LiU-ITN-TEK-A--12/073--SE

Adaptive Hardware-accelerated Terrain Tessellation

Albert Cervin

2012-11-14

Linköpings universitet
TEKNISKA HÖGSKOLAN

Department of Science and Technology
Linköping University
SE-601 74 Norrköping, Sweden

Institutionen för teknik och naturvetenskap
Linköpings universitet
601 74 Norrköping

LiU-ITN-TEK-A--12/073--SE

Adaptive Hardware-accelerated Terrain Tessellation

Examensarbete utfört i Medieteknik
vid Tekniska högskolan vid
Linköpings universitet

Albert Cervin

Handledare Stefan Gustavson
Examinator Jonas Unger

Norrköping 2012-11-14

LINKÖPING UNIVERSITY
ELECTRONIC PRESS

Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Albert Cervin

Abstract

In this master thesis report, a scheme for adaptive hardware terrain tessellation is presented. The scheme uses an offline processing approach where a height map is analyzed in terms of curvature and the result is stored in a resource called density map. This density map is then bound as a resource to the hardware tessellation stage and used to bias the tessellation factor for a given edge. The scheme is implemented inside Frostbite™ 2 by EA™ DICE™ and produces good results while making the heightfield rendering more efficient. The performance gain can be used to increase the rendering detail, allowing for better visual appearance for the terrain mesh. The scheme is currently implemented for hardware tessellation but could also be used for software terrain mesh generation. The implementation works satisfactory and produces good results with a reasonable speed.

Sammanfattning

I den här rapporten för examensarbete presenteras en algoritm för att utföra adaptiv hårdvarutessellation av terräng. Algoritmen använder sig av ett offline-steg där ett höjdfält analyseras med avseende på kurvatur och resultatet lagras i en densitets-karta. Den här densitets-kartan används sedan som en resurs i hårdvarutessellationen där den påverkar en tessellationsfaktor för en given triangel-kant. Algoritmen har implementerats i spelmotorn Frostbite™ 2 skapad av EA™ DICE™ och producerar goda resultat samtidigt som den gör rendering av terrängen effektivare. Detta medför att detaljnivån för terrängrenderingen kan ökas, vilket i sin tur leder till en visuell förbättring. Algoritmen är för närvarande endast implementerad för hårdvarutessellation men skulle också kunna användas för mjukvarugenerering av terrängens geometri. Algoritmen fungerar tillfredsställande och producerar goda resultat med en acceptabel hastighet.

Thanks

I want to thank EA DICE, for giving me this opportunity and the Frostbite team for making me feel welcome.

I want to thank my supervisor Mattias Widmark for his patience with my questions, Johan Åkesson also for his patience when Mattias was not available and for making the whole master thesis work possible in the first place.

I furthermore want to thank friends and family for all the support that you have provided!

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

The Frostbite™ 2 terrain system is a highly scalable terrain system. With the introduction of tessellation hardware with DirectX 11/OpenGL 4 class graphics cards, detail displacement mapping was implemented in the Frostbite™ 2 terrain system. The problem with this approach is that it is a brute-force algorithm that does not take the shape of the terrain into account.

In this chapter, the foundations of terrain rendering will be described as an introduction to the subject. It will furthermore present a number of earlier approaches to adaptive terrain rendering.

1.1 Terrain Rendering

Terrain rendering is a challenging task for real time applications since the terrain typically needs to be very large in order to be convincing. The memory and rendering cost makes it impossible to use a large mesh structure, often referred to as a *polygon soup*. To solve this, a heightfield function is often used to describe the shape of the terrain. This function can have values described in a texture (a height map) giving the height for a given world space position, $z = f(x, y)$. A height map is however limited in terms of resolution and can not be infinitely large. For smaller terrains this is generally not an issue, but to be able to support very large terrains, it is necessary to use level of detail support for the height map.

1.1.1 Mesh Generation

The terrain mesh is typically generated in runtime by placing a mesh grid on top of the height field and then displacing the vertices vertically according to the height map. To be able to support large terrains it is necessary to not generate too many primitives in this control mesh. To account for this, a level of detail[1] scheme is needed also for the control mesh. This level of detail is often based on camera distance and can also contain other measures such as view angle, curvature, etc. Since the mesh is dependent on the viewer, it is generated procedurally at runtime.

1.1.2 Adaptive Terrain LOD

For adaptive terrain mesh generation on the CPU, there are many algorithms. The most widely known method is perhaps ROAM, presented by Duchaineau et. al. in 1997 [5]. ROAM means Real-time Optimally Adapting Meshes and uses two priority queues to drive a series of split and merge operations, producing an optimal mesh for a particular view.

ROAM is a dynamic mesh representation based on triangle bintrees. Triangle bintrees are the triangle counterpart of a binary tree. At the lowest LOD, the tree consists of one triangle, the root triangle. The base triangulation is precomputed and the bintree is then defined recursively by splitting each triangle along an edge formed from the apex vertex of the triangle. Series of split and merges can then be used to obtain any triangulation of the mesh. The splits and merges can also be animated using vertex morphing where a lower LOD triangle is morphed into a higher LOD triangle or vice versa.

Split Queues

The split and merge operations in the bintree stucture provides a way to achieve any triangulation and there is no need to take special care to avoid cracks or T-vertices. With the split and merge framework in place, a measure to control the triangulation is needed. Duchaineau et. al. uses a priority queue for this purpose, that tells which triangles to split. First, all triangles in the bintree are put into the priority queue. Then the triangle with the highest priority is found in the queue and it is split. The split queue is then updated by removing the newly split triangle and adding any created triangles. This is then repeated as long as the triangle mesh is too small or inaccurate, and will create a triangle mesh that minimizes the maximum priority in the queue (often an error measure).

[1]Hereafter LOD

Frame-to-Frame Coherence

The above works well for a static view, but for an interactive view, the frame-to-frame coherency has to be taken into account to get a good framerate. Duchaineau et. al. uses the observation that the changes in priority from one frame to the other are in general relatively small. They introduce a second priority queue, the merge queue. This queue contains all mergeable triangle diamonds (two neighboring triangles from the same LOD) for the current triangulation. The priorities in this queue are obtained by using the maximum of the two diamond triangle priorities. A condition is now added to the algorithm to check if a triangle should be split or if one should be merged. This way, the algorithm becomes incremental in the sense that it produces an optimal mesh based on the mesh for the previous frame. The worst case for this algorithm is when very few triangles are common from one frame to the next and the remedy for this case is to fall back to the original algorithm as if the current frame was the first frame.

Error Metrics

To be able to use the priority queues they need to have some kind of metric attached to them to drive the prioritization. Duchaineau et. al. base this error metric on the geometric screen space distortion for the triangle. That is, how far is the surface point from where it is supposed to be in screen space. In practice this is done by calculating an upper bound for the maximum distortion. For each triangle in the triangulation, a local upper bound on the distortion can be found by projecting the wedgie of the triangle into screen space. A triangle wedgie is defined as the volume of world space that contains points (x, y, z) of the triangle T in a way such that $(x, y) \in T$ and $|z - z_T(xy)|$, where $z_T(x, y)$ is the height value as described by the height map at position (x, y).

1.1.3 Chunked LOD

Thatcher Ulrich proposed a new technique for rendering large terrains adaptively in 2002 [9]. The technique generates static meshes as a preprocessing step which are stored at different LOD levels in a quadtree. In runtime, the needed LOD is calculated and rendered from the quadtree. When quadtree nodes with different LOD meet, there will be cracks at the borders. Ulrich proposes a hybrid solution to the problem using vertical skirts that are simple triangles that extend vertically at the edge of the patch to cover the crack that occurs. This means that the bottom edge of the skirt has to extend below the full LOD of the mesh at the edge and has to extend below any possible simplifications of it. The skirts belong to a chunk and is contained in them and may be textured using the chunk texture.

Texturing is simple for this LOD scheme. When preprocessing, each chunk is assigned a static texture. This makes it possible to have a consistent resolution that is at least one texel per screen pixel.

The rendering of the terrain chunks is done in a view-dependent manner. This means that for a view, chunks are chosen from the quadtree structure to match the desired fidelity of the terrain model. Each chunk (node) in the quadtree has an associated maximum geometric error and a bounding volume. This makes the calculation of which node to use

$$\rho = \frac{\delta}{D}K \tag{1.1}$$

where ρ is the maximum screen space error that this particular node will result in, δ is the maximum geometric error associated with the chunk and D is the distance from the camera to the closest point on the chunk. Furthermore, K is a perspective scaling factor that takes viewport size and field-of-view into consideration. K is computed as

$$K = \frac{\text{viewport\_width}}{2\tan\frac{\text{horizontal\_fov}}{2}}. \tag{1.2}$$

To render a chunk, the quadtree is traversed from the root with a predefined maximum tolerable screen space error. If the current chunk in the traversal is acceptable by means of screen space error calculated by equation 1.1, the chunk is rendered. If the screen space error of the current chunk is too large, the tree traversal continues with the children of the node.

Avoiding Pops

When a parent node in the quadtree is replaced by child nodes, there will be a distinct pop between the two different LOD levels. This can be solved by adding a small morph to the vertical coordinate of each vertex. The morph parameter is uniform over the whole chunk. For a chunk, a vertex morph target has the same horizontal coordinate and the vertical coordinate is calculated by sampling the height of the parent chunk at these known horizontal coordinates.

When the chunk is rendered, the morph parameter is calculated in such a way that it is always 0 when the chunk is about to split and 1 when the chunk is about to merge. This means that the shape of the chunk will be consistent over LOD switches. The morph parameter can be calculated with the help of the previously defined error metric ρ in equation 1.1

$$t_{\text{morph}} = \text{clamp}(\frac{2\rho}{\tau} - 1, 0, 1). \tag{1.3}$$

Equation 1.3 will give $t_{\text{morph}} = 0$ exactly at the distance where a chunk is split into four smaller ones and $t_{\text{morph}} = 1$ exactly at the distance where four child tiles are merged into one. The equation comes from the fact that δ of the parent node is 2δ for the child nodes.

Paging

The chunked LOD system also supports paging of out-of-core chunks. This means that only chunks needed for the current view are kept in main memory. Chunks are then swapped out and read from disk as they are needed. Therefore, it is necessary to keep a pool of terrain chunks in main memory such that nodes that has not been used for some time can be freed.

1.1.4 CDLOD

Another, more recent approach is the CDLOD approach proposed by Filip Strugar in 2010 [7]. This algorithm also organizes the height map into a quadtree just as Chunked LOD by Ulrich. The selection algorithm then assures that the on-screen triangle complexity is kept constant, regardless of the distance to the viewer.

LOD Transition

CDLOD means continuous distance-dependent level of detail and this is accomplished by using a continuous morph between LOD levels. In contrast to the approach proposed by Ulrich [9], CDLOD does not use any stitching geometry to avoid cracks in LOD switches. Instead, the higher level mesh is completely transformed into the lower level mesh before the switching occurs. This means that there is no popping when changing LOD levels. It also allows for a simpler rendering since only one rectangular grid mesh is needed to render everything. This LOD transition approach is also better as a platform for hardware tessellation since there will be no sudden changes in the underlying heightfield mesh, resulting in less popping artifacts.

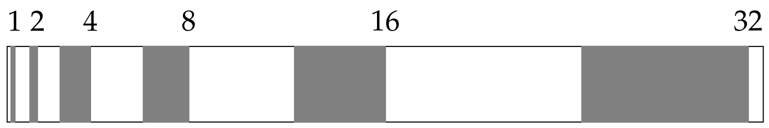


Figure 1.1: The range table for six LOD ranges with relative sizes at the top. The morph area of each range is shown in gray.

Rendering the Terrain

The first step in rendering terrain with the CDLOD terrain system is to select an appropriate node from the quadtree structure. This step is performed every time the view is changed. To make rendering more efficient, the quadtree is laid out such that each depth level in the quadtree corresponds to a LOD level. The reason this makes rendering of the terrain simpler is that the same single fixed mesh can be used to render all nodes. Since nodes are stored in a quadtree, each node has four child nodes, with each of the child nodes occupying a fourth of the area of the parent node. This means that the corresponding world space area will have four times the triangles.

The distances covered by each LOD layer is precomputed and stored in a table. The distance covered by a level should be two times larger than the previous one. This is since each node has four children and due to the way perspective projection (which is assumed) works. The last 15-30% of the areas are used for the mesh morphing and is thus called morph areas. The range table layout is illustrated in figure 1.1.

