lowerFront = NULL;
    lowerBack = NULL;
    threshold = 0.0;
}

TriangleQue::~TriangleQue()
{
    // Deallocate the upper queue
    while(upperFront!=NULL) {
        tmpItem = upperFront;
        upperFront = upperFront->nextItem;
        delete tmpItem;
    }

    // Deallocate the lower queue
    while(lowerFront!=NULL) {
        tmpItem = lowerFront;
        lowerFront = lowerFront->nextItem;
        delete tmpItem;
    }
}

void TriangleQue::InsertMiddle(QueData newData1, QueData newData2, float priority)
{
    if(priority<threshold) {
        // Insert to front of lower queue
        if(lowerFront == NULL) {
            // No previous elements in lower queue
            lowerFront = new QueItem(newData1,newData2,NULL,NULL,priority);
            lowerBack = lowerFront;
        } else {
            lowerFront = new QueItem(newData1,newData2,lowerFront,NULL,priority);
            lowerFront->nextItem->prevItem = lowerFront;
        }
    } else {
        // Insert at front of lower queue
        if(upperFront == NULL) {
            // No previous elements in lower queue
            upperFront = new QueItem(newData1,newData2,upperFront,NULL,priority);
            upperBack = upperFront;
        } else {
            upperFront = new QueItem(newData1,newData2,upperFront,upperFront->nextItem,priority);
            upperFront->nextItem->prevItem = upperFront;
        }
    }
}
void TriangleQue::InsertMiddle(QueItem* newItem) {
    if (newItem->itemPriority < threshold) {
        //Insert to front of lower queue
        if (lowerFront == NULL) {
            //No previous elements in lower queue
            newItem->nextItem = NULL;
            newItem->prevItem = NULL;
            lowerFront = newItem;
            lowerBack = newItem;
        } else {
            newItem->nextItem = lowerFront;
            newItem->prevItem = NULL;
            lowerFront->prevItem = newItem;
            lowerFront = newItem;
        }
    } else {
        //Insert to back of upper queue
        if (upperBack == NULL) {
            //No previous elements in upper queue
            newItem->nextItem = NULL;
            newItem->prevItem = NULL;
            upperBack = newItem;
            upperFront = newItem;
        } else {
            newItem->nextItem = NULL;
            newItem->prevItem = upperBack;
            upperBack->nextItem = newItem;
            upperBack = newItem;
        }
    }
}

float TriangleQue::Remove(QueData removeData) {
    //Find a queue item containing the removeData, and remove it from the queue
    //Start looking in the upper queue, if not found there proceed with looking
    //in the lower queue, return the priority of the found item.
    float returnPrio = 0.0;
    QueItem* tmp = upperFront;
    while (tmp->nextItem != NULL && tmp->data1 != removeData && tmp->data2 != removeData) {
        tmp = tmp->nextItem;
    }
    if (tmp != NULL) {
        if (tmp->data1 == removeData || tmp->data2 == removeData) {
            //We found it in upper queue
            if (tmp == upperFront) {
                //It was the front item
                upperFront = upperFront->nextItem;
                if (upperFront == NULL) {
                    upperBack = NULL;
                } else {
                    upperFront->prevItem = NULL;
                }
            } else if (tmp == upperBack) {
                //It was the back item
                upperBack = upperBack->prevItem;
                if (upperBack == NULL) {
                    upperFront = NULL;
                } else {
                    upperBack->nextItem = NULL;
                }
            } else {
                //It was an item in between the front and back items
            }
        }
    }
}
tmp->prevItem->nextItem = tmp->nextItem;
tmp->nextItem->prevItem = tmp->prevItem;
}
returnPrio = tmp->itemPriority;
delete tmp;
return returnPrio;
}

//Element was not found in upper queue, look in lower queue
while(tmp->nextItem!=NULL && tmp->data1!=removeData && tmp->data2!=removeData) {
    tmp = tmp->nextItem;
}
if(tmp!=NULL) {
    if(tmp->data1==removeData || tmp->data2==removeData) {
        //We found it in lower queue
        if(tmp==lowerFront) {
            //It was the front item
            lowerFront = lowerFront->nextItem;
            if(lowerFront == NULL) {
                lowerBack = NULL;
            } else {
                lowerFront->prevItem = NULL;
            }
        } else if(tmp==lowerBack) {
            //It was the back item
            lowerBack = lowerBack->prevItem;
            if(lowerBack == NULL) {
                lowerFront = NULL;
            } else {
                lowerBack->nextItem = NULL;
            }
        } else {
            //It was an item in between the front and back items
            tmp->prevItem->nextItem = tmp->nextItem;
tmp->nextItem->prevItem = tmp->prevItem;
        }
        returnPrio = tmp->itemPriority;
delete tmp;
return returnPrio;
}
return returnPrio;
}

