Adaptive GPU-based terrain rendering

Adaptives GPU basiertes Terrain Rendering

Diploma thesis

Timo Schmiade

Lehrstuhl für Computergraphik und Multimediasysteme
Fachbereich 12 Elektrotechnik und Informatik
Universität Siegen

Erstgutachter: Prof. Dr. Andreas Kolb
Zweitgutachter: Dr.-Ing. Christof Rezk Salama
Betreuer: Dipl. Math. Martin Lambers

23rd May 2008

Eidesstaatliche Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Ich bin mir bewusst, dass eine unwahre Erklärung rechtliche Folgen haben kann.

Ort, Datum Vorname, Name

Abstract

The diploma thesis at hand presents a survey on the development of terrain rendering algorithms during the past ten years and examines to which extent Shader model 4 can contribute to terrain rendering algorithms. Therefore, a new approach based on the features of Shader model 4 named RAG is introduced. RAG is a purely GPU-based terrain rendering algorithm which refines a coarse input mesh by recursively employing the new geometry shader stage introduced with Shader model 4. Each triangle is directly tesselated within the rendering pipeline, depending on how well it approximates the given height map texture. RAG provides fast CPU-independent terrain refinement, but guarantees to the mesh's accuracy can only be given to a certain extent. This is due to the purely edge-based refinement decisions which are applied to each triangle and guarantee a consistent mesh topology.

Zusammenfassung

Die vorliegende Diplomarbeit veranschaulicht die Entwicklung von Terrain Rendering Algorithmen innerhalb der vergangenen zehn Jahre und untersucht inwiefern Shader Modell 4 für das Terrain Rendering verwendet werden kann. Dazu wird ein neuer, auf den Möglichkeiten von Shader Modell 4 basierender Algorithmus names RAG vorgestellt. RAG ist ein vollständig GPU-basierter Algorithmus, der ein grobes Mesh rekursiv verfeinert, indem er die mit Shader Modell 4 eingeführte Geometry Shader Stage ausnutzt. Jedes Dreieck wird direkt innerhalb der Rendering Pipeline abhängig davon, wie gut es die gegebene Höhentextur bereits approximiert, tesseliert. RAG stellt eine schnelle und CPU unabhängige Terrain Verfeinerungstechnik dar, kann aber nur begrenzt Garantien über die Exaktheit der Höhentextur-Approximation geben. Dies liegt daran, dass rein kantenbasierte Verfeinerungsentscheidungen getroffen werden müssen, um eine konsistente Meshtopologie zu garantieren.

Contents

1 Introduction

Since the early 90's, terrain rendering has been one of the intensely developed subjects in the domain of computer graphics. New algorithms to performantly display digital landscapes have emerged with every new generation of hardware in general and graphic cards in particular.

The desire to display more and more detailed landscapes is the central reason that this topic is far from being ultimately explored. The fields of application are numerous and versatile. For example, an exactly rendered surface can help a pilot to safely land an airplane in a situation of poor visibility. Landscape measurements can be conducted at the computer rather than on location. Finally, virtual reality projects and video games are able to arouse a higher state of immersion for the user.

But with improved presentation, higher demands for computer develop. More detail means more complexity for the processor, the graphic card and the system memory. Each new generation of hardware increases the performance and functional range of these components, but it takes more than mere hardware improvements to compensate today's fast-growing data sizes and increasing demands to detail and accuracy. Therefore, algorithms that exhaust the full potential of every available computer component are required to fulfil a user's demands.

During the past ten years, many terrain rendering technologies have been developed and tested. Some of them are outdated in the meantime, some others are still used and improved. To provide an overview of the research that has been done so far, I present some of the most important algorithms in the following section 2 of this work. The first two presented approaches in that chapter are not GPU-based, but are yet important to illustrate the development of terrain rendering during the past years.

As implied by the title, this thesis deals with the display of terrain using GPU features, which are introduced along with Shader models. Section 3 reviews what such a Shader model is, and which features are supported by the newest Shader model with the version number 4 in comparison to its predecessor.

Section 4 begins by discussing the problems of adapting a classical CPU-based terrain rendering algorithm to a GPU implementation. This reveals some of the limitations of GPU-based approaches. Following this, a new algorithm, based on Shader model 4, named Recursive Adaptive Geometry Shader (RAG) is developed and its workflow is described in detail. The main objectives of RAG are to be fast, accurate and efficient and to avoid the waste of resources.

This approach transports almost all work traditionally conducted by the

CPU to the GPU, exploiting the new geometry shader stage. Following a discussion of problems caused by the new approach, I will compare RAG to an algorithm that has been developed in parallel to it in the last part section 4.

The following section 5 first details my implementation of RAG. All important aspects will be covered to clarify how an implementation can look like and which points are to be considered. Following this, performance measurements recorded with this implementation are presented.

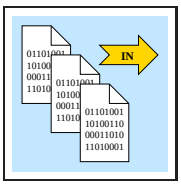
An evaluation of this thesis follows in the final section 6. I will discuss the extent to which my set goals have been reached and which improvements can be pursued in future research.

2 Terrain rendering algorithms

This section presents a selection of terrain rendering algorithms from the past 15 years. Summarising all the algorithms that have been developed during that time is by far beyond the scope of this work, however the selected algorithms presented here give a good overview of the technological history of terrain rendering.

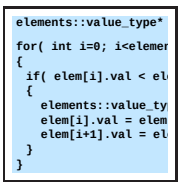
Each algorithm has very different characteristics. For instance, the first algorithm (ROAM) has no support for the features of programmable graphic hardware, due to such hardware not being available at the time of its development. On the other side, Geometric Clipmaps makes vast use of those possibilites, to the degree that the processor is hardly stressed at all.

Bearing this in mind, a comparative evaluation of the presented approaches is very difficult. Nevertheless, my summary of each algorithm will attempt to rate it according to the following criteria:



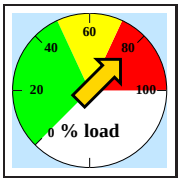
Input data

Older algorithms are often based on a fine triangle mesh which is then reduced for better display performance, while new approaches mostly take a texture with encoded height values as an input.



Programming considerations

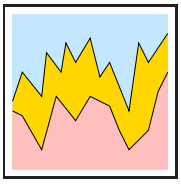
Each algorithm – be it new or old – bears certain challenges, either for its implementation or for its runtime stability. The ones with the largest impact will be enlistet here.



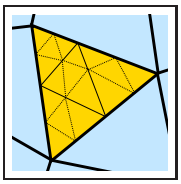
CPU / GPU load

This criterion describes how stressed both the computer's processor and graphic card are while the respective algorithm runs.

RAM usage / bus transfer



Under this point, the amount of system RAM needed as well as the data transfer from system RAM to the graphic card will be subsumed. Especially the latter one is crucial for a fast working algorithm, as the more data that needs to be transferred, the longer the graphic card has to wait before it can render a new image.



Accuracy of the rendered terrain

Finally, the quality of the resulting mesh is examined here. For many applications it is satisfactory to have a nice looking terrain. Nevertheless will I specify the peculiarities of each resulting mesh here.

As will be demonstrated at the end of this section, none of the detailed approaches is perfect and each is better suited for certain fields of application and worse for others.

2.1 Real-time Optimally Adapting Meshes (ROAM)

The first ROAM approach – which has evoked many pursuing and improving algorithms – was introduced by M. Duchaineau et al. in 1997 [9]. ROAM optimises the display of terrain by rendering regions of little local detail (flat regions), and regions far away from the point of view, with less polygons. Thus, the overall number of polygons is reduced to maintain high rendering performance while the error that is introduced is being kept as small as possible.

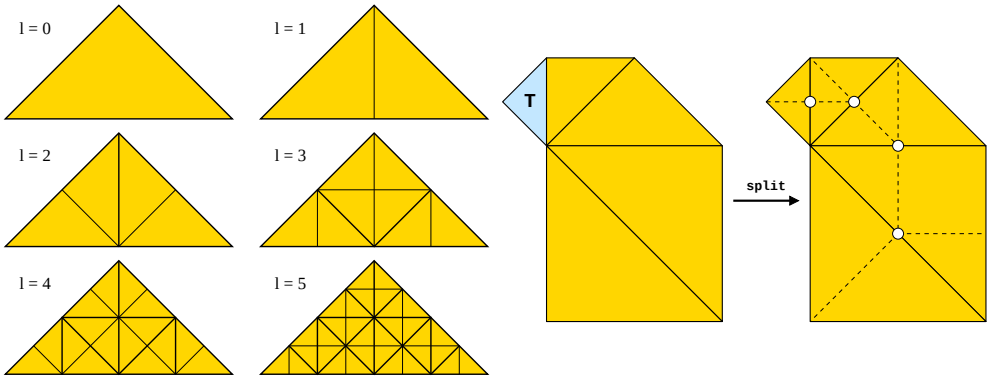
Preprocessing

The declared input for the algorithm is a pre-defined multiresolution terrain representation. The algorithm however, operates on a triangle binary tree (bintree) structure, which needs to be generated beforehand.

The root triangle of this triangle bintree is defined right-isosceles. Each finer level of subdivision is recursively generated by splitting its respective root along an edge formed from its apex vertex to the midpoint of its base egde. Thus, all triangles created become right-isosceles as well. Figure 1.a shows the first six levels of the triangle bintree described.

A concrete mesh in world space is created by assigning world-space positions from the input multiresolution terrain to each bintree vertex throughout all recursion levels. The input data can be discarded afterwards.

After the triangle bintree has been created, the next step of the preprocessing is to calculate a view-independent error value for each triangle in the finest bintree level. Then, error bounds for chunks of triangles are recursively evaluated by traversing the bintree bottom-up. These error bounds will be used later on to determine the necessity for each triangle to gain more detail.



(1.a) The first 6 recursion levels of a triangle bintree.

(1.b) A split-operation throughout four bintree levels induced by triangle T.

Frame-wise updates

The authors of the paper emphasise a key fact about bintree triangulations, that assures the mesh's continuity during split- and merge-operations: The neighbours of a given triangle T are always either from the same, the next or the previous subdivision level, depending on the edge on which they overlap with T.

Due to this characteristic, split- and merge-operations will not break the continuity of the mesh as long as they are done recursively for neighbouring triangles in the respective cases. Figure 1.b shows an example in which a split-operation is done recursively throughout four levels of the triangle bintree.

Each frame, the algorithm iteratively optimises the triangle mesh from the previous frame according to the viewer's position and orientation. This is done by two priority queues, one for splitting triangles to increase detail and another one for merging them to reduce their quantity. Both queues must be updated with each frame adjusting to the viewer's movement.

To determine the priority of each triangle, a wide range of metrics can be applied, predominantly it will be an error metric like the visible screen area, the local curvature or the face orientation of a triangle. The authors, however, have chosen the maximum geometric distortion in screen-space as a base metric for the two queues. To make the calculation of the screen-space error as efficient as possible, the authors use the precalculated view-independant error bounds from the preprocessing and convert them to screen-space errors during priority calculations.

The only requirement for the queues' priorities is that they are strictly monotonic, meaning that a given triangle always has a smaller priority to be split than its parent triangle. In the author's case, this positions the triangle

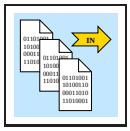
with the highest screen-space distortion on the top of the split-list after each update.

Beginning with this highest priority triangle, the splitting queue adds detail to the mesh by splitting triangles and avoids cracks by recursively splitting neighbouring triangles. Each step reduces the maximum priority of all triangles and – at the same time – the maximum error of the whole mesh. This is due to the triangle with the highest priority being the one that delivers most detail gain.

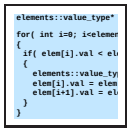
As already mentioned, all mergeable triangle pairs are kept in a second queue, which is processed to remove detail from the mesh if it is too accurate or too large. In such cases, the triangle pair with the lowest priority is merged, thus reducing the number of polygons in the mesh bottom-up. In the author's case, this processing merges the triangles with the lowest screen-space error to ensure that the performance does not suffer from too many triangles being handled.

The frame-wise update is aborted whenever one of the following occurs: The available frame-time expires, the mesh has gained an acceptable detail, the desired number of triangles has been reached or the highest screen-space error in the bintree has been reduced sufficiently. Finally, the calculated mesh is rendered.

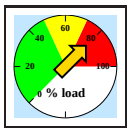
Summary



- Pre-defined multiresolution terrain representation

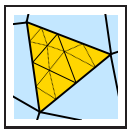


- No special efforts needed to preserve continuous triangulations and prevent "thin" triangles
- Algorithm expects priorities to change smoothly from frame to frame, quick turnarounds or similar actions are not supported



- Preprocessing step needed to create initial mesh
- High CPU load during runtime, GPU not programmed

- The triangle bintree must reside in memory
- A complete mesh is transferred to the graphic card each frame



- Algorithm produces guaranteed bounds on geometric screen-space distortions when used with the respective error metric
- The number of triangles in the mesh can be directly chosen

2.2 Batched Dynamic Adaptive Meshes (BDAM)

The basic goals of the BDAM algorithm are to relieve the processor from the complicated mechanics of manipulating an existing mesh at runtime and avoiding unnecessary transfers of data to the graphic card. Cignoni, Ganovelli, Gobbetti, Marton, Ponchio and Scopigno introduced this algorithm in 2003 [3], writing follow-up papers in the same year and in 2006.

In contradiction to the previously presented ROAM algorithm, the smallest entity that is transferred to the graphic card is not a single triangle, but a small surface patch. Transferring these small patches is considered more effective by the authors than transferring single triangles or complete recalculated meshes.

The algorithm claims to combine the best characteristics of Triangulated Irregular Networks (TINs – with Networks meaning meshes in the domain of computer graphics) and meshes based on right triangle hierarchies. Instead of iteratively adjusting an existing mesh from frame to frame like ROAM, BDAM precalculates right-isosceles triangle shaped TINs (in the following referred to as triangular patches) in different levels of detail and assembles them to a complete representation of the terrain during runtime.

Preprocessing

The preprocessing consists of three parts, each of which constructs a hierarchy that is needed during the runtime of the algorithm. The three hierarchies are:

1. A binary tree, whose nodes are triangular patches
2. A texture quadtree
3. A binary tree of bounding volumes for the triangular patches

The most complex construction algorithm is the one for the binary tree of triangular patches. In the paper, the authors build the tree through a simplification rather than a refinement process. However, they note that

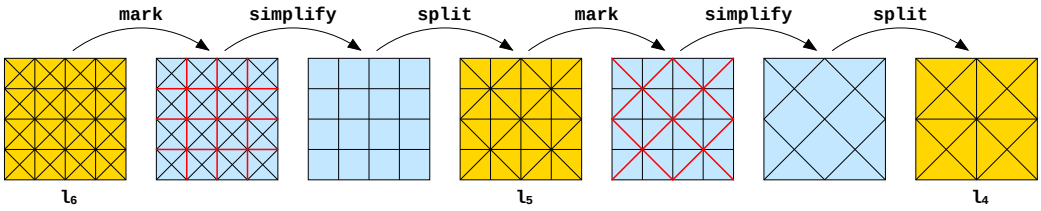


Figure 2: The iterative simplification illustrated for the refinement levels l_6 to l_4. Edges containing unmodifiable vertices are marked (red), generating square-shaped sub-meshes. These are then simplified and split afterwards, generating the triangular patches for the next level. Intermediate states are shown in blue.

their algorithm can be replaced by an equivalent one which suits the required premises and produces the structure as a refinement. The iterative simplification process is illustrated in figure 2.

