Examensarbete
LITH-ITN-MT-EX--05/062--SE

Extending a battlefield simulator with large scale terrain rendering and flight simulator functionality

Daniel Johansson

2005-11-29

Department of Science and Technology
Linköpings Universitet
SE-601 74 Norrköping, Sweden

Institutionen för teknik och naturvetenskap
Linköpings Universitet
601 74 Norrköping

LITH-ITN-MT-EX--05/062--SE

Extending a battlefield simulator with large scale terrain rendering and flight simulator functionality

Examensarbete utfört i medieteknik
vid Linköpings Tekniska Högskola, Campus
Norrköping

Daniel Johansson

Handledare Johan Hedström
Handledare Ken Museth
Examinator Ken Museth

Norrköping 2005-11-29

Avdelning, Institution
Division, Department

Institutionen för teknik och naturvetenskap

Department of Science and Technology

Datum
Date

2005-11-29

Språk Language	**Rapporttyp** Report category
☐ Svenska/Swedish	Examensarbete
☒ Engelska/English	☐ B-uppsats
☐ _______________	☐ C-uppsats
	☒ D-uppsats
	☐ _______________

ISBN

ISRN **LITH-ITN-MT-EX--05/062--SE**

Serietitel och serienummer **ISSN**
Title of series, numbering

URL för elektronisk version

Titel
Title Extending a battlefield simulator with large scale terrain rendering and flight simulator functionality

Författare
Author Daniel Johansson

Sammanfattning
Abstract Simulation of modern battlefield scenarios on consumer PC:s deal with a number of limitations, many of them related to the limited performance of a normal PC compared to workstations and servers. Specifically, the visualization of realistic large scale outdoor environments is problematic because of the large amount of data required to describe its contents. This becomes especially problematic in simulations of fast moving vehicles such as aircrafts, where one needs to maintain high frame rates while having high visual detail for orientation and targeting. This thesis proposes a method of generating realistic outdoor environments from actual geological data and then rendering it efficiently using an improved level of detail algorithm within a proprietary battle simulation framework. We also show how to integrate an open source Flight Dynamics Model (FDM) into the simulation framework for future hybrid simulations involving aircrafts.

Nyckelord
Keyword Visualization, Terrain rendering, Level of detail control, Battlefield simulation

Simulation of modern battlefield scenarios on consumer PC:s deal with a number of limitations, many of them related to the limited performance of a normal PC compared to workstations and servers. Specifically, the visualization of realistic large scale outdoor environments is problematic because of the large amount of data required to describe its contents. This becomes especially problematic in simulations of fast moving vehicles such as aircrafts, where one needs to maintain high frame rates while having high visual detail for orientation and targeting. This thesis proposes a method of generating realistic outdoor environments from actual geological data and then rendering it efficiently using an improved level of detail algorithm within a proprietary battle simulation framework. We also show how to integrate an open source Flight Dynamics Model (FDM) into the simulation framework for future hybrid simulations involving aircrafts.

1 Introduction

Real time simulations of battlefield scenarios are becoming more widespread and used in the defense research community. Many kinds of tactical and strategical simulations can be performed in this manner, which in the end results in a more cost efficient training and the possibility to explore new research topics. The simulation concept is especially useful for maneuvers that involves many participants, but have focus on training only one or a few specialized positions. The demand for a robust, fast and flexible application framework in these kinds of real time military simulations is high, and some commercial and non-commercial applications exist for these purposes. The immersion, realism and scalability of these applications are however rather limited, especially if one only considers software that runs on consumer hardware. The simulation framework used in this thesis is developed by the department for Man-System Interaction at the Swedish Defence Research Agency (FOI). It is intended to be a simple but yet powerful simulation environment for research applications. It uses a client-server system for communication between active agents, and is not unlike many modern computer games.

The visibility range and movement speed of the scenario participants define the scale of the active battlefield area. If we are to consider a hybrid simulation environment, where many different types of agents interact, the ability to change the level of detail of the environment becomes very important. Different agent types need different environment representations since the available processing power and data bandwidth needed to keep consistent frame rates can be insufficient.

In outdoor graphics scenarios the most important visual component is the terrain geometry. Therefore this report focuses mainly on a method for displaying large sets of terrain geometry (> 25km2) with high detail (> 0.1 sample/m). This functionality was also missing in the current version of the MSI simulation framework. Section 3.1 describes the algorithm that this part of the thesis is heavily based on. Section 3.2 describes the additions end extensions that I have made to this algorithm.

Section 4 describes the integration of the JSBSim Flight Dynamics Model into the MSI framework for flight simulation functionality, which is a very important part that was missing in the previous version of the framework. Finally some possible future extensions and improvements to the application are also discussed.

2 Related Work

Terrain LOD algorithms can be categorized by looking at the way hierarchical mesh refinement operations are used to adapt the surface tessellation. Among the more common ones are:

Irregular meshes provide the best approximation for a given number of faces. One drawback with these methods is that they require tracking of adjacency information and refinement dependencies. Some methods uses Delauney triangulations while others allow arbitrary connectivity's. Some examples are [Cohen-Or, D., Levanoni, Y. 1996.], [Hoppe, H. 1998.] and [El-Sana, J., Varshney, A. 1999.].

Binary tree hierarchies use recursive bisection of right-angle triangles to greatly simplify memory layout and traversal algorithms. The main drawback with these methods is that they involve random memory accesses and require immediate-mode rendering. For example [Duchaineau, M et. al. 1997.], [Röttger, S. et.al. 1998.] and [Lindstrom, P., Pacucci, V. 2002.].

Tiled blocks partition the terrain into square patches that are tessellated at different resolutions. The main challenge is to stitch the block boundaries seamlessly. Examples are [Bishop, L. et. al. 1998.], [Wagner, D. 2004.] and [de Boer, W., 2000.].

The terrain renderer module implemented in this thesis project is heavily based upon an algorithm by Frank Losasso and Hughes Hoppe [Losasso, F., Hoppe, H. 2004.], hereby referred to as "Geometry Clipmaps". It is built upon the principle that the rate at which the terrain geometry is being sampled from the height map is higher close to the viewer, and lower far away. The terrain is sampled in nested regular grids centered about the viewer. This approach has similarities with the methods in the tiled blocks category above, and common the LOD treatment of texture images [Williams, L. 1983.], called "mipmapping". A mipmap is a texture filtered into a texture pyramid of power-of-two grids. The mipmap level rendered is a function of screen space derivatives and not the image itself. Another related method is the texture clipmap [Tanner, C. et. al. 1998.] which is a view-dependent subset of the mipmap image pyramid with fast incremental updates, which allows the exploration of huge texture images.

Recently a technical article published by Nvidia [Asirvathan, A., Hoppe, H. 2005.] shows that it is possible to further move the Geometry Clipmap algorithm on to the GPU. They show how to do normal calculation, prediction from coarser detail levels, and fractal detail synthesis directly in the GPU. They also present some details about how to efficiently pack and unpack data between different parts of the rendering pipeline.

In the area of battlefield simulators The Netherlands Organization for Applied Scientific Research (TNO) has a hybrid Land/Air simulator used for training Forward Air Controllers. The FACSIM - Forward Air Controller Training Simulator is a networked simulator environment running on high end workstations and involves both aircraft and ground troops.

3 Geometry Clipmap algorithm

3.1 Original implementation

3.1.1 Description

Geometry Clipmaps is a view-dependent Level Of Detail (LOD) algorithm aiming to minimize screen-space geometric error, the difference between the actual terrain and the terrain geometry representation in pixels, while trying to keep as high frame rates as possible to achieve visual continuity. The screen space geometric error combines the effects of viewer distance, surface orientation and surface geometry. The Geometry clipmap method will discard contributions of surface geometry and surface orientation, and will therefore only be based on the position of the viewer.

3.1.2 Data structure

The geometry clipmap consists of a pyramid of rectangular geometry patches of resolution n × m that are stored as vertex buffers [NVIDIA. 2003] in video memory. The geometry patches are regular; all vertices in a geometry patch are regularly spaced in straight rows and columns. These geometry patches are called detail levels. All detail levels have the same initial number of vertices, but have different spatial size so that triangles in lower detail levels have larger area.