When the array of LOD ranges has been calculated it is used to select a subset of the terrain quadtree that best represents the terrain at a certain view. To determine this subset, the quadtree is traversed recursively from the root. If a node falls in the selected range, the children of that node is traversed to find the highest lod that matches the distance. A node can also be selected partially over an area. This is to ensure that not all child nodes has to be rendered if only a few are in LOD range. An example of a selected quadtree subset is shown in figure 1.2.

Frustum culling can also be performed when traversing the tree to select nodes for rendering.

After a subset of the quadtree has been selected, it is rendered by iterating through a list with the selected nodes and their data. The actual rendering is not very complicated and consists of a single grid mesh of fixed dimensions that is transformed in the vertex shader to cover the desired terrain area.

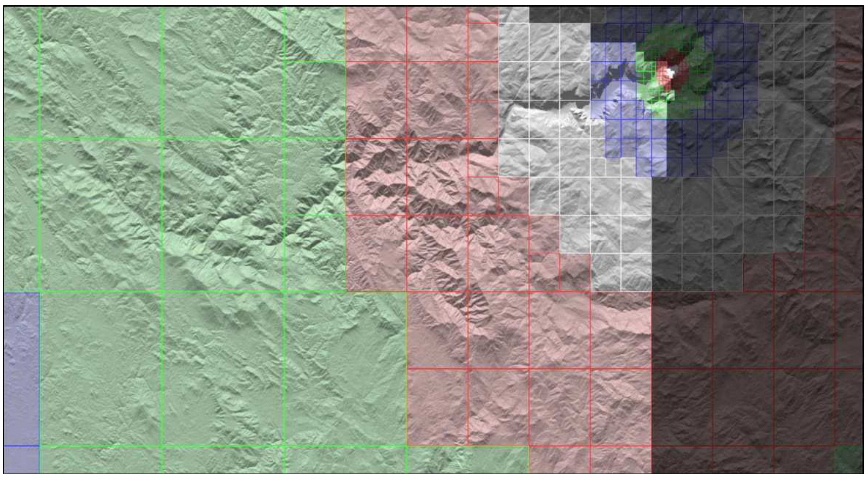


Figure 1.2: Example of LOD quadtree selection. Darker nodes are frustum culled. Image from [7].

Morph Implementation

In the CDLOD algorithm, each vertex is morphed individually based on a per-vertex LOD metric. This is not the case in the Chunked LOD approach by Ulrich [9] where the morph is uniform over a chunk. The morphing operation is done in the vertex shader and each node can be morphed to match a node either one level higher or one level lower in the quadtree. The morph is performed in such a way that every block of 8 triangles are smoothly morphed into a corresponding block of 2 triangles. This morphing will result in smooth transitions with no seams or T-junctions (T-vertices).

The first step is to approximate the distance between the observer and the vertex. The vertex position used in this approximation can be approximated or sampled from the height map. However, it is important that the approximation or sampling is consistent on both sides of a LOD edge to avoid cracks. The vertex is then morphed based on the distance from the vertex to the viewer.

After this morphing, the height is sampled from the heightmap and the vertex is displaced vertically.

Streaming

As was the case with the Chunked LOD algorithm, the CDLOD algorithm also supports streaming of quadtree nodes to lower the memory costs for rendering large terrains.

1.2 Detail Displacement Mapping

A heightfield based terrain is essentially a flat mesh that is displaced with a displacement map, the height map. Displacement mapping can be described as

$$P'(x,y,z) = P(x,z) + D(y) \tag{1.4}$$

in the heightfield case. It is also possible to displace a 3D mesh by a 3D vector which is called vector displacement. However, the control mesh sent to the displacement mapping algorithm is important. Ideally, it has one vertex per displacement map sample. This is for practical reasons not always possible but with the introduction of hardware tessellation in Direct X 11 consumer graphic cards it is possible to generate sufficiently dense meshes effectively. This means that it is also possible to combine a CPU LOD with a GPU LOD scheme where extra detail is added. The CPU LOD can in this case use a coarser generated control mesh that is then tessellated by hardware to get a higher resolution mesh. This can save CPU time needed for other parts of the application.

1.2.1 Character Detail Displacement Mapping

Detail displacement mapping is often used in character modeling. Tools such as ZBrush use subdivision surfaces combined with vector displacement mapping. With the introduction of tessellation hardware in consumer graphics cards, this technique has become increasingly interesting for real-time applications. The most popular subdivision scheme is perhaps the Catmull-Clark scheme. Catmull-Clark subdivision surfaces cannot be used directly since patches that contains extraordinary vertices consists of an infinite set of polynomials. For this reason, Loop et. al. [6] proposes two schemes to approximate Catmull-Clark subdivision surfaces. There are also approaches that do not use Catmull-Clark surfaces. One such example is the PN-Triangles approach suggested by Vlachos and Peters [10].

Chapter 2

Background

This chapter will first give a mathematical background in the field of differential geometry on surfaces. The density value described later in the report will be based on curvature, so a mathematical foundation is needed. It will then describe the Frostbite$^{\text{TM}}$ 2 terrain system to provide the necessary understanding for the implementation of the density map algorithm.

2.1 Differential Geometry Background

Since the heightfield is essentially a 2.5D surface, differential geometry for surfaces is highly relevant to the problem. This section will provide a mathematical background to the differential geometry used throughout the report.

2.1.1 Introduction

The field of differential geometry on surfaces is well studied and well described in books such as *Differential Geometry of Curves and Surfaces* by do-Carmo [4], which can be consulted for a more complete introduction on the subject.

Consider a continuous surface $S \subset R^3$ given in parametric form

$$\mathbf{x}(u,v) = \begin{pmatrix} x(u,v) \\ y(u,v) \\ z(u,v) \end{pmatrix} \tag{2.1}$$

where x, y, z are differentiable functions in u and v. A tangent plane to S is spanned at $\mathbf{x}$ by the two partial derivatives $\mathbf{x}_u$ and $\mathbf{x}_v$. The normal vector at $\mathbf{x}$ is then given by $\mathbf{n} = \frac{(\mathbf{x}_u \times \mathbf{x}_v)}{\|\mathbf{x}_u \times \mathbf{x}_v\|}$.

First fundamental form

The first fundamental form is defined as coefficients of the dot product on the tangent space of S. The dot product is

$$\mathbf{I}(a\mathbf{x}_u + b\mathbf{x}_v, c\mathbf{x}_u + d\mathbf{x}_v) = Eac + F(ad + bc) + Gbd \tag{2.2}$$

where E, F and G is the coefficitents of the first fundamental form. If written as a metric tensor, the first fundamental form becomes

$$\mathbf{I} = \begin{bmatrix} E & F \\ F & G \end{bmatrix} = \begin{bmatrix} \mathbf{x}_u \cdot \mathbf{x}_u & \mathbf{x}_u \cdot \mathbf{x}_v \\ \mathbf{x}_u \cdot \mathbf{x}_v & \mathbf{x}_v \cdot \mathbf{x}_v \end{bmatrix}. \tag{2.3}$$

Second fundamental form

The second fundamental form was introduced by Gauss and considering the surface defined in 2.1 the second fundamental form can be defined as

$$\mathbf{II} = edu^2 + 2fdudv + gdv^2 \tag{2.4}$$

and written in matrix form this becomes

$$\mathbf{II} = \begin{bmatrix} e & f \\ f & g \end{bmatrix} = \begin{bmatrix} \mathbf{x}_{uu} \cdot \mathbf{n} & \mathbf{x}_{uv} \cdot \mathbf{n} \\ \mathbf{x}_{uv} \cdot \mathbf{n} & \mathbf{x}_{vv} \cdot \mathbf{n} \end{bmatrix}. \tag{2.5}$$

With the first and second fundamental form defined, it is possible to measure length, angles, area and curvatures on the surface.

Normal curvature

Let $\mathbf{t} = a\mathbf{x}_u + b\mathbf{x}_v$ be a unit vector in the tangent plane at $\mathbf{p}$ which is represented as $\mathbf{t} = (a, b)$ in some local coordinate system. Then the normal curvature can be defined as the curvature of the planar curve that is the result of intersecting the surface S with a plane through $\mathbf{p}$, spanned by $\mathbf{n}$ and $\mathbf{t}$. The normal curvature in a direction $\mathbf{t}$ can be written as

$$\kappa_n(\mathbf{t}) = \frac{\mathbf{t}^T II \mathbf{t}}{\mathbf{t}^T I \mathbf{t}} = \frac{ea^2 + 2fab + gb^2}{Ea^2 + 2Fab + Gb^2}. \tag{2.6}$$

The maximum and minimum normal curvatures κ_1 and κ_2 are called principal curvatures. The corresponding direction vectors $\mathbf{t}_1$ and $\mathbf{t}_2$ are called the principal directions. Worth to note is that these two directions are always perpendicular to each other.

Weingarten equations

With the first and second fundamental form given, the derivative of the unit normal $\mathbf{n}$ can be described in terms of the first derivatives of the position vector $\mathbf{r} = \mathbf{r}(u, v)$. With the coefficients of the first and fundamental forms E, F, G, e, f, g respectively.

$$\mathbf{n}_u = \frac{Ff - Ge}{EG - F^2}\mathbf{r}_u + \frac{Fe - Ef}{EG - F^2}\mathbf{r}_v \tag{2.7}$$

$$\mathbf{n}_v = \frac{Fg - Gf}{EG - F^2}\mathbf{r}_u + \frac{Ff - Eg}{EG - F^2}\mathbf{r}_v \tag{2.8}$$

The shape operator

If the Weingarten equations are written in matrix form, the *Weingarten curvature matrix* (alt. second fundamental tensor) is obtained

$$\mathbf{W} = \frac{1}{EG - F^2} \begin{bmatrix} eG - fF & fG - gF \\ fE - eF & gE - fF \end{bmatrix}. \tag{2.9}$$

As described above, the Weingarten equations describe the directional derivative of the unit normal. This means that the normal curvature can be described as

$$\kappa_n(\mathbf{t}) = \mathbf{t}^T \mathbf{W} \mathbf{t}. \tag{2.10}$$

If $\mathbf{t}_1$ and $\mathbf{t}_2$ defines a local coordinate system, $\mathbf{W}$ becomes a diagonal matrix

$$\mathbf{W} = \begin{bmatrix} \mathbf{t}_1 & \mathbf{t}_2 \end{bmatrix} \begin{bmatrix} \kappa_1 & 0 \\ 0 & \kappa_2 \end{bmatrix} \begin{bmatrix} \mathbf{t}_1 & \mathbf{t}_2 \end{bmatrix}^{-1} \tag{2.11}$$

which in turn means that the normal curvature can be written as

$$\kappa_n(\mathbf{t}) = \kappa_n(\phi) = \kappa_1 \cos\phi^2 + \kappa_2 \sin\phi^2 \tag{2.12}$$

where ϕ is the angle between $\mathbf{t}_1$ and $\mathbf{t}_2$.

Curvatures

From the above definitions it is possible to express two curvature measures. The mean and Gaussian curvature.

The mean curvature is defined as the mean value of the principal curvatures

$$K = \frac{\kappa_1 + \kappa_2}{2} = \frac{1}{2}\text{trace}(\mathbf{W}) \tag{2.13}$$

and the Gaussian curvature is the product of the principal curvatures

$$H = \kappa_1 \kappa_2 = \det(\mathbf{W}). \tag{2.14}$$

Laplace operator

The Laplace operator Δ is defined as the divergence of the gradient $\Delta = \nabla^2 = \nabla \cdot \nabla$. In Euclidian space this is the sum of second order partial derivatives.

$$\Delta f = \text{div}\Delta f = \sum_i \frac{\delta^2 f}{\delta x_i^2} \tag{2.15}$$

This concept however does not work for functions defined on surfaces. For that, the *Laplace-Beltrami*-operator is used. This operator is defined as

$$\Delta_S f = \text{div}_S \Delta_S f \tag{2.16}$$

where S is a manifold surface and f is the function defined on the surface.