Starting from the highest detail refinement level, a triangular patch is created for each node of the bintree. The algorithm must make sure that no vertices lying on the contour of a triangular patch are modified. Otherwise, this would introduce cracks when two triangular patches are placed next to each other. The authors use a modified version of quadric error metrics [11] to create the triangular patches from the input dataset.

Based on the existing finest bintree level, each coarser level l_{i+1} is recursively built bottom-up from level l_i. First, all vertices lying on the hypotenuse of each triangular patch's contour are marked as unmodifiable, so that they persist in level l_{i+1}. This proceeding avoids cracks in the mesh when triangular patches from different bintree levels are placed next to each other. The marking creates a set of square-shaped sub-meshes, each of which cover the area of four triangular patches from level l_i.

The square-shaped sub-meshes are then simplified in the same manner as the triangular patches from the highest detail refinement level. Additionally to the vertices lying on the contour of the sub-meshes, the ones located on a diagonal now need to be locked as well, so that the sub-mesh can be split into two triangles during the next step without introducing cracks. These very triangles represent the triangular patches for level l_{i+1}.

The second preprocessing step creates a quadtree structure of textures. The highest detail quadtree level contains the tiled original image. Each tile covers the size of two triangular patches within the respective bintree. Coarser levels are obtained by filtering the original image with a factor of two to adapt to the larger area that is covered by the corresponding triangular patches.

While creating hierarchies for triangular patches and textures, an object space error is calculated for each surface and texture element. These view-independent errors are calculated bottom-up by using the respective previous level and represent a measure for how exact the approximation of the original – not simplified – surface or texture is.

For geometry, the error is the maximum height difference between the simplified mesh and the original. For simplicity, the authors use the magnification factor of a filtered texture as its object-space error, noting that there are better estimation methods for texture error.

The last action of preprocessing is to build a hierarchy of bounding volumes for the triangular patches. Each bounding volume of a patch must include all children bounding volumes. Furthermore, two triangular patches adjacent along their hypotenuses share the same bounding volume which encloses both of them. This hierarchy is used to calculate view-frustum culling and estimate screen-space errors during runtime.

Frame-wise Updates

At runtime, the geometry and texture hierarchies are traversed until a desired screen space error is reached. A screen-space error is calculated by applying a monotonic projective transformation to a patch's bounding volume and combining its corresponding object-space geometry and texture errors.

The algorithm starts at the top level of the texture and geometry trees and visits the texture nodes recursively. While descending the texture hierarchy, the corresponding elements of the geometry trees need to be identified. Whenever the texture screen space error has reached an acceptable bound, its refinement part is stopped and the texture is bound. The algorithm then continues to refine the geometry by traversing the geometry bintrees until the geometry screen-space error becomes acceptable as well. The patches are then sent to the graphic pipeline and rendered.

When refining, it has to be kept in mind that one step in the texture quadtree corresponds to two steps in the geometry bintrees. Otherwise, the textures could not be exactly assigned to triangular patches. Figure 3, which has been taken from the author's paper, illustrates this characteristic.

In relation to this, all evenly numbered levels in the geometry hierarchy are skipped, while the texture quadtree is traversed. As a result, when the algorithm starts the geometry refinement, it does not begin with the nodes in the geometry hierarchy that were identified to match the texture, but with the parent nodes.

This must be done as the matching level of detail for geometry might have been missed while skipping every second level. If a parent node meets

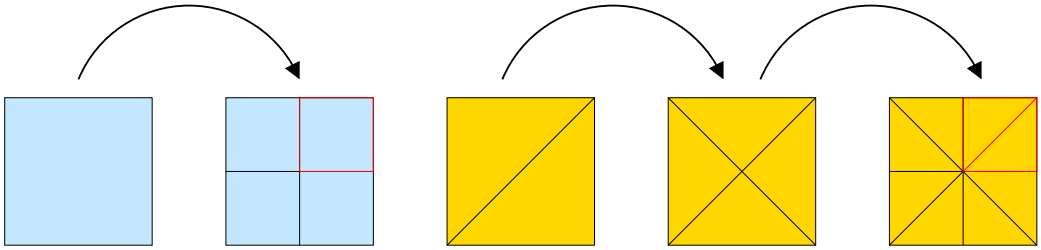
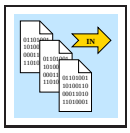


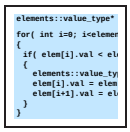
Figure 3: The left side shows one refinement step in the texture quadtree, while on the right two refinement steps in the triangle bintrees are displayed. Both a texture element and its associated triangles are hightlighted in red.

the error criterion, it has to be clipped to the region of the texture that has already been bound.

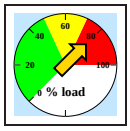
Summary



- Not specifically mentioned in the author's paper, but it must be possible to create a TIN from the input data

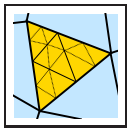


- The algorithm is very complex
- A culling technique is virtually free, because bounding volumes need to be calculated anyway



- Very complex preprocessing step including the generation of hierarchies for terrain patches, textures, object space errors and bounding volumes
- Compared to similar algorithms (e.g. ROAM), relatively little CPU load at runtime

- All terrain must fit in RAM; calculating additional patches would take too much time
- Reduced transfer to the graphic card (e.g. compared to ROAM)



- Better terrain approximation with similar number of triangles than purely right triangle based approaches, because the patches are TINs

2.3 Geometry clipmaps

Geometry Clipmaps were originally introduced by Losasso and Hoppe in 2004 [15]. The algorithm presented in this chapter is a follow-on paper by Asirvatham and Hoppe providing a GPU-based implementation [1] based on the original work. This approach is very different from the ones presented before and is one of the first terrain rendering algorithms based on programmable graphic hardware.

ROAM and BDAM both create TINs approximating a given terrain as exactly as possible while using as few vertices as possible. Geometry Clipmaps however focus less on accuracy but more on relieving the CPU by calculating as much of the landscape to be rendered by using the GPU.

The algorithm renders a fine rectangular, constant, uniform grid (referred to as clipmap level l_0) with its center at the x- and y-coordinates of the camera. Around this grid, ring-shaped meshes (referred to as clipmap levels l_1 to l_{L-1}) with increasing spatial extent and a fixed number of vertices in width and height are constructed. All L clipmap levels receive the z-coordinates for their vertices with the help of a vertex shader program that interpolates them from a given height map texture.

Preprocessing

The height information of the terrain is prefiltered into a single channel mipmap pyramid with L levels, corresponding to the same number of clipmap levels. A method of how to determine L is not given by the authors, but I assume that it depends on the desired viewing distance combined with the selected number of vertices n that each clipmap level has along its edges (see below).

Because the complete pyramid is likely to exceed the available graphic memory for huge terrains, the algorithm caches a square window of $n \times n$ samples within each mipmap level. The authors note that for the algorithm to work properly, n must be an odd number. They decide to define it by $n = 2^k - 1$ to support hardware that is optimised for power of two texture sizes. The complete pyramid is stored in graphic memory.

The main task of the preprocessing though, is to construct a small set of constant vertex- and index-buffers (referred to as footprints) which will construct the clipmap levels during runtime. The authors note that this approach has been taken to reduce memory requirements for the vertex- and index-buffers as well as to enable view-frustum culling, which would be more difficult with complete ring-shaped meshes.

There are three types of footprints, as depicted on the right side of figure 4:

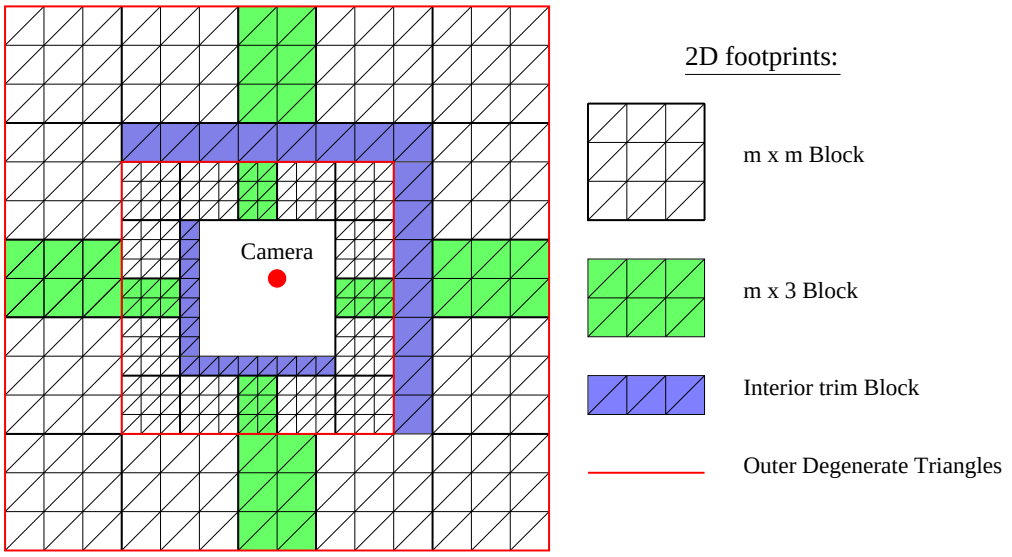


Figure 4: The construction of a mesh with geometry clipmaps, with example values of $n = 15$ and $m = 4$ vertices.

A block of $m \times m$ vertices, a block of $m \times 3$ vertices and a so-called "Interior trim" block, with m being defined as $m = (n + 1)/4$. A concrete clipmap level is constructed by arranging and scaling these footprints at runtime. The "Interior trim" block would be the only footprint to need rotation, so it is split up into four separate vertex- and index-buffers which can be reused without rotation.

As the footprints will obtain the height values for their vertices from a height map, it is sufficient to generate them with x- and y-coordinates only and store them in graphic card memory. The complex structure of the footprints is induced by the choice of n, which causes each clipmap level L_i not to be exactly centered to level L_{i+1}.

As each of the L clipmap levels has a fixed number of n vertices along its edges, an almost constant triangle size in screen-space is provided due to the increasing distance from the viewer.

The left side of figure 4 shows two exemplary cascaded clipmap levels that construct a terrain mesh. For each use, the footprints are transformed to the right position and scale, depending on the current clipmap level. It should be noted that the values of n and m would be more in the region of $n = 255$ and $m = 64$ in a real application.

Whenever the camera is moving, the clipmap levels move correspondingly and are updated with new data. Additionally to interpolating the correct

height for each vertex from the clipmap pyramid, vertex and pixel shaders take care of smoothing the transitions between the different grid levels.

These transitions suffer from T-junctions, which are fixed by inserting strings of degenerate (zero area) triangles (marked as red lines in figure 4) on the outer perimeter of each clipmap level.

Regular Updates

As opposed to the earlier presented algorithms, Geometry Clipmaps does not need to perform updates every frame, but merely when the viewer moves enough. Even then, the coarser a clipmap level is, the fewer updates it needs, as the relative viewer movement within the levels decreases exponentially with the size of the grids.

If an update has to be performed, it involves two steps: The first one is to update the mipmap pyramid. This is performed by applying a fragment shader which first predicts the height from a coarser level of detail and then adds residuals. This approach allows the addition of terrain detail through different sources, like compressed detail information of the actual terrain or a synthesis algorithm (e.g. based on a gaussian noise function). For each clipmap level, the height information of the next coarser level and the signed difference to the detailed height are stored in a single texture to reduce texture lookups in the shaders.

The second part is the creation of a normal map for the terrain. The authors choose this map to have twice the resolution as the height map because they found it to be too blurry otherwise. For performance reasons, the normal map contains the normals of the current level of detail as well as the ones of the next coarser level. This is accomplished by choosing the map to be a 4 channel, 8 bit per channel texture, rescaling both normals so that their z-value is 1 and then writing only x- and y-values to the texture. Before being used in a shader, these values obviously require decompression.

Rendering

During rendering, the vertex shader smoothes the transitions between two different clipmap levels by linearly blending the terrain height at the borders. The required information is not interpolated during rendering, as it would take too much time due to the many required texture fetches. Instead, the needed height information is efficiently stored in the height map (as stated in the previous subsection). The factor calculated for blending the fine and coarse heights is then passed on to the fragment shader.

In the fragment shader itself, the coarse and fine normals of each point are

unpacked from the normal map (again, stated in the previous subsection) and blended with the blending factor from the vertex shader. Afterwards, they are used to shade the terrain. The colour of the terrain is merely interpolated from a z-based 1D colour texture, which of course might be replaced by a common color texture.

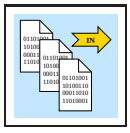
Optimisations

The authors of the paper introduced many optimisations which additionally increase the performance. Some of these optimisations, such as the building of the clipmap levels from footprints and the extraordinary storage of height values and normals in textures have already been mentioned. As the sheer number of optimisations presented in the paper concludes, they seem to be crucial for the algorithm. I will touch upon two more of them that are also presented in covered paper:

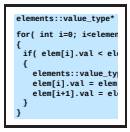
Depending on the height of the camera above the terrain, not all clipmap levels are drawn because the detail of those levels would not be recognisable anyway and might even lead to aliasing effects. As all of the terrain data resides in video memory, one readback from the graphic pipeline has to be done once in a while to decide which levels to draw. If clipmap levels are omitted, the one with the highest detail needs to be rendered as a full square instead of a ring-shaped mesh.

When updating the normal and geometry textures, the authors use wrap-around adressing. This way, they avoid translating existing data and it is sufficient to change small parts of the textures during the regular updates instead of recalculating them completely.

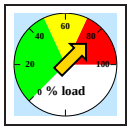
Summary



- 2D height map

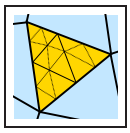


- Terrain structure is simple and constant, thus easy to understand and create
- No need for modifications during runtime
- Transitions between different clipmap levels bear T-junctions which have to be concealed



- Almost no precalculations required
- Heavy utilisation of the GPU, relieving the CPU
- Irregular updates can be accelerated utilising the GPU
- View-frustum culling is possible

- The original height map is stored in system RAM
- A window of the height map including a mipmap pyramid and the 2D footprints need to be stored in graphic memory



- Optimisations reduce accuracy
- Level of detail depends only on the distance to the viewer

2.4 Geometry Image Warping

Similar to Geometry Clipmaps, the approach of Geometry Image Warping by Dachsbacher and Stamminger [7] uses a height map texture as the data source. Instead of rendering constant meshes, however, this algorithm warps quad meshes to enforce geometry detail where it is needed while keeping topology constant and rendering as simple as possible, thus increasing performance.

Unlike the other presented algorithms, this one only gathers rough geometry information from the height map and adds procedural detail during runtime. Acutally, the height map needs to be coarse, because the CPU will operate on its values every frame. As this thesis is about terrain rendering, rather than terrain synthesis, this section will mainly present the rendering part of the algorithm.

To enable the warping process and high-performance rendering, Geometry Image Warping makes use of geometry images. These are textures whose pixels can be directly interpreted as vertex coordinates, implicitly defining a square-shaped mesh. The advantage of this approach is that classically image-based operations such as up- and down-filtering can be applied to a mesh without the need of conversions.