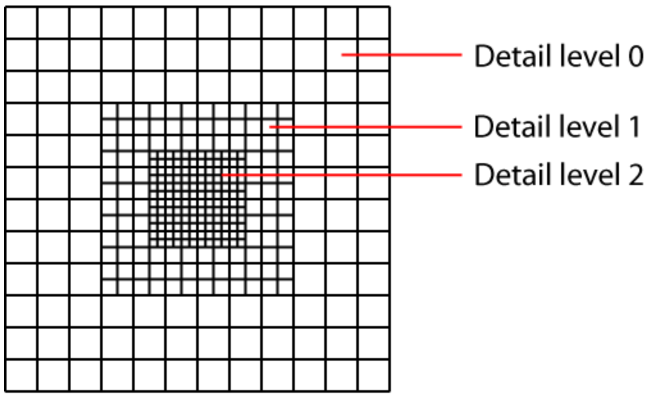

Figure 1: Clipmap viewed from above, higher detail levels are placed in the center of the lower detail level

Each detail level is accessed toroidally in video memory, which is with 2D wrap around addressing with modulo operations in (x, y). This allows for simple incremental updates by copying rectangular data blocks to the virtual edge of the detail level.

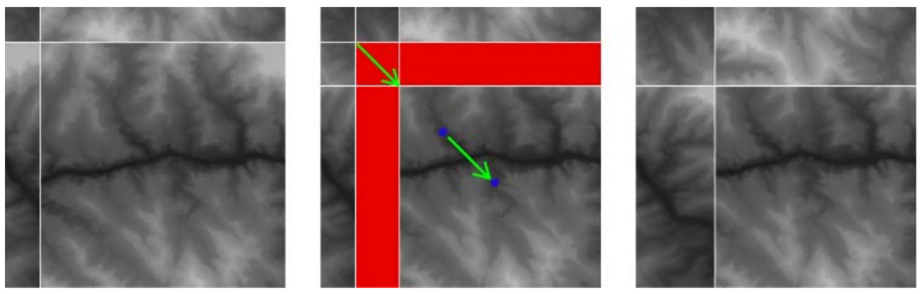

Figure 2: Toroidal wrap around addressing, when viewer moves new data is copied over old data in the red area

Each vertex stored in the vertex buffer contains a four component vector V = (x, y, z, zc), where (x, y, z) is the position of the vertex in world space coordinates and (zc) is the height value on the surface at the same (x, y) value in the coarser detail level. This value is used later on when we morph the detail levels to preserve visual continuity at the edges.

For each level in the clipmap, a set of rectangular regions are defined. The clip region is the spatial extent in (x, y) of the vertices stored in the vertex buffer for that level. The active region is the region that is centered exactly on the viewer, and is therefore the desired rendering area for that level. During viewer motion, the clip regions are filled with sampled height data at the edges in the direction of motion. We try to match the active regions as closely as possible, within a predefined sampling budget. The effect is that finer detail regions will fall behind if the viewer moves too fast.

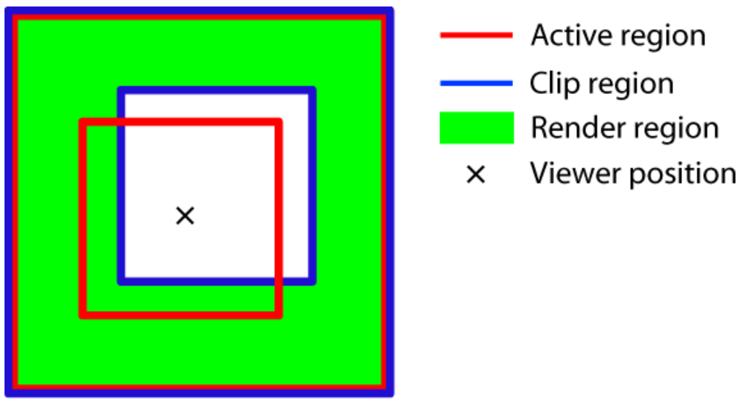

Figure 3: Active and Clip regions

Finally, the render region, is the hollow frame between clipregion(n), and clipregion(n+1). This is the area that actually should be rendered, and the hollow area is present to allow room for the render region for the finer detail level.

The render region is normally split into four rectangular regions - index regions. These exist to simplify the vertex index generation and to perform culling of areas not present in the viewport. The index regions shown in the figure below are the maximum possible area of the index buffers used for indexing the vertices when rendering the vertex buffer. The actual index buffers are usually smaller though due to view frustum culling described later in the report. A special case also occurs for the finest detail level in the detail level pyramid, where the hollow frame in the middle becomes unnecessary because no finer detail levels needs to be rendered. In this special case, only one index region is needed.

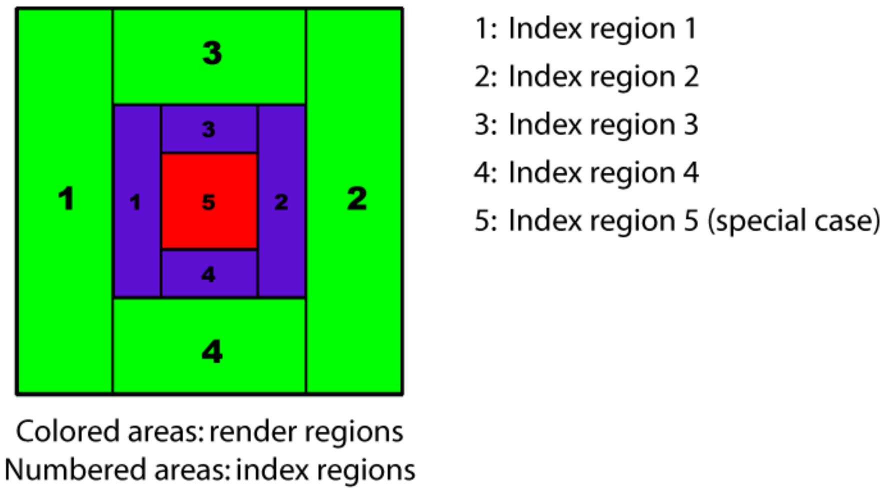
3
3
1
1
5
2
2
4
4
1: Index region 1
2: Index region 2
3: Index region 3
4: Index region 4
5: Index region 5 (special case)
Colored areas: render regions
Numbered areas: index regions

Figure 4: Render and Index regions

Previous terrain LOD schemes for terrain geometry were based not only on the view, but also on the local geometry. This approach often results in irregular meshes with random-access traversals and poor cache coherency, which conforms poorly to the GPU vertex pipeline. Since the Geometry Clipmap algorithm uses regular meshes it can easily utilize triangle strips to upload geometry to the graphics processor, which saves memory bandwidth and guarantees efficient rendering. The triangle strips method currently is the most efficient method of rendering triangle based geometry on today's graphics hardware. It is simply a list of triangles that share each others vertices. Each triangle is built using two vertices from the previous triangle, plus one new vertex.

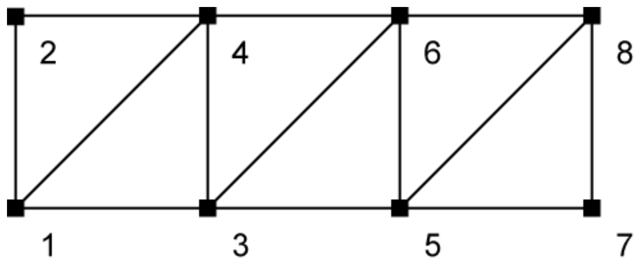

Figure 5: Triangle strip

3.1.3 GPU vertex memory management and toroidal updates

Vertex buffer objects [NVIDIA. 2003.] is a method in OpenGL for storing geometry data on the video memory of the graphics card. Since incremental updates can be performed for vertex buffer objects with the glBufferSubData() call and modern graphics cards have lots of RAM that can be used to cache geometry and textures, a vertex patch abstraction class that caches all terrain geometry directly on the GPU was written. GPU memory needs to be statically preallocated at the initialization because of performance issues with dynamic memory allocation. Since the vertex data needs to be stored in a 1D array, the 2D vertex data also needs to be layed out into a 1D array before uploading to vertex memory. A "vertex scratch pad" is therefore allocated which is simply a chunk of data big enough to contain the vertex data within the clip region at all times. Modulo operations are used to access the 2D grid in the 1D array of data, and also for wrap-around addressing at the edges of the "scratch pad". Four memory offsets are stored as member variables in the vertex patch abstraction class to keep track of where in memory the spatial min and max of the vertex data is stored. These memory offsets are used to generate the primitive index lists needed for the rendering calls, and to calculate where in the static 1D array the access points for the data are.