If this operator is applied to the coordinate function $\mathbf{x}$ it evaluates to

$$\Delta_S \mathbf{x} = -2H\mathbf{n} \tag{2.17}$$

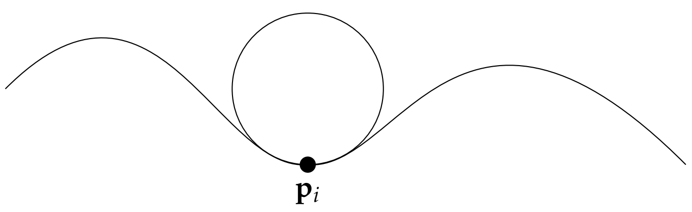


Figure 2.1: An osculating circle.

which is the mean curvature normal. This means that the mean curvature can be calculated by applying the Laplace-Beltrami operator to a surface.

Discretization

Polygonal meshes are not smooth surfaces, but rather piecewise linear approximations. The definition of the curvature tensors also require the existence of second order derivatives. To be able to calculate differential properties on a polygonal surface, discretization has to be done. A common approach for computing discrete differentials is to consider spatial averages over a local neighborhood $N(\mathbf{x})$ for a point $\mathbf{x}$ on the surface. The size of this neighborhood affects the stability of the calculations. A larger neighborhood will smooth the calculations, making them less sensitive to noise. The neighborhood size is often measured in ring size. A one-ring neighborhood means the ring of directly connected neighbor vertices and a two-ring neighborhood means vertices that are directly connected and vertices that are in turn connected to these vertices.

A common approach to estimate the curvature tensor at a vertex is to first discretize the normal curvature. Given vertex positions $\mathbf{p}_i$, $\mathbf{p}_j$ and the normal $\mathbf{n}_i$, the normal curvature in the direction along the edge between $\mathbf{p}_i$ and $\mathbf{p}_j$ is

$$\kappa_{ij} = 2\frac{(\mathbf{p}_j - \mathbf{p}_i)\mathbf{n}_i}{\|\mathbf{p}_j - \mathbf{p}_i\|^2} \tag{2.18}$$

Geometrically this can be interpreted as fitting the osculating circle interpolating $\mathbf{p}_i$ and $\mathbf{p}_j$ with normal $\mathbf{n}_i$ at $\mathbf{p}_i$. This is illustrated in figure 2.1.

2.1.2 Heightfield Differentials

Laplacian

The Laplacian of a heightfield function is the sum of the second order partial derivatives of the surface. With a heightfield described by $z = h(u, v)$, the discrete Laplace filter becomes

$$\nabla h = \frac{\delta^2 h}{\delta u^2} + \frac{\delta^2 h}{\delta v^2}. \tag{2.19}$$

Curvatures

For a heightfield function $z = h(u, v)$, the discretization of curvature measures can be derived by considering the surface S again but this time with a heightfield function.

$$\mathbf{x}(u, v) = \begin{pmatrix} u \\ v \\ h(u, v) \end{pmatrix} \tag{2.20}$$

With this definition, the derivatives for the heightfield function becomes

$$\begin{aligned} \mathbf{x}_u &= (1, 0, h_u), & \mathbf{x}_v &= (0, 1, h_v) \\ \mathbf{x}_{uu} &= (0, 0, h_{uu}), & \mathbf{x}_{vv} &= (0, 0, h_{vv}) \\ \mathbf{x}_{uv} &= \mathbf{x}_{vu} = (0, 0, h_{uv}) \end{aligned} \tag{2.21}$$

and the unit normal

$$\mathbf{n} = \frac{(-h_u, -h_v, 1)}{\sqrt{1 + h_u^2 + h_v^2}} \tag{2.22}$$

The coefficients of the first fundamental form is given by (equation 2.3)

$$\mathbf{I} = \begin{bmatrix} \mathbf{x}_u \cdot \mathbf{x}_u & \mathbf{x}_u \cdot \mathbf{x}_v \\ \mathbf{x}_u \cdot \mathbf{x}_v & \mathbf{x}_v \cdot \mathbf{x}_v \end{bmatrix} = \begin{bmatrix} 1 + h_u^2 & h_u h_v \\ h_u h_v & 1 + h_v^2 \end{bmatrix} \tag{2.23}$$

and the coefficients of the second fundamental form becomes

$$\mathbf{II} = \begin{bmatrix} \mathbf{x}_{uu} \cdot \mathbf{n} & \mathbf{x}_{uv} \cdot \mathbf{n} \\ \mathbf{x}_{uv} \cdot \mathbf{n} & \mathbf{x}_{vv} \cdot \mathbf{n} \end{bmatrix} = \frac{1}{\sqrt{1 + h_u^2 + h_v^2}} \begin{bmatrix} h_{uu} & h_{uv} \\ h_{uv} & h_{vv} \end{bmatrix} \tag{2.24}$$

With the coefficients for the first and second fundamental form in place, recall that the mean curvature is given by the mean value of the principal curvatures or by the trace of the Weingarten matrix. With the above coefficients, the Weingarten matrix is

$$W = \frac{1}{EG - F^2} \begin{bmatrix} eG - fF & fG - gF \\ fE - eF & gE - fF \end{bmatrix} \tag{2.25}$$

which gives the mean curvature

$$\begin{aligned} H &= \frac{1}{2}\text{trace}(\mathbf{W}) \\ &= \frac{1}{2}\frac{eG - fF + gE - fF}{EG - F^2} \\ &= \frac{1}{2}\frac{1}{\sqrt{1 + h_u^2 + h_v^2}}\frac{h_{uu}(1 + h_v^2) + h_{vv}(1 + h_u^2) - 2h_{uv}h_u h_v}{1 + h_u^2 + h_v^2} \\ &= \frac{h_{uu}(1 + h_v^2) - 2h_{uv}h_u h_v + h_{vv}(1 + h_u^2)}{2(1 + h_u^2 + h_v^2)^{3/2}}. \end{aligned} \tag{2.26}$$

With the help of finite differences, this mean curvature equation can be used to retrieve curvature information from a heightfield function. The formula for Gaussian curvature is obtained in a similar fashion but instead from the determinant of **W**.

$$\begin{aligned} K &= \frac{1}{2}\det(\mathbf{W}) \\ &= \frac{h_{uu}h_{vv} - h_{uv}^2}{(1 + h_u^2 + h_v^2)^2} \end{aligned} \tag{2.27}$$

The Laplace-Beltrami Operator

The Laplace-Beltrami operator is, as mentioned above, an extension to the Laplace operator for use on surfaces. The Laplace-Beltrami operator evaluates to the mean curvature normal since

$$-\frac{\nabla s_{\mathbf{x}}}{2} = \check{\mathbf{n}} = \frac{\Delta A}{2A}. \tag{2.28}$$

This means that the mean curvature can be calculated by evaluating the Laplace-Beltrami operator on the surface.

Taubin [8] proposed a uniform discretization to this operator by considering a surface signal to be a function $x = (x_1, \ldots, x_n)^t$ defined on the vertices of a polyhedral surface. The Laplacian of the surface can then be discretized as the weighted averages of the neighborhood.

$$\Delta x_i = \sum_{j \in N_1(i)} w_{ij}(\mathbf{x}_j - \mathbf{x}_i) \tag{2.29}$$

where w_{ij} are positive weights defined for each vertex pair that sum up to one, $\sum j \in N_1(i) w_{ij} = 1$. There are many ways to choose these weights and a very simple choice is to set w_{ij} to the inverse of the number of vertices in the chosen neighborhood. This can in some cases produce sufficiently good results. However, these weights do not take the local geometry around x_i into consideration which means that the approximation will be bad for irregularly tessellated meshes. It will consider vertices that are moved from the barycenter of the region as curvatures, even though the area is completely flat. This will produce good tessellation patterns but a bad approximation of the Laplace-Beltrami operator.

A better approximation of the operator is obtained if the area of the neighborhood is considered.

$$\nabla_S f(v) = \frac{1}{A} \sum_{v_i \in N_i(v)} (\cot \alpha_j + \cot \beta_j)(f(v_i) - f(v)). \tag{2.30}$$

This means that the final sum is divided by the sum of the polygon areas in the chosen neighborhood. $\cot \alpha_j$ and $\cot \beta_j$ are the angles between the current vertex v_i and the next and previous vertices in the ring, v_{j+1} and v_{j-1} respectively.

The measure can however be improved further, by instead considering the Voronoi area of the neighborhood. This gives the discretization

$$\nabla_s f(v) = \frac{1}{A_v} \sum_{v_i \in N_i(v)} (f(v_i) - f(v)). \tag{2.31}$$

where A_v is the Voronoi area of the neighborhood

$$A_v = \frac{1}{8} \sum_{j \in N_1(i)} (\cot \alpha_j + \cot \beta_j)|\mathbf{v}_i - \mathbf{v}_j|^2. \tag{2.32}$$

The mean curvature is then $H(v) = \frac{1}{2}\|\Delta_s f(v)\|$.

The same approach can also be used to get a more accurate discrete estimate for the Gaussian curvature

$$K(v_i) = \frac{1}{A_v}\left(2\pi - \sum_{v_j \in N_1(v_i)} \theta_j\right), \tag{2.33}$$

where θ_j is the angle of the incident triangle at v_j. Geometrically, the Gaussian curvature can be interpreted as the deviation from 2π in the one-ring neighborhood and the formula is a direct consequence of the Gauss-Bonnet theorem. If both the mean and Gaussian curvatures are known, it is possible to calculate the principal curvatures from the two

$$\kappa_{1,2}(v) = H(v) \pm \sqrt{H(v)^2 - K(v)}. \tag{2.34}$$

2.2 The Frostbite™ 2 Terrain System

This section will describe the terrain system in Frostbite™ 2. For a more in-depth view of this system, consult the presentation by Widmark from Game Developers Conference 2012 [11].

The terrain system in Frostbite™ 2 is a highly scalable terrain system and has support for level-of-detail in many different parts of the system. The terrain system is height-map based and generates terrain procedurally at runtime. To be able to handle very large terrains, the heightfield raster is divided into tiles that can have different resolutions. Typically, the tiles residing in the playable area of a level has a higher spatial resolution than tiles at the outer edges of the level.

The scalability in Frostbite™ 2 is defined in terms of arbitrary view distance, LOD and speed. Arbitrary view distance means that it must be possible to vary view distance from 0.06m up to 30 000m. Furthermore, the level of detail must be arbitrary and handle 0.0001m and lower. The terrain must also be viewable at different speeds ranging from walking to jet planes.

2.2.1 Data Layout

All data in the terrain system is laid out in a quadtree structure. This layout is similar to the layouts proposed by Ulrich [9] and Strugar [7] and is also

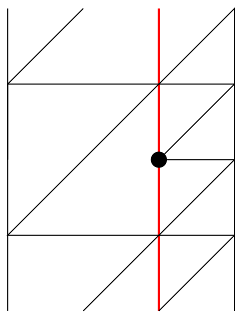


Figure 2.2: A T-vertex (marked in black) at a LOD edge (red).

similar to many flight simulators. Nodes that are closer to the root of the tree describe data with a lower level of detail.

All nodes in the quadtree structure has binary data associated with it but not all nodes have their binary data loaded. In runtime, heightfield tiles for example, are stored in a virtual texture atlas and streamed from disk as they are needed. This makes it possible to support very large terrains whose memory and processing requirements scale well. However, a fraction of the nodes has their binary data in memory all the time. These nodes are needed for multiplayer server simulations.

T-Vertices

A T-vertex is a vertex that is at the border between two differing levels of detail. The tile with the higher level of detail has a vertex in between two vertices in the tile with the lower level of detail. This vertex will create a T-shape, that can result in a crack when the heightfield mesh is displaced. The case is shown in figure 2.2.

To remedy this situation in the Frostbite™ 2 terrain engine, a stitching algorithm is applied to fix LOD switch edges. This is done with index permutations and the original vertices in the mesh are not changed.

2.2.2 Level of Detail

The terrain system has two mechanisms for supporting different level of detail on the procedurally generated heightfield mesh. One is the CPU LOD scheme and the other scheme is implemented on top of the CPU scheme and uses hardware GPU tessellation. This scheme is naturally only active on hardware that supports it. Currently, this means only Direct X 11 graphics hardware.

CPU-Level of Detail

The CPU approach to level of detail is based on the quadtree structure described in section 2.2.1. The terrain mesh is, as mentioned, generated procedurally in runtime and the level of detail is based on the distance to the camera.

