TERRAIN IMPOSTORS

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

William Hess

December 2010

COMMITTEE MEMBERSHIP

TITLE:	Terrain Impostors
AUTHOR:	William Hess
DATE SUBMITTED:	December 2010
COMMITTEE CHAIR:	Zoë Wood, Ph.D.
COMMITTEE MEMBER:	Chris Clark, Ph.D.
COMMITTEE MEMBER:	John Clements, Ph.D.

Abstract

Terrain Impostors

William Hess

Interactive software applications which need to render large terrain meshes can suffer from slow frame rates if the geometry of the terrain is sufficiently dense. However, the viewing angle to many distant features of the terrain does not change rapidly with respect to time. If the movement of the viewing position is limited to continuous motion and restrained to a known speed, many terrain features may be rendered once in high detail and reused for several frames.

This thesis proposes a method to increase the rendering speed of large complex terrains by splitting the terrain into contiguous chunks. If a given chunk is far enough away from the camera and its viewing angle will not change quickly, it is rendered into an image buffer. This buffer is then used to texture map a simplified version of the terrain mesh. The simplified and textured mesh is rendered in place of the original chunk of geometrically complex terrain. The simplified mesh is used to approximate parallax effects as the viewing angle changes in small increments. This technique is shown to as much as double the rendering speed of large terrain meshes without reducing the quality of the final image.

Contents

List of Figures

Chapter 1

Introduction

Often in nature we can view vast landscapes where terrain in the distance is many miles away. It is very desirable to be able to replicate these scenes in computer graphics. In a scientific application it would be valuable to view a natural landscape from a vantage point that would be impossible or impractical for a human to occupy. In an entertainment application users may want to view vast landscapes like those in nature, but created by artists. In both these applications users would like to explore the terrain in a realistic, interactive way. In both scientific and entertainment applications, the most common way of representing these terrains is using heightfields: 2D arrays of elevation points.

While heightfields are a convenient way of representing a terrain, they have severe scalability problems when converted into a triangle mesh. Parts of the terrain that are near the viewpoint should be dense with information. Each triangle of a terrain mesh close to the viewpoint corresponds to a significant number of pixels on the screen. However, as the mesh becomes distant from the viewpoint, the number of pixels that a triangle of the mesh represents is either one or it does not affect the final image at all. Drawing many triangles which do

Figure 1.1: A large terrain in natrure.

not affect the final image severely hurts performance. In order to render an n by m terrain mesh, $2 \times (n-1) \times (m-1)$ triangles must be drawn to the screen. Terrain meshes have $n \times m$ space and time complexity requirements. This makes large terrains very difficult to render interactively.

To be interactive, these applications need to be able to render terrain meshes faster than 30 frames per second. When rendering rates become slower than 30fps it is difficult to navigate and view the terrain mesh. The graphics hardware used in chapter 5.3 was only able to render a 4079 by 4079 wide mesh consisting of 33,554,432 triangles at a rate of 12 frames per second. If a heightfield consists of elevation points spaced one foot apart, a 4097 wide mesh represents an area of just 0.6 square miles. In order to make viewing this size of terrain interactive, we need to be able to maintain higher than 30 frames per second when rendering.

1.1 Terminology

There are two graphics techniques that are essential to Terrain Impostors. The following sections explain these techniques and how they are used.

1.1.1 Level of Detail

When objects appear far away from the viewpoint in a scene, they occupy very few pixels of the screen compared to when they are close. When this happens, many triangles in a complex mesh will not affect any pixels on the screen. Rendering these triangles hurts performance without adding to the final image. To address this, several versions of an object are created that consist of a different number of triangles. The versions of the object with fewer triangles will not appear as detailed when viewed up close, but are a suitable replacement for the original object when viewed at a distance. This is called Level of Detail.

Level of Detail is a very popular technique for improving rendering speed. In entertainment applications artists often manually create each of these Level of Detail models. An alternative way of generating Level of Detail models is to use a mesh simplification algorithm. Mesh simplification works by collapsing edges of the original mesh and placing the resulting vertex in a position that reduces some error metric. This error metric represents the disparity between the original, high triangle count mesh and the simplified mesh. The resulting simplified mesh omits some details of the original mesh but maintains the overall shape of the object.

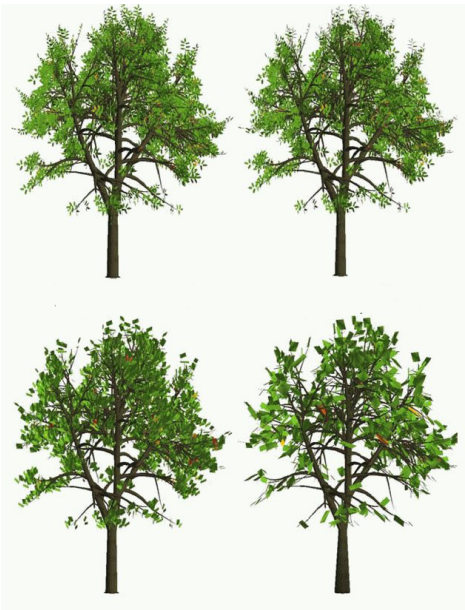

Figure 1.2: This is a tree model at four different Levels of Detail. The top left tree is the original mesh with a high triangle count. The bottom right is the least detailed and only suitable for distant views.

1.1.2 Impostors

An alternative way to simplify distant objects is to use Impostors. If how an object is viewed on the screen is not likely to change over time, it would be advantageous to cache an image of the object and draw the image every frame in place of the object. This is how Impostors work. A complex object is rendered once into a texture. This texture is then applied to a billboard, a rectangle in space which always faces the viewpoint. This reduces an object with any number of triangles to an object with only two triangles. As the viewing angle to the Impostor changes, the Impostor texture needs to be refreshed. If the Impostor does not get refreshed the object looks distorted.

Impostors have become a less popular way of rendering complex objects at a distance compared to Level of Detail models. One reason is that a separate

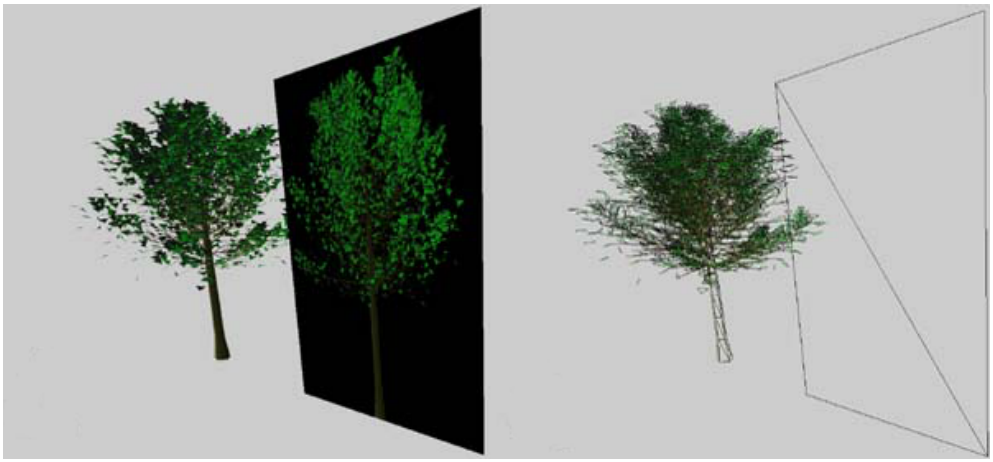

Figure 1.3: This is a tree model at four different Levels of Detail. The top left tree is the original mesh with a high triangle count. The bottom left is the least detailed and only suitable for distant views.

Impostor must be generated for every instance of the object visible in the scene. A scene with 300 trees would require at worst 300 Impostor textures, since every tree model might be viewed from a different distance, angle or have different lighting. By comparison, Level of Detail requires only 4 meshes no matter how many trees appear in the scene.

1.2 Current Techniques

Several techniques exist to make rendering terrain meshes more scalable, but nearly all of them reduce the rendering quality of the original mesh in some way in order to achieve high framerates. The most simple of these is to limit the distance at which terrain can be viewed. Terrain features farther than a defined radius from the viewpoint are not drawn to the screen. In order to view these terrain features the viewpoint must be moved so that they fall within the viewing radius. This is trivial to implement and limits the amount of geometry to be rendered

to a known maximum, guaranteeing constant frame rates. This technique is an easy way to visualize heightfields for scientific data where the dataset is too large to be drawn in its entirety with an acceptable frame rate.

A more complex approach is to use mesh simplification on the terrain mesh. It is a common technique in computer graphics to create Level of Detail meshes using mesh simplification. Level of Detail means that a coarse, simplified mesh is used in place of the original, high-resolution mesh when the object appears far away from the viewpoint. The justification for Level of Detail is that a coarse mesh is not noticeably different from the original mesh when it is viewed at a large distance. This technique can be applied to terrain meshes, but has several unique problems.

For a model of a tree or a person, the simplified mesh can be easily used in place of the original mesh because its geometry is disconnected from the scene it is rendered in. All of the object's geometry falls within a small bounding volume. Therefore it is easy to replace a tree mesh completely when it becomes sufficiently far from the viewpoint. Terrain meshes by comparison do not fall into small bounding volumes and they are entirely connected. A terrain mesh often spans the entire length of the scene. One might be tempted to split the terrain into smaller chunks and simplify each of these individually, but this creates gaps and discontinuities between high and low detail meshes. These gaps can cause undesirable visual effects. Terrain mesh simplification therefore becomes a difficult problem.

The common way of simplifying a terrain mesh is by collapsing edges incrementally in real time as the viewpoint changes. Unfortunately if not done with great care these edge collapses can be observed by the user. Peaks and valleys in the terrain are noticeably smoothed out and the resulting coarse mesh often

lacks a significant amount of detail.

Another common technique for visualizing terrain meshes is to limit the movement of the viewing position to within a known region. This allows application designers to precompute backdrops that appear to have high detail, but cannot be inspected closely because they fall outside of the region that constrains the viewing position. Very large amounts of geometry can be represented with rasterized images or hand-crafted Level of Detail models. By constraining the viewing position these models or images only need to be visually plausible for a small range of viewing angles. The constrained region also places seams between high and low detail meshes at fixed locations so that any gaps can be mended ahead of time.

This technique is particularly useful and popular in entertainment applications. By defining which parts of the scene are always high detail and which are always low detail, the entire scene can be loaded onto the graphics hardware and need not be modified for the rest of the application. This has excellent performance characteristics as well as provides high visual quality.

This technique, while very useful in certain applications, is not suitable for scientific visualization and an increasing number of entertainment applications. It is not practical to limit the viewing position to a small area in a scientific visualization application and the data in these applications is gathered rather than created by artists. In entertainment applications there is an increasing demand for large terrains that are fully explorable. In these applications the viewing position can no longer be constrained to a small region. The viewpoint needs to be able to move anywhere within the terrain mesh.

1.3 Proposed Technique

This paper proposes a new technique that combines the high visual quality of static backdrops with the full range of movement allowed by Level of Detail techniques.

Similar to Level of Detail, this technique creates simplified versions of the terrain mesh. Unlike the Level of Detail techniques however, this technique is not concerned with maintaining consistent borders between chunks of simplified terrain. Chunks of terrain can be simplified independently of each other and need not be modified during runtime. This technique also uses very coarse meshes. These meshes would be unacceptably coarse if used directly to represent the terrain.