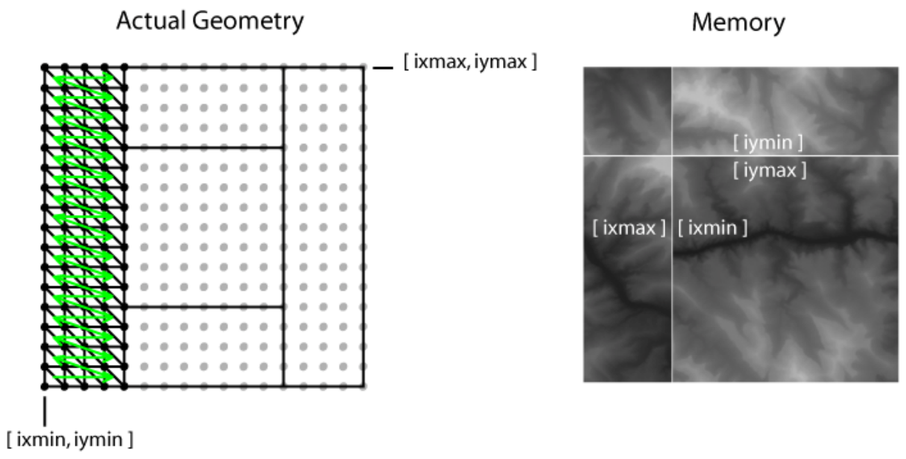

Figure 6: Vertex patch with memory offsets

The "tearing" of the mesh that would be present if one uses the glDrawArrays() rendering call can be avoided by using the glMultiDrawArrays() call. This call will split up the stream of indexes into several separated primitives at a specified interval. It will therefore avoid the long triangles between rows that will appear in the glDrawArrays() call due to that the long sequence of indexes will be interpreted as one single primitive.

The accessing functions for the vertex patch class can do incremental add/remove operations on the edges of the patch, or overwrite the whole patch at once. The incremental operations are very useful during user movement, where we only need to add or remove small L-shaped or rectangular sections of data at the edges of the patches to keep the clip regions fully updated and in sync with the active regions.

3.1.4 Sample limit

To guarantee smooth frame rates at all times even when the user moves fast over the terrain, a sample limit per frame is used. It guarantees that the CPU does not need to fetch and interpolate more samples from the height map than within a predefined budget. The clipmap is processed in a coarse to fine order, so the result will be that during fast user motion, finer levels will fall behind and are actually cropped against the coarser. This sample budget may have to be adjusted on each client using the clipmap algorithm. It depends on many parameters, such as CPU and memory speed, and the speed of the data bus between the main memory and the graphics memory.

3.1.5 Geomorphing

The algorithm described so far suffers from gaps between the render regions at different levels. To avoid these discontinuities at the borders between levels, geomorphing is needed, i.e. interpolation between lower and higher frequency detail levels. This morphing function is a function of the spatial (x, y) grid coordinates of the terrain vertices relative to those of the viewport (v_x, v_y). The transition width w is chosen arbitrarily, but through experimentation we have noted that a transition width of n/10 works well, where n is the grid resolution. We use a vertex shader to perform the actual geomorphing, which uses ordinary linear interpolation for morphing the height value

$$z' = (1-\alpha_b)z + z_c$$

with the border blend parameter α_b as

$$\alpha_b = \max(\alpha_x, \alpha_y)$$

$$\alpha_x = \min\left(\max\left(\frac{|x - v_x| - \frac{x_{\max} - x_{\min}}{2} - w - 1)}{w}, 0\right), 1\right)$$

and similarly for α_y. One could of course use other types of interpolation functions, such as the smoothstep function for example, but we have not noted any visual improvements when trying this. v_x denotes the continuous position of the viewport in the extent of the current active region $(x_{\max}, x_{\min})$. The desired behavior of α_b is that it evaluates to 0 everywhere except at the transition region where it linearly ramps up to 1.

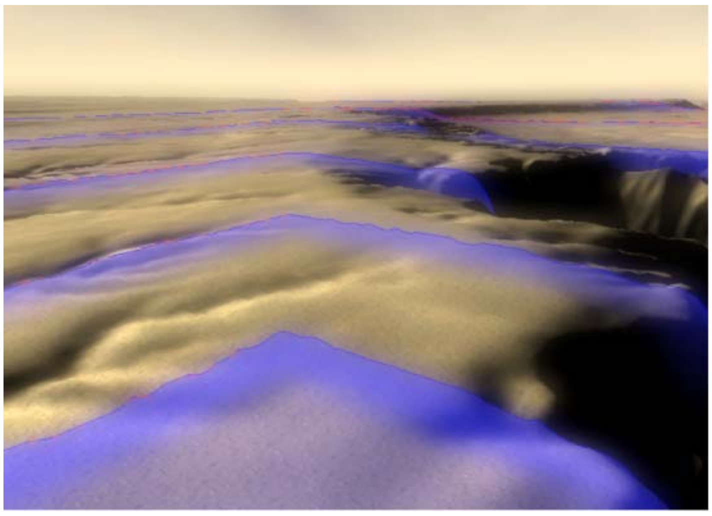

Figure 7: Border blend parameter

3.1.6 View culling

Since only parts of the terrain geometry are visible to the viewer, we can reduce the amount of rendered geometry by culling the hidden geometry. Back faces are automatically culled by OpenGL if the correct OpenGL rendering state is set, but geometry that lies outside of the camera frustum is not automatically discarded. We therefore need to perform this in software. To do this we calculate the intersection of the geometry present in a clipmap level and the camera frustum. To simplify this we maintain [zmin, zmax] for each clipmap level, and together with the rectangular extent of that level we build an axis aligned bounding box. We intersect this box with the camera frustum, and the resulting convex set is projected onto the XY plane. An axis aligned rectangle is created from this projection, and the vertex indices are generated by using this rectangular region instead of the whole active region of the clipmap level. To be sure that the viewport always is filled with terrain geometry the reindexing/culling algorithm needs to be performed each frame.

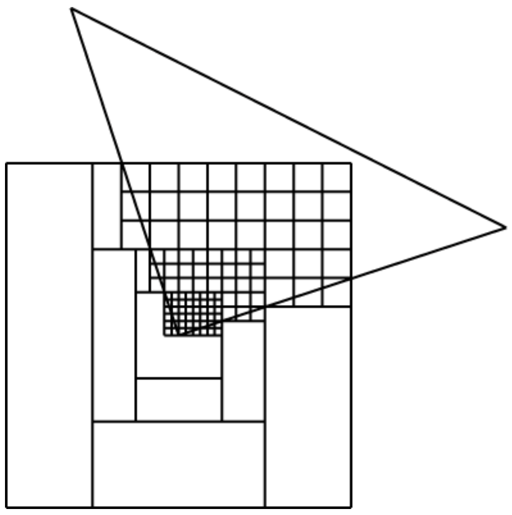
Figure 8: The camera frustum culls the index regions.

3.1.7 Texture maps

The clipmap data structure also stores texture maps. The texture maps are accessed and updated in the same manner as the vertex buffers for the geometry, but different OpenGL calls are used since these maps are treated as textures instead of vertex data. These textures do not need to have the same resolution as the terrain geometry, but they should be updated in the same incremental manner as the terrain when it is being updated. For instance for a normal map used for lighting the terrain, the resolution should be at least twice the resolution of the geometry buffer since one normal per vertex is too blurry. For other kinds of textures, such as the base color of the ground, the resolution of the texture can be much lower. Depending on what kinds of textures that are embedded into the clipmap structure, different kinds of shading methods can be used. We use simple phong lighting using normal maps for higher detail.

3.1.8 Data compression

The original geometry clipmap method uses a lossy compression scheme to fit terrain data sets within the RAM of the client. This is needed to avoid disk paging hiccups when processing very large elevation data sets. Since elevation data images often are very coherent in their nature, this offers a great opportunity for efficient data compression. To interact efficiently within the geometry clipmap structure the decompression algorithm needs to support region-of-interest queries at any power-of-two resolution.

In the geometry clipmap algorithm a simple pyramid compression scheme is used. Finer detail is predicted from the coarser one using interpolatory subdivision, and the residuals are encoded using a lossy image coder. All detail levels are quantized uniformly. They use the image coder described by [Malvar, H. 2000.].

3.1.9 Procedural detail synthesis

Procedural terrain synthesis allows the generation of terrains with infinite detail and extent. In this implementation procedural synthesis is used to create detail when the elevation data becomes too sparse. This creates more realistic and visually compelling renderings. The synthesis method used by the geometry clipmap algorithm is fractal noise displacement, by adding uncorrelated Gaussian noise to the upsampled terrain at the sub sample level. The noise variance is scaled at each level l to equal that in actual terrain, i.e. the variance of the residuals mentioned in the previous section. A table of precomputed Gaussian noise values is stored, and is indexed with modulo operations on the vertex coordinates.