The quadtree structure ensures that the step between two neighboring patches is at most one level of detail. This makes removing T-vertices (avoiding cracks) simpler since it is always possible to know that the neighboring triangle patch is only half or double the size of the current one. This means that all possible index permutations needed to stitch the edges as described above, can be stored in advance. Andersson [1] calls this a restricted quadtree.

2.2.3 Virtual Texturing

Virtual texturing (sometimes mega-texturing) was proposed by John Carmack [3] and is used where one large texture would simply not provide enough detail for a reasonable size of the texture. Virtual texturing makes it possible to have a very large texture by placing smaller parts of the big texture in an atlas which is a large texture that can fit a fixed number of tiles from the original texture.

The Frostbite™ 2 terrain engine uses something that is called *Procedural Shader Splatting* [1]. This means that shaders are applied based on masks that can be painted by artists. However, this makes rendering of the terrain slow (10-20ms) [11] and the solution for this is to render the results into a virtual texture. The frame-to-frame coherency can thus be used and the rendering can be split into multiple passes. With this optimization, a full screen rendering of the terrain takes 2.5-3ms on the Playstation™ 3 [11].

2.3 DirectX 11 Hardware Tessellation

The DirectX 11 API introduces two new shader types into the pipeline; the hull shader and the domain shader. The hull shader is run once per input primitive and the primitive can be a triangle or a quad. From the hull shader, the API expects a tessellation factor per edge and one for the inside of the primitive. These factors decide how many new vertices the tessellation stage should create along each of the edges and the center area. The calculations are performed in a patch-constant function since tessellation

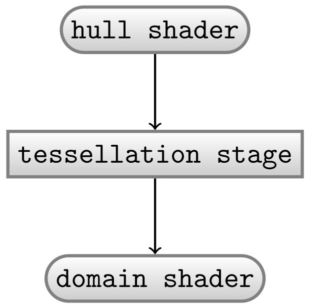


Figure 2.3: Direct X 11 tessellation flow.

factors are constant over the whole patch and the patch-constant function only runs one time per patch.

It is also possible to do surface calculations in the hull shader. This can be done, for example, to approximate subdivision surfaces as described by Loop et. al. [6].

To obtain a view-dependent level of detail for the hardware tessellation in the Frostbite™ 2 terrain engine, the clip space length of an edge is considered. To get this length, a sphere is placed around the mid-point of the edge, covering the edge. This sphere is then projected into clip space and the tessellation factor is calculated to fit a desired number of triangles to this edge. The desired number of triangles is specified in pixel size of the resulting triangles. This maintains a constant screen space size of the triangles meaning that triangles that are far from the viewer and thus small in clip space, are not tessellated as much as closer ones.

After this stage, the triangle size is clamped to a minimum specified horizontal size. The reason for using horizontal size is that the heightfield is horizontal, meaning that there will be only a single heightfield sample for a completely vertical triangle, leaving no need for a high tessellation. There is furthermore no need to tessellate down to smaller triangles than the resolution of the height map.

After the hull shader, the information is fed to the fixed-function tessellator. This is implemented in hardware which makes it significantly faster than a software tessellation approach. As mentioned, it uses the tessellation factors together with a selected type of partitioning. The partitioning types are `fractional_odd`, `fractional_even`, `integer` and `pow2`. `fractional_odd` and `fractional_even` means that the tessellator allows floating point numbers. If `fractional_even` is used, 2.1 is topologically the same as 4, the next even number. However, the two extra vertices will be placed closer and closer to their final positions as the tessellation factor approaches 4. When the tessellation factor goes above four, the topology matches that of

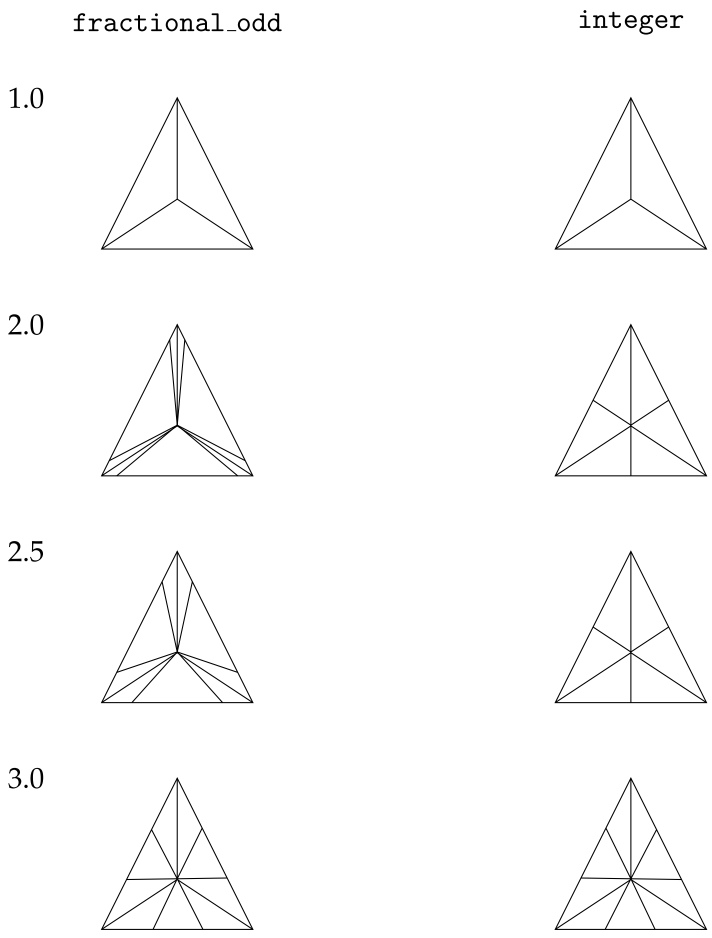


Figure 2.4: Tessellation patterns for fractional odd (left) and integer (right) partitioning.

tessellation factor 6. This is illustrated in figure 2.4.

It can be noted from figure 2.4 that for odd numbers such as 1.0 and 3.0 the `fractional_odd` partitioning is equivalent to integer partitioning. If `fractional_even` would have been used, the fractional partitioning would have matched the integer partitioning at even integers. In all cases in figure 2.4, the inside tessellation factor is 1.0.

2.3.1 Inside Tessellation Factor

The tessellation factors for triangle edges are quite self-explanatory. The factor for the inside of the triangle on the other hand could use some more explanation. If the inside tessellation factor is odd, the inside will consist

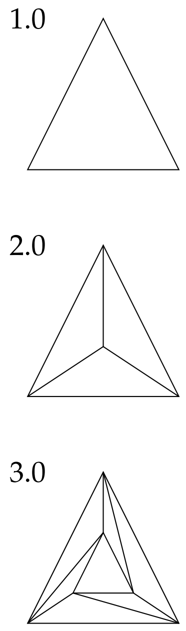


Figure 2.5: Tessellation patterns for the inside of a triangle.

of $\frac{N+1}{2}$ concentric rings for a tessellation factor of N. The innermost ring will in this case be a single triangle. If the tessellation factor for the inside of the triangle is even, the inside will consist of $\frac{N}{2}$ concentric rings for a tessellation factor of N. The inner ring in this case will be a single vertex. The inside tessellation factor for quads are a bit different and perhaps easier to understand. It has two tessellation factors for the inside, one along u and one along v. This will give a regular grid of the size specified by the two tessellation factors. The triangle case is illustrated in figure 2.5.

`integer` partitioning means that the tessellator uses floor to determine the number and the placing of the new vertices. This means that transitions between tessellation levels will not be smooth to the eye. `pow2` tessellation means that the tessellation factor is floored to the closest power of two number, leaving the tessellator with even fewer levels than `integer` partitioning. In the density map algorithm, `fractional_odd` partitioning is used to ensure smooth transitions.

After the tessellation stage, the new vertices are passed to the domain shader which is run once for each newly generated vertex. In the domain shader the vertex is displaced according to heightfield information. To be able to displace vertices in a good way, the heightfield resolution has to be sufficient. This means that an input patch has to correspond to more than one heightfield sample. Otherwise, all newly generated vertices will have the same height, making the tessellation unnecessary. This is the reason for the

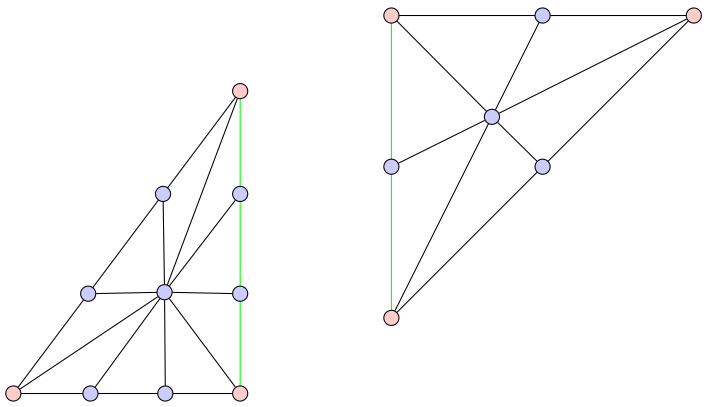


Figure 2.6: Two triangles with different tessellation factors and integer partitioning. The left triangle has all edges set to 3 and inside set to 1. The right triangle has all edges set to 1 and inside set to 1. Vertices added by tessellation are illustrated in blue.

triangle size clamping described above.

2.3.2 Crack-Free Tessellation

The CPU LOD scheme in Frostbite$^{\text{TM}}$ 2 guarantees that the input mesh that is fed to the tessellation stage is always crack-free. The next possible source of cracks is if the tessellation factors for an edge does not match up. This will result in a broken edge since the number of new vertices on the edge are different depending on which side of the edge is considered. The solution to this problem is simple: make sure tessellation factors on both sides of an edge match up.

Consider the two (tessellated) triangles in figure 2.6. If these triangles shared an edge and the vertices was then displaced, there would be a crack in the edge. This is since each patch is treated separately by the tessellation pipeline, meaning that the vertices on the left side triangle will have different heights than the vertices on the right side triangle resulting in the case in figure 2.7 where the edges marked with green (although any edge would give the same result) in figure 2.6 has been displaced.

It is possible to see that figure 2.7 describes the problem with T-vertices. The important conclusion from this is that it is absolutely essential for the tessellation factors on both sides of an edge to match up.

When tessellation is combined with tiling, this means that LOD switches in the input mesh that coincide with tile borders will create cracks if the tiled data is not continuous at the borders. This continuity is achieved in

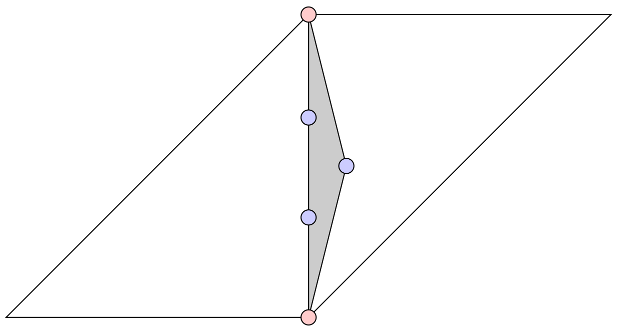


Figure 2.7: The two triangles in figure 2.6 sharing an edge. The resulting crack is illustrated in gray.

the heightfield by using an odd sample border. This odd sample is a one-sample border on the tile, placed on the right and lower edge. This border is not considered in rendering, thus resulting in the border lying "under" the first pixel in the adjacent tile and having the same value. This gives continuous and crack-free data since both the heightfield and the density map are sampled with point sampling.

2.3.3 The Terrain Pipeline

Game data is not used in raw format by Frostbite™ 2. This would work but would be way too slow. To address this, all data is pre-processed into a format that is efficiently readable by the engine and this pre-processing of data is handled by the pipeline stage of the engine. The terrain pipeline is responsible for building terrain assets into an efficient runtime format and has components for building the height field, terrain decals, terrain mesh scattering, etc.

The heightfield part of the terrain pipeline reads raw data, that has been sculpted by artists in the terrain editor, and generates run-time data. The runtime layout of data is discussed above in section 2.2.1.

Chapter 3

Method

This chapter will describe how the density map algorithm was implemented in the Frostbite$^{\text{TM}}$ 2 engine. More specifically, it will describe the resources created by the new algorithm and in detail how the algorithm works.

