FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Improved Persistent Grid Mapping

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Peter Houska
Matrikelnummer 9907459

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Prof. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Daniel Scherzer

Wien, 26.11.2019

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Improved Persistent Grid Mapping

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Peter Houska
Registration Number 9907459

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Prof. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Daniel Scherzer

Vienna, 26.11.2019

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Peter Houska
Ferrogasse 80/9, 1180 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

I am grateful that I had the privilege of enjoying the creative and intellectually engaging environment that prevails at the Vienna University of Technology, especially at the Research Unit of Computer Graphics.

The feedback of my supervisors Michael Wimmer and Daniel Scherzer was instrumental during the development of the key ideas, and also helped improve the clarity of the text later on. I was lucky that I could tap into the knowledge of two of the leading scientists in the field of real-time temporal coherence.

I dedicate this thesis to my family. Mira, Jan and Miriam, you're the best!

Abstract

We propose a novel heightmap-based terrain rendering algorithm that enhances the *Persistent Grid Mapping* (PGM) method. As in the underlying method, we cache a regular triangulated grid in video memory and use the GPU to project the mesh onto the ground plane each frame anew. Each vertex in the grid is then displaced according to the sampled heightmap value along the ground plane's normal vector. The perspective mapping of the grid results in a view-dependent, continuous level-of-detail approximation of the terrain dataset.

PGM is a simple and elegant terrain rendering algorithm, however, as the camera hovers over the terrain, projected vertex positions slide over the terrain. This leads to the underlying static terrain surface changing shape slightly from frame to frame. We address these swimming artifacts by introducing four improvements: tailoring the projected grid, which pushes most otherwise culled vertices back into the view frustum, redistributing grid vertices according to an importance function for more faithful mipmap selection when sampling the heightmap, local terrain edge search for vertices within a certain proximity to the camera, and exploiting temporal coherence between frames. While our algorithm cannot guarantee a maximum screen-space error, it nevertheless reduces PGM's inherent temporal aliasing artifacts considerably.

Kurzfassung

In dieser Arbeit wird eine Verbesserung des *Persistent Grid Mapping* (PGM) Verfahrens vorgestellt. Wie in der zugrundeliegenden Methode wird ein reguläres Gitter von Eckpunkten im Speicher der Graphikkarte abgelegt. Das Gitter wird für jedes zu berechnende Bild auf die Grundebene projiziert, wobei die Eckpunkte anschließend entsprechend den in einem Höhenfeld gespeicherten Werten, entlang der Normale auf die Grundebene verschoben werden. Die perspektivische Abbildung des Gitters resultiert in einer blickpunktabhängigen, kontinuierlichen Detailstufen Annäherung der Geländedaten.

PGM ist ein simpler und eleganter Geländevisualisierungsalgorithmus bei dem jedoch Probleme auftreten, sobald sich die Kamera über das Terrain bewegt. Da die projizierten Eckpunkte des Gitters frei auf der Grundebene zu liegen kommen, scheint sich das an sich statische Gelände leicht zu bewegen, bzw. zu schwimmen, wenn die Abtastungsfrequenz der Höhendaten nicht ausreichend hoch ist. Es werden vier Verbesserungen vorgestellt, die sich dieses Problems annehmen: Anpassung des zu projizierenden Gitters an die Kamera um dessen Abtastinterval zu verkleinern indem keine Bereiche des Gitters an nicht sichtbares Gelände zu verschwenden, Verzerrung des Gitters entsprechend einer Gewichtungsfunktion, um exzessiver Gitterzellenstreckung und damit verbundener schwieriger mipmap Selektion zum Horizont hin entgegenzuwirken, lokale Suche und darauffolgende Verschiebung von Abtastpositionen hin zu lokalen Geländekanten, sowie die Nutzung von temporaler Kohärenz von aufeinanderfolgenden Bildern. Auch wenn PGM selbst nach Anwendung unserer Verbesserungen weiterhin keinen maximalen Pixelfehler im Bildbereich garantieren kann, so erreichen wir mit unserem Algorithmus dennoch eine deutliche Verbesserung der Bildqualität, vor allem durch die Verminderung der temporalen aliasing Artefakte die PGM innehat.