3.2 Improvements

3.2.1 Pure vertex shader displacements

Graphics cards that support the shader standard "Shader Model 3.0" has the ability to perform displacement mapping directly on the hardware using a functionality called vertex texture fetches [Gerasimov, P. et. al. 2004.]. This means that the texture memory on the graphics card can be used at the vertex program level, which deals with the transforms and the lighting of vertices before the rasterization process. Since the z value of the terrain is sampled from the terrain height map in the vertex program, the big problem with this method is to also calculate or sample z_c which is the height value at the grid vertices of the coarser detail level. Remember from section 3.1.5 that this is needed to keep continuity at the edges. If we could use a mipmap to store the elevation data in texture memory, this problem would be solved easily since we would be able to access all detail levels in the clipmap pyramid with just one bound texture unit. But due to restrictions on current graphics cards we cannot create mipmaps for floating point textures and use them at the vertex program level, so this problem needs to be solved in a different manner. The way this is performed is instead by comparing the vertex position in (x, y) against the vertex grid of the coarser detail level. Then we can generalize the vertex position to three cases, and use a unique interpolation method for each of these three cases.

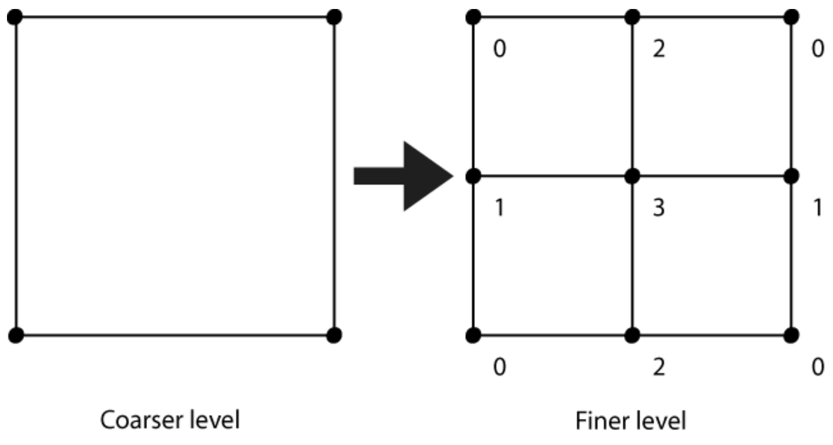

Figure 9: SM3.0 vertex classification

If a vertex is of case 0 where it corresponds to a vertex in the coarser detail level, no interpolation is needed. The height value z is equal to z_c and is directly sampled from the heightmap.

If a vertex is of case 1 or 2 it is on the line between two vertices that only differ in one dimension. We can sample these two vertices and linearly interpolate between them to get the height value on the surface of the coarser detail level, z_c.

If a vertex is of case 3 it is on the surface between the four neighboring vertices that make up the polygon. We need to sample these four vertices and perform bilinear interpolation to get the height value on the surface of the coarser detail level, z_c.

This method is implemented in a vertex shader and is in fact very similar to the CPU based method used for calculating z_c. The advantage of using the vertex shader instead of the CPU to calculate z_c is that the CPU can be offloaded some work and can perform other tasks instead. This algorithm is very parallel in its nature, and is therefore more suited to be done on a parallel processor such as the GPU. However, with the current generation of graphics cards, the GPU version of this algorithm is in fact slower that the CPU version. This is probably due to the fact that vertex texture fetches have very high latencies, and we need to perform up to four fetches for each vertex depending on what vertex type is being processed. The dynamic branching part of the code that classifies the vertex type may also be somewhat slow. Hopefully this will improve on next generation graphics cards.

3.2.2 Interpolated level clipping based on viewer height

High frequency detail levels can be discarded when the viewer is sufficiently high above the ground. To avoid visual "popping" when a detail level is clipped, finer detail levels are blended against the coarser before the actual discarding takes place. The height blend alpha parameter that is used for this blending is calculated in an ad-hoc manner as

$$\alpha_h = \left\{ \begin{array}{c} \dfrac{h}{0.4 * edge(n)}; n = n_{\max} \\ \dfrac{h}{(0.4 * edge(n+1) - 1)}; n < n_{\max} \end{array} \right\}$$

where

$$edge(n) = \max\left(edge_x(n), edge_y(n)\right)$$

and h is the height above the ground of the viewer. n is the current level in the clipmap being rendered, and $n_{\max}$ is the finest detail level number. The variables $edge_x$ and $edge_y$ is length of the sides of the current clip region in the clipmap. This is basically just a linear function depending on the height above the ground and the size of the clip region. The value α_h evaluates to a value between 0 and 1 when h is between edge(n) and edge(n+1). The reason that we have a special case when n = $n_{\max}$ is that otherwise we would use a level that not exists in the clipmap and would get an index out of bounds error. Therefore at the finest level α_h evaluates to a value between 0 and one when h is between edge(n) and 0.

The final blend between two detail levels becomes

$$\alpha = \min(\max(\alpha_b + \alpha_h, 0), 1)$$

when we add up the contribution of both the border blend alpha parameter and the height blend alpha parameter. The clamping by the min and max functions makes sure that we get a value between 0 and 1 when we add these blending parameters together.

3.2.3 Detail textures and detail masks

To increase visual realism, we need to add finer detail and color information to the terrain geometry. If we use regular textures to achieve this, the scale of the geometry makes the amount of texture data needed for sufficiently high detail quite extensive. There are however alternative techniques to achieve high visual detail.

The method we have chosen is one used in many PC games and is frequently called detail mapping. It uses a base texture with low detail to assign a base color to the ground, and then multiplies it with high frequency textures with different patterns or characteristics. These high frequency textures can be tiled and repeated over the surface, often the best results are achieved using several detail textures tiled in different frequencies over each other. We can also use masks to stance out which areas that should have a certain high frequency texture, and therefore create environments with many different realistic surfaces and combinations of these.

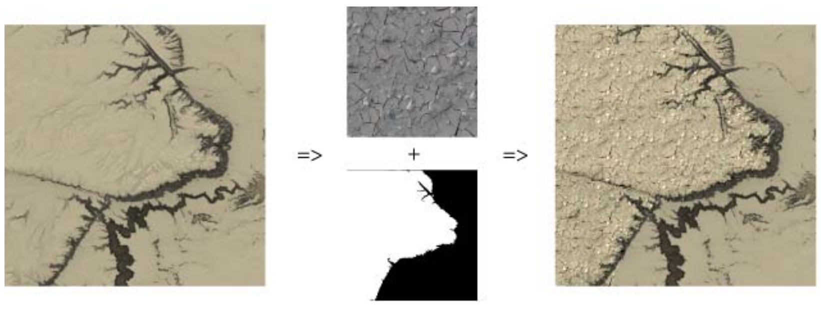

Figure 10: Detail textures and mask

The detail masks could be drawn by hand, which could be tedious for large environments, or generated automatically. The algorithm we have written for automatically generating masks takes the heightmap as input data, together with a number of parameters. We classify areas in the heightmap by either height or slope and write these masks to each channel in a 4 channel color texture with 8 bits per channel. The mask M is then fetched from the texture in the same way that the elevation data is sampled, and embedded into the vertex color attribute of the vertex. It is then treated as a varying 4-component vector between the vertex and the fragment shaders, and is therefore automatically interpolated between vertices. In the fragment shader, the following equations are used to mask out the detail texture Cd_n and multiply them with the base color texture Cb using the mask channel M_n. C_n is the accumulated surface color. Usually we use eight detail textures and mask channels so in that case the final surface color would be C_8.

$$C_n = \begin{cases} C_b; n = 0 \\ 2 * C_{n-1} * M_{n-1} * Cd_{n-1} + \frac{(1 - M_{n-1})}{2}; n > 0 \end{cases}$$

Texture coordinates are generated in the vertex shader, and are planar projections along the 3 axises. We use projections along all 3 axises to be able to map different detail textures on steep surfaces such as hillsides and cliffs. They will then use different axises for projecting the texture on than what a normal reasonably flat terrain would use. In the configuration file that is loaded at initialization of the terrain, one can specify exactly what detail texture and mask combination that should be used. One also specifies on which two axises the texture should be projected on as well as the tiling frequency.

3.3 Implementation details

3.3.1 Combining the Geomipmap and Geoclipmap methods

For the battle simulation applications that this terrain renderer is targeted for, the ability to visualize vegetation, buildings and roads is of high priority. In the MSI simulation framework this functionality and a terrain renderer based on [de Boer, W., 2000.] is available. This existing implementation has problems with visualizing large terrains (> 25km, 0.1sample/m) and this is the reason that the clipmap method is implemented. However, the geomipmap method is tightly integrated into the MSI application framework, is fast and stable, and has many ready extra functions not related to rendering. For instance collision detection used in rigid body simulation is present in this framework. It is also tightly integrated with the battlefield level loading system. Therefore we have chosen to merge these two methods into a hybrid, which uses the clipmap method in areas of low visual interest, and the geomipmap method in important areas, for instance target areas and areas with many important landmarks.