3.1 The Density Map

Already in the beginning of the project, the decision was made to create a new type of asset, the density map. This map would describe the curvature of the terrain mesh and thus how high the triangle density should be in a specific world space region of the final terrain mesh. The density map is stored in a texture atlas like the heightfield which means that it is, as the heightfield, also divided into smaller streamable tiles.

This density map is bound as a shader resource to the hull shader in the tessellation stage which then can read density information from it and use in a suitable way. This follows the ideas presented by Ian Cantlay in 2008 [2].

3.1.1 Resolution

A heightfield tile consists of 133 samples per side. Out of these, two samples on each side are explicit borders. One extra sample on the lower and right edges of the tiles are also present for continuity. This means that the non overlapping data area is 128 samples. If the density map would have the same resolution as the height field, it would have four samples per input primitive edge. This would be a waste of resources since there should essentially only be one density map value per input primitive edge. This means that the non-overlapping data area of a density map tile only has to contain $128/4 = 32$ samples. With one sample border (on all sides) plus the always needed odd pixel (on the right and lower tile edge) this will result in density map tiles with 35 samples per side.

3.1.2 Bit Depth

The heightfield data is 16 bit unsigned integer. However, the density map does not need that amount of bit resolution. 8 bit unsigned integers are enough to represent the density. This is since the tessellation factors only has 64 distinct levels, and since the density value is only used as a scaling factor, 16 bit resolution is not needed. 256 distinct scaling values are enough for scaling 64 values.

3.2 A First Runtime Implementation

At the first stage, an implementation that was run in the engine itself, was made. The algorithm was run on a tile in the heightfield as it was uploaded to the GPU which means that the algorithm was run on isolated heightfield tiles. In this implementation, the filter consisted of a discrete Laplace filter. The reason that this was implemented was that it was necessary to test the performance impact and also the possibility of a runtime algorithm.

3.2.1 Limitations

The obvious limitation of a runtime implementation is the lack of local information. Each heightfield tile is processed without knowledge of neighbors. This means that it is impossible to enforce continuity and still preserve the correctness of the filter. However, since there is an overlap between heightfield tiles, it is possible to generate continuous and crack-free density map tiles. As long as the filter is consistent the result of the filtering will be the same on both sides of an edge.

The real problem appears when tiles from differing LOD are neighbors. This means that the border of a tile at one level of detail has to match the border of a tile at another level of detail. This can simply not be solved in a good way without neighborhood information. It would certainly be possible to have neighborhood information in runtime but it would probably be slow and would also introduce extra requirements on the streaming system.

...	b	b	b	b	b
...	**b**	a	b	b	b
...	b	b	b	b	b
...	b	b	b	b	b

Table 3.1: Data for a region of the heightfield.

3.3 Pipeline Implementation

Due to the limitations described above, the decision was made to move the implementation to the pipeline stage of the engine. This would allow for more neighborhood information than the runtime implementation and the real-time requirements would be gone, allowing for more sophisticated filtering.

The first step in the pipeline implementation is to obtain the needed amount of height field data to be able to create a density map tile. This is achieved by using the world space coverage of the source heightfield tile and then creating a density map tile with the same coverage. This tile is then expanded in world space to have the necessary neighborhood information (in this case it is 16 samples per side) for creating a continuous density map tile. All heightfield samples are sampled by world space positions, to make sure that the world space alignment is correct. It is also done without taking borders into consideration, only considering non-overlapping data.

3.3.1 Filters

The implementation comes with five different filters that represent different combinations of speed and accuracy. The filters are run once per pixel in the source data and the result is max resampled to currently a fourth of the resolution. That is, each density map sample is the maximum of the filtered value for four heightfield samples. However, the heightfield is sampled at texel centers, which means that the smallest spatial unit in the density map has to be four samples. To accomplish this, one extra pixel overlap is needed for the filters. This is since many of the filters are derivative based and will not catch changes in the heightfield that is only in one dimension. Consider a heightfield sample with a non-zero value a and all others with a significantly smaller value $b << a$, laid out as table 3.1.

This means that the derivatives of the sample labeled with **b** becomes (approximated with central differences and the assumption that distance between samples is 1 meter)

$$h_x = \frac{h_{i+1,j} - h_{i-1,j}}{2\Delta} = \frac{a-b}{2} \tag{3.1}$$

$$h_y = \frac{h_{i,j+1} - h_{i,j-1}}{2\Delta} = \frac{b-b}{2} = 0 \tag{3.2}$$

$$h_{xx} = \frac{h_{i+1,j} - 2h_{i,j} + h_{i-1,j}}{\Delta^2} = \frac{a-2b+b}{1} = a-b \tag{3.3}$$

$$h_{yy} = \frac{h_{i,j+1} - 2h_{i,j} + h_{i,j-1}}{\Delta^2} = \frac{b-2b+b}{1} = 0 \tag{3.4}$$

$$h_{xy} = h_{yx} = \frac{h_{i+1,j+1} - h_{i+1,j-1} - h_{i-1,j+1} + h_{i-1,j-1}}{4\Delta} = \frac{b-b-b+b}{4} = 0 \tag{3.5}$$

where Δ is the distance in meters between the two samples, h is the heightfield function and $h_{i,j}$ is the value for h at position i, j. With these derivatives in place, the expression for the mean curvature becomes

$$\begin{aligned} H &= \frac{h_{xx}(1+h_y^2) - 2h_{xy}h_xh_y + h_{yy}(1+h_x^2)}{2(1+h_x^2+h_y^2)^{3/2}}. \\ &= \frac{(a-b)(1+0) - 2\cdot 0 \cdot \frac{a-b}{2} \cdot 0 + 0 \cdot (1+(\frac{a-b}{2})^2)}{2(1+(\frac{a-b}{2})^2+0^2)^{3/2}} \\ &= \frac{a-b}{2(1+(\frac{a-b}{2})^2)^{3/2}}. \end{aligned} \tag{3.6}$$

With the above expression for mean curvature it can be seen that if $b << a$ the denominator of equation 3.6 will become much larger than the numerator, resulting in a very low curvature where there essentially should be a very high curvature. Say for example that $a = 100$ meters and that $b = 50$ meters. This will give a numerator of 50 and a denominator of 44300, resulting in a curvature of approximately 0.0011 even though the curvature *should* be much larger.

Since the density map is downsampled with respect to the heightfield, this is generally not a problem, due to the fact that the density map is max filtered. However, a problem occurs when this case coincide with an edge in the downsampled 4x4 region of the heightfield. The solution for this is to let the filter run 1 sample into the neighboring 4x4 region.

Laplace Filter

The simplest usable filter is the Laplace filter which is essentially a sum of the second order partial derivatives of the heightfield function as described by equation 2.19. The filter is implemented using discrete central differences and the real strength of it is speed and the major downside is that it is not accurate.

Gaussian Curvature Filter

The Gaussian curvature filter is implemented by estimating first and second order derivatives by central difference schemes. The equation for this filter is described by equation 2.27. The filter is fast and also more accurate than the Laplace filter. However, it is not as accurate as the mean curvature filter.

Mean Curvature Filter

The mean curvature filter is like the Gaussian curvature filter implemented by first estimating the first and second order derivatives with central differences. The equation for calculating the mean curvature is given by equation 2.26. The mean curvature filter has the same speed but better accuracy than the Gaussian curvature filter.

Laplace-Beltrami Filter

All three of the above filters are only applicable to height fields due to the fact that height fields only has one height value for each world space position. This is not true in general for polygonal surfaces.

To provide a more general filtering alternative, the Laplace-Beltrami operator was implemented (see section 2.1.2). This filter has two alternative implementations. One implementation uses the area of the triangles in the one-ring neighborhood and the other implementation uses the Voronoi area of the neighborhood. The benefit of using the Voronoi area is that it gives a better approximation of curvature.

As can be seen in section 2.1.2, the calculation of the Laplace-Beltrami operator involves cotangent computations. This is implemented by dividing the dot product with the length of the cross product of two vectors

$$cot(a,b) = \frac{a \cdot b}{|a \times b|}. \tag{3.7}$$

The calculation of Voronoi area contains more cotangent calculations since it is defined as a sum of cotangents over the one-ring neighborhood. The large amount of cotangent calculations necessary is what makes this filter the slowest of the implemented filters.

Normalization

Since heightfield tiles have different resolution, tiles with lower resolution have a larger spatial distance between sample points. This will result in a kind of artificial smoothness on the terrain. This is not desirable as high curvature terrain in the distance has a high impact on the perceived appearance of the terrain. To compensate for this, the height is normalized by the spatial resolution of the tile so that height differences are scaled down less on tiles with a lower resolution.

The final curvature value of the different filters is also scaled to lie approximately in the range $[0, 1]$. The scaling constants for this step was generated through a series of experiments where a suitable curvature level was selected for a specific heightfield tile.

3.3.2 Preprocessing

The filtering stage of the algorithm is the stage that uses the most of the total execution time of the algorithm. Therefore, the main max filtering is done in a "preprocessing" step. The reason it is called preprocessing step is that it runs before the inner nodes are generated, compensating for artifacts that would otherwise appear in the later stages of the algorithm. The max filter size in this step is controllable by artists and the algorithm also preprocesses borders to avoid aliasing artifacts that come from the fact that borders has to be point-sampled (see below).

The border preprocessing filter is also a max filter and the size of it is controlled by the number of LOD levels to skip and the size of the max filter. Since the resolution of a parent tile is half of the resolution for the children of it, the size for the border preprocessing filter becomes

$$s_b = s_f * 2^{l_{\text{skip}}} \tag{3.8}$$

where s_b is the size of the border filter, s_f is the size of the original filter and l_{skip} is the number of LOD levels to skip. For example if l_{skip} is set to 1, it means that the borders will approximately match borders as they would look at one LOD lower than the highest LOD.

3.3.3 Border Generation

The inner nodes of the quadtree structure are generated by resampling from child nodes. The resampling filter is a max filter, resulting in larger parts with a high curvature as the triangle size increases. This means that aliasing artifacts are avoided. However, the problem with resampling child nodes is that continuity constraints are violated. It would be possible to fix continuity for tiles on the same level of detail but it would be impossible to have continuous switches between LOD levels. This practically means that when the tiles at the highest LOD level has been resampled for continuity, the borders of that tile can not be changed. It furthermore has to be point sampled to generate border data for the inner nodes. The preprocessing of borders described above makes sure that the resulting aliasing artifacts are reduced. So the resulting implementation uses point sampling for the borders and then runs a max filter on the inner area of the nodes.

3.3.4 Parameters

There are five parameters that can be used to customize the behavior of the density map generation algorithm.

Preprocess Density Map

This is a toggle parameter to enable or disable the preprocessing of the density map. This parameter is not meant to be changed apart for debugging purposes. It can be helpful to turn preprocessing of the density map off to see what the algorithm actually picks up in terms of curvature.

Curvature Gain

The parameter called *curvature gain* controls the overall gain of the curvature values. If this value is set to 2 all curvature values will be doubled. This parameter is implemented by simply scaling the final curvature value by the value of this parameter.

Curvature Exponent

This parameter controls the shape of the distribution curve for density map values. The curve is fitted so that incoming density map values with a value of 1 is mapped to outgoing values of 1. This means that a higher curvature exponent will give a larger difference between high and low curvature values. This parameter can be set to a higher value if the level has a few distinct crests and is otherwise flat. This will result in the flat areas being considered low curvature areas and the higher curvature areas will have an even higher curvature value.

Density Map Filter

This parameter controls the filter type to use. The available filter types are described in section 3.3.1.

Density Map Filter Size

This parameter controls the size of the preprocessing filter. The size of the downsampling filter used to create the initial density map is fixed and not affected by this parameter. A larger filter size will result in the curvature "bleeding" out to larger areas. This parameter can be used if the filtering is too exact.

3.4 Hull and Domain Shader

The hull shader is where the created assets are used. The density map atlas is bound as a shader resource and for each control primitive edge fed to the hull shader, the texture is sampled.

To determine the density for an edge, the density map atlas is sampled. The value at that world space point is interpreted as a scale factor for the edge tessellation factor of that edge. A value of 1 means that the calculated tessellation factor should remain unchanged. This, in turn, means that the density map algorithm is a simplification algorithm.

After the density map value has been sampled, a tessellation factor for the edge is calculated by considering the size of the edge in clip space. That is, the edge projected onto the screen. This length is then used to fit a desired number of new vertices to the edge.