QueItem* TriangleQue::GetFront() {
//If upper queue is not empty, return front of it, else return the front of
//the lower queue.
return (upperFront!=NULL) ? upperFront : lowerFront;
}

QueItem* TriangleQue::GetBack() {
//If lower queue is not empty, return the back of it, else return the back
//of the upper queue.
return (lowerBack!=NULL) ? lowerBack : upperBack;
}

QueItem* TriangleQue::RemoveFront() {
if(upperFront!=NULL) {
    //Remove the front of the upper queue
    tmpItem = upperFront;
    upperFront = upperFront->nextItem;
    if(upperFront == NULL) {
        upperBack = NULL;
    } else {
        upperFront->prevItem = NULL;
    }
return tmpItem;
} else if(lowerFront!=NULL) {
    //The upper queue was empty, remove the front of lower queue
    tmpItem = lowerFront;
    lowerFront = lowerFront->nextItem;
    if(lowerFront == NULL) {
        lowerBack = NULL;
    } else {

lowerFront->prevItem = NULL;
return tmpItem;
}
return NULL;
}

QueItem* TriangleQue::RemoveBack()
{
if(lowerBack!=NULL) {
    //Remove the back of the lower queue
    tmpItem = lowerBack;
    lowerBack = lowerBack->prevItem;
    if(lowerBack == NULL)
        lowerFront = NULL;
    else
        lowerBack->nextItem = NULL;
    return tmpItem;
} else if(upperBack!=NULL) {
    //The lower queue is empty, remove back of upper queue
    tmpItem = upperBack;
    upperBack = upperBack->prevItem;
    if(upperBack == NULL) { 
        upperFront = NULL;
    } else {
        upperBack->nextItem = NULL;
    }
    return tmpItem;
}
return NULL;
}

Vertex.h

*****************************************************************************/
* Vertex.h                                                                   *
* Author: Anders Dahlbom                                                     *
* Last modified: 030603                                                      *
*                                                                            *
* Purpose: Class for capturing the properties of a vertex.                   *
*                                                                            *
*          This class can also be used as a vector                           *
*                                                                            *
******************************************************************************/

#ifndef VERTEX_H
#define VERTEX_H

#include "Matrix4f.h"

class Vertex {
public:
    Vertex();
    Vertex(float in_x, float in_y, float in_z);
    ~Vertex();

    //Addition, if used as vector
    Vertex operator+(const Vertex& v) const;
    //Subtraction, if used as vector
    Vertex operator-(const Vertex& v) const;
    //Division by constant, if used as vector
    Vertex operator/(const float& f) const;
    //Multiplication, if used as vector
    Vertex operator*(const float& f) const;
    //Comparison
    bool operator==(const Vertex& v) const;
    //Addition of current, if used as vector
    const Vertex operator+=(const Vertex& v);
    //Subtraction of current, if used as vector
    const Vertex operator-=(const Vertex& v);
    //Division of current, if used as vector
    const Vertex operator/=(const float& f);

    //Set the elements of this vertex
    void Set(float x, float y, float z);
    //Transform this vertex by a matrix
    void Transform(Matrix4f* matrix);
//Get the X, Y, and Z components
float GetX(void);
float GetY(void);
float GetZ(void);

//Calculate the length, if used as vector
float LengthOf(void);

//Calculate the squared length, if used as vector
float SquaredLength(void);

//Compute the dot product, if used as vector, (u)*(v)
float DotProduct(const Vertex& v) const;

//Compute the cross product, if used as vector, (u)x(v)
Vertex CrossProduct(const Vertex& v) const;

float p[3]; //The elements
};
#endif

Vertex.cpp

#include <math.h>
#include <vcldb.h>

#include "CommonInclude.h"
#include "Vertex.h"

Vertex::Vertex()
{
    p[0] = 0.0;
p[1] = 0.0;
p[2] = 0.0;
}

Vertex::Vertex(float in_x, float in_y, float in_z)
{
    p[0] = in_x;
p[1] = in_y;
p[2] = in_z;
}

Vertex::~Vertex()
{
}

Vertex Vertex::operator+(const Vertex& v) const
{
    Vertex temp[p[0],p[1],p[2]];
temp.p[0] += v.p[0];
temp.p[1] += v.p[1];
    return temp;
}

Vertex Vertex::operator-(const Vertex& v) const
{
    Vertex temp[p[0],p[1],p[2]];
temp.p[0] -= v.p[0];
temp.p[1] -= v.p[1];
    return temp;
}