The two methods are combined by simply cutting out an area in the height map used by the clipmap method where we are interested in having higher visual detail. This cut out area of heightmap data will be used in the geomipmap method instead, or will be replaced by another heightmap file. We smooth the borders of the cut out area in the original height map to prevent sharp edges in the transition between the two methods. Note that when rendering, the clipmap method will still produce geometry in this area of interest, but it will be below the geometry of the geomipmap method. This is due to that it is not trivial to discard geometry in arbitrary areas with the geometry clipmap algorithm. We decided that the overhead of rendering the extra terrain would not be large and could therefore be accepted.

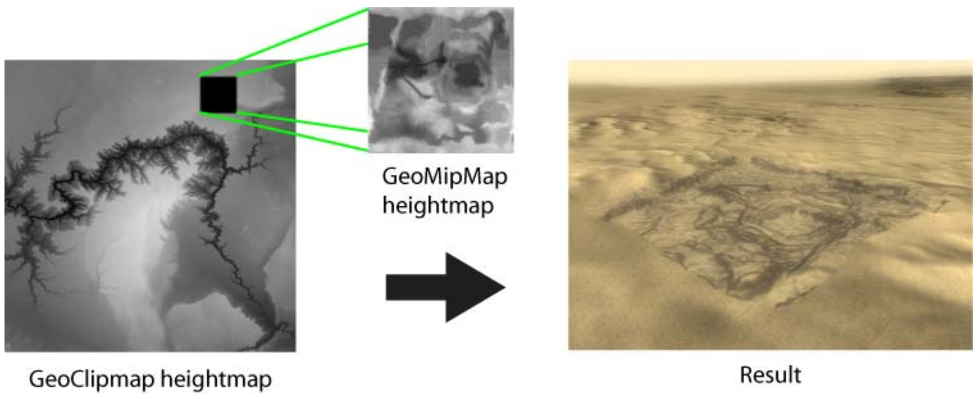

Figure 11: Merging geoclipmap and geomipmap terrains

3.3.2 Class Structure

We have tried to encapsulate common functions for all types of terrain geometry that used heightmap data in the base class HeightMapTerrain. A few examples of functions that resides in this class are heightmap file loading, normal map generation, detail mask generation and detail layer setup. The two specific terrain type classes GCMTerrain and GMMTerrain are then inherited from the HeightMapTerrain class.

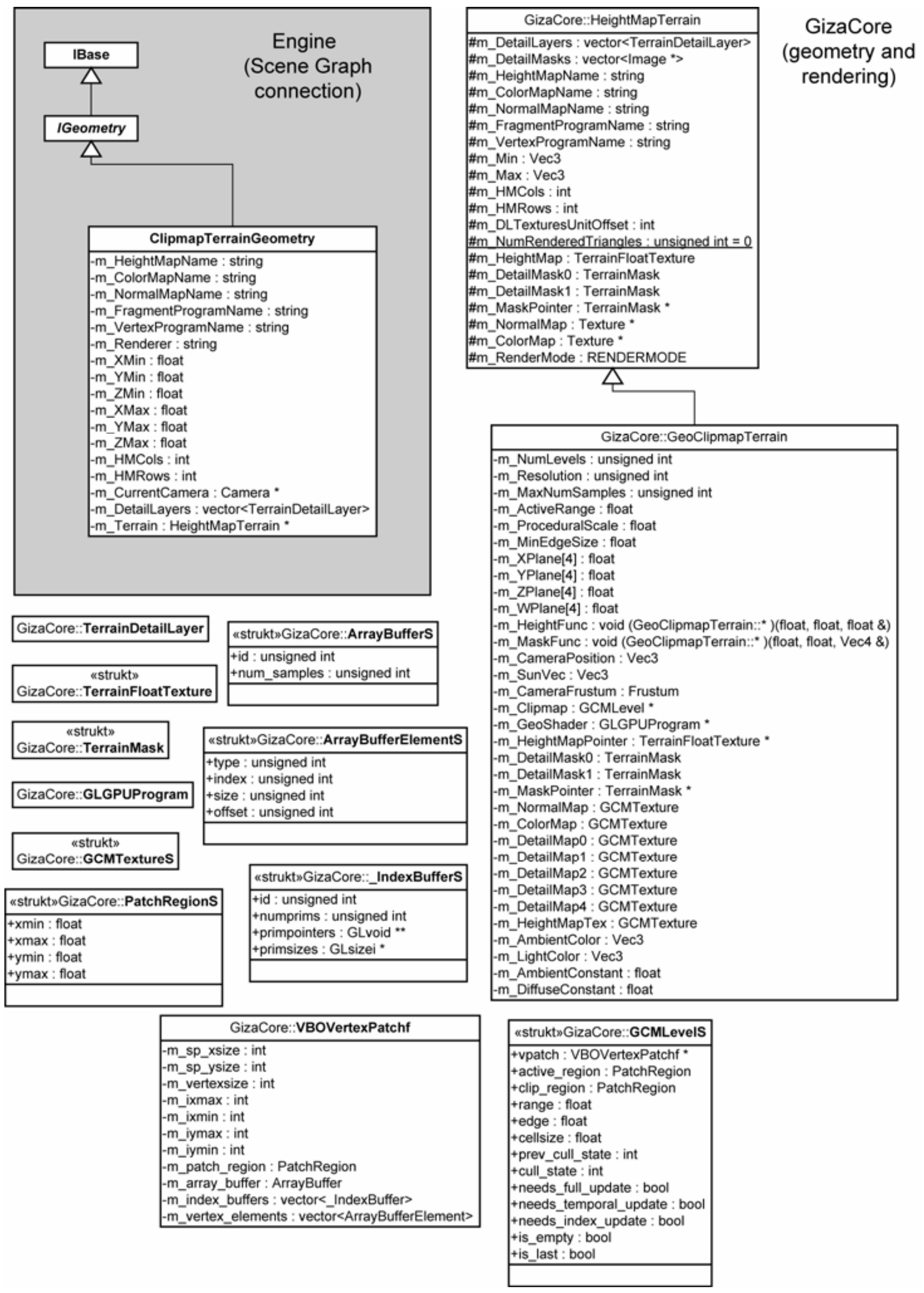
Engine (Scene Graph connection)
IBase
IGeometry
ClipmapTerrainGeometry
-m_HeightMapName : string
-m_ColorMapName : string
-m_NormalMapName : string
-m_FragmentProgramName : string
-m_VertexProgramName : string
-m_Renderer : string
-m_XMin : float
-m_YMin : float
-m_ZMin : float
-m_XMax : float
-m_YMax : float
-m_ZMax : float
-m_HMCols : int
-m_HMRows : int
-m_CurrentCamera : Camera *
-m_DetailLayers : vector<TerrainDetailLayer>
-m_Terrain : HeightMapTerrain *
GizaCore (geometry and rendering)
GizaCore::HeightMapTerrain
#m_DetailLayers : vector<TerrainDetailLayer>
#m_DetailMasks : vector<Image *>
#m_HeightMapName : string
#m_ColorMapName : string
#m_NormalMapName : string
#m_FragmentProgramName : string
#m_VertexProgramName : string
#m_Min : Vec3
#m_Max : Vec3
#m_HMCols : int
#m_HMRows : int
#m_DLTexturesUnitOffset : int
#m_NumRenderedTriangles : unsigned int = 0
#m_HeightMap : TerrainFloatTexture
#m_DetailMask0 : TerrainMask
#m_DetailMask1 : TerrainMask
#m_MaskPointer : TerrainMask *
#m_NormalMap : Texture *
#m_ColorMap : Texture *
#m_RenderMode : RENDERMODE
GizaCore::GeoClipmapTerrain
-m_NumLevels : unsigned int
-m_Resolution : unsigned int
-m_MaxNumSamples : unsigned int
-m_ActiveRange : float
-m_ProceduralScale : float
-m_MinEdgeSize : float
-m_XPlane[4] : float
-m_YPlane[4] : float
-m_ZPlane[4] : float
-m_WPlane[4] : float
-m_HeightFunc : void (GeoClipmapTerrain::*)(float, float, float &)
-m_MaskFunc : void (GeoClipmapTerrain::*)(float, float, Vec4 &)
-m_CameraPosition : Vec3
-m_SunVec : Vec3
-m_CameraFrustum : Frustum
-m_Clipmap : GCMLevel *
-m_GeoShader : GLGPUProgram *
-m_HeightMapPointer : TerrainFloatTexture *
-m_DetailMask0 : TerrainMask
-m_DetailMask1 : TerrainMask
-m_MaskPointer : TerrainMask *
-m_NormalMap : GCMTexture
-m_ColorMap : GCMTexture
-m_DetailMap0 : GCMTexture
-m_DetailMap1 : GCMTexture
-m_DetailMap2 : GCMTexture
-m_DetailMap3 : GCMTexture
-m_DetailMap4 : GCMTexture
-m_HeightMapTex : GCMTexture
-m_AmbientColor : Vec3
-m_LightColor : Vec3
-m_AmbientConstant : float
-m_DiffuseConstant : float
GizaCore::TerrainDetailLayer
«strukt» GizaCore::TerrainFloatTexture
«strukt» GizaCore::TerrainMask
GizaCore::GLGPUProgram
«strukt» GizaCore::GCMTextureS
«strukt»GizaCore::ArrayBufferS
+id : unsigned int
+num_samples : unsigned int
«strukt»GizaCore::ArrayBufferElementS
+type : unsigned int
+index : unsigned int
+size : unsigned int
+offset : unsigned int
«strukt»GizaCore::PatchRegionS
+xmin : float
+xmax : float
+ymin : float
+ymax : float
«strukt»GizaCore::_IndexBufferS
+id : unsigned int
+numprims : unsigned int
+primpointers : GLvoid **
+primsizes : GLsizei *
GizaCore::VBOVertexPatchf
-m_sp_xsize : int
-m_sp_ysize : int
-m_vertexsize : int
-m_ixmax : int
-m_ixmin : int
-m_iymax : int
-m_iymin : int
-m_patch_region : PatchRegion
-m_array_buffer : ArrayBuffer
-m_index_buffers : vector<_IndexBuffer>
-m_vertex_elements : vector<ArrayBufferElement>
«strukt»GizaCore::GCMLevelS
+vpatch : VBOVertexPatchf *
+active_region : PatchRegion
+clip_region : PatchRegion
+range : float
+edge : float
+cellsize : float
+prev_cull_state : int
+cull_state : int
+needs_full_update : bool
+needs_temporal_update : bool
+needs_index_update : bool
+is_empty : bool
+is_last : bool