This technique also makes use of offscreen render buffers. In OpenGL these buffers are called Frame Buffer Objects. The Frame Buffer Object and its DirectX equivalent are supported on nearly all commodity graphics hardware sold at the time of this writing. These offscreen buffers are used to render Impostors of distant chunks of terrain in full detail. The renderings are saved for later use as textures in multiple frames.

These Impostors are applied as textures to the coarse chunks of terrain. When rendered from the original viewing position, the terrain appears almost no different from the full resolution mesh (see figure 5.4 for an example). This would also be the case if the offscreen buffer were applied to a flat piece of geometry called a billboard. For a small viewing angle change the flat image is an acceptable visual approximation of the high resolution mesh. If the viewing angle becomes too great then the "flatness" of the image becomes apparent.

By applying the offscreen buffer to the coarse mesh, we achieve a much better

visual approximation of the terrain as the viewing angle changes. The coarse mesh provides the parallax effect needed for larger viewing angle changes while the offscreen buffer provides the rich visual detail of the full resolution mesh.

Parallax effect is the observation that parts of an object that are close to the viewpoint move quickly relative to the camera when the viewpoint changes. For example, when driving a car, the fence along the road seems to pass by very quickly, houses behind the fence move by at a reasonable speed and hills in the distance hardly seem to move at all. An Impostor that is projected onto a flat surface has no parallax effect.

1.4 Justifications

There are several reasons for why this is an appropriate technique for both scientific visualization as well as entertainment applications.

1.4.1 Static Content

Heightfield data in these applications is almost always static. In scientific applications, data is usually gathered first and then visualized using the application. In entertainment applications the heightfield represents a static terrain which cannot be modified at runtime. Given these assumptions it should be appropriate to render the meshes into an offscreen buffer and reuse the buffer as an approximation over time.

1.4.2 Camera Movement

In both scientific and entertainment applications, the viewing position usually moves in a continuous path or in small, well-defined increments. In addition, the movement of the camera is usually limited to a maximum speed. For these applications we can define an upper limit on the movement speed of the viewing position. By limiting the movement speed, we can calculate a lower bound for the time an Impostor can be reused. This lower bound allows us to guarantee that we won't need to refresh any given Impostor for at least that amount of time.

1.4.3 Video Memory

In 1999 commodity graphics hardware had at most 32 megabytes of video memory [1]. Also at the time 800 pixels by 600 pixels was the most common resolution used on personal computers [14]. Since then video memory available on commodity graphics hardware has increased to 2 gigabytes: a 64x increase in just over 10 years. Over the same time the most common screen resolution has increased to 1280x1024. This is only a 2.7x increase in the number of pixels displayed.

In 1999 the ratio of screen pixels to video memory was roughly 70 bytes per pixel. In 2010 that ratio is now as much as 1.6 megabytes of video memory available per pixel. This trend in memory size relative to screen resolution has accommodated the high storage requirements of impostors. As video memory per screen pixel increases, impostors should become more effective for real-time rendering.

Chapter 2

Related Work

2.1 Mesh Simplification

Since mesh simplification is used for Level of Detail terrain meshes and is also used in this paper, it is worth mentioning the common methods. Hoppe demonstrates in [8] that an appropriate simplified mesh can be attained by only collapsing edges of the original mesh. This is important for incremental Level of Detail meshes so that a transition from a high resolution mesh to a low resolution mesh can be described as a series of sequential edge collapses.

In ROAM [5] an incremental mesh simplification technique specifically for terrain is introduced. This technique works by representing the terrain mesh as a binary triangle tree. Any triangle within this data structure can be split or merged to adjust the complexity of the geometry. A drawback to this technique is that any triangle within the mesh cannot differ from its neighbors by more than one level of triangulation. The splitting of one triangle to increase detail may cause a chain reaction of splits. This prevents high detail triangulation from

being in close proximity to low detail parts of the mesh. Another drawback is that all edges must fall on 45 degree increments with respect to the xz-plane. This makes it very difficult to represent a terrain such as a steep cliff that falls at a 30 degree angle on the xz-plane. A general technique such as [8] would be able to represent such a mesh with only a few triangles while ROAM will require high triangulation along the cliff's edge.

2.2 Limited Movement

Several popular game engines make use of the technique of limited viewpoint movement. A Skybox consists of a cube that surrounds the scene and does not move relative to the camera position. This has the effect of appearing infinitely far away from the observer. The cube normally appears as a hand-drawn sky texture. This technique is still used to draw sky, which is appropriately approximated as being infinitely far away. In older games the skybox was also used to draw far away scenery, but because this scenery did not have any parallax effect Skyboxes are now rarely used for anything but rendering sky.

Valve's Source engine [16] makes use of a 3D Skybox. Level designers reserve an unreachable part of the scene for the 3D Skybox. Within this unreachable area the level designers create all of the geometry that falls outside the area the player may move within. This geometry is created at a 1:16th scale so that very large scenes can be created without needing to increase the capabilities of the game engine. The effect is that the player of the game appears inside of an extremely large and detailed scene. In the Frostbite Engine [2] a similar technique is used for their very large terrains. The Frostbite engine claims a 32x32 kilometer viewable area while only 2x2 kilometers are explorable.

2.3 Image Based

Impostors are a technique for replacing an object in a scene with a cached rendering of that object. Impostors are usually applied to a flat surface which always faces the viewpoint called a billboard. Impostors have the benefit of being able to simplify an arbitrary amount of geometry and detail into a flat image which can be reused for many frames. Impostors have a drawback of requiring the rendered image to be stored as a texture. While impostors are easier for graphics hardware to render, they require a significant amount of video memory. Level of Detail meshes have generally been preferred to Impostors because only one mesh needs to be saved in memory per type of object in the scene, while Impostors require a separate texture for every instance of an object in the scene.

In [15] Impostors are used to represent buildings in an urban environment. Distant structures in their scene are rendered into textures and the depth map of the texture is analyzed to generate an appropriate low resolution mesh on which to project the texture map. This mesh places appropriate vertices at places where the depth map has large disparities in order to give a good parallax effect.

Layered Depth Images [7] also transforms a rendering of a scene in order to use it for multiple different viewpoints. Two things make LDI different. First, the scene is not simply rendered as a single 2D image. Instead of discarding occluded pixels when a scene is rendered, pixels at multiple depths are saved. This allows parts of the scene which would have been occluded from the original viewpoint to be visible when the viewing angle is adjusted. The second difference is that LDI uses a technique called Image Splatting to display the rendered image.

Image Splatting renders an image by treating every pixel of the image as a separate piece of geometry. The depth of the pixel when it was drawn is used to

place the pixel in space. If the pixel appears closer to the viewpoint than it did in the original rendering then its size is increased to occupy a larger part of the screen. The result is a good visual approximation of a scene from small changes in the viewpoint. The large increase in geometry used in Image Splatting makes it unsuitable for real-time rendering.

2.4 Hybrid Techniques

Debevec's technique for rendering architecture [4] uses a very similar technique for representing complex objects. In Debevec's work, the objects are not high resolution meshes but actual physical landmarks. Sparse photographs are taken of these landmarks and projected onto coarse meshes that approximate the landmark. At first a linear combination of the photographs based on the viewing angle is used to texture the coarse mesh, but it is found to be an unsuitable approximation of the actual landmark because the photographs used are extremely sparse. Stereo Correspondence is used to calculate a depth value for pixels in the photographs and combined with Image Splatting recreate the high detail surfaces of the landmarks.

In [3] a very similar technique to the one proposed is used. LOD-Sprite rendering renders a flat image for the entire high-resolution terrain mesh from the current viewpoint. This image is used to texture map a low resolution version of the mesh for several frames until an error metric determines that the scene needs to be redrawn. Polygons which were not fully visible when the high resolution image was rendered are instead mapped with the original texture used in the scene. While LOD-Sprite addresses the same basic idea of terrain Impostors, it does not address many of the scalability concerns that are discussed in this

paper such as incremental updating using many smaller Impostors and blending between foreground and background.

Chapter 3

Algorithm

The algorithm is composed of several parts. First, the desired terrain mesh must be simplified in chunks and these chunks must meet several requirements in order to be suitable for this technique. These requirements and the simplification method used are explained in section 3.1. Next, a view frustum must be created in order to render the chunk of terrain into an offscreen buffer. Section 3.2 explains how that frustum is derived. Once we have a view frustum that encapsulates the chunk we want to draw, we need to know how many pixels the Impostor requires to accurately represent the terrain. Section 3.3 describes how to calculate the Impostor dimensions. After the Impostor has been rendered into a texture, we need to apply it onto the simplified mesh we generated earlier. Section 3.4 explains how the texture coordinates for the simplified mesh are calculated. Once the program is running, Impostors will need to be redrawn. Section 3.5 explains how to calculate an update criteria for each Impostor and how to define a lower limit for the time an Impostor will be valid.

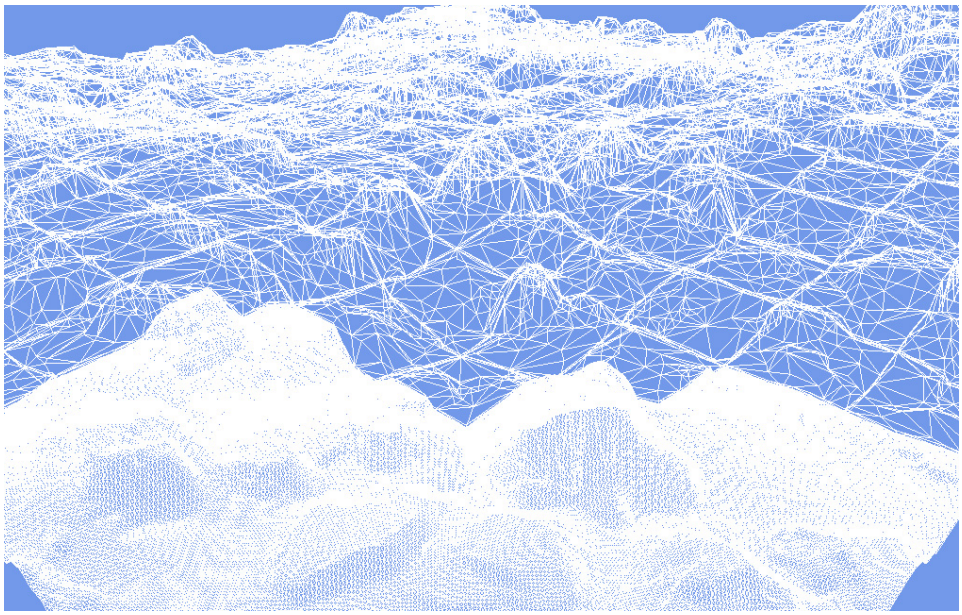

Figure 3.1: Wireframe images of both the simplified and high resolution chunk meshes are shown together.

3.1 Terrain Simplification

The mesh simplification algorithm used is very similar to that used in Progressive Meshes. Meshes are simplified by performing a series of edge collapses until some criteria has been met. In Progressive Meshes an error metric is used to determine when the mesh has been simplified enough. Instead, our chunk meshes are simplified until they are composed of a fixed number of triangles. While this may be slightly inefficient by assigning more triangles to chunks which may not need them for a good approximation, every chunk is reduced to less than 1% of its original triangle count. Forcing all chunks to be the same size makes framerates more predictable as well. Future work may make better use of incorporating the error metric to decide when to stop simplifying.