Contents

CHAPTER 1

Introduction

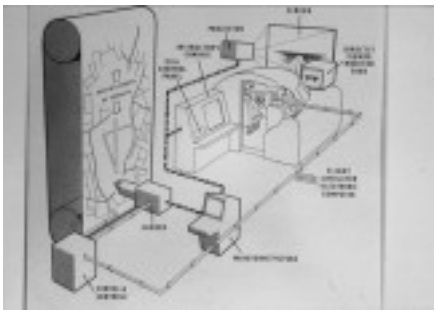
Displaying terrain has many applications in computer graphics, which range from scientific simulations and visualizations to video games and movies. Interaction in real time is often a critical requirement. While terrain rendering is a demanding task in its own right, the landscape is usually only one of many objects that need to be displayed. This implies that the respective rendering algorithms must be designed in such a way that they do not occupy all of the available system resources.

1.1 Origins of Terrain Visualization

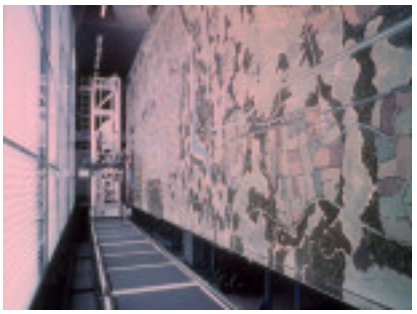
Looking back in time, we see that terrain visualization is especially linked to the simulation genre, as displaying terrain at real-time frame rates is naturally a key element for immersive simulations. Around 1970, first digital flight-, naval- and tank simulators were built for education and training. Most systems were specifically designed for this single application, and some even featured a multi-directional movable cabin.

Before digital systems emerged, simulator systems used rather simple visualizations, for example static panorama photographs mounted on a movable dome. In the 1950's a new system appeared, which had the scenery printed to maps that were mounted on a moving belt. A fixed optical probe was aimed at the map and rotated according to the simulated aircraft's orientation. The camera's picture was then projected onto a screen in front of the cockpit, thereby giving visual feedback to the pilot, cf. Figure 1.1(a). Around 1962 the moving belt model was replaced with a rigid model of the landscape with a freely movable camera. Another improvement was that the camera now transmitted color pictures, cf. Figure 1.1(b). This method for displaying scenery was successfully employed well into the 1970's.

The increasing performance of early digital computers made it possible for the General Electric Company to create the first simulator using computer-generated images for the space program. McDonnel-Douglas and Evans & Sutherland soon followed with their own flight-simulator systems. These early systems used vector displays, thereby producing crisp wireframe



(a) Moving Belt Model



(b) Rigid Model

Figure 1.1: Using a miniature map or model and a camera to mirror the simulated aircraft's view. In (a) the camera's position is fixed (apart from rotational movement) and translations are realized by the moving belt onto which the map is printed. In (b) a rigid model represents the scenery and the camera moves over the terrain, mirroring the simulated aircraft's global pose. Images courtesy of [42]

(a)

(b)

(c)

Figure 1.2: The CT5 simulator from 1981 by Evans & Sutherland. The movable cockpit is shown in (a), while (b) and (c) depict two snapshots from virtual flythroughs. Images are taken from the video available at `http://www.youtube.com/watch?v=06mbwNg1Vw4`

images. Soon, the creators of simulators switched to raster displays, as they could display anti-aliased lines faster [2]. Figure 1.2 exemplarily shows several screenshots from a flight-simulator system from 1981 with impressive visuals for the time. While not depicted in the picture, this simulator even supplies the pilot with a wide viewing angle.