Geometry images can be efficiently rendered using graphic hardware by directly interpreting the texture as a quad mesh. The authors note that an OpenGL extension is required to enable this interpretation. Unfortunately, they do not mention which extension it is, but it is likely to be `ARB_pixel_buffer_object` [10], which has become part of the standardised OpenGL extensions in December 2004.

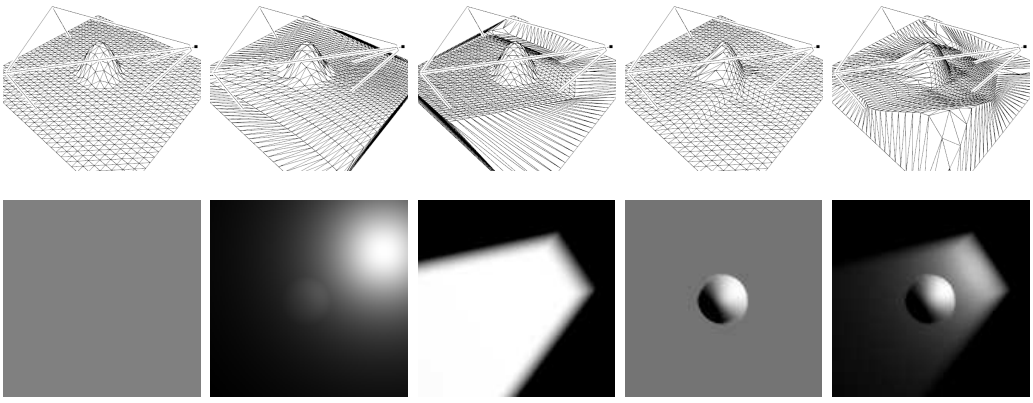


Figure 5: Different important maps (bottom row) and their influence on an example quad mesh (top row).

Preprocessing

Much as for Geometry Clipmaps, the preprocessing part of the algorithm is rather simple. Basically all that needs to be done is to read the height map into system RAM from which the frame-wise updates can read. The authors call this texture sketch atlas.

Frame-wise Updates

The frame-wise update is separated into several steps, most of them are done on the CPU. The first action in every update is to copy a square region from the sketch atlas which encloses the current view-frustum. This region is called the sketch map.

After that, another texture is created, which indicates how important every region of the sketch map is. The importance encoded in this importance map depends on certain surface characteristics, namely the surface orientation, the distance of the surface to the viewer and the location of the surface compared to the view-frustum. For each point of the geometry, it defines an approximate value for how many grid points should be around that point. Figure 5 demonstrates how such importance maps and the resulting quad meshes can look like.

The geometry image for the current frame is constructed by writing world x- and y-coordinates to the R- and G-channels of a new texture and writing the height from the sketch map to the B-channel.

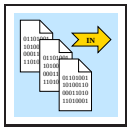
The geometry image is then warped according to the importance map, so that regions with a high importance are enlarged and regions with low importance shrink. Calculating this physically correct would include iteratively solving a spring-mass system, with the geometry image's pixels being the masses and their respective importances representing the springs.

However, the authors choose an alternative, yet less accurate approach: They interpret the geometry image as piece-wise linear functions, first for the rows, then for the columns of the image. Their warping algorithm operates on each of these linear functions, moving the control points depending on their calculated importance. Afterwards, the warped functions are uniformly resampled.

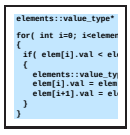
The authors claim that the described warping algorithm delivers acceptable results – even when not applied iteratively – which enables a frame-wise recalculation on the CPU, at least for coarse input maps. To allow for meshes with higher resolution than the geometry image, the warping algorithm can be adapted to output an up-sampled geometry image.

The resulting geometry image is rendered directly, being interpreted as a quad mesh. Procedural detail is added in the fragment shader during rendering through an adjusted bump-mapping technique. Due to the fixed topology of the mesh, no error bounds whatsoever are given by the authors. Additionally, the original terrain data is resampled twice, which introduces even more inaccuracy.

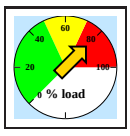
Summary



- 2D height map

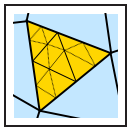


- Collision detection is difficult to implement
- Because the original height data is resampled during rendering, swimming artifacts can occur



- Almost no precalculations required
- CPU / GPU load almost balanced, CPU is not stressed too much as calculations are linear
- Update calculations can be performed using the GPU

- RAM must only hold the input height map, a small window of it, the importance map and the geometry image
- Geometry image (quad mesh) is transferred to the graphic card each frame



- Although visually appealing, the correctness of the rendered terrain cannot be guaranteed, even error bounds are not given by the authors

2.5 Generic Adaptive Mesh Refinement (GAMeR)

The last presented algorithm is detailed in an article entitled "Generic Adaptive Mesh Refinement" [2] and is authored by T. Boubekeur and C. Schlick. Unlike the other approaches, this is not specifically a terrain rendering algorithm, but – as the title denotes – a generic approach to refine meshes. As it can be applied to any kind of meshes, this algorithm still suits the topic of this thesis.

The algorithm accepts a mesh of coarse triangles and replaces each triangle with an Adaptive Refinement Pattern (ARP – A finer approximation of the original triangle) depending on the triangle's required level of detail. This is achieved by exploiting the features of Shader model 4 in cooperation with a set of precalculated fine meshes that are stored in graphic memory.

Preprocessing

Because the calculation of the ARPs is too complex to solve within a shader, it has to be solved in a preprocessing step. Given a maximum refinement depth l for each of the triangle's edges, a three dimensional matrix of l^3 ARPs is required to cover all possible combinations of refinement within a triangle. This matrix is called the ARP pool.

To conserve video memory, a single vertex buffer of the highest refinement ARP can be created, while all other ARPs are encoded as index buffers for that vertex buffer. The vertices within the vertex buffer are encoded in barycentric coordinates. These are used to interpolate information (like the world-space position) from the original triangles prior to rendering. Additionally, utilising barycentric coordinates makes worrying about the orientation of a triangle unnecessary.

Each element $\{i, j, k\}$ in the ARP pool corresponds to a triangle whose refinement configuration is as follows: Edge 1 has refinement depth i, edge 2 has refinement depth j and edge 3 has a refinement depth of k. An edge with refinement depth i will be split i times, resulting in 2^i vertices. The

interior of the triangle must contain a valid tesselation, respecting the given number of vertices for each edge.

The authors note that creating ARP elements with refinement depths $i \neq j \neq k$ is rather difficult. One approach noted by the authors is to uniformely refine a basic triangle mesh until the minimum level of detail of the three edges is reached. Then, the triangles at the other edges are split until the required refinement depth has been reached. In unfavourable cases, this simple approach results in many oblique (non-right) triangles at the mesh's border.

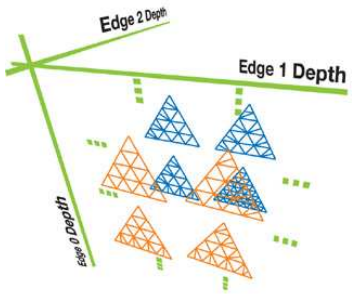


Figure 6: Detail of the ARP pool.

This described effect can be avoided by applying better algorithms for the ARP creation, which on the other hand leads to a higher primitive count and – depending on the algorithm – a longer preprocessing time. Figure 7 shows two refinement topologies featuring the same depth-tag configuration $\{3, 4, 5\}$, generated by different algorithms. Note that the right version consists of less oblique triangles while using more triangles than the left version. Both images shown in this section are taken from the discussed paper.

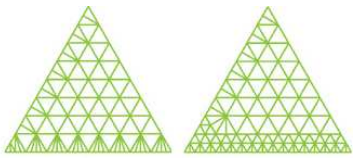


Figure 7: 2 refinement topologies for the same depth configuration.

Frame-wise Updates

The algorithm only needs frame-wise recalculations if the refinement depths change dynamically. In that case, the refinement depth tag of each vertex needs recalculation. This tag measures how fine the mesh should be tesselated in the vicinity of the corresponding vertex. Amongst others, the authors name camera-to-vertex distance, local curvature and the semantic importance of the vertex as applicable metrics for calculating the depth tag.

The depth tags of two adjacent vertices are interpolated to gain the depth configuration for the edge they span. Thus, a crack-free mesh topology is implicitly ensured because triangles sharing an edge will compute the same depth configuration for their common edge.

From the CPU's point of view, the rendering loop can be described with this short pseudo-code:

```
GLuint ARPPool[MaxDepth][MaxDepth][MaxDepth];

void render(Mesh M)
{
  if(dynamic)
    for each Vertex V of M do
      V.depthTag = computeRefinementDepth(V);
  for each CoarseTriangle T of M do
  {
    sendToGPU(T.attributes);
    bind(ARPPool[T.V0.depthTag][T.V1.depthTag][T.V2.depthTag]);
    drawElement();
  }
}
```

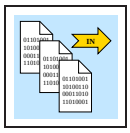
Note that the attributes of each vertex need to be explicitly transferred to the GPU (e.g. via uniform variables) as the original coarse triangle is not drawn. Instead, only an index to the ARP pool is selected and the corresponding fine mesh is drawn. The metrics for the depth tag computation can be selected freely and are even dynamically exchangeable during runtime.

The vertex shader consists of three parts: The tesselation, which uses the barycentric coordinates of the ARP to interpolate the attributes of the coarse input triangle, a user-defined displacement function and the shading as a last step. The following pseudo-code outlines these operations:

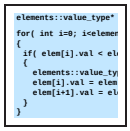
```
const uniform vec3 p0, p1, p2; // the triangle's vertices
const uniform vec3 n0, n1, n2; // the triangle's normals

void main()
{
  // tesselation by barycentric interpolation
  float u = gl_Vertex.y, v = gl_Vertex.z, w = gl_Vertex.x;
  gl_Vertex = vec4(p0*w + p1*u + p2*v, gl_Vertex.w);
  gl_Normal = n0*w + n1*u + n2*v;

  // apply displacement function
  float d = displace(gl_Vertex.xyz);
  gl_Vertex += d*gl_Normal;

  // shading and output
  ...
}
```

The fragment shader is completely unused by the algorithm itself, so any kind of shading can be applied.

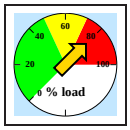
Summary



- 2D height map and coarse triangle mesh

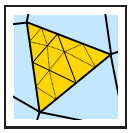


- Vertex attributes (including position) need to be transferred separately from the mesh topology
- CPU needs to determine ARP pool index for every triangle



- If refinement depths change dynamically, per-frame updates must be performed by the CPU
- Only vertex shader is stressed, no fragment operations are needed

- The ARP pool needs to be stored in video RAM
- Per triangle, an index into the ARP and the triangle's attributes are transferred to the GPU each frame



- No error bounds are discussed in the article

2.6 Recapitulation

This chapter has presented a review of the development of terrain rendering algorithms in the past 15 years. A significant change in newly emerging approaches can be observed when graphic cards with programmable vertex and fragment processors became available for the first time. These gave computer engineers the possibility of handling big amounts of calculations on an extra processor that purely focuses on graphic matters, thus relieving the CPU.

Early algorithms take meshes that constitute the finest available approximation of the terrain as input. This approach is derived from classic land surveying techniques: Positional information is stored in trigonometric meshes. Such a mesh can be seen in figure 8 which is taken from [4]. Algorithms like ROAM or BDAM strive to maintain a steady rendering rate while keeping the input mesh as detailed as possible.

Nowadays, height information is also gathered with distance sensors and high resolution color cameras mounted on airplanes or even satellites. While the first of these can gather height information directly, images taken with colour cameras can reveal height information by utilising interferometry. Two images of the same target area are taken from slightly shifted points of view. From the differing colour information of these two images, height information can be computed, which is basically how human eyes procede.

Both of these approaches deliver their results as maps, which not only contain more detail than classical surveying techniques, but are also more intuitive to handle. Yet, these new data sources call for new rendering techniques. Such algorithms have to solve different problems than the classic mesh-based approaches. They often have plenty of available height information, but they need to generate meshes on their own to illustrate this information.

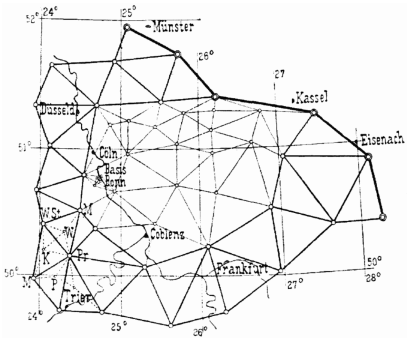


Figure 8: A geodetic mesh

The three presented algorithms based on height maps have different approaches to visualise such maps: Geometry clipmaps uses static meshes which are finer in detail the closer they are to the viewer. Geometry Image Warping warps a constant quad mesh to allow for enough detail in respective areas. And finally, GAMeR refines triangles with precomputed meshes to increase mesh detail.

The development of terrain rendering algorithms tends towards constantly relieving the CPU and increasingly reducing bus transfer to a minimum. Like this, the processor has more time to perform other necessary calculations (e.g. like physical ones) and a steady rendering rate can be achieved.

In the next chapter, I present the new features of Shader model 4, which include new options to improve terrain rendering.

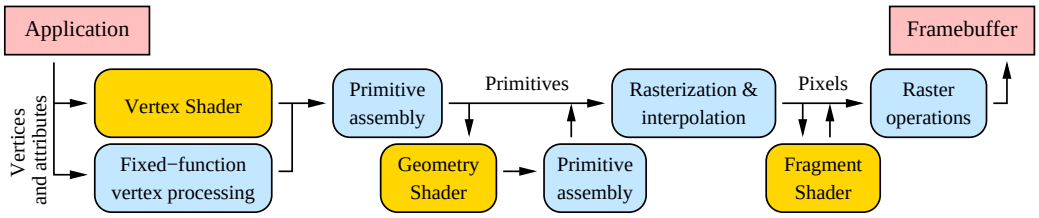


Figure 9: The simplified OpenGL rendering pipeline with Shader model 4.

3 Shader model 4

Before presenting the features of Shader model 4, I will detail what a Shader model is. The following quote from [5] summarises the term accurately:

> "A shader in the field of computer graphics is a set of software instructions, which is used by the graphic resources primarily to perform rendering effects. Shaders are used to allow a 3D application designer to program the GPU (Graphics Processing Unit) programmable pipeline, which has mostly superseded the older fixed-function pipeline, allowing more flexibility in making use of advanced GPU programmability features."

Due to constant and on-going development, each new generation of graphic cards provides new features to program the GPU. These new features, such as the set of available instructions, the hardware definitions and the supported data formats are combined in the term Shader model.

Beginning with the release of Shader model 1 in November 2000 (as part of Microsoft's DirectX 8.0), more and more of the fixed-function pipeline has been abandoned and replaced with a programmable complement. The first release already supported vertex and pixel shaders, replacing the fixed vertex and fragment processing stages. Since then, each new Shader model has featured new instructions and fewer restrictions, extending the functionality of the previous Shader models (for more details, see [6]).