Pseudo-code for the algorithm is presented in algorithm 3.1.

Algorithm 3.1 Pseudocode for the shader density map algorithm.

```
density[1] = getEdgeDensity($p_2$.worldPos, $p_3$.worldPos)
density[2] = getEdgeDensity($p_3$.worldPos, $p_1$.worldPos)
density[3] = getEdgeDensity($p_1$.worldPos, $p_2$.worldPos)

screenspaceTessFactor[1] = calcTessellationFactor($p_2$.worldPos, $p_3$.worldPos)
screenspaceTessFactor[2] = calcTessellationFactor($p_3$.worldPos, $p_1$.worldPos)
screenspaceTessFactor[3] = calcTessellationFactor($p_1$.worldPos, $p_2$.worldPos)

edgeTessFactor[1] = max(1.f, screenspaceTessFactor[1] $*$ density[1])
edgeTessFactor[2] = max(1.f, screenspaceTessFactor[2] $*$ density[2])
edgeTessFactor[3] = max(1.f, screenspaceTessFactor[3] $*$ density[3])

insideTessFactor = max(edgeTessFactor[1], edgeTessFactor[2], edgeTessFactor[3])
```

The result of this algorithm is the tessellation factors for each of the triangle edges and an additional tessellation factor for the inside of the triangle. These factors are then passed to the tessellation stage of the tessellation pipeline as described in section 2.3.

3.5 Destruction

The terrain in Frostbite™ 2 also supports terrain destruction, meaning that the shape of the terrain mesh can be affected by different events. Since a crater (or other displacement) in the height field during runtime changes the curvature value for the affected area, the corresponding density map is invalid. To account for this, a corresponding world space rectangle is calculated for the density map and a density is applied based on the size of the terrain displacement.

A scaling factor that is based on experimental results is also applied to the density value to match the range of existing density values. However this value is not in any sense correct but is based on observations that craters need to be tessellated harder than they actually would if they were tessellated based on curvature since the craters are visually important for gameplay.

Chapter 4

Result

The result of the implementation is a stable, reasonably fast and accurate algorithm for estimating curvatures from a heightfield and applying the estimate as a simplification measure for an existing hardware tessellation algorithm.

4.1 Runtime Results

The runtime implementation consisted of a second order derivative filter. This implementation hooked into the terrain heightfield streaming and could run in real time. The result was an algorithm that worked well for isolated LOD levels. Borders between LOD levels could not be accounted for since the necessary neighbor information was not available.

4.2 Pipeline Results

The pipeline implementation is what is currently used in Frostbite™ 2. This implementation can be more sophisticated than the runtime counterpart since it does not have to run in real-time. The algorithm is however reasonably fast for large terrains and the speed of the algorithm is largely dependent on the choice of filter. Usage of the Laplace-Beltrami filters will cause the algorithm execution time to increase by orders of magnitude.

The algorithm accuracy is good and it can be tweaked by artists to achieve desirable results. For example, if a level is relatively flat and only has a few important crest lines, these crest lines can be boosted by either modifying

Num Samples	Second Order Difference	Gaussian Curvature	Mean Curvature	Laplace Beltrami	Laplace Beltrami (No Voronoi Area
1M	0.14s	0.18s	0.31s	1.35s	1.14s
2M	0.27s	0.35s	0.63s	2.69s	2.25s
3M	0.41s	0.54s	0.94s	4.36s	3.57s
4M	0.56s	0.74s	1.28s	5.98s	4.89s
8M	1.13s	1.46s	2.56s	11.04s	9.30s

Table 4.1: Filter speed comparison

the profile shape of the density map values or by applying a total curvature gain over the whole level.

4.2.1 Filter Performance

A comparison between filter speeds using different number of heightfield samples is shown in table 4.1

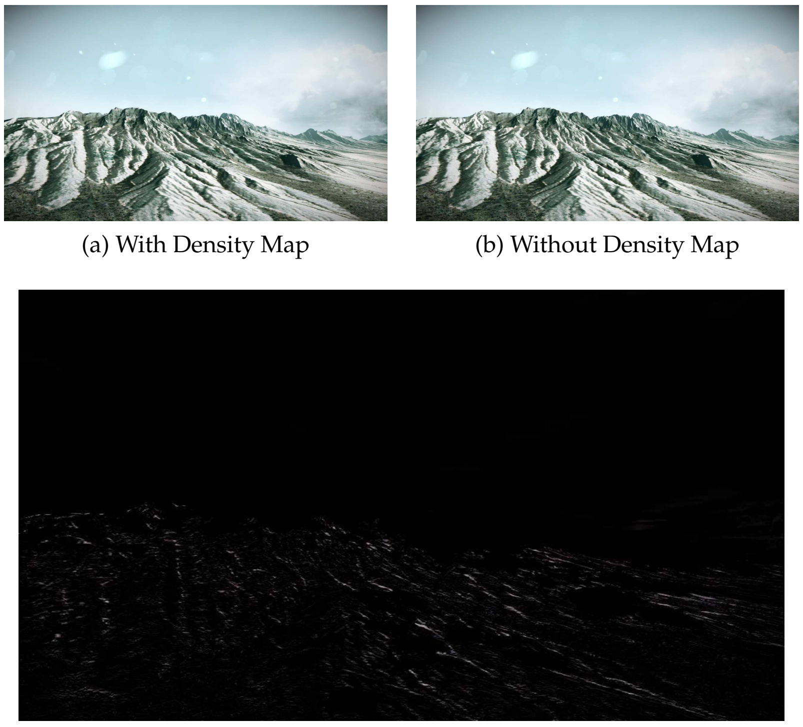
All performance figures was generated on an Intel Xeon X5650 2.67 GHz with 24 Gb of RAM.

4.2.2 Visual Quality

The visual quality of the filter was evaluated by comparing the resulting image from the density map algorithm with the image generated by the previous non-adaptive tessellation algorithm. The comparison was done by considering difference images where one image is subtracted from the other one. All kinds of decimation and simplification algorithms introduce an error and the visual difference is a good measure for this kind of error. The results for the visual quality comparisons can be seen in figure 4.1 - 4.3 where two settings for the terrain rendering was varied.

Triangle width

The first setting varied was the setting for triangle width of tessellated triangles. This setting controls how wide (in pixels) the generated tessellated triangles should be. The expectation was that the density map algorithm would be significantly faster, allowing for lower triangle widths with equal


(a) With Density Map

(b) Without Density Map

(c) Difference

Figure 4.1: Difference image for a triangle size of 12 pixels and 4 patch faces per side.

performance. This would arguably give better visual performance with maintained execution time for the tessellation algorithm.

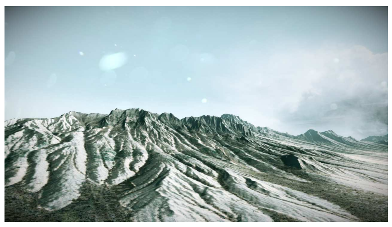
Patch Faces Per Side

This setting controls how high the resolution in the CPU controlled terrain mesh should be. Each patch consists of a number of triangles and this parameter controls how many faces there should be on each side of the patch. A higher number will give a better looking terrain but will also leave less work for the tessellation algorithm since the triangles that are fed to the tessellation shaders are already small.

4.2.3 Visual Stability

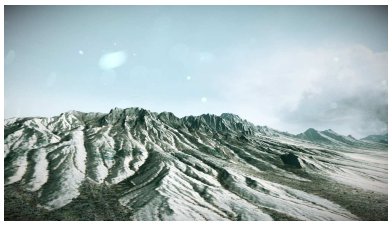
In many cases, when moving farther away or closer to the terrain, popping artifacts occur. However, these popping artifacts can be reduced by

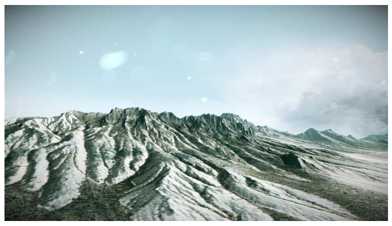
(a) With Density Map


(b) Without Density Map

(c) Difference

Figure 4.2: Difference image for a triangle size of 6 pixels and 4 patch faces per side.


(a) With Density Map


(b) Without Density Map

(c) Difference

Figure 4.3: Difference image for a triangle size of 6 pixels and 8 patch faces per side. This is the recommended setting for using the density map algorithm.

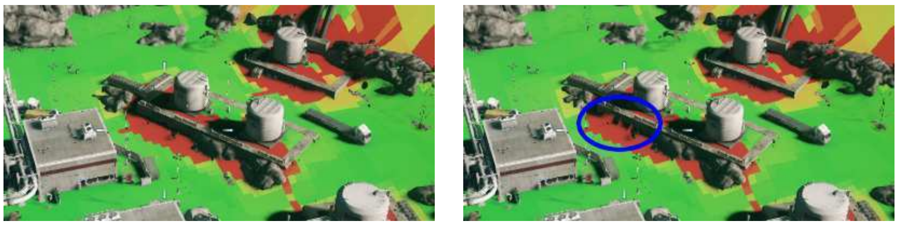

(a) With Density Map (b) Without Density Map

Figure 4.4: Static mesh inserted into the terrain. Left side shows the result with density map and right side without the density map. The terrain is colored with density map colors to make artifacts easier to see.

decreasing the triangle width either for the tessellation or for the CPU LOD algorithm. With the density map, the triangle width for the hardware tessellation can be lowered without losing performance. Also considering that the density map is max filtered on lower LOD, the terrain appearance from a distance will be even more stable.

The need for stable terrain appearance under varying view distance is large in places where static meshes like buildings are placed in the terrain. Popping in these areas can create visually disturbing artifacts like walls being intersected by the terrain mesh. An example of this case can be seen in figure 4.4.

It can be seen that the fact that the density map approach can use smaller triangles in tessellation, will give a better appearance.

4.2.4 Runtime Performance

The runtime performance of the density map algorithm was evaluated by a series of experiments where different parameters for the terrain rendering were varied and compared against the results for the earlier brute-force tessellation algorithm. The experiments was also performed to find a set of parameters that produced the best visuals combined with the best performance (the so called “sweet spot”).

The runtime performance was then measured in terms of the time spent for filling the G-Buffer used in the deferred rendering algorithm. This number is more reliable than for example measuring the frame rate. The G-Buffer measure is also an averaged measure over a number of frames and not an instantaneous number.

A series of experiments were carried out, first varying the triangle width to see the effect that the number of patch faces per side would have on the

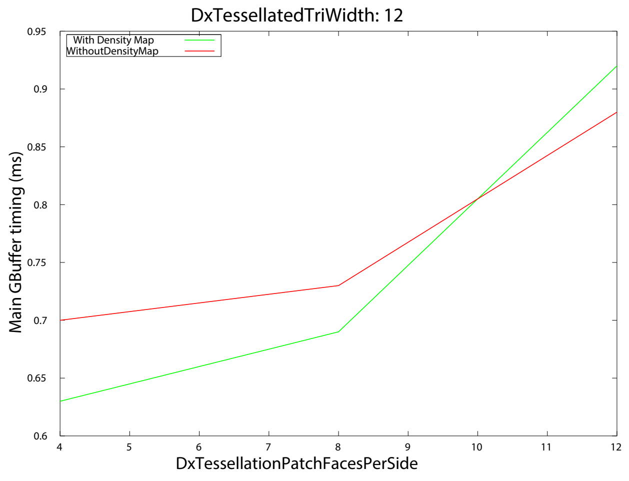


Figure 4.5: Patch faces per side varied for a triangle width of 12 pixels.

performance.

From figures 4.5 - 4.8, it can be seen that once the number of patch faces per side goes over 8, the performance drastically decreases. This is especially true for larger triangle sizes. It should be added that a triangle size of 4 pixels is not very likely to be usable and is included for completeness.

In the next series of experiments, the triangle width was varied for different number of patch faces per side (see figures 4.9 - 4.11).

From this series of experiments it can be seen that 6 is the lowest practical triangle width that can be used and it can also be seen that the density map algorithm manages to keep the times down longer than the original non-adaptive algorithm. This is the expected results for the algorithm.

The impact that the density map algorithm has on the number of generated triangles is illustrated in figure 4.12.

It should be noted that figure 4.12a is wireframe and figure 4.12 shows that the number of triangles is reduced significantly when considering the overall frame.