Two additional criteria are imposed on the mesh simplification. First, the edges of the chunk must be maintained. This means that when viewed from

$xEdge \leftarrow v_2.x \bmod chunkWidth$

$zEdge \leftarrow v_2.z \bmod chunkWidth$

if $xEdge \neq 0$ **and** $zEdge \neq 0$ **then**

  return true

else if $xEdge = 0$ **and** $zEdge = 0$ **then**

  return false

else if $xEdge = 0$ **then**

  return $v_1.x = v_2.x$

else

  return $v_1.z = v_2.z$

end if

Figure 3.2: This is the routine to determine the validity of an edge collapse from v_2 onto v_1. This check maintains the square shape of simplified chunks of terrain.

above in an orthogonal projection, each chunk should maintain a square shape. This prevents gaps from forming between chunks in the xz-plane. There may still be discrepancies in the y-axis between chunks of terrain because each chunk is simplified independently. Maintaining the edges allows gaps between chunks to be more easily sealed using techniques mentioned in section 4.2.

To determine whether an edge collapse would disrupt the square shape of the mesh, a subroutine is called and given the x and z values of each vertex and returns a boolean value determining whether or not the edge collapse is valid. To simplify our algorithm, there are only two possible outcomes of an edge collapse. One vertex involved in the collapse must be collapsed onto the other. Given two vertices v_1 and v_2 who occupy the same edge, the routine in figure 3.2 will determine if collapsing v_2 onto v_1 is legal.

The routine is very simple. First we assign the value of $v_2.x$ and $v_2.z$ modulo the chunk width to $xEdge$ and $zEdge$ respectively. If either of these values is equal to 0 then v_2 lies on a chunk boundary in the x or z direction. If v_2 does not fall on a border between chunks in either axis then it is ok to collapse. This is the normal case for the algorithm. The second condition is true if v_2 is on a border in both directions, meaning it is one of the corners of the mesh. Collapsing a corner onto another vertex is never a valid edge collapse. In the third case v_2 falls on a boundary in the x direction, but is not a corner. If both v_1 and v_2 have the same x value then they must fall on the same boundary. Note that v_1 may either be an edge vertex like v_2 or it may be a corner. This allows collapses to take place along the boundaries of meshes without disrupting the edge. The last case is the only other possibility and performs a similar check to the third case for the y value..

Another criteria for collapsing is that no triangle in the simplified mesh should have a normal vector with a nonpositive y value. This implies that no two triangles in the simplified mesh may overlap in the xz-plane. Combined with the previous criteria we can say that any vertex from the original terrain mesh is either included in the simplified mesh, corresponds to only one triangle in the y-axis or falls on an edge between two triangles in the y-axis. Using this knowledge it is easy to determine which points of the original mesh correspond to any given triangle in the simplified mesh. This is useful for calculating the error of an edge collapse.

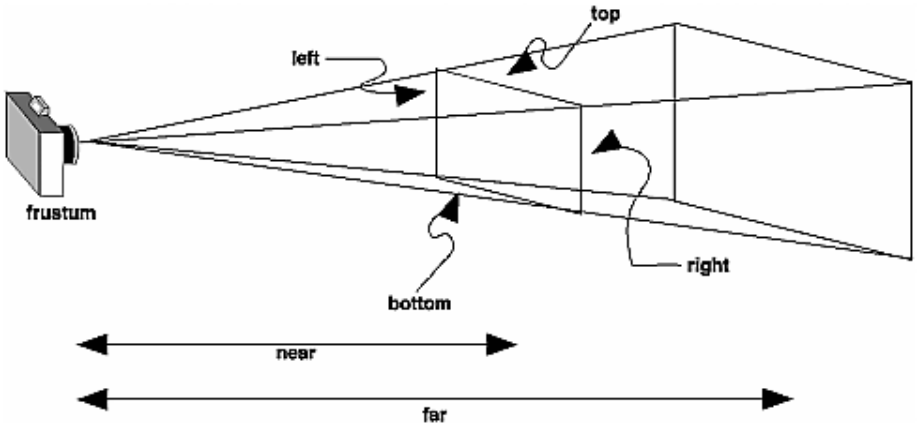

Figure 3.3: A view frustum with labeled dimensions.

3.2 View Frustum

Once the high resolution terrain and its simplified chunks have been acquired, Impostors can be generated. Rendering the chunk of terrain into an offscreen buffer requires an appropriate view frustum and dimensions for the offscreen buffer which correspond to the number of pixels the chunk occupies on the screen. These values need to be recalculated every time an Impostor is rendered from a different viewpoint.

A view frustum is a volume which defines where objects are drawn onto the screen. The view frustum is defined by 6 planes. The top, bottom, left and right planes define the edges of the screen or viewport. Each of these planes tapers outward to achieve a perspective transformation. The view frustum contains a larger part of the scene the farther it is from the viewpoint. The near and far planes prevent objects from being drawn too close or too far from the viewpoint.

Since this part of the algorithm must be run in real-time and in small enough increments that it can be calculated between frames, it is not practical to test

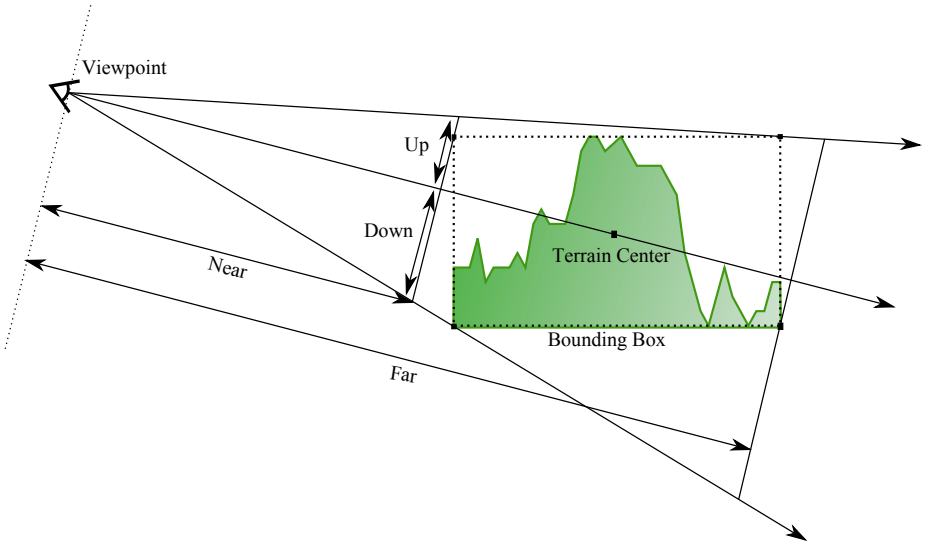

Figure 3.4: A diagram of how a view frustum is tightly fitted to a bounding box. The $near$, far, up and $down$ values are labeled.

each vertex of the terrain mesh in order to find an appropriate view frustum. For this reason an axis-aligned bounding cube is constructed with corners equal to the corners of the terrain mesh whose y values have been replaced with the minimum and maximum height values within the terrain. Each of these corners is used to create the view frustum. The center of this bounding cube is also calculated.

The view frustum can be constructed using 6 values. These values are $near$, far, $left$, $right$, up and $down$. $near$ and far define the distance from the viewpoint to the near and far clipping planes. These values are calculated first by constructing a plane which contains the viewpoint and has a normal vector parallel to the direction from the viewpoint to the center value calculated earlier. The distance between this plane and each of the 8 corners of the bounding volume are calculated and the minimum and maximum values are saved as $near$ and far

respectively.

In order to find left and right, each of the corners of the bounding box are projected onto a plane. This plane is constructed so that the viewpoint and the center of the bounding box lie within it, and is oriented so that it should bisect our view frustum horizontally. Each projected point and the center is translated so that the viewpoint is considered the origin. The cross product between the center point and each of the translated points is calculated. The points whose cross product is largest and smallest are saved as the leftmost and rightmost points respectively. A similar procedure is followed to find the upmost and rightmost points with the plane oriented so that it bisects the view frustum vertically.

The leftmost, rightmost, upmost, downmost and center points are then projected onto the near clipping plane. This is a plane with a normal parallel with the viewing direction and *near* distance away from the viewpoint. The distance between the projected center point and each of the other projected points yields the values for $left$, $right$, up and $down$. Using this data we can construct an appropriate view frustum and orient it using the viewing position, center position and up vector, which is simply a unit vector on the y-axis. The projection matrix constructed by this frustum and orientation is saved for use in section 3.4.

3.3 Impostor Allocation

In addition to the view frustum, we also need to know how many pixels should be allocated for each Impostor. To calculate the dimensions of the offscreen buffer two more pieces of information are needed. The dimensions of the main viewport for the application $width$ and $height$ and the field of view of the viewport fov_y

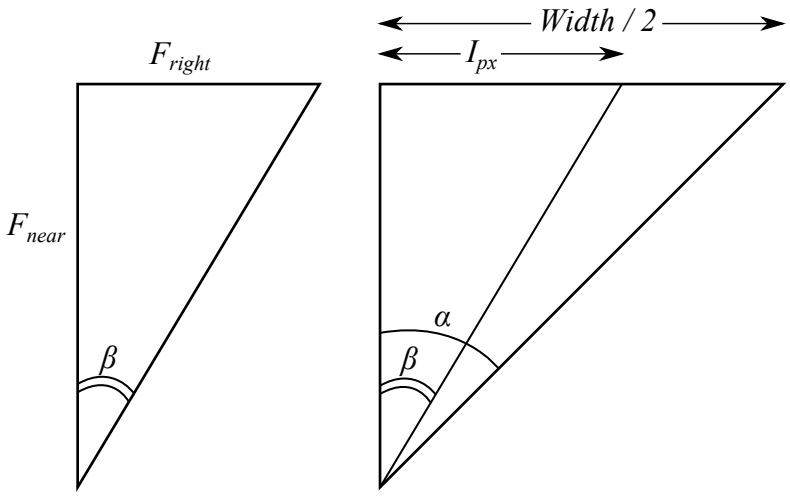

Figure 3.5: This diagram shows the relationship between the view frustum for the chunk of terrain and the view frustum for the main application. These frustums can be compared to calculate an appropriate number of pixels for the Impostor. I_{px} is the number of pixels required for the right section of the view frustum.

and fov_x.

In figure 3.5 α is $fov_x/2$ and the number of pixels associated with α is the width of the screen resolution divided by two. It is assumed that the view frustum for the application's viewport is symmetrical. I_{px} is the number of required pixels for the Impostor based on the right half of the view frustum. F_{right} and F_{near} are the values we calculated earlier to create the view frustum. Using the triangle on the right we can derive the relationship in equation 3.1:

$$I_{px} = \frac{Width \times \tan\beta}{2\tan\alpha} \tag{3.1}$$

Since α is a constant value, $\tan\alpha$ is easily calculated. We can calculate a value for $\tan\beta$ using the triangle on the left and the values from the view frustum.

Replacing these values we get equation 3.2:

$$I_{px} = \frac{Width \times F_{right}}{2F_{near} \times \tan\left(fov_x/2\right)} \tag{3.2}$$

If we perform this calculation for F_{right} and F_{left} and sum the number of pixels calculated for each side of the frustum, we will have the required pixel dimensions of our Impostor. Combining the equations for F_{right} and F_{left} and repeating the process for F_{up} and F_{down}, we get equations 3.3 and 3.4:

$$I_{width} = \frac{Width \times (F_{right} + F_{left})}{2F_{near} \times \tan\left(fov_x/2\right)} \tag{3.3}$$

$$I_{height} = \frac{Height \times (F_{up} + F_{down})}{2F_{near} \times \tan\left(fov_y/2\right)} \tag{3.4}$$

Where I_{width} and I_{height} are the dimensions of the Impostor.