Shader model 4 was introduced along with DirectX 10.0 in November 2006. Just like previous revisions, it mainly features improvements to pre-existing functionalities and reduced restrictions for programming the GPU. As this section is concentrating on the new possibilites of Shader model 4, it should be noted that these minor enhancements can be reviewed further on the internet, for instance at [6].

Having said that, Shader model 4 offers completely new features to GPU programmers. Firstly, Shader model 4 programs now support a full set of

IEEE-compliant 32-bit integer and bitwise operations. In Shader model 3, all integers were inefficiently converted to floats, while the new model allows for algorithms such as compression or packing techniques and bitfield program-flow control. The bitwise C-style operations include negation, shift, `AND`, `OR` and comparison operators. All previously available functions like `abs`, `sign`, `min`, `max` and `clamp` functions now support signed and unsigned integers.

Texture arrays allow complex texture blending operations to be done in one draw call through indexing of textures. The special case of using 6 textures in an array to construct a texture cube can be utilised to easily construct a cube map (see [16]). Also, values from textures can now be interpolated as integers instead of them being converted to floats ranging from 0 to 1.

Instanced rendering supports the repeated rendering of the same mesh without having to re-transfer it to the graphic card. A vertex shader can decide how to transform each instance into world space depending on the given instance ID, thus reducing bus load.

The most promising feature of Shader model 4 concerning this work is the introduction of a new shader stage: the geometry shader. This new stage is settled after the primitive assembly, but before clipping and rasterisation (the location within the pipeline is illustrated in figure 9). It is called for every primitive rendered, enabling the programmer not only to manipulate the primitive, but also to skip it completely or insert new ones on the fly.

Geometry shaders work on complete primitives[1] and can emit either points, line- or triangle-strips. The data output of a geometry shader must not exceed a certain hardware limit, which is 1024 floats for modern graphic cards, such as those from nVidia's GeForce 8 series.

Another new feature of Shadel model 4 is the capturing of output from vertex and geometry shader stages. Intercepting their emitted vertices allows them to be reused in later passes. The rendering pipeline can optionally be stopped at these points, avoiding possibly unnecessary fragment stage passes. For HLSL, this technique is called draw-auto, while the name for GLSL is transform feedback.

Finally, it should be noted that all shader units on graphic cards supporting Shader model 4 are dynamically utilised as vertex-, geometry- or fragment-shaders as necessary. As the graphic driver controls the utilisation depending on the respective load distribution, programmers do not need to worry about equally distributing computations to the different shader types.

[1] As for now: points, lines and triangles, the latter two with or without the also newly introduced adjacency information

4 Recursive Adaptive Geometry shader

This section introduces a new approach called Recursive Adaptive Geometry shader (RAG) to render terrain data with the features of Shader model 4.

The basic concept is to calculate an edge-based refinement for each triangle in a geometry shader and tesselate the triangle directly within the rendering pipeline. This can be done recursively by rendering the results of the geometry stage into a vertex buffer and refeeding them into the pipeline.

I begin by summarising the motivation for the new approach. After that, the successive subsection "Realisation" details on how RAG refines triangles and how it is applied recursively. The "Preprocessing" and "Frame-wise updates" subsections describe run-time proceedings of the algorithm, clarifying them with some pseudo-code. Following this, problems evoked by RAG are discussed in detail.

After having covered the new approach itself, a simultaneously developed algorithm that employs a process similar to RAG's is presented and compared to RAG.

4.1 Motivation

The primal intention of the thesis is to find out to which extent the features of Shader model 4 can be utilised for terrain rendering. It is specifically expected that employing the geometry shader stage can transport dynamic mesh refinement completely to the GPU, thus relieving the CPU. Having the latter relatively idle is considered an advantage, because other calculations like complex physic simulations often strive for CPU time simultaneously.

Basically, there are three characteristics of terrain rendering algorithms that need to be met: accuracy, speed and mesh topology. For the user, what counts is that the resulting 3D rendering approximates the input data as accurately as possible and has an appealing look, while still being interactive. Naturally, an algorithm always has to compromise between these properties to some extent, as higher accuracy and a better look require more computations, which in turn need more time.

Concerning terrain rendering, the characteristic of accuracy is usually measured by the screen-space error. This metric is determined by computing the screen-space distance of a rendered pixel to the point where it should have been placed in an optimal case (hence the name). How much screen-space error is acceptable depends on the application and it is therefore advisable to have this property configurable.

The speed of a terrain rendering algorithm is usually measured in frames per second. The time it takes to calculate one frame is the sum of the CPU

calculation time, the time it takes to transport necessary information from RAM over the system bus to the graphic card and finally the GPU time. Each of the three can be optimised in different ways. For instance, the CPU can provide all necessary information to the GPU and then start calculating the next frame without waiting for the result.

A bad mesh topology can have several forms of negative impact on the resulting image. Meshes bearing T-junctions can reveal gaps in the terrain, which either show the scene's background colour (typically black) or objects located underneath the terrain which should otherwise be hidden. Degenerate triangles are oblique ones with (almost) zero area and can also cause rendering artifacts. These artifacts are often not as noticable as gaps, but can still impair the visual quality of the rendered terrain.

4.2 Realisation

As motivated by the previous section, the goal of this thesis is to find out how the features of Shader model 4 can be utilised to obtain a performant terrain rendering algorithm. The geometry shader stage introduced with Shader model 4 can operate on incoming triangles and is able to generate new geometry dynamically within the rendering pipeline. These characteristics let the geometry shader perform refinement operations which in other algorithms would have to be computed by the CPU.

A classical CPU-based mesh refinement algorithm that refines meshes on a triangle basis was introduced in an article named "Continuous LOD Terrain Meshing Using Adaptive Quadtrees" [17] by Thatcher Ulrich in the year 2000. It recursively iterates over all triangles in a uniform mesh of right-isosceles triangles, splitting each triangle edge once at maximum per recursion. The resulting sub-triangles are always right-isosceles as well, which constitutes a good property for rendering. As the refinement decisions applied are edge-based, T-vertices are implicitly avoided.

Because this approach delivers refined triangle meshes with good topology, mimicking its proceeding in a GPU implementation is basically a good idea. I therefore split up this algorithm into two basic parts – tesselation and recursion – and discuss how each part can be adapted to suit a geometry shader based realisation.

Both parts are discussed in the remainder of this section. The result is a new terrain rendering algorithm entitled Recursive Adaptive Geometry Shader, referred to as RAG.

Ulrich's algorithm requires a triangle data structure like winged-edge or half-edge to operate. Unfortunately, no algorithm is known yet that implements such structures to be used in a shader program. As I will show in the

following subsection, this averts an exact adaption of Ulrich's algorithm in a GPU-based implementation.

4.2.1 Tesselation

The goal of the tesselation is to split triangles that do not approximate the input height map exactly enough into smaller ones. To maintain a continuous mesh topology at the same time, neighbouring triangles must share the same tesselation along their common edge. As triangle-based refinement decisions can result in a differing refinement for the common edge of adjacent triangles, refinement decisions must be taken purely edge-based to guarantee a continuous topology.

Consequently, the first step to refine a triangle is to decide which of its edges need to be split. This refinement decision basically depends on the application, but predominately various error metrics are applied. Possible error metrics include the height difference between an edge's spanning vertices as well as estimated object- and screen-space errors between the coarse and the refined edge.

Depending on the outcome of the refinement decision, a new vertex is inserted at the midpoint of each respective edge. This binary decision results in $2^3 = 8$ possible tesselations for each triangle. There are several reasons for not inserting more than one vertex along each edge, the most important of which are:

Simplicity:
The most important reason is that 8 cases of tesselation can be handled quite comfortably. Increasing the number of splits by one per edge already yields $3^3 = 27$ different cases for each triangle, including complicated refinement configurations.

Expensive conditionals:
More cases of tesselation require more conditional statements within the geometry shader, which are still very time-consuming on current graphic hardware.

Geometry shader output limit:
The output of the geometry shader stage is currently limited to 1024 floats per call. With one split per triangle edge, this limit is never exceeded, avoiding problematic cases of incomplete refinements.

After the edges which are to be refined have been determined, a valid tesselation for the triangle must be generated. Ulrich's algorithm forwards

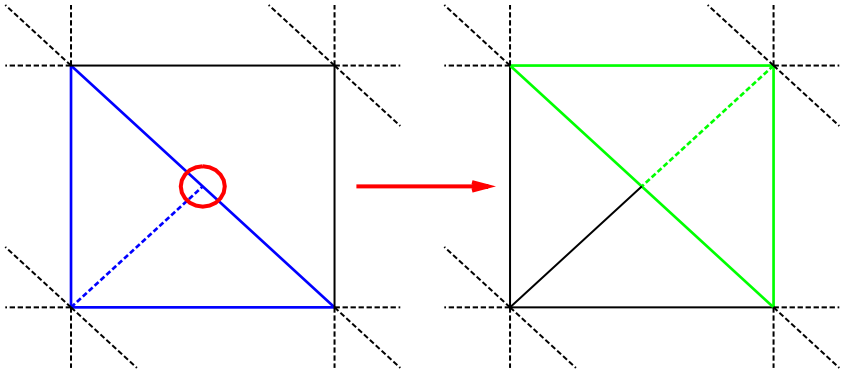


Figure 10: An edge split in triangle A (blue) enforces an edge split in triangle B (green) to avoid a T-vertex.

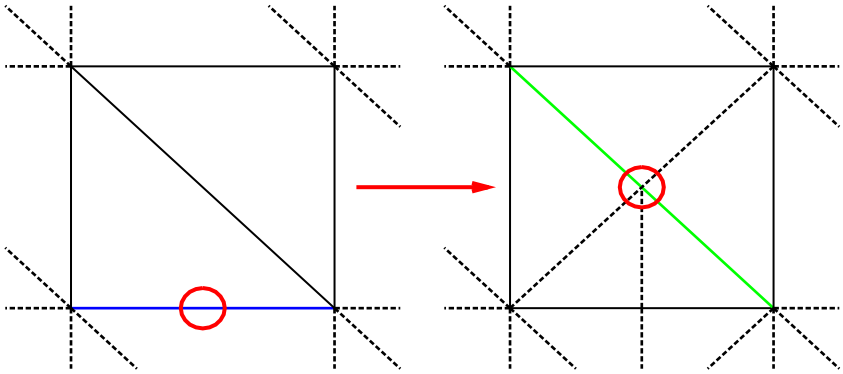


Figure 11: A needed split on edge A (blue) enforces a split on edge B (green).

the splitting of an edge to the adjacent triangle sharing the edge, just as shown in figure 10: Triangle A needs a split at its hypotenuse to add necessary detail, creating a T-vertex (circled in red) which is likely to evoke a crack in the mesh. To avoid this, an edge split is enforced in the neighbouring triangle B.

As already noted, no algorithm is yet known to implement triangle data structures such as winged-edge or half-edge in a shader program. This concludes that it is not possible for RAG to communicate a split to a neighbouring triangle. In most cases, this is not problematic: As RAG makes edge-based refinement decisions anyway, neighbouring triangles compute the same refinement for their common edge.

Problems only arise in cases where edges that do not need refinement are split nonetheless in order to maintain a right triangulation. Such a situation is illustrated in figure 11. The split on edge B is merely performed to keep the triangulation right-angled and the split is communicated to the neighbouring triangle to avoid a crack in the terrain.

As RAG cannot enforce a triangle split in the triangle sharing edge B and that triangle will not split edge B unless one of its other edges needs refinement by chance, a T-vertex is most likely being created. There are two ways to deal with this problem, both of which conclude a restriction for the algorithm.

One way is to maintain a right-angled tesselation and abandon possibilities concerning the splitting decisions. The other way is to drop the purely right-angled tesselations and have freedom in choosing the splitting decisions. Both approaches are discussed below, followed by a rating conclusion.

Right-angled tesselations

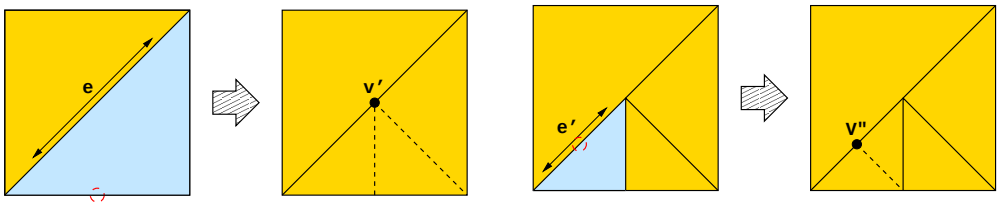
Though it is not possible to communicate a split to another call of a shader, RAG can still produce right-angled triangulations, but at a cost. The idea is to intentionally insert invisible T-vertices on the hypotenuse in problematic cases. If the hypotenuse does not need an edge split, but one or both of the triangle legs do, then an additional vertex is inserted in to the hypotenuse just as in Ulrich's algorithm, so that a right-angled tesselation is formed.

The difference is that instead of the height value of this vertex being interpolated from the height map, it is linearly interpolated from the positions of its adjacent vertices. That way, the vertex will not create a gap in the terrain, as the vertex lies exactly on the edge between the two original vertices, but the triangulation will still remain right-angled.

This approach demands an additional prerequisite to work and results in a disadvantage concerning the refinement decisions. The prerequisite is that all height values entering the recursion loop must remain constant. Otherwise, the inserted invisible T-vertices would become visible as soon as they have their "real" height value from the height map assigned to them.

To avoid this, the basic mesh sent to the graphic card must be initialised with height values from the map before it enters the recursion loop. This has the positive side-effect that the shader can perform faster as it has to conduct texture lookups only for newly generated vertices as opposed to looking up a height value for every output vertex.

At the same time, this implies the denoted disadvantage:



(12.a) First recursion **(12.b)** Second recursion

Figure 12: An example for problems emerging from an enforced right-angled triangulation with RAG. The current triangle has a blue background, vertices inducing a split are circled in red.

> When using the right-angled approach, the refinement decisions for a triangle edge must be strictly monotonic. This means that any edges resulting from the split of an edge that did not require refinement must never require refinement themselves.

To clarify this restriction, consider the situation displayed in figure 12.a: RAG is using the screen-space error of not inserting a vertex in the middle of edge e as the refinement decision. In this situation, one leg of the blue triangle needs refinement (red circle), its hypotenuse e, however, does not. To achieve a right-angled tesselation, RAG inserts the invisible T-vertex v' in to the hypotenuse, as explained above.

During the next recursion step (figure 12.b), the same refinement metric is applied to edge e'. If e' is split[2], the newly inserted vertex v'' is very likely to become a visible T-vertex as the neighbouring triangle has not performed the first split and thus cannot insert v'' as well.

Partially-oblique tesselations

Abandoning purely right-angled tesselations, edge-based splitting decisions can be taken without constraints to the applied error metrics. The reason is that all tesselations can be performed without applying a split to an edge that principally does not need one. Following this, T-vertices are implicitly avoided as neighbouring triangles will always perform the same refinement for their common edge.

Figure 13 displays all possible tesselations for Ulrich's approach (left) and RAG (right). Note that five out of the eight tesselations are identical for both algorithms, only the ones including a split on one or both of the triangle legs that do not include a split on the hypotenuse differ.