Meanwhile, in the consumer market, computer games pushed the available hardware to the limits, accounting for progress in hardware development and subsequent computer price drops. The flight-simulator genre was especially popular in the '90s. Due to the lack of graphics accelerator cards, software rendering was the only available way to render three-dimensional objects, leaving little room for detailed scenes. Terrain on personal computers was therefore preliminarily still depicted as a flat plane with pyramid-like hills. Several alternative ways to render more convincing terrain in games were explored, which are hallmarked by Novalogic's Comanche and Appeal's Outcast. The approach of these games to terrain rendering was to cast a ray for each

(a) Gunship 2000 (1991)

(b) Comanche: Maximum Overkill (1992)

Figure 1.3: Comparing software polygon rendering to software ray casting in two computer games from the early '90s. Images are snapshots from videos from `http://www.youtube.com/watch?v=DaWoJhWpvIM` and `http://www.youtube.com/watch?v=51E_G7NCXVM`, respectively.

screen column and project the individual slabs onto the screen with the according color instead of rasterizing transformed polygons. Figure 1.3 compares Gunship 2000, which uses traditional polygon rendering, to the ray-marching technique of Comanche: Maximum Overkill.

The evolution of graphics hardware initiated a general shift from software- to hardware rendering of digital elevation maps. The employed algorithms mostly mirrored the graphics chips' capabilities of the time, at first merely taking advantage of accelerated rasterization, filtered texturing and depth-buffer support. The important aspect was that filtered textures could be applied to the polygons without sacrificing performance, as would have been the case with software rendering. Later, through hardware transform and lighting, basically the whole OpenGL fixed-function pipeline was implemented in silicon, enabling ever increasing triangle counts. Another paradigm shift was introduced with programmable vertex- and fragment shaders, which freed programmers from the predefined fixed-function pipeline, spawning fundamentally different techniques like terrain ray-marching as opposed to rasterizing polygonal approximations of the terrain surface. Nowadays, terrain rendering algorithms continue to profit from hardware features such as *vertex texture fetch* (VTF) and hardware tessellation, which both enable the reduction of system bus bandwidth requirements. Especially the ability to read from textures in the vertex shader considerably simplified mesh-storage data structures.

1.2 Problem Statement

While it is theoretically possible to represent the terrain as a single object that can be rendered with only one draw call, it is not feasible in practice for several reasons. For example, a huge *digital elevation model* (DEM) simply does not fit into a single vertex buffer. Even if it did fit into video memory, this approach would perform badly. Firstly, distant areas would be rendered at full detail unnecessarily, and secondly, the draw call would feed lots of excess vertex data into the graphics pipeline for parts of the landscape which are currently not in view. The prevalent approach to cope with these two problems is to subdivide the terrain into several subsections or patches. This subdivision enables culling invisible patches, and to represent patches that are

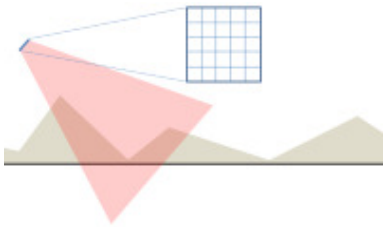
far away with fewer triangles. Approximating full detail patches with coarser geometry, and selecting the appropriate detail level at runtime is the task of *level-of-detail* (LOD) algorithms. Both view-frustum culling and LOD management help reduce the triangle count considerably.

One particularly elegant approach to solve both problems mentioned above, i.e., visibility and geometry simplification of distant terrain areas, is the *Persistent Grid Mapping* (PGM) method [32, 33]. Unlike the previously mentioned method of subdividing the heightmap's two-dimensional domain into a number of patches, PGM uses a single patch to represent the visible part of the terrain. Another notable difference is that instead of using one discrete detail level per patch, the level of detail varies continuously across this one patch in a view-dependent manner. In the context of PGM, the patch is referred to as the persistent grid. The key idea of PGM is to project the persistent grid onto the ground plane from the viewer's position. Each projected grid vertex is then displaced along the ground plane's normal vector according to the sampled heightmap value, cf. Figure 1.4 (a) through (d).