3.4 Texture Mapping

By saving the projection matrix for later use, we can derive appropriate texture coordinates for each vertex of the simplified mesh. By applying the projection matrix to a vertex, we get a value in the range [-1, 1] for the x and y values. These values correspond to where the vertex would appear in screen space using the view frustum we derived earlier. If we translate these values to [0, 1], the x and y values become appropriate s and t values. s and t can be used as texture coordinates for each vertex of the mesh. If the simplified terrain mesh is viewed from angles near to the angle the Impostor was rendered from, it should look very similar to the original mesh.

3.5 Update Criteria

As the viewing position moves, it is important to redraw the offscreen buffers. Some of the Impostors may be more inaccurate than others and should be redrawn sooner. We can roughly quantify how inaccurate every terrain Impostor is by comparing the current viewing angle to the viewing angle when the Impostor was rendered. An easy way to calculate this value is to compute the dot product between two unit vectors. The first unit vector points from the center of the chunk of terrain to the viewpoint when the Impostor was generated, and the second unit vector points from the center of the chunk of terrain to the current viewpoint. In this situation -1 would be the most inaccurate while 1 would indicate it does not need to be upated at all.

For very large Impostors this may be a poor criteria. The viewpoint can move towards the center of a large Impostor until it is very close. The error would still be minimal even though the edges of the Impostor may be very distorted. For this reason it would be advantageous to do this calculation for each corner of a bounding box surrounding the chunk of terrain and use the maximum value. This would prevent such a situation since the viewpoint cannot move towards all corners of the box at once and each corner represents the maximum possible error of any section of the terrain represented by the Impostor.

3.5.1 Minimum Update Rate

It is also important to guarantee that we will not need to redraw Impostors at a rate higher than our minimum frame rate of 30 fps. If we need to update our Impostors at a rate greater than 30 frames per second then the technique will be counterproductive because of the overhead required to render to an offscreen

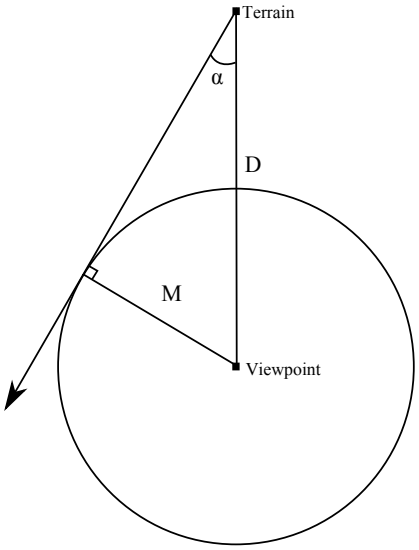

Figure 3.6: This diagram shows the relationship between the movement of the viewpoint and the viewing angle of an object viewed at distance. α is the largest appropriate viewing angle for the object. D is the distance from the viewpoint to the object. M is the shortest distance the viewpoint must travel in order to exceed a viewing angle of α.

buffer. If we define a maximum viewing angle α at which any piece of terrain Impostor is a good approximation of the original mesh, we can combine that with the speed of the viewing position S to obtain a rate at which the chunk of terrain must be redrawn that will always maintain a good visual approximation.

Let us define D to be the distance from the original viewing position to the chunk of terrain. If α is the largest appropriate viewing angle for the chunk of terrain then let M be the distance the viewpoint must travel to cause the viewing angle to become greater than α. This relationship is illustrated in figure 3.6. From figure 3.6 we can derive equation 3.5:

$$\sin \alpha = \frac{M}{D} \tag{3.5}$$

We can substitute M for $S \times t$ where S is the maximum speed of the viewpoint and t is the time required for the viewing angle to become greater than α. Substituting and rearranging equation 3.5 gives us equation 3.6:

$$t = \frac{D \sin \alpha}{S} \tag{3.6}$$

We can substitute D, α and S with values from our application to determine the smallest amount of time that a single rendering of a chunk of terrain can be reused for. As an example, if we set D to be 1000 units, S to be 50 units per second and α to be 5 degrees, then we obtain a value of 1.74 seconds for t.

If our application is normally rendered at 60 frames per second, 1.74 seconds becomes a minimum of 104 frames that the chunk can be reused for. By redrawing the chunk of terrain at least once every 104 frames we can guarantee that the terrain will always maintain a good approximation of the original high resolution mesh.

If we apply this equation to many chunks of terrain that are distributed around the viewpoint, we see that most chunks will not approach this minimum refresh rate. Since the terrain chunks are spread uniformly across the xz-plane the viewing angle will change at a much slower rate for some chunks (particularly those the viewing position is moving towards or away from). In addition, the viewing position may not always be moving at its maximum speed or in a uniform direction. All of these factors increase the time for which a chunk of terrain can maintain a good approximation without needing to be redrawn.

Chapter 4

Implementation

The implementation of this technique involves several practical and performance concerns. Section 4.1 explains how the terrain meshes which are used in this paper were generated. Section 4.2 explains how cracks between chunks were mended to appear as one contiguous terrain. Allocating Impostors individually was found to perform badly. Section 4.3 explains how Impostor textures are packed into one large texture atlas. Not only do chunks need to be mended together, but parts of the foreground which are drawn in full resolution need to be transitioned into Impostors. Section 4.4 explains how this transition is achieved.

4.1 Terrain Generation

First a suitable terrain mesh is needed for rendering. For this implementation, techniques from [13] were used to generate and erode large heightmaps. This terrain generation technique was chosen because it creates distinct terrain features that protrude from the mesh. These features are useful for demonstrating the characteristics of terrain Impostors.

4.2 Cracks

Since terrain meshes are large, contiguous meshes, one of the first concerns that needs to be addressed in this technique is the ability to conceal cracks between chunks. There are two forms of cracking that can appear. The first is a result of the mesh simplification algorithm and the second is from the pixelation of Impostors.

By making mesh simplification easy and simplifying each chunk of terrain independently, significant gaps can appear between neighboring geometry. To the user, these gaps would be very obvious and undesirable. To prevent this from happening, a "skirt" is added to each chunk.

A skirt is a set of polygons added to the edge of the chunk of terrain that extend its geometry downwards. This skirt fills any gaps between neighboring simplified meshes. The effect of the skirt is that in the gaps where no geometry was visible, there is now a face which can be mapped with the appropriate pixels from the farther away Impostor.

Sometimes the opposite effect happens, where the closer simplified mesh fails to cover all of the pixels in its Impostor. To prevent this from happening, a second skirt is added, but this time it is very small, points upwards and faces inwards towards the center of the chunk of terrain. This is useful for situations where the simplified mesh is too low around its edges, preventing pixels from the Impostor from being displayed on the screen. This skirt is small enough that it is does not cause a noticeable parallax effect on the terrain.

Another form of crack occurs due to the pixelation of Impostors. If an Impostor for a chunk of terrain includes only geometry from that chunk, it creates ragged, pixelated edges where the terrain suddenly ends. The ragged edges of

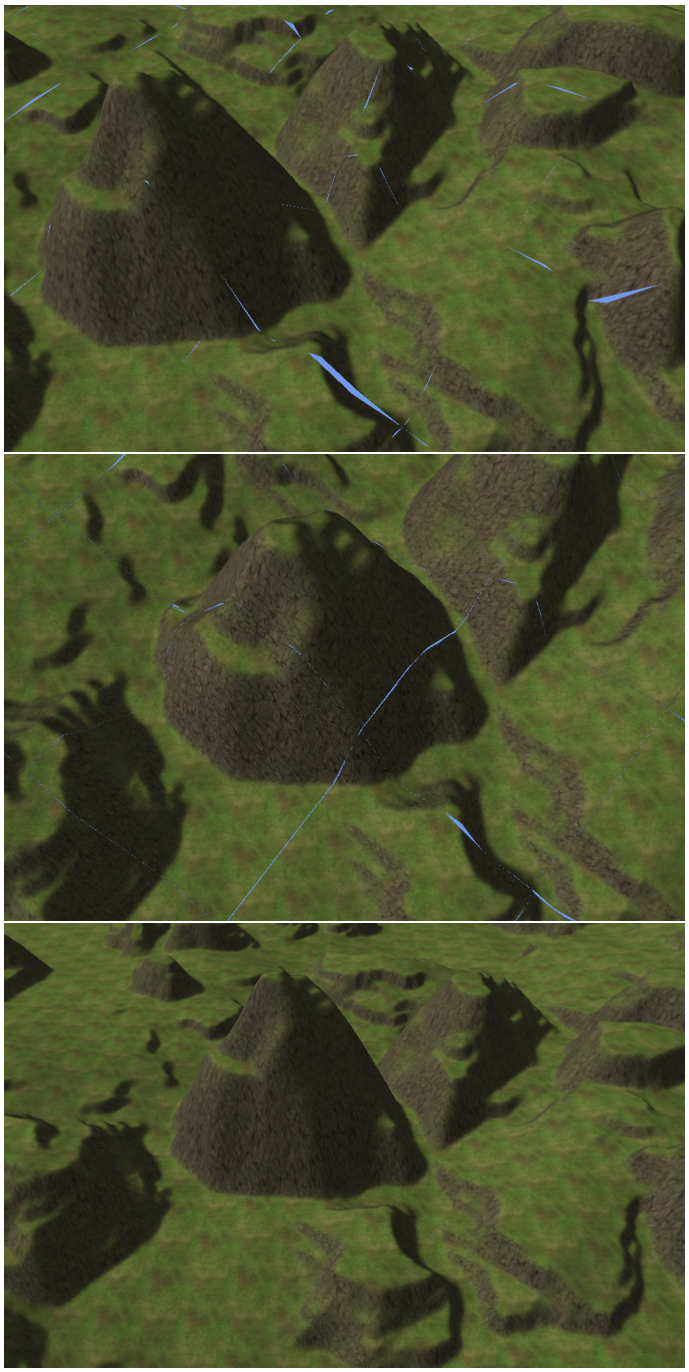

Figure 4.1: Here the same section of terrain is shown with different types of cracking. Top: the geometry is rendered without any skirt triangles and gaps are present between adjacent meshes. Middle: Pixelations are present in the edges of the Impostors due to only rendering the geometry associated with the individual chunk. Bottom: In this scene both seaming techniques are used and no cracks are visible.

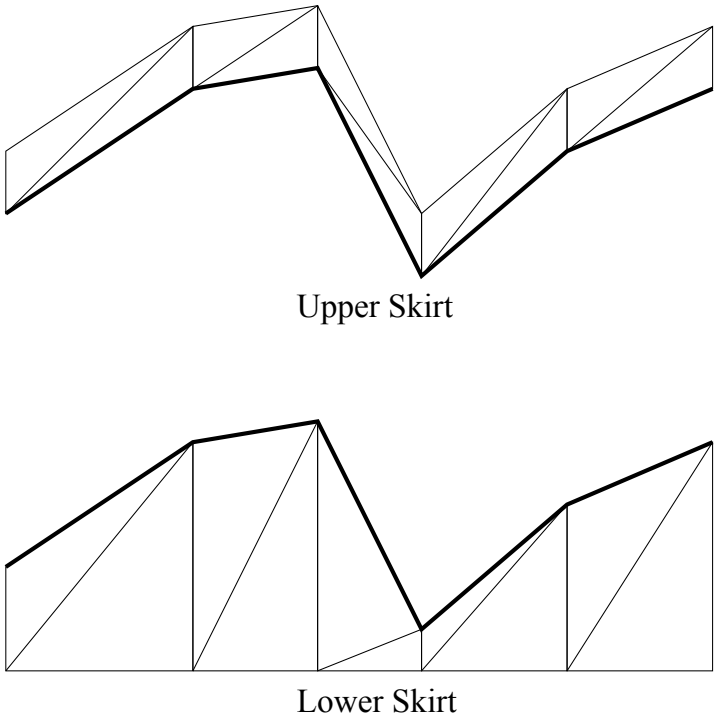

Figure 4.2: The bold line represents the edge of a simplified terrain mesh viewed from the side. The upper mesh skirt is created by adding triangles to each of these edges that rise vertically a small number of units from the original mesh. The lower skirt is created by dropping triangles down to a set minimum value for the entire chunk.

neighboring Impostors create seams that are visible by the user. To prevent this from happening not only is the individual chunk's geometry rendered to create the Impostor, but each of the 8 chunks surrounding it are rendered as well. Most of the geometry of these chunks is clipped out by preventing all but a small border of terrain around the chunk to be drawn in the Impostor. Enough neighboring geometry is kept near the edges so that the pixelation and ragged edges are no longer present.