[2]If there is a local rise in the terrain, like a tower or a hill, for instance

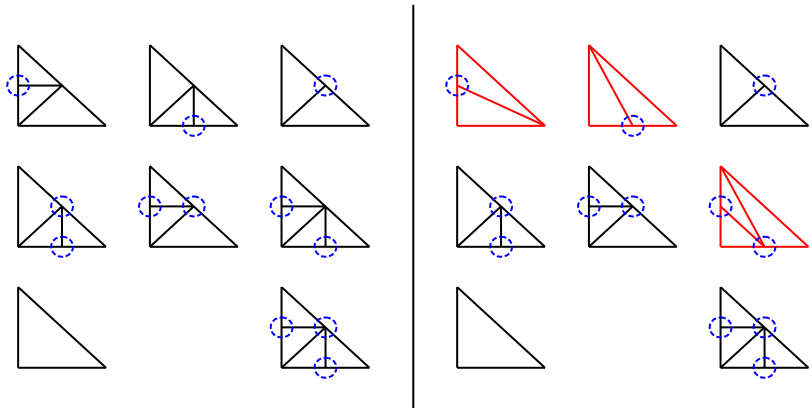
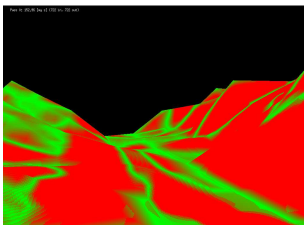
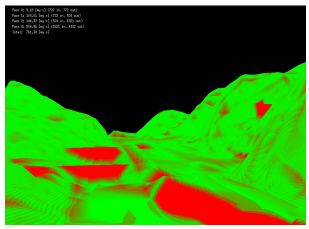


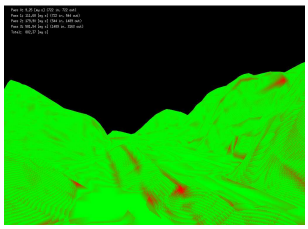
Figure 13: Possible triangulations in Ulrich's algorithm (left) and RAG (right). Edges forcing a split are marked with blue circles, oblique tesselations are shown in red.



(14.a) No refinement



(14.b) Right-angled tesselation



(14.c) Oblique tesselation

Figure 14: The images show the same scene with colour-coded screen-space error, green representing no error, red 10 pixels or more. Images 14.b and 14.c show the respective tesselation approaches with 3 recursions applied.

These differing tesselations can evoke degenerated triangles, a problem which is discussed in section 4.5.2.

Conclusion

The constraint to the refinement decisions introduced by the right-angled tesselation approach can have severe impacts on the accuracy of the rendered terrain, which is illustrated in figure 14. Figures 14.a to 14.c show the same perspective view into a terrain with colour-coded screen-space error. The leftmost image shows the scene without any refinement applied, the others apply a purely right-angled tesselation (center image) and a partially-oblique

tesselation (right image) with three recursion steps each.

In figure 14.b, RAG uses the monotonic refinement decision of the height difference between two edge-spanning vertices to decide whether to split an edge. The height difference that causes an edge to be split is set to 0.01, with heights of the terrain ranging from 0.0 to 0.5. After three recursion steps, the algorithm outputs 4332 visible triangles and requires 761 μs to finish.

In the right figure 14.c, a non-monotinic splitting decision is employed, namely the screen-space error of not inserting a midpoint vertex on an edge. The screen-space error enforcing an edge to be split is set to 2 pixels, with an OpenGL canvas size of 1024×768 pixels. After three passes, the algorithm outputs 3163 visible triangles, requiring 882 μs to finish.

As expected, the results of this comparison reveal that the version based on non-monotonic decisions creates a more accurate approximation of the terrain with less triangles while requiring some more time to calculate. The increased calculation time is due to the more exact error estimation metric applied. It should be noted, that all images have been taken with a research implementation of RAG, that can most likely be further improved to yield better results for both approaches.

My conclusion from the presented tesselation approaches is that the right-angled approach should only be applied if mesh topology is far more important than the terrain's accuracy. The oblique tesselations do not require a constraint to the refinement decisions and are therefore favourable in most applications.

4.2.2 Recursion

Until now, I have only discussed one refinement step. Ulrich's algorithm however is a recursive one that increases the mesh resolution iteratively. Shader programs however, cannot use recursion, not even statically. The only way to apply a shader recursively is to employ multiple render passes, only asserting that the data format does not change in between the calls.

Therefore, the output of a geometry shader pass must be captured in a buffer to be reused in a consecutive pass. A technique that provides this functionality for OpenGL is transform feedback [13]. To increase performance, the rendering pipeline can be aborted after the application of the geometry shader if more than one pass is scheduled.

The special advantage of this course of action is that the fragment stage needs to be passed only once even if the approach employs several rendering passes. This becomes very important when complex fragment shaders are applied to decorate the terrain surface, as their costs remain constant with this proceeding.

Only two pieces of information need to be stored in the transform feedback buffer: the vertex positions and the texture coordinates. The vertex positions are interpolated while tesselating triangles and are passed to the fragment shader on the last recursion step. The texture coordinates are used to interpolate height values for new vertices from the height map.

After a rendering pass is complete, the transform feedback buffer is bound as a vertex buffer for the next rendering pass. To allow for more than one pass, a second buffer is used as transform feedback buffer for the second pass. These two buffers take turns in operating as source vertex buffer and target transform feedback buffer. More than these two buffers are never required.

The described proceeding can be used as a dynamic level of detail algorithm. As RAG purely operates on the GPU, the CPU merely has to start it each frame and can then concentrate on other calculations. If the CPU regularly checks whether the transform feedback buffer is ready (e.g. by checking if the `GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN_NV` query is ready) and there is enough time left for the current frame, it can reapply RAG to the result of the previous frame to further improve terrain detail.

4.3 Preprocessing

Only very few preprocessing steps need to be performed for this algorithm. The most important step is the creation of a uniform triangle mesh that will serve as a basis for refinement during rendering. This mesh is provided with texture coordinates and is uploaded to graphic memory along with the height map texture.

The mesh need not have a high resolution, as it will be refined anyway, but it should be noted that – depending on the refinement decisions applied – mesh artifacts can occur in unfavourable situations. This can happen if the coarse mesh does not have enough vertices or – speaking from another point of view – the height map features too high frequencies. More detail on these problems can be found in section 4.5.

If early clipping is to be used – which is highly advisable to reduce the polygon count for later recursions – necessary information, such as the modelview-projection matrix for view-frustum culling, need to be updated whenever they change. The variables can be sent to the shaders by using uniform variables.

4.4 Frame-wise updates

The following pseudo-code sums up the basic advancement of the CPU-side rendering loop:

```
VertexBuffer VB0, VB1;

void render() {
  numPasses = calculateNeededPasses();
  if( numPasses > 1 )
    discardFragmentStage(true);

  setVertexBufferTarget( VB0 );
  renderCoarseMesh();

  for( i=1; i<numPasses; i++ ) {
    if( isLastPass(i) )
      discardFragmentStage(false);

    if( isOdd(i) ) {
      setVertexBufferTarget( VB1 );
      renderVertexBuffer( VB0 ); }
    else {
      setVertexBufferTarget( VB0 );
      renderVertexBuffer( VB1 ); }
  }
}
```

The `calculateNeededPasses` method determines the recursion depth of the algorithm. This can be a fixed number or a dynamically calculated one that depends on situational circumstances such as the viewer's distance to the terrain. Furthermore, the refinement loop can be aborted early if no additional detail is needed.

One method to determine such a situation would be to compare the output triangle count of the geometry shader from two consecutive passes. If the difference is too small, the refinement can be stopped and the mesh rendered. Another method is presented in section 4.5.2.

After the number of passes has been computed, the coarse mesh is rendered and the resulting mesh is stored in the first vertex buffer. If only one pass is rendered, the algorithm finishes here, otherwise it alternately renders the contents of one vertex buffer and stores the results in the other. This way it never needs more than two buffers, no matter how many loops are actually passed. Before applying the last pass, the fragment shader stage is reactivated in order to be applied to the refined mesh.

The essential workflow of the algorithm in the graphic pipeline is illustrated in figure 15 while the pseudo-code below outlines the advancement of

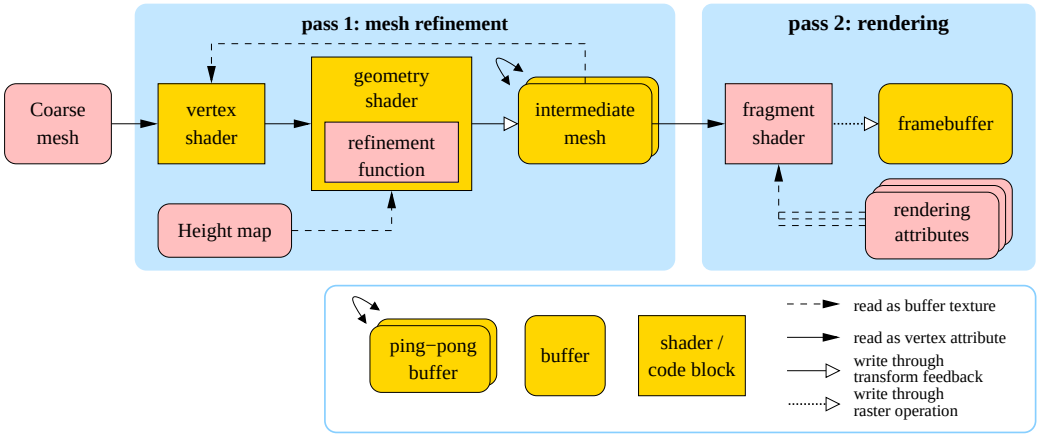


Figure 15: Outline of the RAG algorithm. Pink boxes mark application dependant parts.

the geometry shader:

```
uniform Triangle T;

function main()
{
  if( T.isInvisible() )
    return;

  int tess = 0;
  if( edgeNeedsRefinement( T.edge0 ) )
    tess |= 1;
  if( edgeNeedsRefinement( T.edge1 ) )
    tess |= 2;
  if( edgeNeedsRefinement( T.edge2 ) )
    tess |= 4;

  outputTriangles( tess );
}
```

In each call, the geometry shader first checks if the triangle is visible at all and skips it completely if not, thus reducing the polygon count for subsequent recursions. Such a check can be done by applying the modelview-projection matrix to each of the triangle's vertices and checking whether they are located inside the clipping space or not.

This particular visibility check, however, does not account for partialy visible triangles, whose vertices are all located outside the clipping space. Therefore, I would like to refer to well-known view-frustum culling techniques at this point, regarding the given algorithm as an example.

When the visibility test has been passed, each edge's refinement is calculated depending on the applied refinement decisions. The tesselation that will be performed is stored in the integer variable `tess` by utilising Shader model 4's integer operations. Depending on which edges need to be refined, the `outputTriangles` method creates a refined tesselation of the input triangle.

More detail on how to implement a geometry shader for RAG is presented in section 5.1.

4.5 Problems

As already denoted in section 4.3, RAG can produce terrain and rendering artifacts in certain situations, especially if the coarse mesh has too few vertices. This chapter will present problems caused by RAG, their origins and suggestions to minimise them.

The first two detailed problems are typical for most terrain rendering algorithms taking height maps as input and derive from the applied refinement decisions. The last problem of oblique triangles is common to GPU-based approaches and specifically pronounced for RAG.

4.5.1 Under-refined triangles

This is a problem known to most terrain rendering algorithms that rely on a height map texture as input data and derives from the resampling of the height data. This usually leads to triangles which would need further refinement but will not receive it because the vertices that the refinement depends on share similar attributes.

For example, consider a height map with high frequencies and refinement decisions depending on the height difference between two adjacent vertices of the coarse mesh. In that case, two adjacent vertices might share a similar height, which concludes no refinement for the algorithm. The height map, however, can still bear height differences between the vertices, resulting in an under-refined mesh. An example is provided in figure 16.

To avoid this problem, the height map can be scanned before rendering, checking its frequencies and adjusting the mesh resolution accordingly. However, it is commonly sufficient to adjust the applied refinement decisions. In the above example, the heights of two triangle vertices are compared with

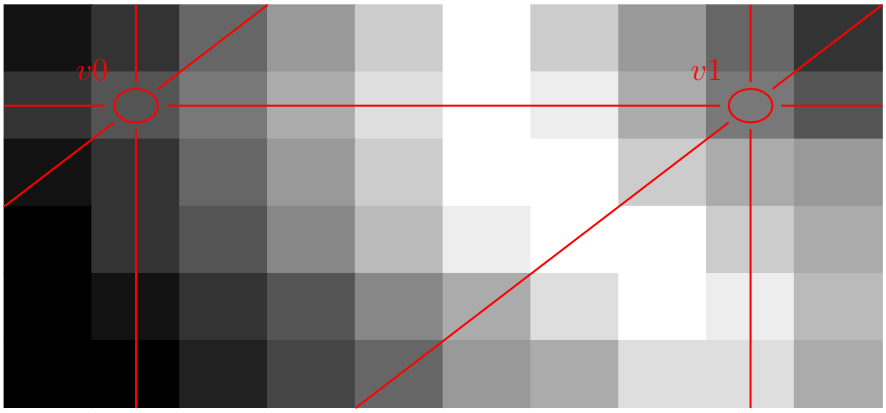


Figure 16: $v0$ and $v1$ share similar height values from the height map and height map detail between them will remain ignored.

each other. If, instead, the heights between the triangle vertices and the according edge midpoint had been compared, the problem would not have arisen in this example.

Generally speaking, inserting more samples along a triangle's edges improves error estimation, further suppressing the described effects. The issue, in this case, is just how much time the user is willing to invest into the error estimation and how exactly the error of an edge is estimated.

The worst case scenario for this type of issue would be a mesh whose vertices all happen to share similar height values, with all terrain detail located between them. In that situation, the coarse input mesh would not be refined at all.

4.5.2 Over-refined triangles

Over-refined triangles occur whenever RAG tries to compensate high height differences in the height map. If the refinement metric decides to split an edge, the situation shown in figure 17 could occur.

The refinement decision applied in figure 17 is based on the world-space height difference between the edge-spanning vectors. RAG attempts to minimise the height difference between the vertices $v0$ and $v1$ by inserting more vertices on their spanned edge. Because the height map resolution is rather low in this example, there is no new data to be gathered from it and the refinement does not stop until both checked vertices receive their height from the very same height map pixel.

As the problem evolves from height map textures with too little resolution r_t, the best solution is to stop the refinement when no new information can

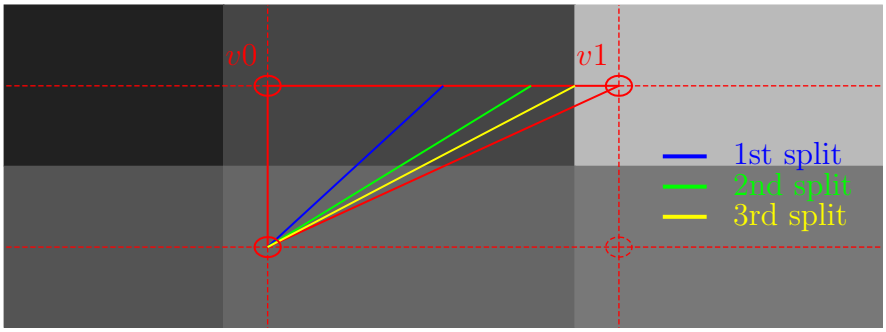


Figure 17: The height difference between $v0$ and $v1$ causes their common edge to be split three times.

be gathered from it. With each recursion step, the initial resolution of the mesh r_m is doubled as the distance between two adjacent vertices is halved. Thus, the maximum number of recursions n can be calculated as follows:

$$2^n \cdot r_m = r_t \quad (1)$$

$$2^n = \frac{r_t}{r_m} \quad (2)$$

$$n = \log_2 (\frac{r_t}{r_m}) \quad (3)$$

If n becomes greater than the right hand side of equation 3, the mesh resolution has exceeded the texture resolution and no further refinement is necessary.