Figure 12: Terrain class structure

3.3.3 Height data conversion

Height maps in the MSI application framework use a single channel, 16-bit float, Photoshop raw format file. Since some of the height data we use comes from the National Land Survey of Sweden it uses a height raster format provided in ASCII text files, and therefore needs to be converted into Photoshop raw.

A command line conversion program was created that converts ASCII height rasters. It can also bake these rasters into large raw files by tiling them next to each other. The output resolution can optionally be specified in the command line, and in that case the conversion process uses bilinear interpolation to avoid aliasing artifacts. We also implemented the ability to add fractal and cellular noise which can be used to create pseudo detail at the sub sample level of the input data.

3.3.4 Missing parts from original paper

In our implementation some parts from the original paper [Losasso, F., Hoppe, H. 2004.] are not fully implemented. These parts are the incremental texture updates (used for normal maps), height data compression and parts of the procedural detail synthesis. All these parts are used to minimize the amount of RAM and GPU RAM used for the application. We left these parts for possible future implementation mainly due to the fact that the time it takes to implement them is quite extensive. The fact that we had lots of RAM in the workstations that runs the application itself also played a role since we could get good result without these features.

4 JSBSim Flight Dynamics Model

4.1 Description

JSBSim is an open source Flight Dynamics Model (FDM) programmed in C++. It is designed for multi-platform compilation and is very modular in its nature. It is the main FDM used in the popular open source flight simulator "FlightGear", and has support for various aircrafts, both civilian and military. It is very configurable and custom aircraft are configured with relative ease. JSBSim can be used as a static library which can be linked into the application that will use it, or it can be run in a standalone mode where it communicates with the host application via TCP/IP.

4.2 Aircraft configurations

The aircraft configuration files (or model specification as they often are called) are stored in the extensive markup language format (XML). This is a good approach since it provides the ability to model different kinds of aircraft without specific program code. The configuration files are of different types depending on what is describes by the file, for instance aircraft specification, engine specification, thruster specification and initialization scripts. Several files can be involved in modeling a specific aircraft.

The set of models that comprise the JSBSim framework includes:

• Aerodynamics

• Equations of motion

• Propulsion

• Flight Control

• Ground Reactions

• Atmosphere

The Aerodynamics model is configured by providing lookup tables of measured coefficients for possible maneuvers of the aircraft, as well as indicating the structure and weight of the aircraft and the location of the propulsion system used. The propulsion system models the engines used by the aircraft, and their characteristics such as their power and fuel consumption. The Flight Control System consists of a collection of components that can be connected to represent the control laws of the airplane. The components consist of different filters, switches, gains, summers, etc.

We will not go into more detail about these configuration files and their contents here, instead we refer to [Berndt, J. 2004.] for more details.

4.3 MSI framework integration

The JSBSim simulation system is integrated into the MSI simulation framework by creating an AirplaneObject class which holds a member variable of the class FGFDMExec which is the entry point to the simulation. The AirplaneObject is derived from the VehicleObject, which is a general class for controllable agents within the MSI framework. It holds functions for connecting user input to actions and much more.

The JSBSim aircraft configuration files and initialization scripts are referenced from the configuration files of the AirplaneObject. They are loaded upon runtime creation of the AirplaneObject. The Flight Control System (FCS) object is accessed from a member function of the FGFDMExec object. The user input, such as a keyboard, mouse or a joystick, are passed directly over to the FCS object by using the FCS object "Set" functions, for example SetThrottleCmd(). After inputs have been copied using these functions, the simulation runs one discrete time step. Thereafter the position and rotation data output from the simulation are fetched using the FCS object "Get" functions and are then stored as current airplane state as member variables in the AirplaneObject. These state variables can then be accessed by the drawing routines to position geometry data.

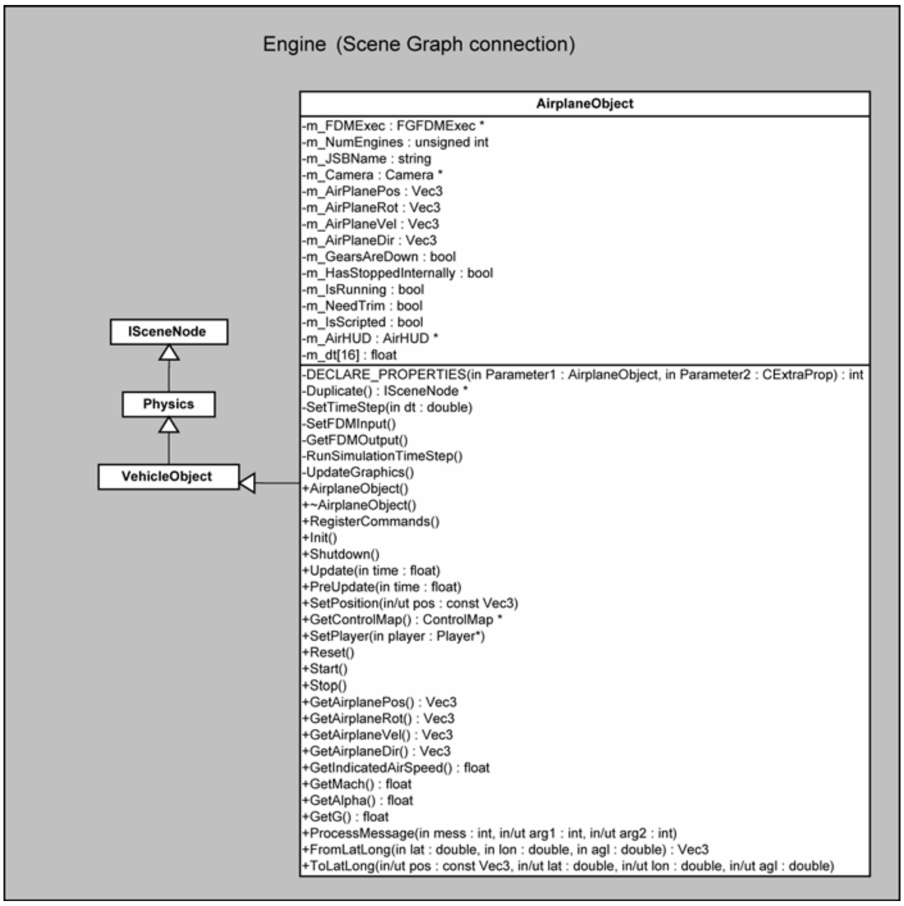

Figure 13: Aircraft object class structure

5 Results

All screenshots are taken on a Dell Inspiron 9300 laptop with an 1.6GHz Intel Centrino processor, 1GB RAM and a "Nvidia Geforce 6800 Go" graphics card with 256 MB of dedicated graphics RAM. The current frame rate when the screenshot was taken is indicated in the lower left corner of each screenshot. The frame rate usually varies between 30 and 60 fps, depending on at which height above the terrain geometry the frame is rendered. This is because at lower height, more detailed geometry is being rendered. The speed at which the user moves will not normally affect the frame rate because of the sample limit per frame. The result of this is that at high speed, higher detail levels fall behind and are not rendered. This will not normally affect the visual appearance much.