4.3 Atlas

Unfortunately, allocating and deleting offscreen buffers every frame is extremely expensive on current hardware and causes very low framerates. To avoid this problem, one large offscreen buffer is allocated at the beginning of the application and used as a texture atlas [11][9]. By combining all of the offscreen buffers into one large texture atlas, the buffer size allocated for each chunk of terrain cannot be changed independently. This requires a new scheme for changing Impostor size for all chunks at once.

To re-allocate the number of pixels assigned to each Impostor, each chunk of terrain creates a view frustum and calculates the desired dimensions for its Impostor. The Impostor sizes are sorted by height and packed into the texture atlas using the First-Fit Decreasing Height algorithm [6]. Fortunately, Impostors are rectangular in shape and similarly sized so packing them somewhat efficiently is a quick process. This calculation is fast enough to be performed in the time between frames of the application and causes no stalls. If for larger scenes this allocation takes longer than one frame to calculate, the calculation can be performed asynchronously in a separate thread.

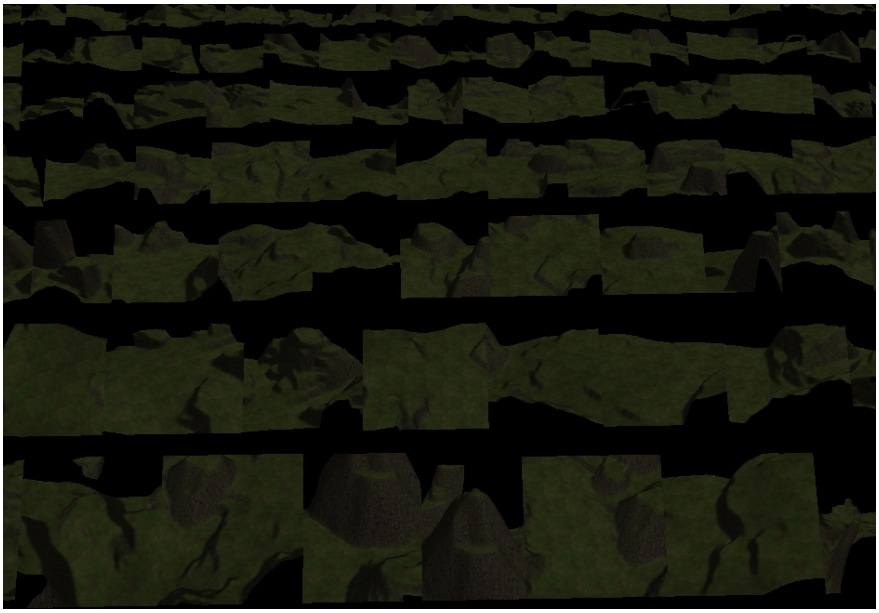

Figure 4.3: Impostor viewports are packed into one large texture which is allocated at the start of the application.

When Impostors are assigned new locations in the texture atlas, the pixels they are assigned will not contain appropriate renderings. Since not all Impostors can be regenerated in one frame, a second texture atlas is required. Each chunk of terrain is allocated a space for an Impostor in both texture atlases. One of the texture atlases will be more out of date than the other. When a new Impostor is rendered for a chunk of terrain it changes its texture mapping from the out of date atlas to the newer atlas.

Once all the Impostors in the newer atlas have been rendered the out of date atlas can be re-allocated. When new assignments have been made for the texture atlas Impostors can begin to be rendered in their new locations. The new atlas becomes out of date and the out of date atlas is cleared and becomes the new one.

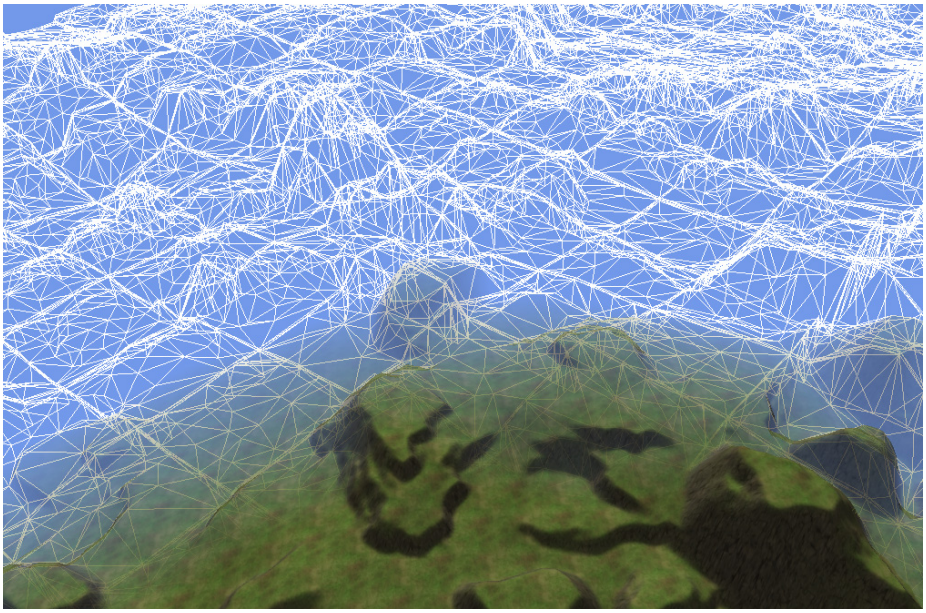

Figure 4.4: The textured foreground geometry is faded into the background Impostor-based geometry which has been rendered as a wireframe for this image.

4.4 Foreground Blending

Terrain in the foreground, especially chunks the viewpoint lies within, should not generate Impostors. Not only will they need to be redrawn more often, but they will also require much more texture space in our atlas. This means that some terrain is drawn as Impostors while other parts of terrain are drawn in the usual way at high resolution. This requires a seamless way of blending between the foreground (regular terrain meshes) and the background (Impostors).

Blending between the foreground and background is achieved with a simple linear alpha channel fade. At an inner radius around the viewpoint, $startFade$, the alpha channel of the foreground terrain begins to drop from 1 (opaque) towards 0. The foreground becomes completely transparent at the outer radius, $endFade$. Any chunks of terrain which lie outside this radius are drawn only as

Impostors. Each chunk of terrain is tested against the radius using its bounding box.

To render the scene, first all Impostors are rendered with a depth value that is offset to be deeper than normal, but accurate relative to all other Impostors. Impostors which fall completely within the $startFade$ radius are not drawn. Foreground chunks of terrain are then drawn over the Impostors. Since the Impostors have been drawn with a deeper than normal depth value, the foreground cannot be occluded by the Impostors, but the foreground will properly occlude itself. This was found to perform better than clearing the depth buffer after drawing the Impostors.

Since the Impostors so closely represent the high resolution chunks of terrain, the fade between foreground and background is not obvious, even when aware of it and attempting to notice the effect.

Chapter 5

Results

In order to judge the fitness of this technique it was necessary to compare rendering speeds using various terrain meshes and parameters.

5.1 Considerations

There are issues with the current implementation that impact the performance results. These issues are non-essential to the technique.

First, two entire texture atlases occupy video memory while only half of the allocated texture atlas space is ever in very active use. A better scheme for resizing Impostors over time may make better utilization of this memory and allow the technique to be used on machines with lower amounts of video memory.

The second performance concern involves the storage of the original mesh data. In the implementation the entire high-resolution terrain mesh is stored in video memory. Combined with the texture atlas, this further increases the need for large video memory. A scheme for keeping in memory only those parts of

the original mesh which are needed for the foreground or for updating Impostors would greatly reduce the required video memory needed to use this technique.

Given these two issues, texture atlas size had to be reduced to allow all required geometry data to be stored in video memory for the larger meshes. This reduces the visual quality of the Impostors during the tests.

5.2 Testing Environment

All performance tests were run on a machine with an Nvidia GeForce 260GTX video card with 768 megabytes of video memory. Four terrain meshes were generated for testing this technique. Each terrain mesh is square shaped and increasing in size. The smallest heightfield consists of 1025 squared height values. Its mesh contains 2,097,152 triangles. Each mesh increases in width by 1024 values. The fourth mesh is generated from a heightfield of 4097 squared values and has a mesh consisting of 33,554,432 triangles. Two simplified versions of each terrain were generated. One version used chunks that were 64 units wide while the other used 128 unit wide chunks. Chunks consist of 157 triangles on average including the skirts and 85 triangles without. The triangle count of the completely simplified version of the 4097 wide mesh with 64 wide chunks is 643,072.

To measure the performance of the technique, the number of milliseconds required to draw each frame is recorded for 10,000 consecutive frames. These frame times are later used to compute an average number of frames per second for each of the experiments.

Since Impostors need only be updated when the viewpoint moves a great enough distance, testing requires that the viewpoint be in motion for the tests.

While frame times were gathered, the viewpoint was set to move in a circular pattern around the terrain. This viewpoint movement covers nearly the entire width of the terrain so that terrain features must have Impostors generated for many possible viewing angles. The viewpoint also has a fixed movement speed of 80 units per second.

5.3 Performance

Figure 5.1 shows general performance results for each of the four terrain meshes used. The framerates for full represent rendering the full terrain mesh with no overhead for rendering Impostors. Each of the other plots show the framerate results of using this technique at varying foreground fading distances and chunk sizes. The first number is the foreground rendering boundary and the second number is the width of the simplified chunks used. By increasing the foreground distance there are less Impostors in the scene, but the full resolution mesh must be used in their place. The Impostors which are removed are also the ones that would need to be updated more often since they are closest to the viewpoint. As the terrain meshes become larger, pushing the foreground boundary back farther results in slightly higher framerates. Pushing the foreground distance back as far as possible increases visual quality by not only rendering more of the high resolution mesh, but by allocating more space in the texture atlas to each individual Impostor since the largest Impostors are no longer present.