There is, however, one scenario of over-refined triangles that cannot be fixed that easily. Terrain features like cliffs and other almost vertical faces tend to generate over-refined triangles a lot. This fact emerges from the nature of commonly applied edge-based refinement decisions: They decide whether to insert a new vertex along an edge based upon the error it would evoke not to insert it.

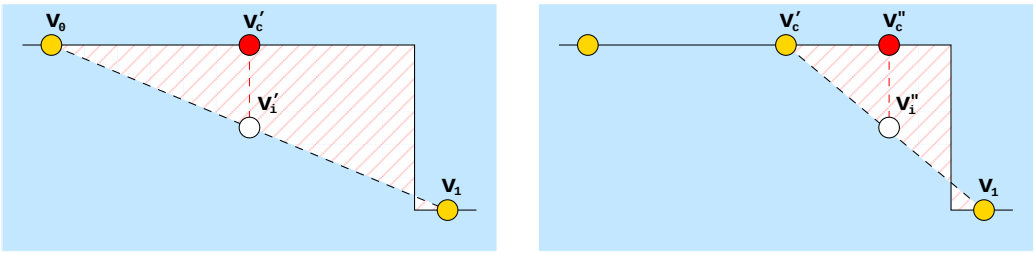


Figure 18: An example of an over-refinement induced by a cliff: Current vertices are shown in red for the (correct) position if inserted, and in white for the (interpolated) position if not inserted.

Given a position function $p : \mathbb{R}^2 \mapsto \mathbb{R}^3$ that translates vertex positions of the coarse mesh into world space[3], two edge spanning vertices $v_0, v_1 \in \mathbb{R}^2$ and a constant height error ε, the decision is usually based on the following equation:

$$\left\| \frac{p(v_0) + p(v_1)}{2} - p(\frac{v_0 + v_1}{2}) \right\| < \varepsilon \tag{4}$$

If the vertex to be inserted does not decrease the terrain approximation error about at least ε, it is not inserted. This approach works very well except for (near) vertical faces. Consider the situation in figure 18: In this example, the vertices v'_c and v''_c are correctly inserted into the mesh to more exactly approximate the given terrain (solid black line).

The problem in this example is that the error measured by equation 4 hardly converges, resulting in more and more vertices being inserted in each recursion step. In the worst scenario, the refinement never stops, creating degenerated triangles.

There are two ways to avoid degenerated triangles in cliff areas: Instead of measuring the difference between the correct and interpolated vertex positions, it would be more exact to estimate the area between the coarse mesh and the mesh containing the newly inserted vertex. The area before inserting v'_c is shown as the streaked area to the left side of figure 18, while the one after the insertion is shown to the right. Applying this refinement decision results in a converging error, even in cliff areas.

To even estimate this area in practice, it is necessary to calculate many height values along each triangle's edges, which would result in a drastic performance loss.

[3] In the most simple case, this function merely reads the height from a map, uses it as z-value for the vertex and leaves the x and y coordinates untouched

Thus, if performance is important for an application, it can be more appropriate to simply reduce ε, or the number of recursions for height maps causing problems. This constitutes the second way of avoiding degenerate triangles in areas with near vertical faces.

4.5.3 Oblique triangles

As described in section 4.2.1, RAG can perform local tesselation decisions for every triangle while maintaining a free choice of the refinement decision being applied. This is bought dearly for the guarantee of receiving only right-angled triangles as in Thatcher Ulrich's algorithm. As already stated, oblique triangles can not be prevented without restrictions to the accuracy of the rendered terrain. However, for many applications, accuracy is the most important characteristic of a terrain rendering algorithm. Therefore, this section describes the effects of oblique triangles and how to reduce their number.

Right-isosceles triangles have the advantage that their area linearly decreases with the length of their edges. The oblique triangles generated by RAG do not feature this property, especially when an already oblique triangle is split along its shortest edge repeatedly, making it a degenerate triangle. Such degenerate triangles (having immensely differing edge lengths and very little to no area) often induce rendering artifacts as textures cannot be correctly interpolated along them.

Figure 13 on page 32 presents all possible tesselations of RAG and Ulrich's algorithm, with the problematic ones coloured red. This concludes that three out of eight possible tesselations result in oblique triangles. Unfortunately, it is not possible to give an exact estimation of how many oblique triangles will occur as the number of oblique triangles generated mainly depends on the given height map and how it is refined.

If it is desired that degenerated triangles (and thus rendering artifacts) are prevented as good as possible, RAG should be configured so that it needs very few recursion steps only. This way, each triangle is refined more seldomly, making oblique triangles degenerate more seldom.

The number of required recursions for a given accuracy depends on several settings for the RAG algorithm. The most obvious approach is to manually reduce the number of recursions applied to a mesh. However, this also reduces the accuracy of the resulting terrain, eliminating this approach for many applications.

Another approach to reduce the number of recursions, that does not have an impact on the mesh's accuracy, is to increase the resolution of the basic mesh that is fed into the pipeline. For example, increasing the number of

vertices along the sides of the coarse mesh by the factor 2 corresponds to doubling the number of refinement tests along each edge.

The most important way to reduce the number of degenerated triangles, however, lies in the error estimation algorithm that is applied to decide whether to split an edge. Scenarios such as detailed in the previous section 4.5.2 tend to be the most common reason for degenerate triangles. Employing a more accurate error estimation algorithm generally results in the refinement of each edge finishing earlier. Thus, edges are split less often, simultaneously reducing the number of degenerate triangles.

4.6 Error metrics

In the domain of terrain rendering, error metrics describe how exactly a rendered terrain approximates its given input data. For algorithms based on height maps, the reference data is a mesh that consists of one mesh vertex per pixel in the map. Such a mesh would extract every possible bit of information from the map, but would quickly exceed the available graphic memory even of modern graphic cards, not to mention the additional time it would take to apply a vertex shader to so many vertices.

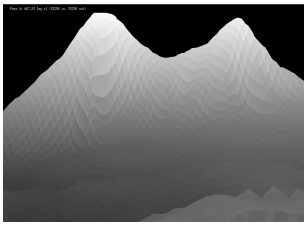
RAG approaches this problem by creating a coarse mesh that can be performantly set up, stored and displayed. It then recursively splits the edges between the vertices. To decide whether to split an edge, it tests if inserting a new vertex in the middle of the respective edge would decrease the pixel placement error in that position about, at least, a given threshold ε.

All vertices belonging to the initial coarse mesh and all additionally added vertices receive their height from the input height map. To ensure that no vertex erroneously receives a wrong height, texture minification and magnification operations must be set to nearest neighbour interpolation. That way, the height of every vertex is reliably the one from the height map.

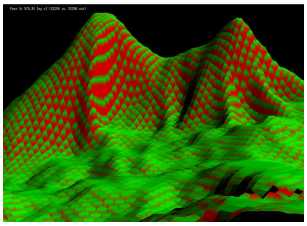
Based on this perception, the only errors that can occur with RAG are the ones between the mesh vertices and inside the triangles themselves. The latter errors can only be addressed indirectly with RAG, by increasing the coarse mesh's resolution. To reduce error between edge vertices, exacter error approximation algorithms – such as the one introduced in section 4.5.2 – must be applied.

An explorative approach to configure RAG to one's desire is to visualise the screen-space error by colour-coding the terrain. It is then possible to monitor the screen-space error directly while adjusting variables such as the input mesh's resolution, ε or even the refinement metric.

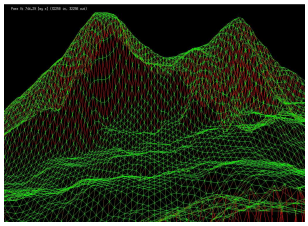
My implementation of RAG features this visualisation, as can be seen in



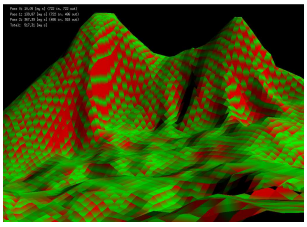
(19.a) 128x128, 0 passes, height map



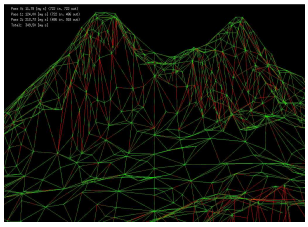
(19.b) 128x128, 0 passes, pixel error



(19.c) 128x128, 0 passes, wireframe



(19.d) 20x20, 2 passes, pixel error



(19.e) 20x20, 2 passes, wireframe

Figure 19: 19.a is decorated with the underlying height map, all others show the screen-space error between 0 (green) and 10 or more (red) pixels. Upper row: Meshes with 128x128 vertices and no refinement applied. Bottom row: Basic mesh with 20x20 vertices and two recursions applied.

figure 19. Additionally, this figure demonstrates with how many passes RAG can approximate a given terrain. The height map in this example (with which the terrain is decorated in figure 19.a) is consciously chosen to be very coarse (128×128 pixels). That way, it is possible to create a reference mesh with the same resolution, representing the best possible terrain approximation (shown in figure 19.b).

Regions colored red mark high pixel-space errors. As already noted, the height map in this example is very coarse. Combined with the values of the height map being interpolated by the nearest neighbour method, red areas are even visible in the rendering of the reference mesh.

Figure 19.d shows a rendering of the same scene based on a coarse mesh of merely 20×20 vertices that is refined with RAG in two recursion steps. Note that this version achieves a similar screen-space error as the reference mesh, with much fewer triangles (visible in the wireframe renderings of the respective images).

4.7 Parallel research

In February 2008, a paper entitled "Dynamic Mesh Refinement on GPU using Geometry Shaders" [14] was released as work in progress by Haik Lorenz and Jürgen Döllner of the University of Potsdam. Their work is based on the GAMeR approach [2] that was detailed in section 2.5. Unlike GAMeR, it utilises the features of geometry shaders to further reduce CPU load.

Similar to the process of its predecessor, this new approach relies on pre-calculated patterns encoded in barycentric coordinates that are stored in the graphic card's memory. During runtime, they replace coarse triangles with a fine mesh covering the same area. Instead of only rendering a single pattern at once, the whole coarse mesh is initially stored in graphic memory. Then, it is recursively refined using the said patterns directly in the graphic pipeline.

A difference to GAMeR concerning the patterns is that with this algorithm their storage cannot be optimised by indexing. GAMeR always renders one pattern that replaces exactly one triangle at a time. As Lorenz and Döllner's approach stores and renders an entire mesh at once, using indices on the pre-calculated patterns can result in conflicts with wrongly reused cached vertex shader results.

Preprocessing

Additional to the patterns, there are more preparations that must be met until the actual algorithm starts. First off, the vertex buffer with the pattern vertices needs an additional access table supporting addresses into the

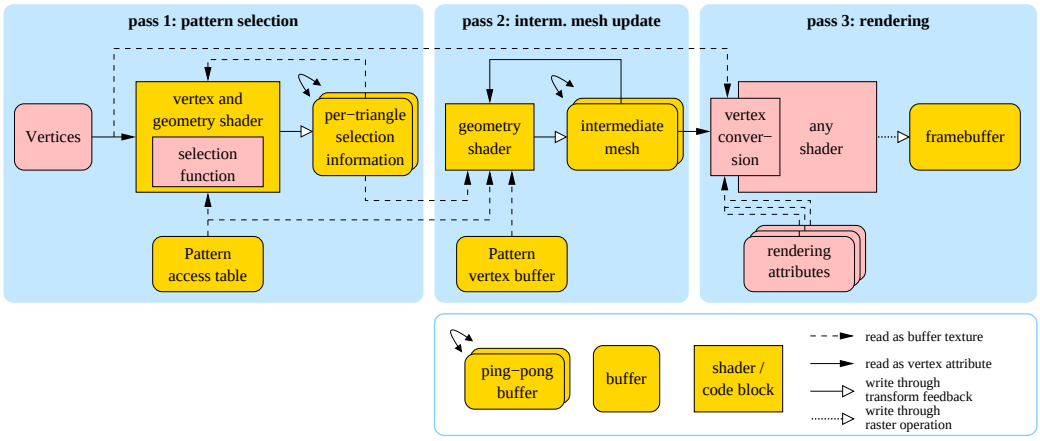


Figure 20: Outline of Lorenz and Döllner's algorithm. The pink boxes mark application dependant parts.

pattern buffer as the patterns have differing sizes.

Four buffers need to be set up that will be used as transform feedback targets. Two of them are intended for the first pass, the other two for the second pass. The buffers for each pass will be used alternately as input and output for the corresponding geometry shaders.

As a last step, the initial mesh needs to be converted to the format of the second pass results and stored as a buffer in graphic memory. Each vertex needs to consist of barycentic coordinates, the ID of the triangle and a sub-triangle ID. The latter one is redundant in this initialisation step, but still necessary for the geometry shader to work properly.

Frame-wise updates

The purely GPU-based refinement process is divided in three passes that are outlined in figure 20, which is directly taken from the paper. Pass 1 – the "Pattern Selection" – is in charge of selecting a pattern for each coarse triangle. As for GAMeR and RAG, it is important for this algorithm to keep the refinement decisions edge based, thus avoiding cracks in the mesh. Being fed with the needed vertex attributes, such as vertex position and normal, a geometry shader emits the tesselation information with three values: The index of the selected pattern p, the number of required sub-triangles s_r and the number of available sub-triangles s_p from the previous frame. The geometry shader's output is captured via transform feedback and the rasterisation stage is discarded.

Pass 2, called "Intermediate Mesh Update", takes the previous pass'es

triangles as input. In the first recursion, this coincides with the initial converted mesh that has been copied to graphic memory. By using 1D buffer textures, the geometry shader reads the results of the first pass with the help of each triangle's ID. Then, it emits the vertices from the pattern pool – again encoded in barycentric coordinates – and augments them with the original mesh's triangle ID and a new sub-triangle ID. The results are either captured in a vertex buffer with the help of transform feedback and further refined by another loop of pass 2 or they are fed to the next pass.

The last pass named "Rendering" essentially does what its name indicates and is mostly application dependant. All that is left to do is to convert the barycentric coordinates of the intermediate mesh from pass 2 into cartesian coordinates so that they can be applied as usual. This is done by using the triangle IDs that are encoded in the intermediate mesh to access the original mesh and interpolate its values. After that, any valid combination of shaders can be applied to the refined mesh.

4.7.1 Comparison

Lorenz and Döllner's approach is very similar to mine, RAG. Both algorithms perform recursive triangle refinement on the GPU using geometry shaders with the restriction that tesselation decisions must always be kept edge-based to avoid cracks in the mesh. Yet, there are some differences that favour one or the other depending on the requirements of the application.