4.2.5 Vertex Count

The number of vertices processed by the domain shader was measured with AMD^{TM} GPU Perf Studio. This was done for two different views (figures 4.13a and 4.13b) and the results are presented in table 4.2.

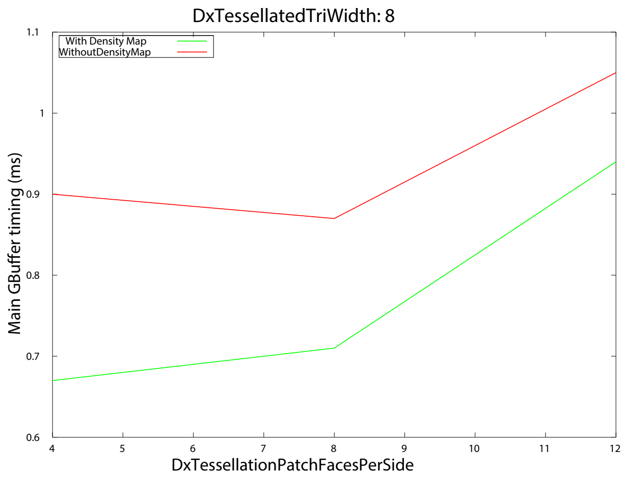


Figure 4.6: Patch faces per side varied for a triangle width of 8 pixels.

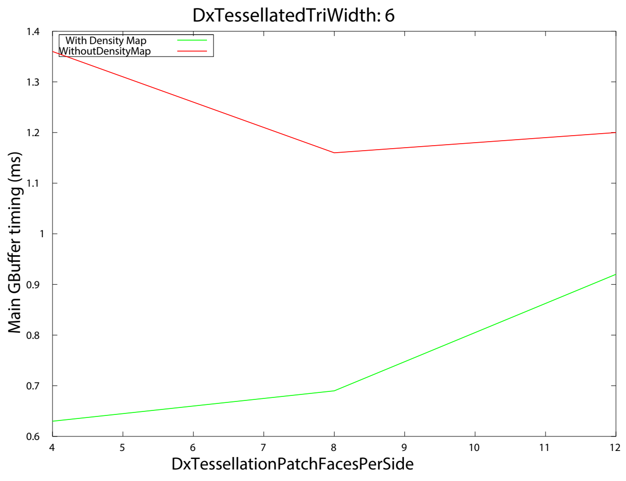


Figure 4.7: Patch faces per side varied for a triangle width of 6 pixels.

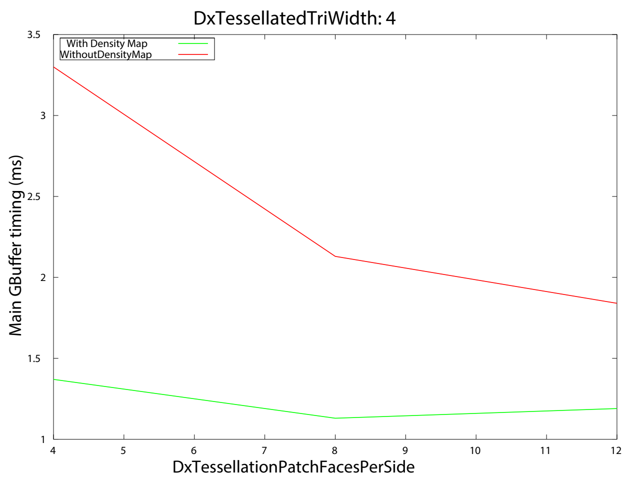


Figure 4.8: Patch faces per side varied for a triangle width of 4 pixels.

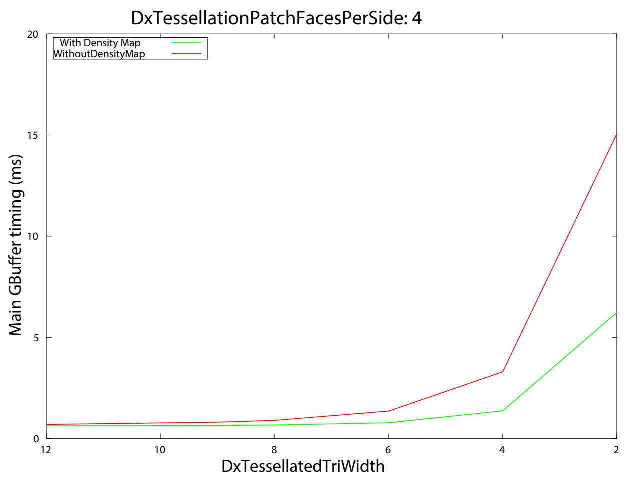


Figure 4.9: Triangle width varied with the number of patch faces per side fixed at 4.

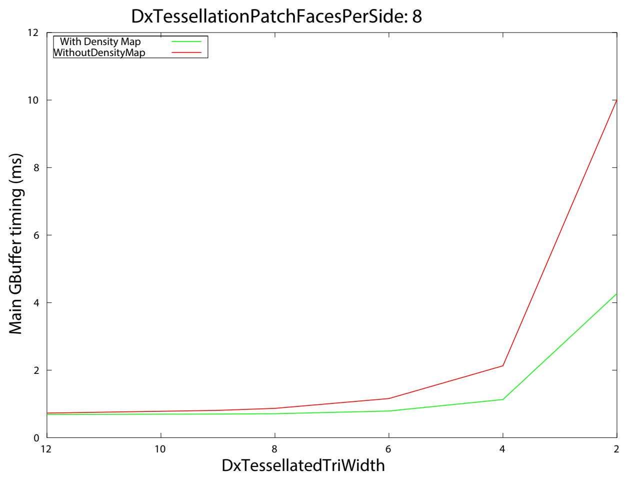


Figure 4.10: Triangle width varied with the number of patch faces per side fixed at 8.

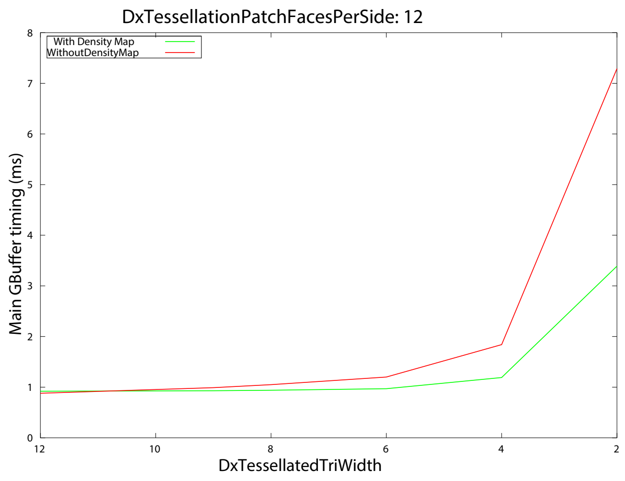
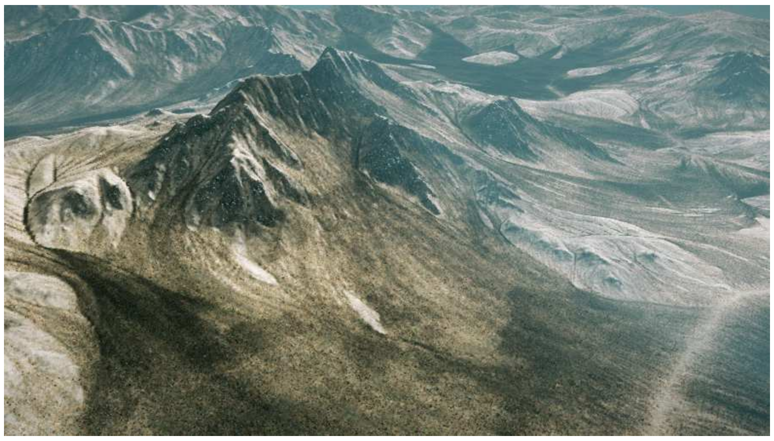
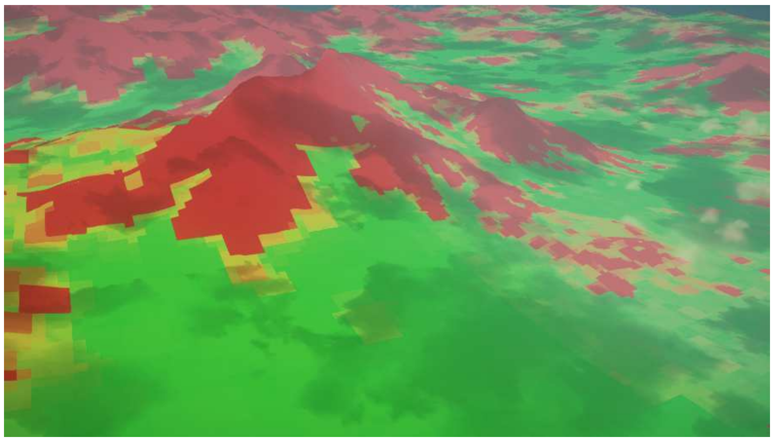


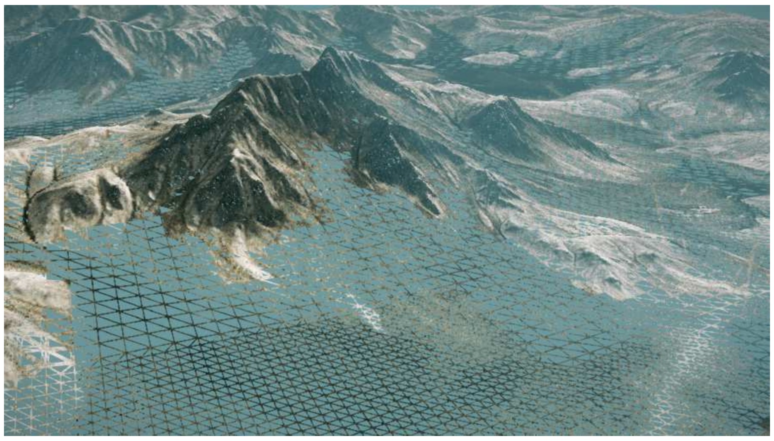
Figure 4.11: Triangle width varied with the number of patch faces per side fixed at 12.



(a) Wireframe terrain mesh without the use of a density map.

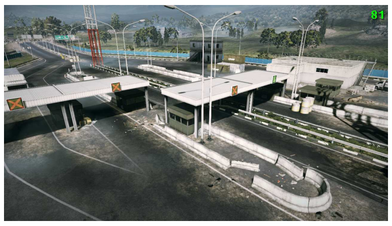


(b) The density map for the region.



(c) Wireframe terrain mesh with the use of a density map.

Figure 4.12: Comparison between the wireframe terrain mesh without the density map and with the density map. The density map for the region is also shown. A red density map color means high curvature and green means low curvature.


(a) Scene 1.

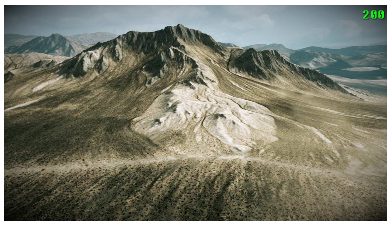

(b) Scene 2.

Figure 4.13: The two scenes used for measuring vertex count. Scene 1 represents a common scene for action and scene 2 represents a terrain view.

Scene	With Density Map	Without Density Map
figure 4.13a	249 222	667 060
figure 4.13b	209 485	511 723

Table 4.2: Number of vertices processed by the domain shader for the views in figure 4.13a and 4.13b.