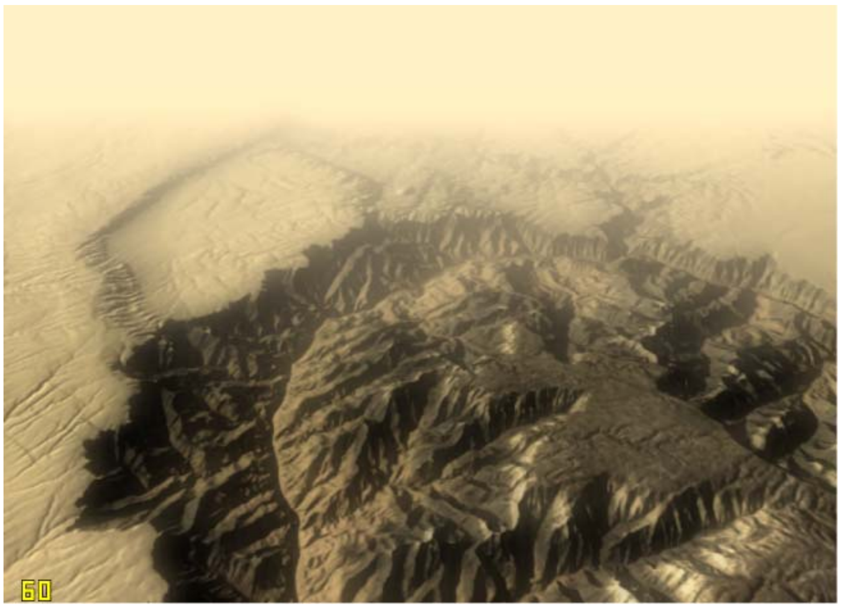

Figure 14: Grand Canyon viewed from approximately 10000m above ground level.

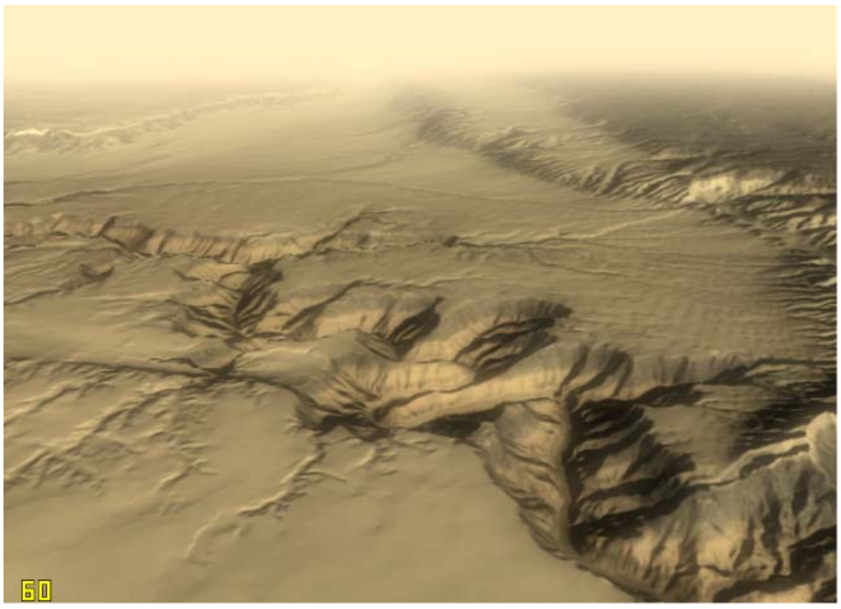

Figure 15: Grand Canyon viewed from approximately 6000mabove ground level.

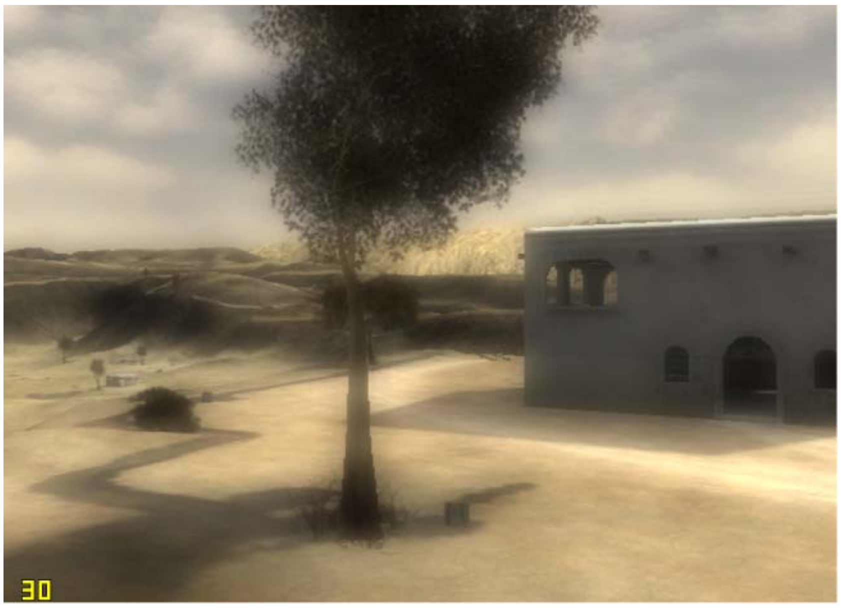

Figure 16: The "Desert Combat" level from Battlefield 1942 is embedded into the Grand Canyon level.

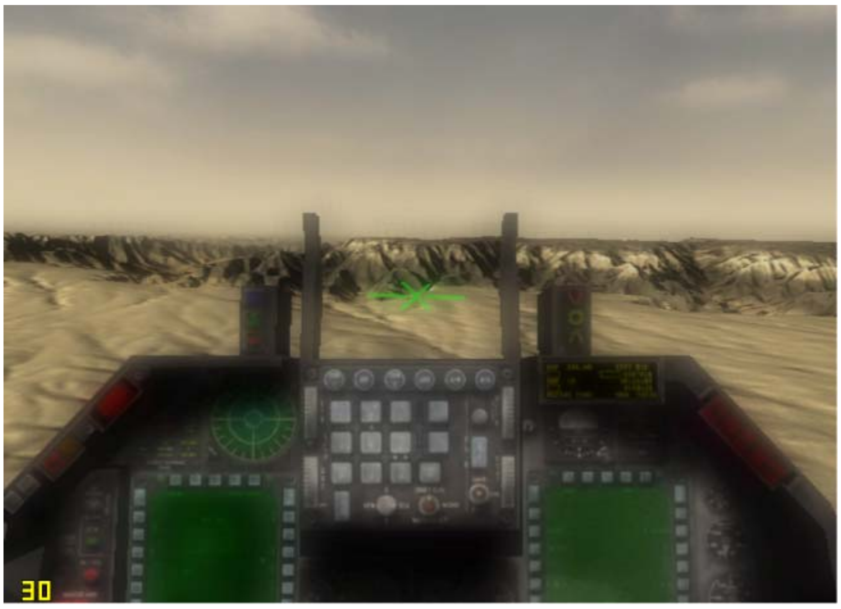

Figure 17: Approaching the Grand Canyon.

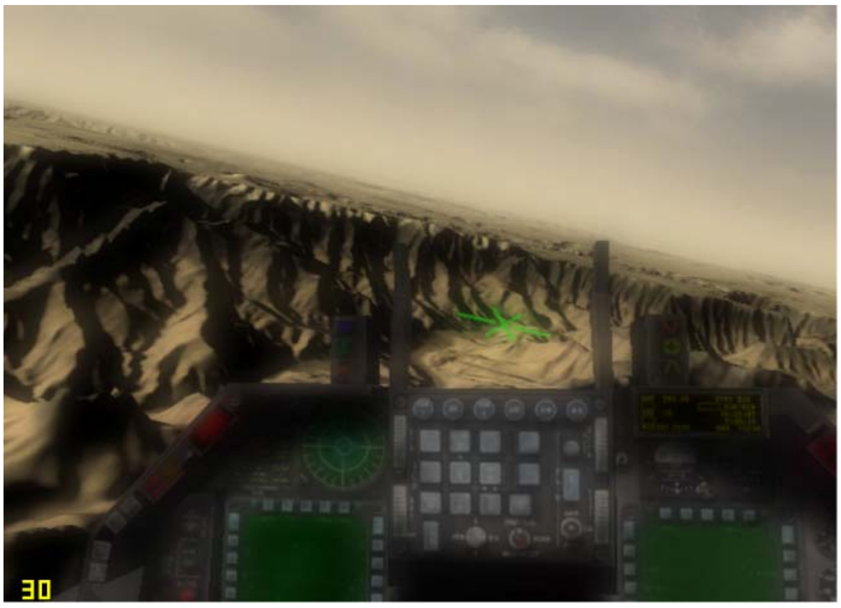

Figure 18: Flyby over Grand Canyon 1.