The 128 wide chunk tests had excellent rendering speeds. The larger chunks were able to increase the rendering speed of the scene by as much as 7.6 times. However, chunks near the viewpoint had much more obvious rendering artifacts due to self-occlusion of Impostors. When the large chunks were viewed from a

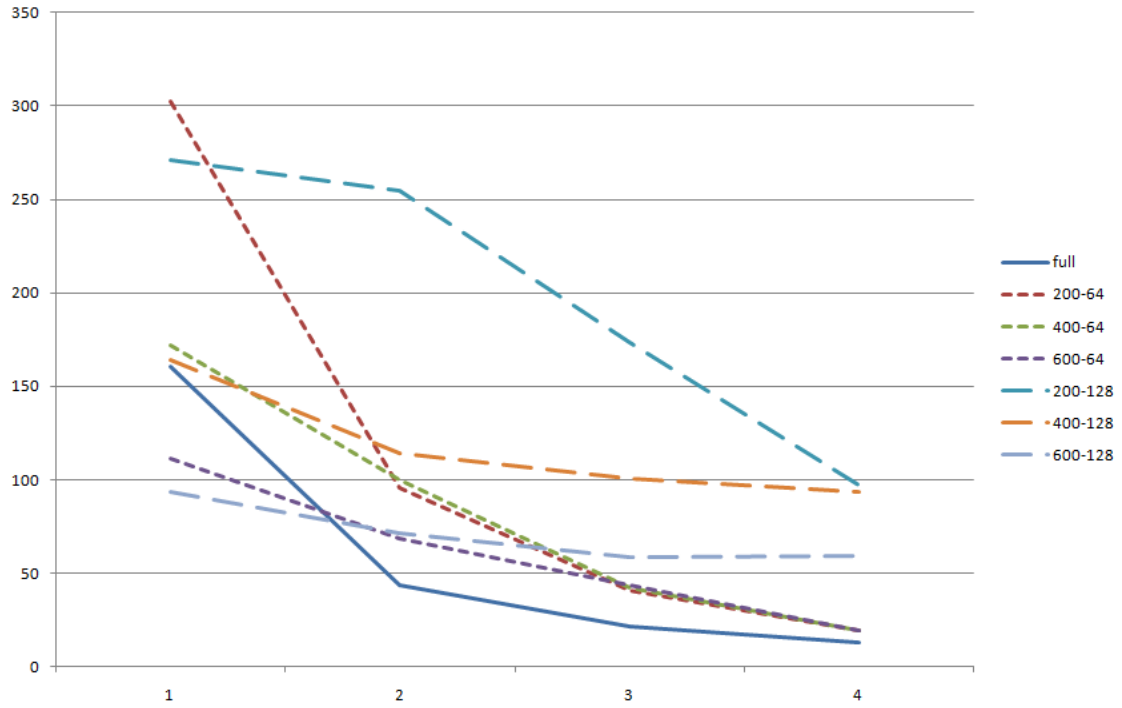

Figure 5.1: Adjusting the radius at which the foreground ends can affect time required to render the scene. For smaller meshes a close foreground boundary yields better results, but in larger meshes a larger foreground boundary is equal or slightly faster. Inf represents rendering the scene using no Impostors.

large distance, this self-occlusion was not as obvious. The 128 wide chunks are too coarse for close viewing but provide a very large performance boost for the larger terrain meshes where most chunks are far away from the viewpoint. Section 6.5 explains a possible extension to this implementation that would combine the good approximation of 64 wide chunks close to the viewpoint and the high performance increase of using larger chunks far away from the viewpoint.

Even in the 4097 wide mesh there is a limit at which extending the foreground boundary maintains a high framerate. Figure 5.6 shows that past 1000 units away from the viewpoint, extending the foreground boundary decreases the framerate when using 64 unit wide chunks. This shows that in any application using this

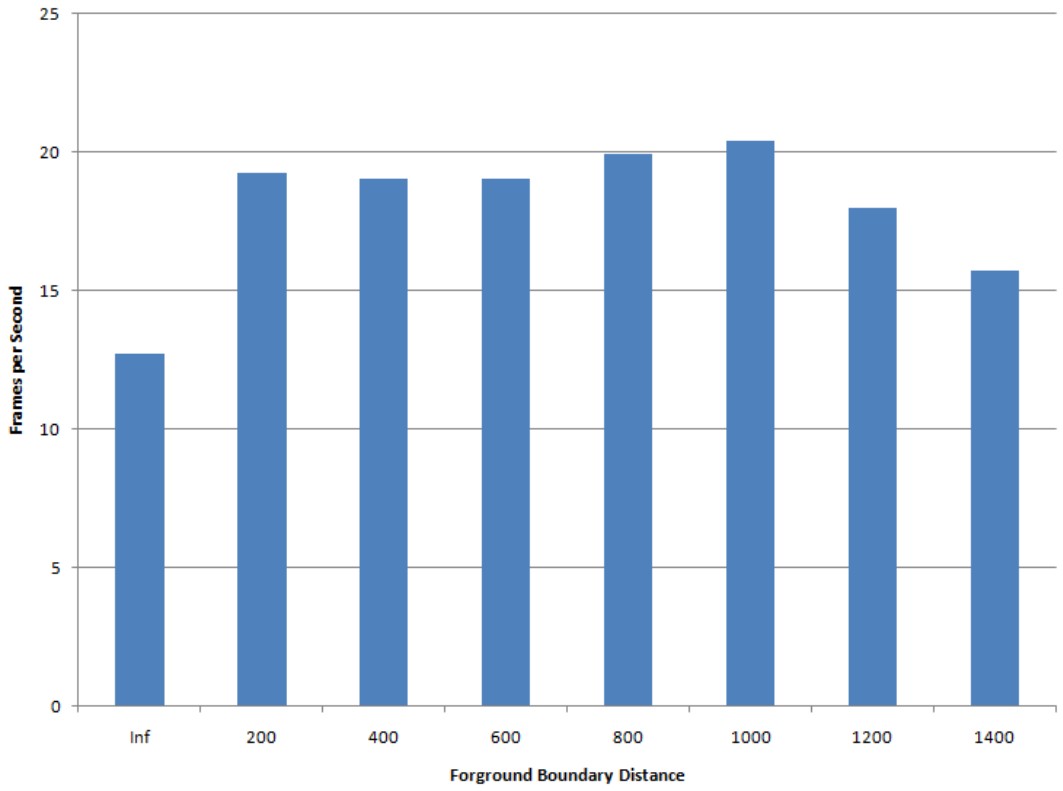

Figure 5.2: The behavior of framerates as the foreground boundary is extended. Approximately the same framerate is maintained until 1000 units away from the camera when the inefficiency of rendering the full mesh begins to lower the framerate.

technique, finding an appropriate value for the boundary is critical to balancing speed and rendering quality.

It is interesting to note that for the 1025 wide mesh, using a foreground boundary of 600 and chunk sizes of 64 the framerate actually decreased. At this distance, nearly all of the full resolution mesh must be drawn every frame in addition to the overhead of having to set up and render Impostors in the scene. This would indicate that being able to cut out large sections of the terrain's geometry is an important factor to being able to overcome the overhead of rendering Impostors.

Foreground Distance	Chunk Width	Average FPS	FPS at one Standard Deviation
Full	N/A	12.6922711	12.60956096
200	64	19.34585674	17.25031183
400	64	19.09715595	15.79657353
600	64	19.17327814	15.54628143
200	128	97.01818182	77.82946639
400	128	93.64908504	80.67043284
600	128	59.26606048	52.84394585

Figure 5.3: This table shows the average frames per second for each test as well as the drop in FPS at one standard deviation away from the average. These results are all from the test of the 4097 wide mesh.

These results show that there is clearly a significant performance decrease when many Impostors must be drawn every frame. With a fixed chunk size, the number of Impostors we must render in our implementation as the terrain size grows is still relative to the square of the size of the scene. Section 6.5 discusses a technique for reducing the number of Impostors and can possibly overcome this overhead and make the number of Impostors which must be rendered very scalable.

It is also important to consider not only the average framerate. If the application suddenly drops to a very low framerate intermittently, that can ruin the interactivity of the applicaiton even though average framerates are very high. To verify that this is not the case, the standard deviation as well as minimum framerates have been calculated for several of the performance tests on the 4097 wide terrain mesh and are shown in Figure 5.3.

Figure 5.3 shows that the standard deviation does increase when using Terrain Impostors, but the framerates at one standard deviation from the average are not so far from the averages that the drop in framerate should be very noticeable. This increase in standard deviation could be reduced even further by updating a

constant number impostors every frame rather than simply updating those that exceed our error threshold.

5.4 Conclusion

A few years ago, this technique would not have been feasible. Commodity graphics hardware did not have nearly the abundance of fast memory that it has now. In addition, OpenGL Frame Buffer Objects have made rendering into a texture effective for real-time applications. Combining these advances in hardware has allowed the implementation to successfully render large terrain meshes faster than a simple geometry-only implementation.

What makes this technique different from Level of Detail approaches is that the quality of the scene does not need to be sacrificed in order to obtain high framerates. As long as enough video memory is available, these large terrains can be rendered quickly with little or no loss in visual quality. With the improvements suggested in section 6.6, this technique may also even reduce the overall video memory requirements for rendering a scene by making the space requirements of elements in the scene proportional to their size in screen space rather than the complexity of their geometry.

It is also important to note that this technique is not mutually exclusive with Level of Detail. While current Level of Detail implementations often visibly reduce the quality of distant parts of the scene, Level of Detail can be used in such a way that there is very little loss in quality but significant improvements in speed. Any improvements to the efficiency of rendering the underlying high-quality terrain will only improve the performance of using Terrain Impostors.

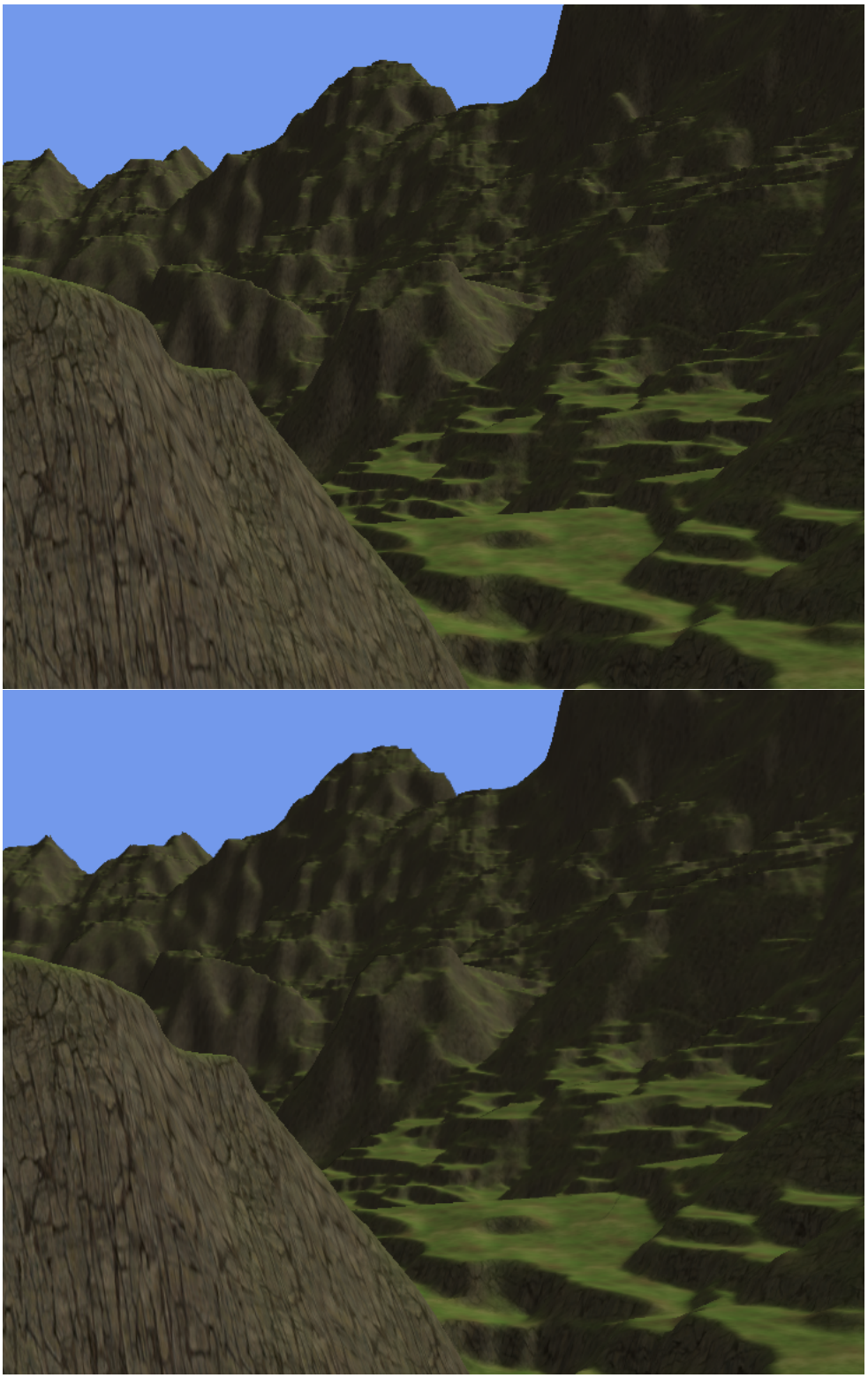

Figure 5.4: Top: The 2049 mesh rendered using the original terrain mesh. Bottom: The 2049 mesh rendered from the same position using Terrain Impostors.