Probably the most essential difference between the two approaches is that Lorenz and Döllner's algorithm can be configured as to the maximum number of output refined triangles t_r per coarse triangle t_c. RAG however only supports a fixed refinement ratio of $\frac{t_r}{t_c} = 4$. Interestingly, Lorenz and Döllner report in their paper that they chose a value of $t_r = 4$ to achieve the best performance and allow for a fast pattern growth.

The following shows the advantages and disadvantages that RAG has in comparison to Lorenz and Döllner's algorithm:

Disadvantages of RAG

- Meshes refined with RAG potentially feature more oblique triangles. The reason is that the refinement patterns used in Lorenz and Döllner's approach can be precalculated to include as few oblique triangles as possible. Depending on the combination of refinement depths within a pattern, they are not completely avoidable, but as their algorithm potentially needs less recursions to reach the same refinement, the oblique triangles degenerate more seldomly.

- Lorenz and Döllner's approach needs less refinement calculations, because it can have a higher $\frac{t_r}{t_c}$ ratio than RAG, thus producing more refined triangles per refinement calculation. This leaves more time to calculate the refinement in a more complex and exact way, like sampling more height values along each edge.

- RAG does not operate on barycentric coordinates, but directly on the coarse mesh. To support terrain synthesis, texture interpolation or similar, texture coordinates must be supplied to the mesh. In Lorenz and Döllner's approach, texture coordinates can be generated on the fly by interpolating the barycentric coordinates of each vertex. However, this only works for certain applications that need local texture coordinates for each triangle, as barycentric coordinates can only generate triangle-relative information.

- RAG needs an input mesh with sorted vertices. Otherwise, the vertices need to be ordered within the shader, which is possible, but slows the process down tremendously. For most applications, this is not really a disadvantage but more of an inconvenience as the vertices can be generated in any order initially, reducing the work to merely adapting the construction method of the coarse mesh.

Advantages of RAG

- RAG uses less buffers than Lorenz and Döllner's algorithm. It only needs two transform feedback buffers, as opposed to four, and it does not need a pattern vertex buffer or access table. This also results in an easier understanding of the algorithm and less required management regarding the correctly bound buffers and shaders.

- If a value of $t_r = 4$ is used for Lorenz and Döllner's approach, RAG reaches the same refinement with less passes because no additional pattern selection pass is required. This saves a lot of time for the graphic card and minimises load on graphic card memory transfer.

- RAG requires less preparations before it can be applied. The creation of the refinement patterns for the other algorithm is a one time job, but it still needs to be done once. Also, RAG does not require a conversion of the input mesh to barycentric coordinates, it just operates directly on the coarse mesh.

- A very important advantage of RAG is that it is actually capable of discarding triangles completely. Lorenz and Döllner's approach merely

keeps a coarse triangle as it is, but it can never remove it from the pipeline. RAG, however, can skip hidden triangles and reduce the number of triangles that need to be handled in subsequent passes. Additionally, the required size of the transform feedback buffers shrinks with this proceeding.

- When using a high $\frac{t_r}{t_c}$ ratio, Lorenz and Döllner's approach can come across a situation in which the hardware limit of 1024 floats of output is exceeded. In that case, their algorithm can introduce temporary cracks which require attention. RAG however does not face such problems, as it always has a maximum output of four refined triangles per coarse triangle which never exceeds this hardware limit. This is probably a rather insignificant disadvantage as the hardware limit is potentially raised in one of the next generations of graphic cards, while Lorenz and Döllner state that a smaller $\frac{t_r}{t_c}$ ratio improves performance anyway.

4.7.2 Conclusion

If Lorenz and Döllner's results are truly correct and the ideal number of emitted triangles per coarse triangle is four, then RAG is most probably the better solution for all applications that require speed over minimally better mesh topology because it reaches the same results with less rendering passes and less GPU memory consumption.

On the other hand, the greater t_r is chosen, the slower oblique triangles degenerate, resulting in a better mesh topology, because less recursions are needed. If an application requires good mesh topology with as few degenerated triangles as possible, Lorenz and Döllner's approach might be favourable because triangles can degenerate faster with RAG.

It should be noted that the chosen value of $t_r = 4$ in Lorenz and Döllner's approach might not be the best choice for future graphic cards or even graphic card drivers. In their paper, they remark that keeping the output of each geometry shader implication as small as possible increases the algorithm's performance. If this should change with future graphic cards or even drivers, their algorithm could prove to be better than RAG as it would produce less oblique triangles while requiring less recursions to create an equally refined mesh.

5 Results

This section demonstrates the practical results of my research. Firstly, my implementation of the introduced RAG algorithm is presented, followed by accuracy and performance measurements recorded with this very implementation.

5.1 Implementation

The program discussed in this and the following section, has been implemented on a PC with an AMD Athlon 64 X2 Dual Core 4200+ processor, 1 Gb of main memory and a GeForce 8800 GTX graphic adapter. The operating system was Gentoo linux and programming was done using C++ (gcc 4.1.2) and OpenGL. I always updated to the newest nVidia drivers as soon as they were available. The latest version applied was 169.12.

Before going into detail on my implementation, it should be noted that the program is only intended for research purposes to test RAG's characteristics and is not a full-featured application. No automatic adjustment of RAG's settings is performed and the program still contains inconveniences which should not persist in a non-scientific environment.

For instance, navigating within the terrain is circumstantially done by adjusting combox box and slider values instead of employing a mouse interaction. There is no handling for large textures yet, so only a single height map can be rendered at once and must fit completely into graphic memory. Also, depending on RAG's settings and the number of recursions applied, it can happen that the transform feedback buffers incur an overflow as the size of these buffers is not dynamically adjusted.

Both tesselation approaches introduced in section 4.2.1 are supported by my RAG implementation. They will be referred to as "shader variant 1" for the shader that outputs only right-isosceles triangles while sacrificing accuracy and "shader variant 2" for the shader that outputs partially-oblique topology.

5.1.1 Graphical user interface

Qt serves as a window toolkit to configure the rendering settings during runtime. My implementation supports two different views on the terrain: Initially, the viewer sees the whole terrain from a distance and can move, zoom and rotate the terrain with the mouse. In addition to the terrain, a perspective view-frustum and a viewer position are drawn, which both are separately adjustable by the graphical user interface. When using a

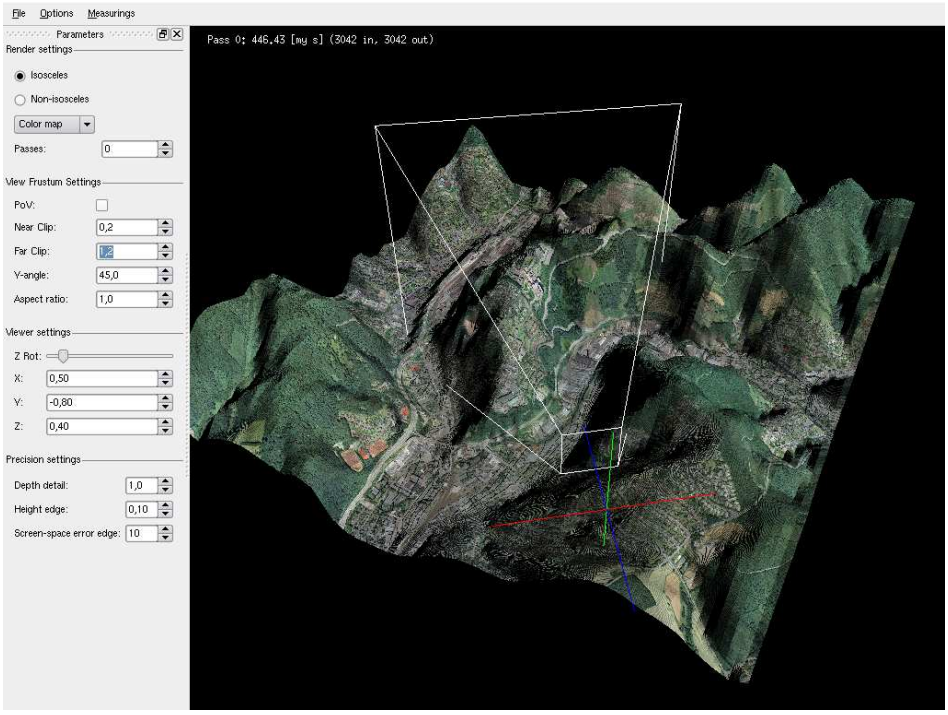


Figure 21: A screenshot of my RAG implementation with a height map of Siegen.

switch, the view changes so that the terrain is displayed as seen through the configured view-frustum.

The rendered terrain has four different colouring options. As a default, each pixel is coloured with its screen-space height error ranging from green, for no error, to red which indicates a height error of 20 or more pixels. The other options are to show the underlying height map the terrain is created from, the normal map that is generated from it, or a separately eligible colour map. Of course, the height map is also selectable through the user interface.

Three toggles select whether to show the terrain as wireframe, to render it flat (by ignoring the height values from the height map and setting all heights to 0) and to enable or disable shading, respectively. A dialog offers options to configure the terrain grid: The number of vertices in width and height of the coarse mesh is adjustable as well as the distance between the vertices.

The last parameter is a constant factor for the heights interpolated from the height map. With these parameters, it is possible to scale the rendered terrain to the wishes of the viewer. This is especially important as all heights interpolated from the height map are transformed so that they range from 0 to 1, which allows for a unified processing of all height maps. With the

options itemised above, it is still possible to receive a correct width-height ratio for the terrain, while always getting an immediate display of the terrain data.

5.1.2 Internals

When switching between shader variants 1 and 2, just a different geometry shader is bound. The shaders take the same input and are handled completely the same from the point of view of the CPU. The number of passes applied can be selected directly in the user interface. If a number of zero passes is selected, a shader without a geometry part is bound so that the initial mesh is rendered without being tesselated at all.

The two shaders have different precision settings that can be set during runtime. Shader variant 2 determines the tesselation of each triangle based on equation 4 on page 40. The only difference is that for my implementation $p : \mathbb{R}^2 \mapsto \mathbb{R}^2$ holds, as it determines the error in screen-space. This screen-space error can be adjusted with a unit of pixels during runtime.

Being based on pure height differences, shader variant 1 uses two other variables: The first determines the height difference in object space between two adjacent vertices that enforces an edge to be split. Ranging between values 0 and 1, the depth detail describes the diminishing weight of the height difference with growing distance to the viewer. A value of 0 means that the height difference is linearly reduced from its set value at the near clipping plane to 0 at the far clipping plane, while a value of 1 implies that the height difference setting is equal anywhere on the terrain.

When the rendering routine is called, the viewing transformations are calculated first, then the needed vertex and texture coordinate buffers are bound. As already stated in section 4.2.1, the vertices sent to the graphic card must either be assumed constant or variable, a mixture of both is very difficult to implement while keeping the mesh topology consistant. I have chosen to see them as constant, so the vertex buffer I bind is a pre-created one with the vertex heights already interpolated from the height map.

With each pass, the vertices for the active shader are alternately read from one vertex buffer and stored in another with the help of transform feedback. Except for the last one, all passes are aborted before the rasterisation stage. Object- and viewing-transformations are only applied to the last pass, otherwise they would accumulate.

To guarantee right triangulations (for shader variant 2 at least in 5 of 8 cases), the geometry shader needs to know which edge represents the hypotenuse of the currently processed triangle. Otherwise, the order in which the triangulation needs to be built is not definite. Figure 22 demonstrates such

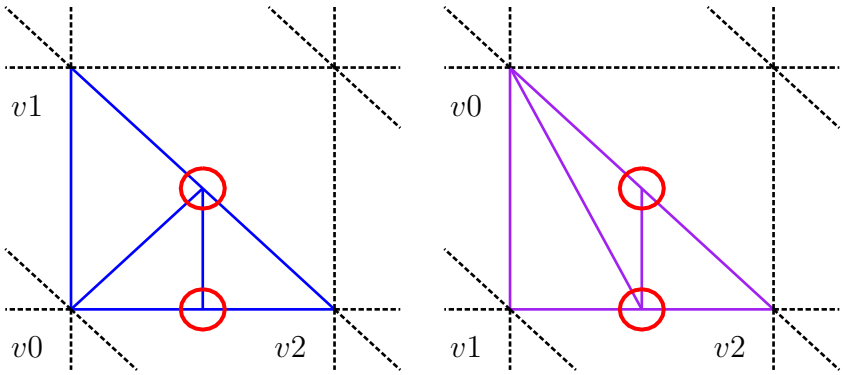


Figure 22: Two different triangulations for the same triangle. Both are created with the same triangle strip $\{v1, v0, \frac{v1+v2}{2}, \frac{v0+v2}{2}, v2\}$ but resulting in different tesselations due to a differing vertex order. The edges enforcing splits are marked with red circles.

a situation. Both presented tesselations are valid and emerge from the same triangle and vertex order. The difference lies merely in the order of the vertices, resulting in an oblique triangulation although a right-angled one can be performed.

I solved this problem by sorting the triangle vertices in advance: When the coarse mesh is created, it is assured that the vertex order is always clockwise and the hypotenuse is located between the second and third vertex of each triangle. Because the geometry shader does not accept triangle strips, only triangles, this is not a performance issue.

With the algorithm being recursive, the geometry shader needs to maintain this rule of triangulation so that the intermediate meshes always remain consistent. Principally, this is unfavourable because a geometry shader is capable of emitting triangle strips which are more efficient than a plain triangle structure. But as transform feedback converts triangle strips to triangles anyway, this is only a disadvantage against non-recursive algorithms, which can output a final tesselation in one step. Such algorithms, however, suffer from the limited output size of geometry shaders.

The two shader variants share common vertex and fragment shader parts and operate very similarly. On each call, the geometry shader first checks the current triangle's visibility by projecting each of its vertices into normalised device coordinates. If all three vertices triangle are located outside the normalised device cube, the triangle is discarded completely, occasionally resulting in a small gap in the corner of the rendering area.

After the visibility test has been passed, the triangulation is calculated as an OR-combined integer using the GLSL step[4] function which proved to compute faster during my tests.

The following pseudo-code illustrates the proceeding of shader variant 2, which determines the triangle tesselation by the screen-space error:

```
uniform float pixelEdge;
vec2 screenPos( vec3 pos ) {...}

int calculateScreenSpaceErrorTesselation()
{
  float error0 = length(
    screenPos( (PositionIn[0] - PositionIn[1]) / 2 ) -
    screenPos( PositionIn[0] ) - screenPos( PositionIn[1] )
  );
  float error1 = length( ... // PositionIn[1] + PositionIn[2];
  float error2 = length( ... // PositionIn[0] + PositionIn[2];

  return 0 |
    int(step( pixelEdge, error0 ))     |
    int(step( pixelEdge, error1 )) * 2 |
    int(step( pixelEdge, error2 )) * 4;
}
```

The function computes the pixel distance between a possibly inserted point in the middle of an edge and the linearly interpolated position that is used if no vertex is inserted. The screen-space difference arises from the height being set for the new vertex: It would be a height interpolated from the height map instead of the linear interpolation of the adjoining vertices' heights.