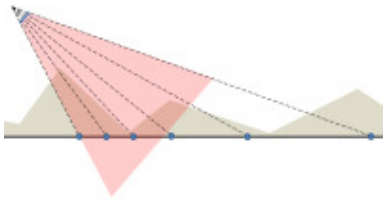
While the projection of the grid is performed from the viewer's position, using the viewing camera directly for the grid's mapping onto the ground plane would lead to problems. Since parts of the ground plane underneath the viewer's camera, yet outside of its view frustum, can still give rise to visible terrain due to the involved heightmap displacement, it is important that the projected grid covers this area as well. Further, we need to guarantee that each grid vertex is correctly projected onto the ground plane, even if the viewing camera is aimed towards the sky. The introduction of a second, auxiliary camera solves both problems. The auxiliary camera basically follows the main camera, yet it is always properly aimed at the ground plane, thus ensuring that the projection always yields a correct intersection in front of the camera. The two-camera setup also prevents missing terrain underneath the camera. Images 1.4 (e) through (h) illustrate how both issues are handled by the auxiliary camera.

PGM possesses many desirable properties, such as automatic view-frustum culling and watertight, continuous LOD by construction. It runs entirely on the GPU with minimal system bus traffic at runtime. As the triangle count of the static mesh does not change, the method guarantees a constant frame rate and pretty much leaves the CPU idle for other tasks. Last but not least, the algorithm is easy to implement.

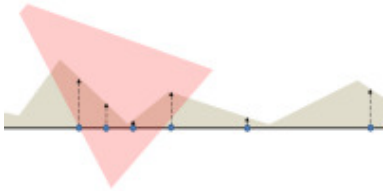
Since the vertex positions do not snap to the regularly spaced heightmap samples, the algorithm exhibits a number of problems, though, such as missing terrain features due to undersampling and the presence of swimming artifacts when the camera is animated. Both issues can be tackled by either increasing the grid's density or by accessing the heightmap at coarser mipmap levels, both of which are impractical. The first approach leads to extremely high polygon counts, while the second results in low-pass filtering of the heightmap, making high-frequency terrain features disappear quickly as projected vertices approach the horizon. Getting rid of the swimming artifacts in practice is thus a non-trivial task. Further, PGM in its basic form cannot guarantee a maximum screen space error. Regardless of the presence of the temporal aliasing artifacts and undersampling, we chose to base our algorithm on the PGM method, since in our opinion these disadvantages are outweighed by the algorithm's simplicity and elegance.



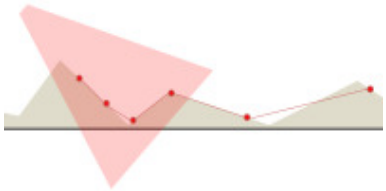
(a) Persistent grid before projection



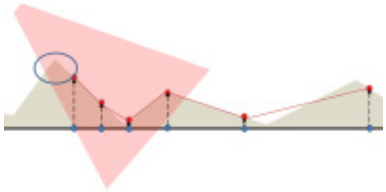
(b) Projecting the grid onto the ground plane



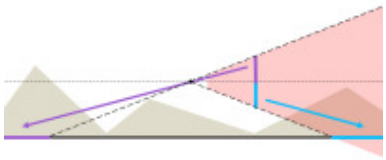
(c) Displacement



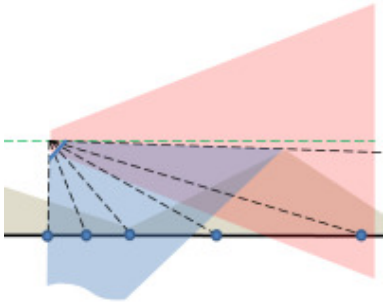
(d) Terrain surface reconstruction



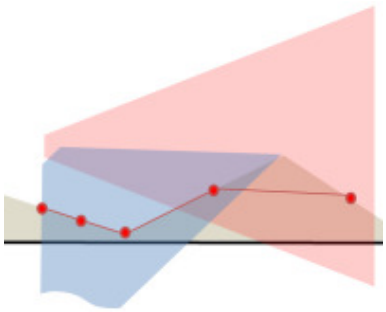
(e) Missed peak close to the camera



(f) Backfiring



(g) Projecting the grid from the auxiliary camera (in blue) instead of the main camera (in red)