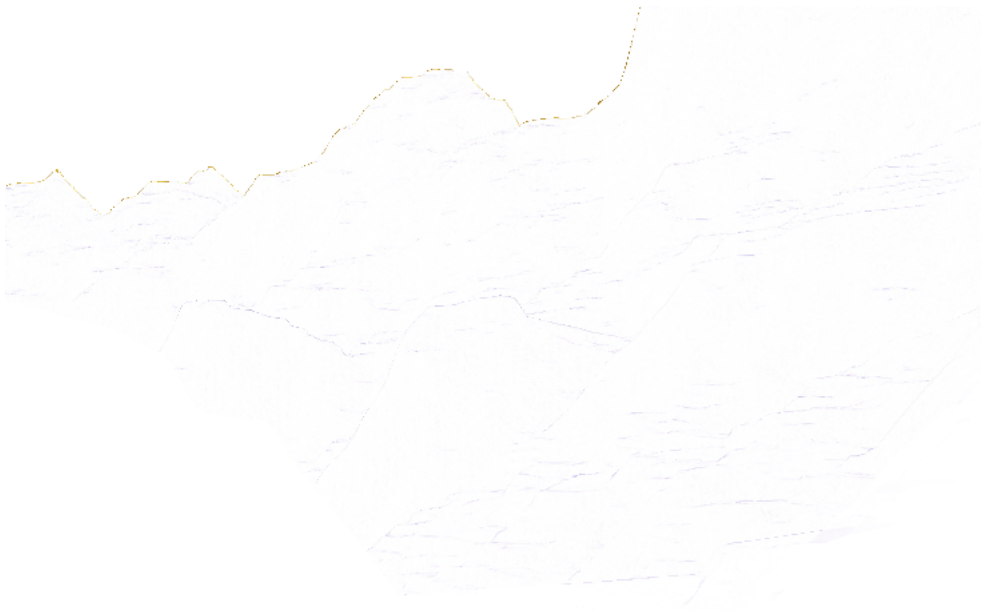

Figure 5.5: This is an inverted difference image showing the pixel differences between the original rendering of the scene in figure 5.4 and the same scene rendered using Terrain Impostors.

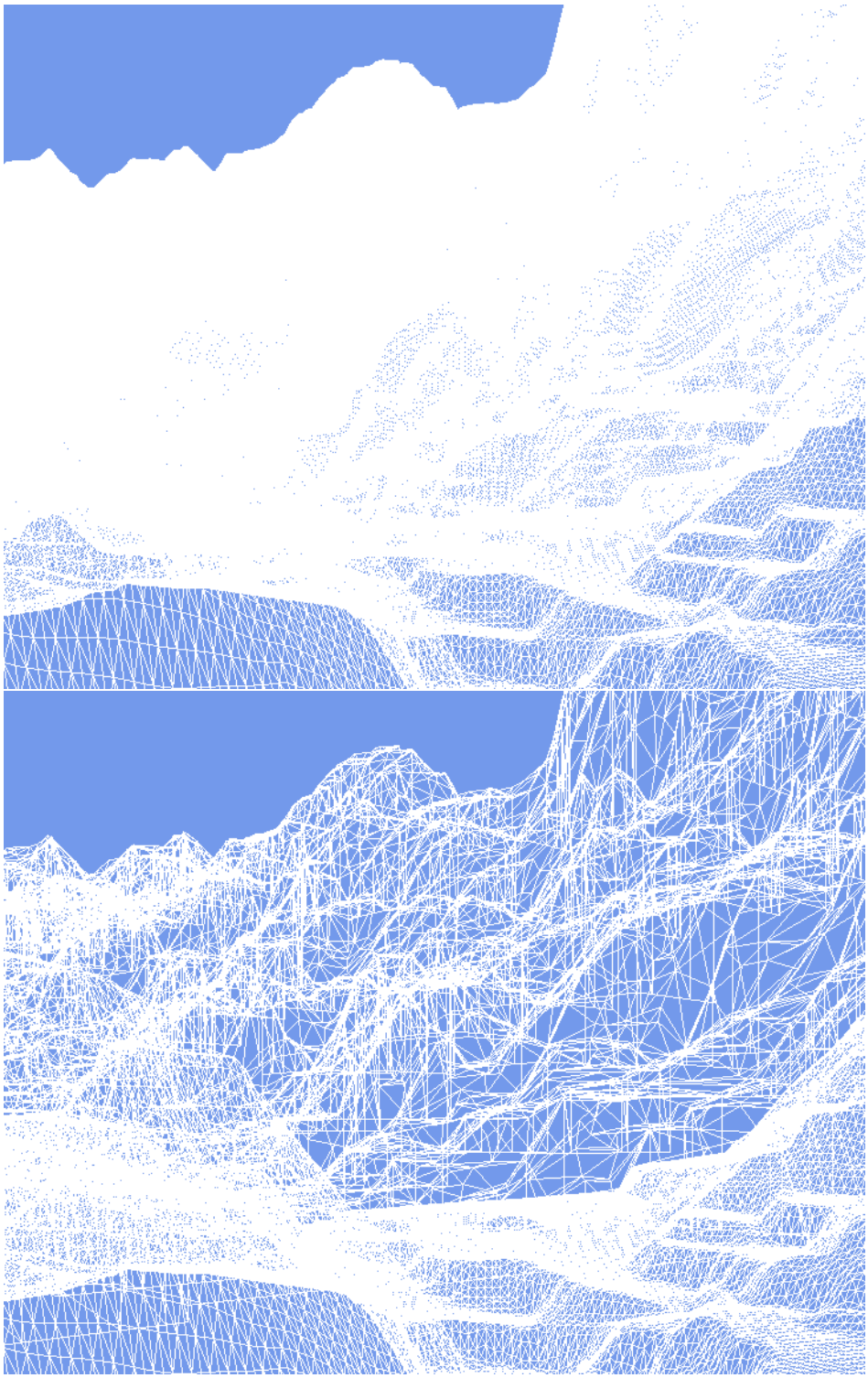

Figure 5.6: Top: The wireframe for the full resolution terrain mesh. Bottom: The wireframe for terrain Impostors using 64 wide chunks. Foreground chunks are drawn in high resolution while background chunks use simplified meshes.

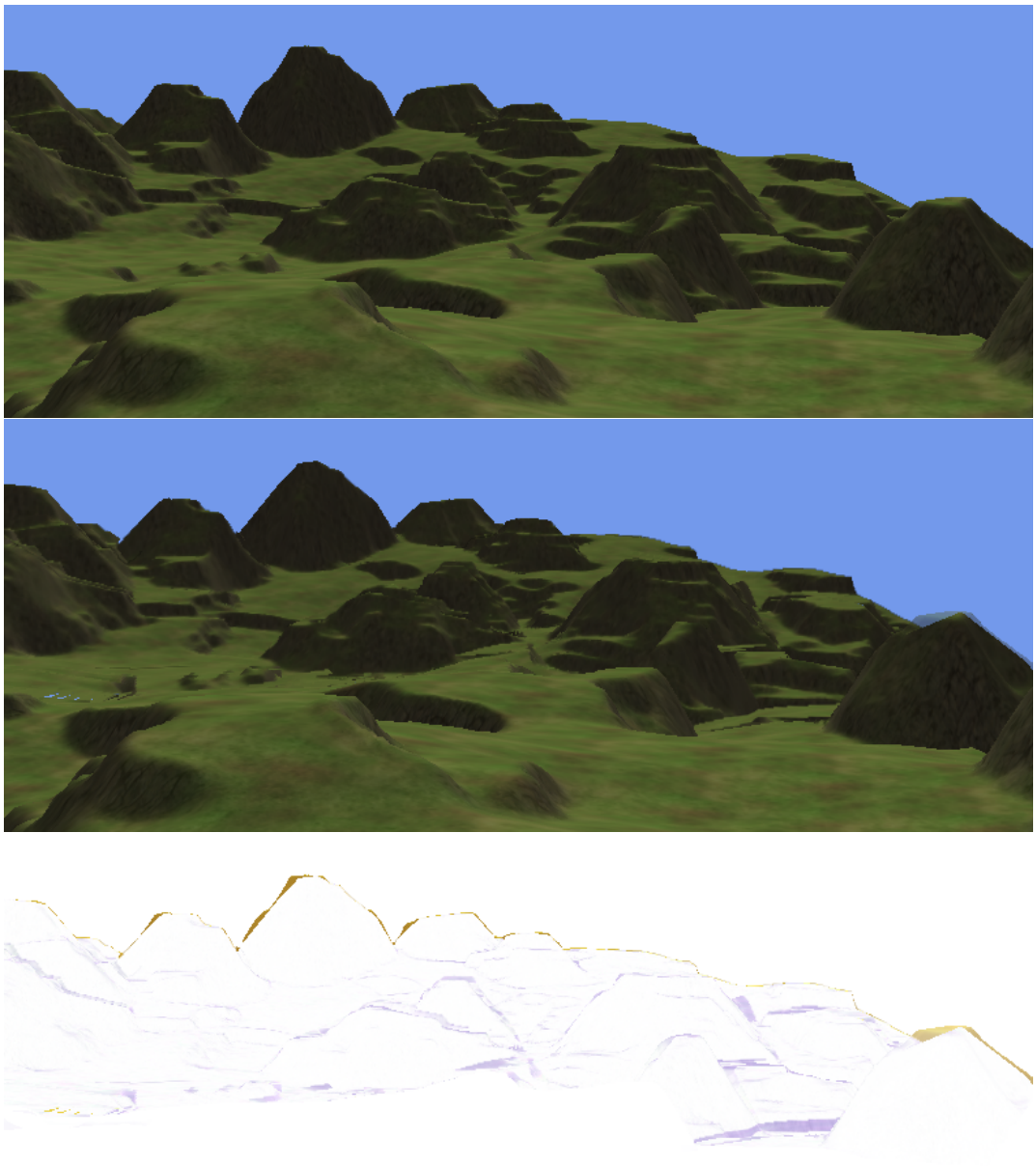

Figure 5.7: Top: A scene rendered using a full resolution mesh. Middle: The same scene rendered using 128 wide chunks. Bottom: An inverted difference image comparing the pixel difference between the two renderings. The larger chunk size exaggerates the self-occluding chunk problem for nearby terrain.