Vertex Vertex::operator/(const float& f) const
{
    if(f==0.0) throw EZeroDivide("Cannot divide Vertex with zero!");
    Vertex temp = Vertex(p[0],p[1],p[2]);
temp.p[0] = temp.p[0]/f;
temp.p[1] = temp.p[1]/f;
    return temp;
}

Vertex Vertex::operator*(const float& f) const
{
    Vertex temp[p[0],p[1],p[2]];
temp.p[0] = temp.p[0]*f;

temp.p[1] = temp.p[1]*f;
return temp;
}

bool Vertex::operator==(const Vertex& v) const
{
    return false;
}

const Vertex& Vertex::operator+=(const Vertex& v)
{
    p[0] += v.p[0];
p[1] += v.p[1];
    return *this;
}

const Vertex& Vertex::operator-=(const Vertex& v)
{
    p[0] -= v.p[0];
p[1] -= v.p[1];
    return *this;
}

const Vertex& Vertex::operator/=(const float& f)
{
    if(f!=0) {
        p[0] /= f;
p[1] /= f;
    }
    return *this;
}

void Vertex::Set(float x, float y, float z)
{
    p[0] = x;
p[1] = y;
p[2] = z;
}

void Vertex::Transform(Matrix4f* matrix)
{
    float td[3];
    for(int i=0;i<3;i++)
    { p[i] = td[i];
    }
}

float Vertex::GetX(void)
{
    return p[0];
}

float Vertex::GetY(void)
{
    return p[1];
}

float Vertex::GetZ(void)
{
    return p[2];
}

float Vertex::LengthOf(void)
{
    float result = (p[0]*p[0])+(p[1]*p[1])+(p[2]*p[2]);
    if(result>0)
    { result = sqrt(result);
        return result;
    }
float Vertex::SquaredLength(void)
{
}

float Vertex::DotProduct(const Vertex& v) const
{
}

Vertex Vertex::CrossProduct(const Vertex& v) const
{
    Vertex temp;
    temp.p[0] = p[1]*v.p[2]-p[2]*v.p[1];
    temp.p[1] = p[2]*v.p[0]-p[0]*v.p[2];
    temp.p[2] = p[0]*v.p[1]-p[1]*v.p[0];
    return temp;
}

VertexExt.h

/******************************************************************************
* VertexExt.h                                                             *
* Author: Anders Dahlbom                                                   *
* Last modified: 030603                                                     *
*                                                                            *
* Purpose: Class for having an extended vertice.                           *
*          This class extends the vertex class, by introducing:            *
*             -Texture coordinate (u,v)                                  *
*             -Normal vector                                              *
*             -Update flag                                               *
*                                                                            *
*           The update flag is used if this vertex has been displaced in    *
*           association with a deformation.                             *
*                                                                            *
******************************************************************************/

#ifndef VERTEXEXT_H
#define VERTEXEXT_H

#include "Vertex.h"

class VertexExt : public Vertex {
public:
    VertexExt();
    VertexExt(float in_x, float in_y, float in_z);
    VertexExt(float in_x, float in_y, float in_z, float u, float v);

    //Has this vertex been changed
    bool HasChanged(void);
    //Set the changed flag for this vertex
    void SetChanged(unsigned char hC) {
        hasChanged = hC;
    }

    float uv[2]; //Texture coordinates
    float n[3]; //Normal vector elements

private:
    unsigned char hasChanged;
};
#endif

VertexExt.cpp

#include "CommonInclude.h"
#include "VertexExt.h"

VertexExt::VertexExt() : Vertex()
{
    hasChanged = 0;
    uv[0] = 0.0f;
    uv[1] = 0.0f;
    n[0] = 0.0f;
    n[1] = 1.0f;
    n[2] = 0.0f;
}
VertexExt::VertexExt(float in_x, float in_y, float in_z) : Vertex(in_x, in_y, in_z)
{
    hasChanged = 0;
    uv[0] = 0.0f;
    uv[1] = 0.0f;
    n[0] = 0.0f;
    n[1] = 1.0f;
    n[2] = 0.0f;
}

VertexExt::VertexExt(float in_x, float in_y, float in_z, float u, float v) :
Vertex(in_x, in_y, in_z)
{
    hasChanged=0;
    uv[0] = u;
    uv[1] = v;
    n[0] = 0.0f;
    n[1] = 1.0f;
    n[2] = 0.0f;
}

bool VertexExt::HasChanged(void)
{
    if(hasChanged==0)
        return false;
    --hasChanged;
    return true;
}