Having decided how to tesselate the triangle, the remaining part of the geometry shader needs to realise the actual tesselation. To do this with as little performance loss as possible, it is advisable to abstain from switch-case constructs as they are converted to if-else statements and cost a severe amount of calculation time. Instead, I used if statements combined with early returns to avoid all else statements.

[4]Quote from [12]: `genType` **step** `(float` *edge*`, genType` *x*`)`: Returns 0.0 if $x < edge$, otherwise it returns 1.0.

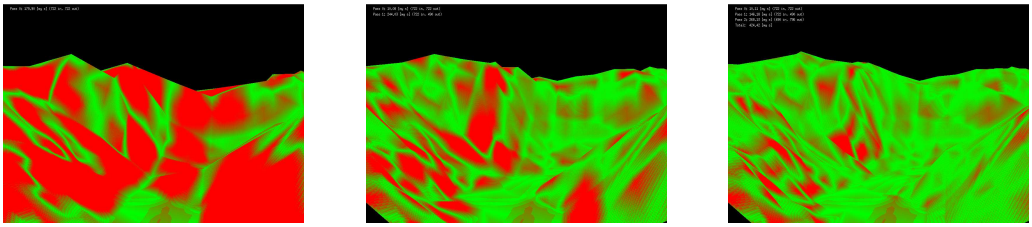


Figure 23: A scene rendered with 0, 1 and 2 recursion steps.

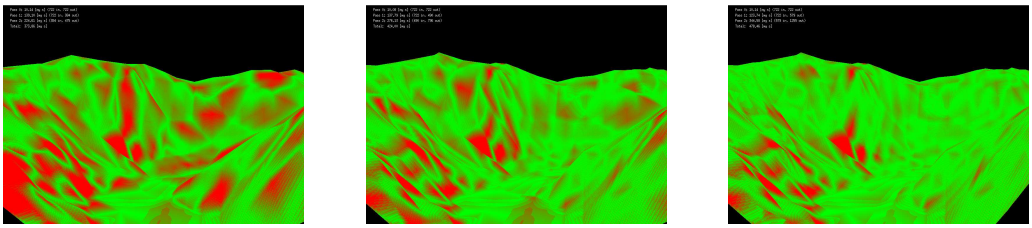


Figure 24: A scene rendered with RAG set to split an edge on a screen-space error of 20, 10 and 5 pixels.

5.2 Measurements

The following images have been produced with the RAG implementation introduced in section 5.1, using the oblique tesselation approach. These visualise the impact that the most important RAG settings have on the accuracy of the terrain. These settings are the number of recursions applied, the set screen-space error on which an edge is split and the resolution of the coarse input mesh.

All of these images show the same scene, with the terrain being colour-coded in green for little to no screen-space error and red for 20 pixels of screen-space error and more.

The sequence of images in figure 23 shows renderings with RAG set to split an edge on a screen-space error of 10 pixels for the midpoint of an edge. 0, 1 and 2 recursion steps are applied to a coarse mesh of 20×20 vertices,

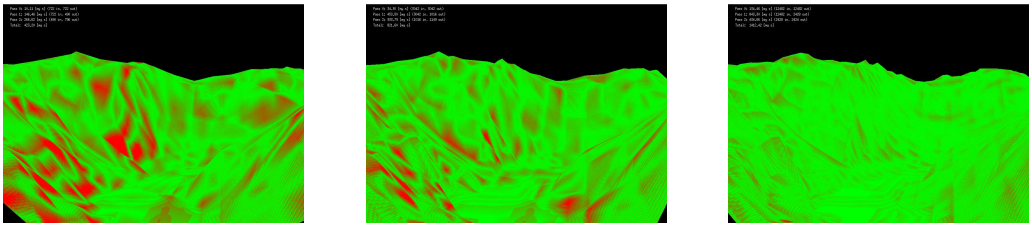


Figure 25: A scene rendered with a coase mesh of 20×20, 40×40 and 80×80 vertices.

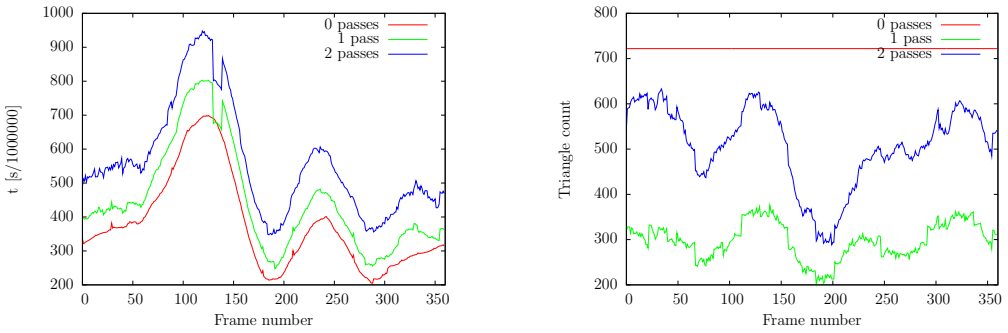


Figure 26: Performance measurements with the same settings as in figure 23.

respectively.

The image series in figure 24 displays the same scene with a fixed number of two recursion steps. The screenshots have been taken with RAG set to split an edge on a screen-space error of 20, 10 and 5 pixels, respectively.

The last image series in figure 25 again shows the same scene with a fixed number of two recursions and RAG set to split an edge on a screen-space error of 10 pixels. The coarse mesh has a resolution of 20×20, 40×40 and 80×80 vertices, respectively.

Now that I have visualised the impact of the most important settings on the accuracy of the algorithm, I will examine which influence they have on the algorithm's performance. To measure the performance, my RAG implementation supports a simple test: After selecting an option from the menu and a file to which the results of the test should be written, it rotates the camera around its current position in 360 steps with each step representing a turn of one degree.

For each step, the scene is rendered and the program waits until the rendering is complete. Then, it stores the time the rendering has taken (measured with an OpenGL query) and the number of visible triangles. After the test is complete, the collected results are written to the given file.

For the upcoming measurements, the camera is placed at the center of the map, just slightly above 0 on the z-axis. The far clipping plane is set up to always exceed the terrain's extents. A height map of Siegen is loaded with an associated colour map and shading is activated. The other settings correspond to the ones of the figures from the accuracy tests. The figures 26 to 28 illustrate the rendering times and number of rendered triangles of the figures 23 to 25, respectively.

Analysing the measurements, some estimations about the typical behaviour of RAG can be made. Note that these hold for the applied refinement

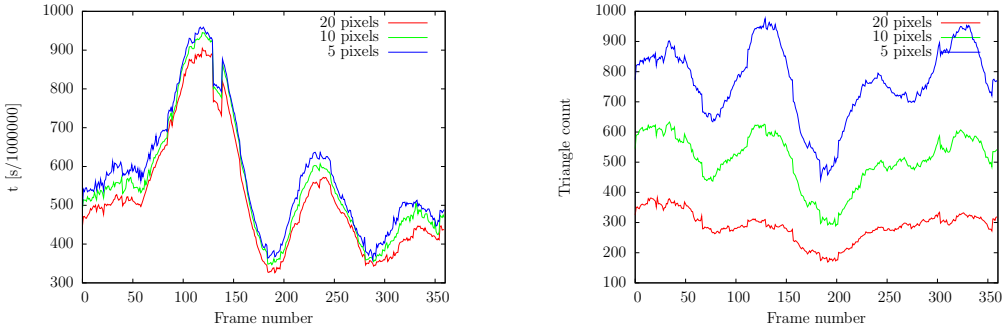


Figure 27: Performance measurements with the same settings as in figure 24.

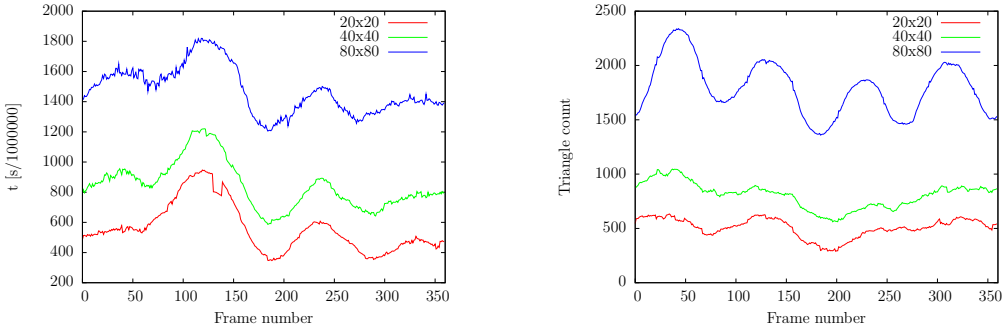


Figure 28: Performance measurements with the same settings as in figure 25.

decision of an edge midpoint's screen-space error and can differ for other error approximations:

- Settings improving the mesh's accuracy always result in higher computation time and a higher number of visible triangles.
- Increasing the number of passes and reducing the screen-space error which forces an edge to be split quickly increases the mesh's accuracy while only requiring insignificantly longer to compute. Unfortunately, these settings stop adding necessary detail to the mesh just as quickly. Some areas with high inaccuracy remain even after more recursion steps and a very small screen-space error set to split an edge.
- As expected, increasing the coarse mesh's resolution reduces the mesh's error constantly, while quickly increasing the rendering time and the number of visible triangles.

6 Conclusions

During this work, I have reviewed some of the most important terrain rendering algorithms that have been developed during the past years and have detailed the new features of Shader model 4. Employing these new features, I have developed a new GPU-based refinement algorithm named RAG and have detailed on its requirements, proceedings and restrictions.

Analysing RAG, I have discussed problems which occur when using this algorithm and have compared it to a similar approach that has been developed in parallel. Finally, I have presented my research implementation of RAG as well as accuracy and performance measurements recorded with the implementation.

Especially the comparison to Lorenz and Döllner's approach has revealed that RAG can compete with current refinement algorithms. However, RAG only offers a certain tradeoff between accuracy, performance and visual quality.

As section 4.5.2 has shown, better error estimation algorithms improve the resulting mesh's accuracy while increasing calculation time. Nonetheless, only errors along an edge can be considered for any error approximation applied, as cracks in the terrain would be introduced by considering triangle-based information. This is a common characteristic of GPU-based refinement algorithms, making these approaches inferior for applications that require accuracy above anything else.

Applying a faster and less accurate error approximation, the algorithm gains speed, but generates many oblique and oftentimes even degenerate triangles. These induce rendering artifacts if too many recursions are applied, thus decreasing visual quality.

Following the conclusions of the previous section, I recommend RAG for applications predominately requiring an interactive terrain rendering, such as video games and virtual reality environments. Due to its high performance and the possibility to increase detail independently from the CPU (as described in section 4.2.2), it is perfectly suited for delivering steady frame rates for these kinds of application.

Environments such as digital land surveying applications or similar, that rely on perfectly approximated height maps would probably not accept any detail being missed. For these, other approaches should most probably be favoured.

In the introduction, I noted that the development of terrain rendering algorithms is far from being utlimately explored. Even with the introduction of Shader model 4 and its new geometry shader stage, this statement still

holds. Shader model 4 offers the refinement of terrain data directly in the rendering pipeline, even enabling completely CPU-independent refinement approaches such as RAG.

However, with the current state of development, the desire for exactly approximated terrain surfaces can not be fulfilled as refinement decisions for triangles are too restricted and thus respecting all detail cannot be guaranteed.

6.1 Future work

RAG is fast, easy to understand and implement, but it still produces adverse topology in certain situations. That is why the most important target for an improvement of RAG should be to produce right-angled triangles only. This seems to be one of the biggest problems of all shader-based refinement algorithms at present, so I am excited how and when this problem is going to be solved.

Another nice feature would be to find a way of increasing the ratio $\frac{t_r}{t_c}$ of refined triangles per coarse triangle without having to use patterns like GAMeR or Lorenz and Döllner's approach. If there was an algorithm that could produce a refined triangulation without an access table and a pattern buffer, it would be very interesting to see whether the performance of RAG would increase through less refinement passes. However, Lorenz and Döllner have reported a decreasing performance for their algorithm with increasing refinement ratio, leaving the results of this feature less promising.

Finally, it would be interesting to observe how RAG behaves with better refinement metrics. The first modification to my proposition would be to add more sample points on each edge, thus improving the error estimation. Maybe it would make sense to move the inserted vertex along the edge, to the sample point with the highest error. It would be exciting to see which impact on the mesh topology such an adjustment would have.

References

[1] Arul Asirvatham and Hugues Hoppe, *Terrain Rendering using GPU-based Geometry Clipmaps*, GPU Gems 2 (2005).

[2] Tamy Boubekeur and Christophe Schlick, *GPU Gems 3 - Generic Adaptive Mesh Refinement*, ch. 5, pp. 93–104, Addison-Wesley, 2007.

[3] P. Cignoni, F. Ganovelli, E. Gobetti, F. Marton, F. Ponchio, and R. Scopigno, *BDAM - Batched Dynamic Adaptive Meshes for High Performance Terrain Visualization*, Eurographics 2003 (2003).

[4] Wikipedia contributors, *Triangulation (Geodäsie)*, `http://de.wikipedia.org/wiki/Triangulation_%28Geod%C3%A4sie%29`, 13.12.2007.

[5] Wikipedia contributors, *Shader*, `en.wikipedia.org/wiki/Shader`, 23.11.2007.

[6] Wikipedia contributors, *High Level Shader Language*, `http://en.wikipedia.org/wiki/High_Level_Shader_Language`, 4.11.2007.

[7] Carsten Dachsbacher and Marc Stamminger, *Rendering Procedural Terrain by Geometry Image Warping*, Eurographics Symposium on Rendering (2004).

[8] Ben Discoe, *Virtual Terrain Project*, `www.vterrain.org`, 15.11.2007.

[9] Mark Duchaineau et al, *ROAMing Terrain*, IEEE Visualization '97 (1997).

[10] Mark J. Kilgard et al, *Documentation of the OpenGL ARB_pixel_buffer_object extension*, `http://www.opengl.org/registry/specs/ARB/pixel_buffer_object.txt`, 08.12.2004.

[11] Michael Garland and Paul Heckbert, *Surface simplification using quadric error metrics*, SIGGRAPH 1997 Conference proceedings (1997).

[12] John Kessenich, *The OpenGL Shading Language*, `http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf`, 07.09.2006.

[13] Barthold Lichtenbelt, Pat Brown, and Eric Werness, *Documentation for OpenGL NV_transform_feedback extension*, `http://www.opengl.org/registry/specs/NV/transform_feedback.txt`, 04.02.2008.

[14] Haik Lorenz and Jürgen Döllner, *Dynamic Mesh Refinement on GPU using Geometry Shaders*, Proceedings of the 16-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2008, February 2008, to appear.

[15] Frank Losasso and Hugues Hoppe, *Geometry clipmaps: Terrain rendering using nested regular grids*, SIGGRAPH 2004 (2004).

[16] Microsoft, *Resource Types (Direct3D 10)*, `http://msdn2.microsoft.com/en-us/library/bb205133.aspx`, 4.12.2007.

[17] Thatcher Ulrich, *Continuous LOD Terrain Meshing Using Adaptive Quadtrees*, 28.02.2000.