The significance of this implementation is to show that this technique can be used to render geometry faster than using the full mesh every frame. The overhead of rendering into a texture is low enough that this method of rendering scenes should be seriously considered over pure Level of Detail techniques. With the extensions mentioned in Chapter 6, real scalability may be possible and we may begin to see much larger scenes and datasets used in interactive applications.

Chapter 6

Extensions and Future Work

Several extensions to Terrain Impostors were considered during implementation but could not be completed. Sections 6.1, 6.2 and 6.3 explain techniques which could be applied to increase the visual quality of Terrain Impostors. Section 6.5 describes a way to make Terrain Impostors more scalable. Section 6.6 explains how Terrain Impostors could be used to render terrains meshes which are much larger than video and even main memory.

6.1 Silhouette Edge Extension

While seams between chunks of terrain extend geometry to make sure all pixels from the Impostor are drawn to the scene, there is another source of lost pixels. A silhouette edge on the simplified terrain mesh may cut off pixels from the Impostor. The current Implementation makes no attempt to recover these lost pixels. This causes a very subtle change by only a few pixels between the full resolution model and the Impostor as it is viewed in the implementation.

To remedy this problem, additional faces can be extended from silhouette edges of the simplified mesh to "catch" the extra pixels that are missing. This or another possible technique for recovering these missing pixels from the Impostor would be a valuable future improvement.

6.2 Self Occluding Chunks

Occasionally a single chunk of terrain can occlude itself in such a way that when the occluded part of the simplified mesh is revealed as the viewpoint moves, the Impostor becomes a poor representation of the original mesh. A peak in a chunk of terrain can appear to leave a "shadow" of itself on the terrain behind it. This is a limitation of using Impostors.

To overcome this problem, a single chunk of terrain can be split into two or more Impostors. The same view frustum can be used to render both Impostors, except that the near and far clipping planes of each can be adjusted so that the chunk is split at a point that reveals as much as possible of the occluded terrain in one Impostor and uses the occluding part of the terrain in the other. The simplified mesh could either be rendered twice using each Impostor or each triangle of the simplified mesh can be textured using a different Impostor.

6.3 Scene Models

The implementation does not use any decorative meshes like trees, grass or bushes, but these are clearly a desirable part of terrain meshes in entertainment applications. In level of detail techniques, detail meshes often disappear from the terrain very noticeably at a defined radius. Grass and bushes can be raster-

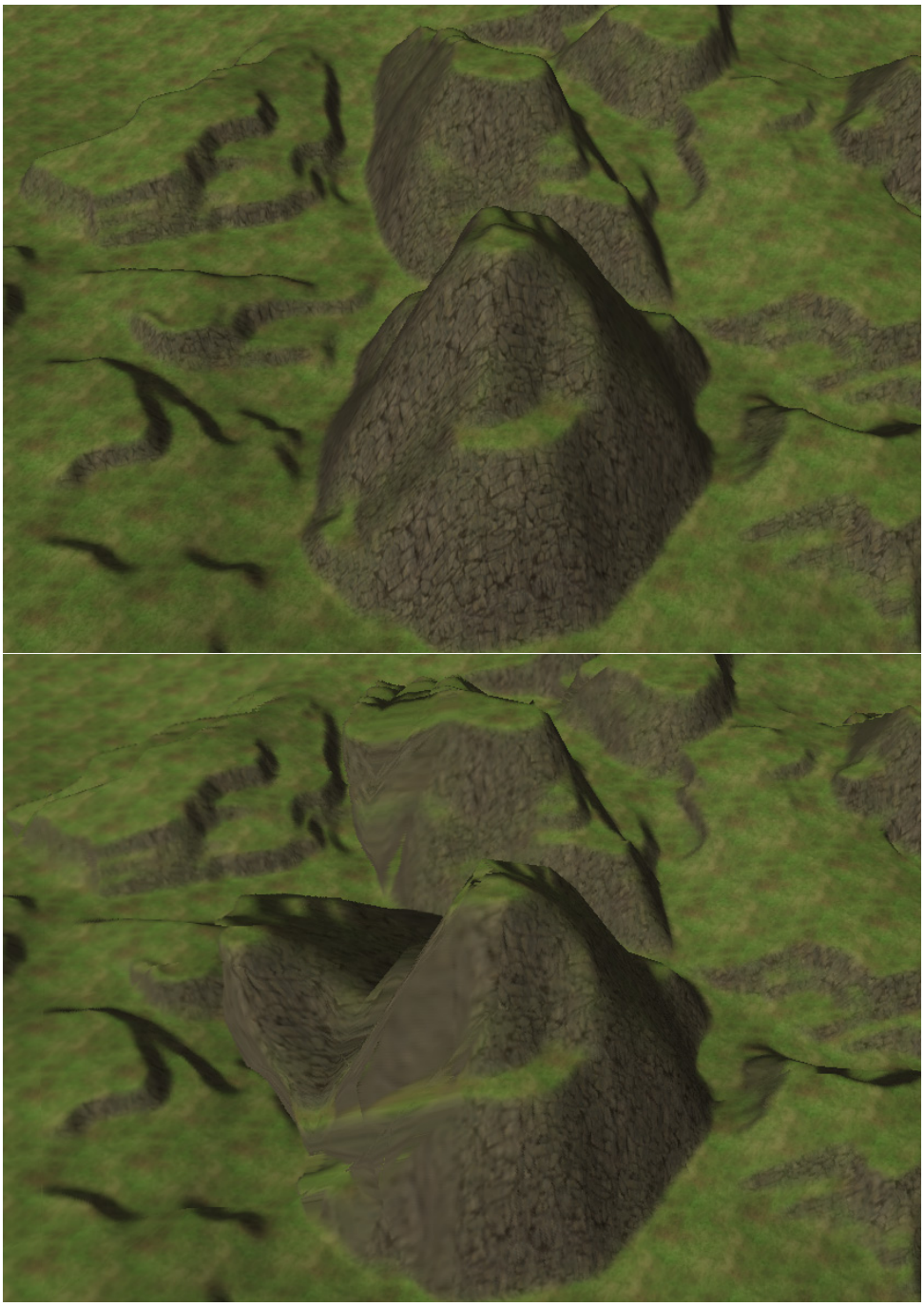

Figure 6.1: Top: A section of terrain rendered normally. Bottom: A terrain Impostor viewed from the side. The hill's imagery is repeated on the terrain behind it. This is the result of a chunk occluding part of its own terrain.

ized into the Impostor like any other part of the terrain and viewed later with no additional overhead. The parallax of small meshes should be appropriately approximated by the same simplified mesh that is used for the terrain. If these meshes increase the height of the horizon of terrain meshes then implementing the silhouette edge extension mentioned in section 6.1 would become even more important.

Larger meshes are more difficult and probably should not be rasterized into the terrain Impostors. Level of detail techniques can be used for large meshes and rendered along with Impostors without difficulty. The simplified meshes still output a coarse depth value. If fine grained depth values are needed, the depth map of the Impostor can be saved and used to output more correct depth values.

6.4 Shader Effects

One drawback of using Impostors is that how lighting affects an object can only be changed as often as its Impostor is redrawn. Very slow lighting changes such as the movement of the sun through the sky may still be acceptable with the slow, incremental updates of Impostors. If dynamic lighting is desired then it may be possible to store a normal buffer in addition to saving the color and depth for every pixel in the Impostor. The color, depth, normal and other desired values can be combined to calculate the lighting equation for every pixel per frame. This would be a similar technique to that used in Deferred Shading [12].

Another possible shader effect would be to use parallax mapping [10] on each of the terrain Impostors. This would have superior parallax effect to the coarse grained parallax from the simplified meshes used in this technique. Parallax mapping would also reduce the amount of geometry in the scene but may hurt

performance by putting more calculations in the fragment shader.

6.5 Large Chunks

As terrain is rendered farther away from the viewpoint, even coarse chunks might only occupy only a few pixels on the screen. When large chunks were used close to the viewpoint, there were very noticeable visual artifacts. Ideally we would like to use the small chunks close to the viewpoint and the larger chunks far away. To make this technique truly scalable, chunk sizes need to increase so that Impostors always occupy a significant number of screen pixels.

A single Impostor which represents a significant portion of terrain may not be able to be updated completely in a single frame. To accommodate this, each smaller chunk of terrain within a group can be rendered individually into the Impostor. If the depth is saved between renderings or the chunks are drawn in decreasing depth order, large Impostors can be constructed incrementally and not interrupt the framerate of the application.

6.6 Out of Core Rendering

One of the most compelling reasons for using Impostors for terrain is that the high resolution mesh for any given Impostor does not need to be in video or main memory while it is being used. If chunks of terrain with update requirements in the order of minutes exist in our scene, it is possible to load terrain into video memory, create an Impostor and then free its memory or use it for another section of terrain.

Combined with techniques for rendering large hierarchies of chunks mentioned in section 6.5, this would allow datasets larger than the memory available on the machine to be viewed accurately and at interactive framerates. This would be very valuable for scientific applications involving very large datasets.

Bibliography

[1] Nvidia graphics cards. http://nvidiagraphicscards.com/.

[2] J. Andersson and DICE. Frostbite: Rendering architecture and real-time procedural shading and texturing techniques. http://developer.amd.com/assets/Andersson-Tatarchuk-FrostbiteRenderingArchitecture(GDC07_AMD_Session).pdf, 2007.

[3] B. Chen, J. Edward, and S. Ii. Lod-sprite technique for accelerated terrain rendering. In *ISBN 0-7803-5897-X. Held in*, pages 291–298, 1999.

[4] P. Debevec. Modeling and rendering architecture from photographs, 1999.

[5] M. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein. Roaming terrain: Real-time optimally adapting meshes, 1997.

[6] J. E. G. Coffman, M. R. Garey, D. S. Johnson, and R. E. Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM Journal on Computing*, 9(4):808–826, 1980.

[7] S. J. Gortler, L. wei He, and M. F. Cohen. Layered depth images, 1997.

[8] H. Hoppe. Progressive meshes, 1996.

[9] T. Igarashi and D. Cosgrove. Adaptive unwrapping for interactive texture painting. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, I3D '01, pages 209–216, New York, NY, USA, 2001. ACM.

[10] T. Kaneko, T. Takahei, M. Inami, N. Kawakami, Y. Yanagida, T. Maeda, and S. Tachi. Detailed shape representation with parallax mapping. In *In Proceedings of the ICAT 2001*, pages 205–208, 2001.

[11] J. Maillot, H. Yahia, and A. Verroust. Interactive texture mapping. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '93, pages 27–34, New York, NY, USA, 1993. ACM.

[12] Nvidia Corporation. Deferred shading. http://http.download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_Deferred_Shading.pdf, 2004.

[13] J. Olsen. Realtime procedural terrain generation. http://oddlabs.com/download/terrain_generation.pdf, 2004.

[14] Refsnes Data. Browser display statistics. http://www.w3schools.com/browsers/browsers_display.asp.

[15] F. Sillion, G. Drettakis, and B. Bodelet. Efficient impostor manipulation for real-time visualization of urban scenery, 1997.

[16] Valve Developer Community. 3d skybox. http://developer.valvesoftware.com/wiki/3D_Skybox.