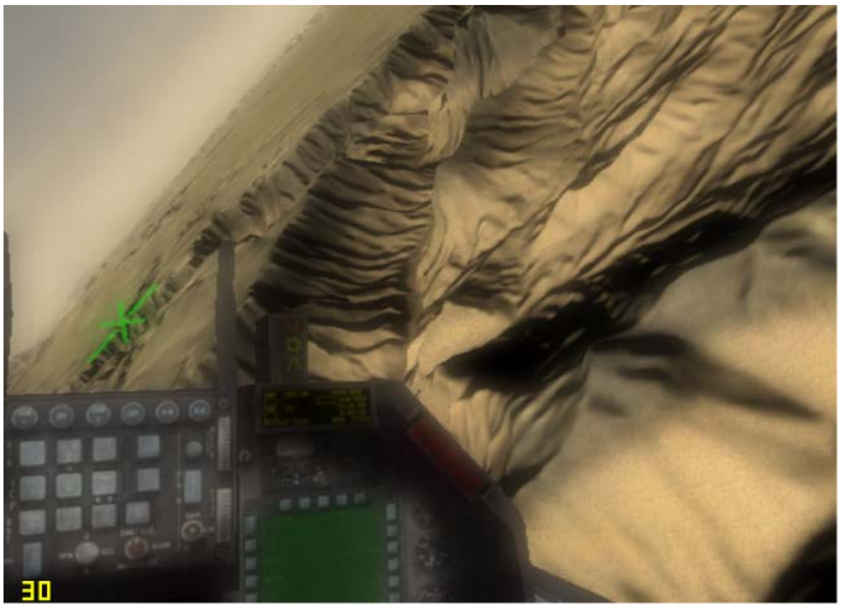

Figure 19: Flyby over Grand Canyon 2.

6 Conclusion and Future Work

6.1 Conclusion

With this thesis we have successfully showed that it is possible to use large outdoor environments with high geometric detail in real time simulation environments. We achieve real-time frame rates (>30fps) when outputting over 500000 vertices per frame on normal consumer laptop. We have also showed that by using an open source Flight Dynamics Model, we can get advanced flight simulation functionality without much programming and configuration.

6.2 Improving the clipmap method further

Since "Shader Model 3.0" also supports if-statements inside vertex programs, a possible extension to the pure vertex shader displacement mapping method would be to use the border blend parameter _b to identify in which areas that this classification needs to be performed, i.e. only in the actual border regions. This would hopefully reduce the running time of this vertex shader, which can become extensive for large terrain geometries.

It would also be interesting to add contribution of local geometry to minimize screen-space geometric error. A possible solution may be to use a sparse irregular grid instead of a regular to avoid using a lot of geometry in low frequency parts of the height map and to create more geometry in high frequency parts of the heightmap. This may be a challenging problem, since we need to maintain visual continuity at the borders between detail levels. This becomes harder when using an irregular grid since the grid spacing is arbitrary instead of fixed as in the regular grid.

6.3 Summary

One advantage with using the Geometry Clipmap method over previous terrain rendering methods is that with the Geometry Clipmap method we do not need to make any preprocessing of the data. If an efficient method for streaming and unpacking compressed elevation and texture map data from disk is used in conjunction with this method, we can use this method to render huge outdoor environments without time and storage space consuming preprocessing steps. Another advantage is that the method is well adapted for the architecture of the current generation of graphics cards. The design of the algorithm allows for incremental updates of geometry and textures which balances the data transfers between the graphics processor and the main system memory so that unwanted lags can be avoided. This is very important for real-time simulators.

A disadvantage of using this method instead of other terrain rendering methods is that it currently does not consider local geometry to calculate screen-space geometric error. This fact requires more geometry to be rendered to achieve the same visual quality as methods which uses local geometry to calculate screen-space geometric error. This was a known limitation when we decided to use this method, but since the geometry pipeline of modern graphics cards seldom is the bottleneck in these kinds of applications, we decided to use this method anyway.

7 Acknowledgements

I would like to thank the following: Johan Hedström and Björn Lindahl for helping me with the simulator framework and the sometimes staggering amount of code base at FOI. Jonathan Borgvall for advising in more user-oriented issues when building the flight simulator prototype. My academic advisor Ken Museth. My friends and family. All the talented and dedicated people at the department for Man-System Interaction at FOI.

References

[Losasso, F., Hoppe, H. 2004.] Losasso, F., Hoppe, H. 2004. Geometry

Clipmaps: Terrain Rendering Using Regular Nested Grids. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004)

[Asirvathan, A., Hoppe, H. 2005.] Asirvathan, A., Hoppe, H. 2005. Terrain Rendering Using GPU-Based Geometry Clipmaps. GPU Gems 2

[Cohen-Or, D., Levanoni, Y. 1996.] Cohen-Or, D., Levanoni, Y. 1996. Temporal of levels of detail in Delauney triangulated terrain. IEEE Visualization. 37-42.

[Hoppe, H. 1998.] Hoppe, H. 1998. Smooth view dependent level-of-detail control and its application to terrain rendering. IEEE Visualization 1998, 35-42.

[El-Sana, J., Varshney, A. 1999.] El-Sana, J., Varshney, A. 1999. Generalized view-dependent simplification. Proceedings of Eurographics 1999, 83-94.

[Duchaineau, M et. al. 1997.] Duchaineau, M., Wolinsky, M., Sigety, D., Miller, M., Aldrich, C., Mineev-Weinstein, M. 1997. ROAMing terrain: Real-time optimally adapting meshes. IEEE Visualization 1997, 81-88.

[Röttger, S. et.al. 1998.] Röttger, S., Heidrich, W., Slusallek, P., Seidel, H.-P. 1998. Real-time generation of continuous levels of detail for height fields. Central Europe Conf. on Computer Graphics and Vis, 315-322.

[Lindstrom, P., Pacucci, V. 2002.] Lindstrom, P., Pacucci, V. 2002. Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. IEEE TVGC 8(3), 239-254.

[Bishop, L. et. al. 1998.] Bishop, L., Eberly, D., Whitted, T., Finch, M., Shantz, M. 1998. Designing a PC game engine. IEEE CG&A 18(1), 46-53. [Wagner, D. 2004.] Wagner, D. 2004. Terrain geomorphing in the vertex shader. In ShaderX2: Shader Programming Tips & Tricks with DirectX 9. Worldwide Publishing.

[de Boer, W., 2000.] de Boer, W., 2000. Fast Terrain Rendering Using Geometrical MipMapping. http://www.whdeboer.com/writings.html

[Gerasimov, P. et. al. 2004.] Gerasimov, P., Fernando, R., Green, S. 2004. Shader Model 3.0: Using Vertex Textures - NVIDIA white paper DA-01373-

001_v00

[NVIDIA. 2003.] NVIDIA. 2003. Using Vertex Buffer Objects - NVIDIA white paper WP-010115- 001_v01

[NVIDIA. 2005.] NVIDIA. 2005. NVIDIA GPU Programming Guide - version 2.4.0

[Tanner, C. et. al. 1998.] Tanner, C., Migdal, C., and Jones, M. 1998. The clipmap: A virtual mipmap. ACM SIGGRAPH 1998, 151-158.

[Williams, L. 1983.] Williams, L. 1983. Pyramidal parametrics. ACM SIGGRAPH 1983, 1-11

[Berndt, J. 2004.] Berndt, J. 2004. JSBSim: An Open Source Flight Dynamics Model in C++. AIAA 2004-4923.

[Malvar, H. 2000.] Malvar, H. 2000. Fast progressive Image Coding without Wavelets. Data Compression Conference (DCC ’00), 243-252

[Borgvall, J., Hedström, J. 2004] Borgvall, J., Hedström, J. 2004 Forward Air Controlling Memo. FOI Internal Research Reports