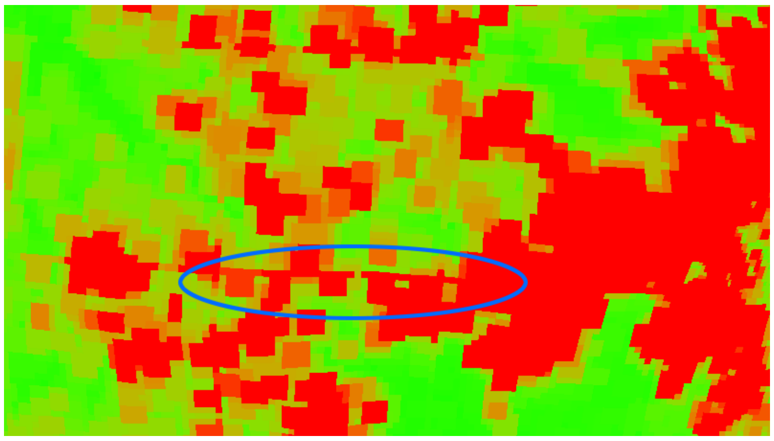
4.2.6 Border Preprocessing

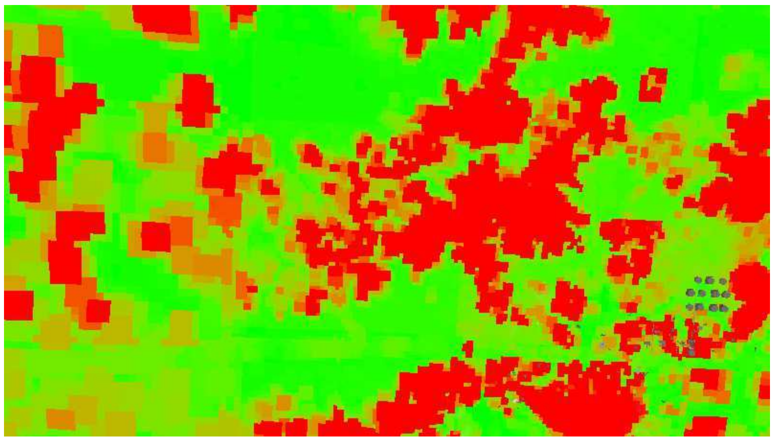
As described in section 3.3.3, borders are preprocessed to match borders at a given LOD. This means that the borders will be correct for this LOD level and aliasing artifacts will be visible on all other levels. If this LOD level is chosen carefully, the border aliasing will be minimized. An example of borders at different LOD are presented in figure 4.14

It can be seen from figure 4.14 that the border for the second highest LOD (figure 4.14b) does not have aliasing artifacts. The other two levels has aliasing artifacts due to the point sampling used when generating inner nodes in the density map quadtree. Aliasing artifacts are annotated with a blue ellipsis.

4.2.7 Workflow Results

The density map algorithm introduces no new workflows that has to be considered. However, there are a few parts where manual tweaking of the density map can be done. The algorithm adds a few parameters to the terrain heightfield asset as described in section 3.3.4 but the workflow is otherwise automatic and no special care has to be taken to use the algorithm.


(a) Highest LOD.


(b) Second highest LOD.

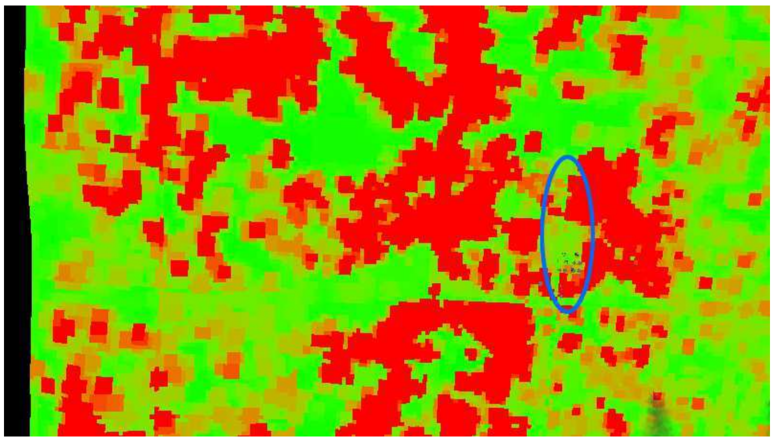

(c) Two levels from highest LOD.

Figure 4.14: Example results from the border generation algorithm. Figure 4.14b shows the second highest LOD which is selected to be correct. Smaller aliasing artifacts can be seen on other levels.

Chapter 5

Discussion

The implementation works satisfactory even though the algorithm nature is simple. This is essentially good since a simple implementation is easier to maintain and understand.

A large part of the actual implementation work was devoted to the five implemented filters that are the heart of the density map algorithm. The range of complexity for the filters is quite large, mathematically. The simplest filter is the Laplacian filter that produces acceptable results and is very fast. However, the mean curvature filter is almost as fast and produces much more accurate results. On the other end, the Laplace-Beltrami filters are very slow but also has very good precision. The default filter in the implementation is the mean curvature filter.

Much of the implementation work was also spent in figuring out suitable schemes for borders between different LOD levels. The conclusion was made early that these cases needed special consideration. The problem is described earlier and I will not go into detail here but the problem is that the borders must be point-sampled when creating inner nodes in the quadtree. This is due to the fact that the borders on each level has to match the world space borders for neighboring tiles on all other levels. Different approaches was discussed to solve this but most of them would suffer from aliasing artifacts. A compromise scheme was selected since it would give the most accurate results for as many levels as possible. This scheme preprocesses the borders to match a certain LOD level, leaving aliasing artifacts in other levels. However, if care is taken, choosing a LOD level that the border should match, the aliasing artifacts will be minimized.

There are more sophisticated schemes for solving these type of problems, but the simplicity of the implemented solution wins in this case since it works in practice and is easy to understand.

5.1 Runtime Implementation

The intention was never that the runtime implementation would be usable and it was made to get familiar with the terrain system in Frostbite™ 2. This was partly due to the performance penalties that this kind of implementation would introduce and partly due to the lack of spatial neighborhood information. The latter issue was the issue that manifested itself first and also the runtime performance of the algorithm was not very bad since a simple filter was used. However, when trying to make borders match up, it soon became apparent that this was a problem that could not be solved in runtime. The streaming nature of the terrain engine made it hard to access the needed information without imposing new constraints on the system.

5.2 Pipeline Implementation

The decision to move the implementation to the engine pipeline was made quite early due to the limitations described. It was also the initial idea that this would be the case and that a pipeline implementation would be the most scalable and a better solution in the long term.

The whole solution to the problem also became more obvious when all information was available in a structured way. The extra execution time allowed for a pipeline implementation also made the use of more sophisticated techniques possible.

Interesting to note is also that the more sophisticated filters like the Laplace-Beltrami operator is much slower than the finite difference Gaussian and mean curvature filters. However, the lack of speed is not quite matched by the accuracy of the Laplace-Beltrami filters. The advantage of these filters is of course that they are usable even for general surfaces and not only for heightfield based surfaces.

5.3 Future Improvements

This section will list a couple of future improvements identified during the implementation. Note that these suggestions are my own suggestions and does not in any way reflect any implementation plans for Frostbite™ 2.

5.3.1 CPU Implementation for Consoles

Playstation™ 3 and Xbox 360™ does not have hardware tessellation[1] like Direct X 11 so instead they use only the CPU algorithm but with a more detailed mesh. This algorithm essentially suffers from the same overtessellation issues as the hardware tessellation algorithm since it is only based on view distance. The same density map resource as for the hardware tessellation algorithm can be used as input to the mesh generation algorithm. Possibly, the terrain mesh generation algorithm could be adapted into something similar to CDLOD described in section 1.1.4.

5.3.2 Terrain Improvements

To achieve maximum visual stability, it would be desirable that the heightfield triangulation would use some kind of vertex morphing to ensure smooth transitions between LOD levels. This would reduce popping artifacts due to LOD switches in the input terrain mesh. An approach similar to the CDLOD algorithm described in section 1.1.4 could possibly be implemented to morph vertices between different LOD levels.

5.3.3 GPGPU

Since the filtering of the heightfield data contains parts that are data-parallel, the mapping to SIMD architectures would certainly be possible. This means that CUDA/OpenCL/Direct Compute can be used to utilize the parallell nature of GPU:s. This could result in large speedups which in turn means that more sophisticated filtering methods like the Laplace-Beltrami operators could be used. There are benefits however of a strictly CPU based approach since it does not impose any hardware requirements.

5.3.4 Other Uses

As described in section 1.2.1, terrain rendering with height fields is a case of displacement mapping. Displacement mapping for characters is often used as an efficient way to achieve highly detailed characters.

Displacement mapping is often used together with subdivision surfaces and hardware tessellation. There are different approaches for subdividing

[1]Xbox 360 actually has a simpler form of hardware tessellation but it is not used in the Frostbite™ 2 terrain engine.

the surface. It is for example possible to approximate Catmull-Clark subdivision surfaces [6] in different ways. This allows for smooth surfaces created from coarser, more storage efficient, control meshes. The subdivided mesh is then hardware tessellated and displaced. A density map based approach would certainly be plausible to use in this case. The displacement map (most commonly a vector displacement map) can be preprocessed to create an accompanying density map. This density map would make sure that the parts that had little or no displacement described in the displacement map, would not be overtessellated. The problem with this approach however is that a simplification algorithm could break the appearance of the subdivision surface.

5.4 Other Reflections

The first thing I realized rather early in the project, is that adaptive hardware terrain tessellation is something that has not been done to great extent before. There is a lot of methods for the CPU side of terrain systems. However, the introduction of hardware tessellation in consumer graphics cards has to be seen as a rather new concept and that might be the reason for the lack of research in this field.

What is striking though, is that a simple algorithm like this implementation can have a rather large impact on performance and thus using the GPU where it is actually needed instead of wasting precious GPU cycles on flat, uninteresting parts of the terrain. As mentioned above it is also possible to use the density map as an input parameter to the CPU based LOD algorithm, allowing for better optimization of the input mesh that is fed to the tessellation shaders. This would result in two stages using the density map as input.

I feel that the implementation turned out good since it is actually usable without any intervention from artists. This means that the process is more or less automatic but can still be controlled if desirable. The performance results is also satisfying since it is possible to use much higher detail for the parts that needs it. It is possible to argue that the introduction of the density map has made the runtime performance less predictable since it will depend on how much high-density areas that are in view, but since it is a simplification algorithm, the performance can never be worse than the non-adaptive algorithm.

One thing that I think turned out well is the generation of inner (lower LOD level) nodes. The compromise between the aliasing on the borders and the correct maximum filtering for the inner area of the node turned out to works well in practice. The scheme is also simple which means that it is easy to maintain and test.

To sum things up, I am very happy with the way the algorithm turned out. It is reasonably fast and produces good results without tweaking. I also think that the algorithm is easy to understand and should therefore be easy to maintain and modify.

Index

Bibliography

[1] Johan Andersson. Terrain rendering in frostbite using procedural shader splatting. Siggraph Presentation `http://publications.dice.se/attachments/Chapter5-Andersson-Terrain_Rendering_in_Frostbite.pdf` (2012-05-31), 2007.

[2] Iain Cantlay. Adaptive terrain tessellation on the gpu. Siggraph presentation `http://www.nvidia.com/object/siggraph-2008-terrain.html` (2012-05-29), 2008.

[3] Jonh Carmack. Quakecon presentation. `http://us.generation-nt.com/john-carmack-quakecon-2005-keynote-complete-transcript-help-29744482.html` (2010-05-29), 2005.

[4] Manfredo P. Do-Carmo. *Differential Geometry of Curves and Surfaces*. Prentice Hall, first edition, February 1976.

[5] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. Roaming terrain: real-time optimally adapting meshes. In *Proceedings of the 8th conference on Visualization '97*, VIS '97, pages 81–88, Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[6] Charles Loop, Scott Schaefer, Tianyun Ni, and Ignacio Castaño. Approximating subdivision surfaces with gregory patches for hardware tessellation. *ACM Trans. Graph.*, 28(5):151:1–151:9, December 2009.

[7] Filip Strugar. Continuous distance-dependent level of detail for rendering heightmaps. *journal of graphics, gpu, and game tools*, 14(4):57–74, 2009.

[8] Gabriel Taubin. A signal processing approach to fair surface design. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '95, pages 351–358, New York, NY, USA, 1995. ACM.

[9] Thatcher Ulrich. Chunked lod: Rendering massive terrains using chunked level of detail control, 2002. Course at SIGGRAPH 02, `http://tulrich.com/geekstuff/sig-notes.pdf` (2012-05-29).

[10] Alex Vlachos and Jörg Peters. Curved pn triangles. `http://alex.vlachos.com/graphics/CurvedPNTriangles.pdf` (2012-05-30), 2001.

[11] Mattias Widmark. Terrain in battlefield 3: A modern, complete and scalable system. GDC Presentation `http://publications.dice.se/attachments/GDC12_Terrain_in_Battlefield3.pdf` (2012-05-31), 2012.