(h) Terrain surface reconstruction

Figure 1.4: An overview of persistent grid mapping. The main camera is indicated by a red view frustum, the sampling camera by a blue one. Images (a) through (d) illustrate the steps involved in the PGM method to visualize the landscape. Note the missed terrain features due to undersampling in (d). Peaks close to the main camera are missed (e) if projection of the grid is performed by the same camera that is also used for rendering. Worse yet, if the camera is aimed above the horizon towards the sky, projection yields intersections behind the camera (f). Both issues are handled by projecting the grid from the properly aimed sampling camera.

1.3 Contributions

This thesis presents a novel technique for rendering heightfield-based terrains. The algorithm builds upon PGM [32,33] and the projected grid concept [30], keeping their advantageous properties while improving on their weaknesses. We rely on the grid's projection onto the ground plane, followed by heightmap-based displacement. The positive properties of PGM are thus all left intact. We concentrate on reducing the aforementioned swimming artifacts. Our method achieves this with a minimal impact on performance. As mentioned in the previous section, these artifacts can be somewhat lessened when the grid's sampling density is increased. However, performance drops are inevitable with the brute-force approach and we observed that towards the horizon the terrain still suffers from undersampling. Our contributions thus circle around the effort of making better use of the grid's resolution and increasing sampling density where needed by vertex redistribution, rather than increasing the grid's triangle count.

We also explore ways to apply *temporal coherence* (TC) methods to further reduce the temporal instabilites. In previous work, TC was applied successfully to remove similar flickering artifacts for example in shadow mapping [48]. *Image-Based Bidirectional Scene Reprojection* (IBTC) [65] extends the basic TC algorithm so that apart from past frames, future frames are taken into account as well. IBTC reconstructs intermediate frames with an image-space search algorithm. We will also sketch a new image reconstruction algorithm, which works well for static geometry.

The key contributions of our method can be summarized in four enhancements to the basic PGM algorithm, which are illustrated in Figure 1.5 (a) through (d). The main contributions are:

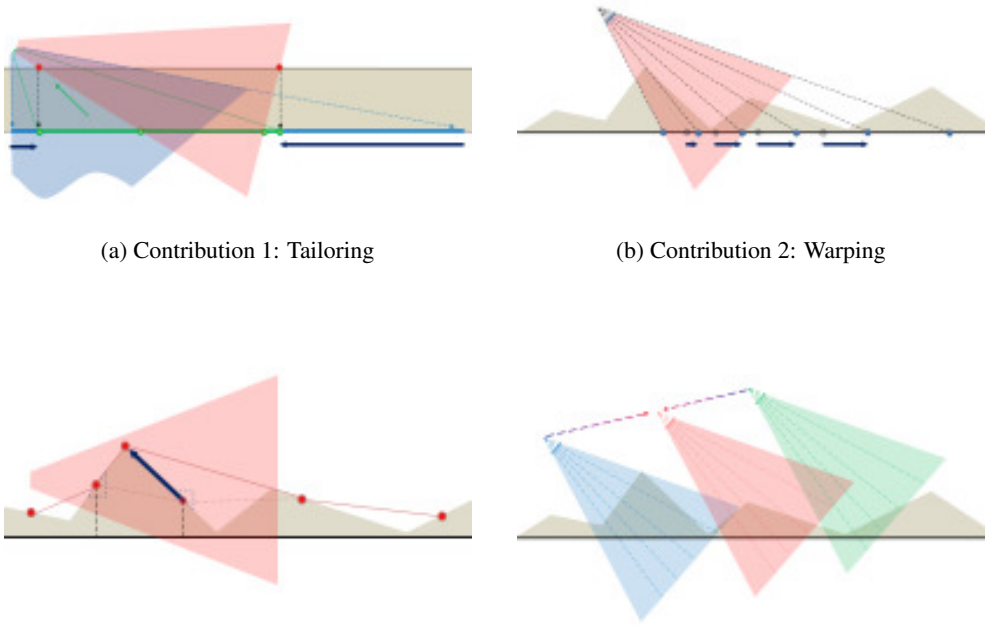
- Tailoring the persistent grid: we efficiently increase the sampling density of the projected grid by pushing vertices which would otherwise fall off the screen back into the main camera's view frustum.

- Importance-driven warping on the GPU: we redistribute grid vertices in order to counteract the excessive stretching of grid cells, which is caused by their projection onto the ground plane.

- Searching for local terrain edges: we adjust the position of vertices in the camera's vicinity depending on local terrain features. Whenever a terrain edge between two sampling positions is missed, the more distant vertex is moved onto the edge, which increases heightmap sampling fidelity.

- Exploiting temporal coherence.

The proposed improvements are orthogonal to each other. Quality can thus be traded for performance by applying only a subset of them. Since our additions still have the same hardware requirements as PGM, they can be easily incorporated into any system that uses the basic algorithm.



(a) Contribution 1: Tailoring

(b) Contribution 2: Warping

(c) Contribution 3: Local Edge Search

(d) Contribution 4: ExploitingTemporal Coherence

Figure 1.5: The four contributions proposed in this work. Dark blue arrows depict how vertices are redistributed in images (a) through (c). Image (a) illustrates that the grid does not need to sample the whole area that the auxiliary camera covers (light blue), but only the area which gives rise to terrain that possibly intersects the main camera's view frustum (green). Image (b) shows how warping redistributes the grid's vertices in order to counteract the stretching of projected grid cells as they approach the horizon. Image (c) demonstrates how vertices in the camera's vicinity are moved to sharp terrain edges which would otherwise be missed. Image (d) motivates that large parts of the terrain remain in view over the course of several frames so that we can profit from this information. This is especially true for static geometry like terrain.

1.4 Improvements over State of the Art

Our first contribution, *tailoring the persistent grid*, has its roots in Claes Johanson's water rendering algorithm [30], which performs basic shrinking of the grid to the main camera's view frustum. However, although the grid projection already performs automatic view-frustum culling, it is performed in a conservative way, so there is room for improvement. Johanson describes how to make better use of the grid's vertices, which is achieved by making adjustments to the sampling camera's transformation matrix. This adjustment essentially corresponds to scaling the grid on the sampling camera's view plane prior to projection. After projection of the adjusted mesh onto the ground plane, the grid thus still maintains a trapezoid shape. Our method refines this process and further minimizes the area that the grid covers on the ground plane by determining the intersection of the terrain volume with the main camera's view frustum. The resulting shape on the ground plane corresponds to a general polygon on the sampling camera's view plane. Fitting the grid to the absolutely necessary area only helps reduce undersampling artifacts, but can't completely remove them.

Our second contribution, *importance-driven warping*, is strongly related to a PGM-based algorithm by Löffler et al. [34], published in 2009. In their work they introduced the idea of warping the grid, prior to projecting it, to counteract excessive stretching of grid cells. The redistribution of grid vertices was calculated on the CPU. Our method builds on that idea, however, it performs the warping step efficiently on the GPU, without requiring any more advanced graphics cards features than render-to-texture and dependent texture lookups, which were both already available in 2009.

To our knowledge, neither the third, i.e., the *redistribution of near grid vertices* based on local terrain features, nor the fourth contribution, i.e., *exploiting temporal coherence*, have been considered in any other projected-grid related publication.

1.5 Structure

We start with an overview of related work and recurring data structures in landscape visualization in Chapter 2. While we propose a terrain rendering algorithm, we do not restrict ourselves to research in this specific field. Since we exploit temporal coherence, we also give a short overview of the state of the art in this currently highly active branch of research in computer graphics.

Chapter 3 constitutes the main part of this thesis. It explains our proposed algorithm, first presenting the basic PGM method and then each of our contributions. The chapter also contains a comparison of our algorithm to several other terrain rendering methods. Most of the presented techniques have been implemented in a single framework, so that we can collect meaningful numbers for a fair comparison of the various algorithms' performance.

Chapter 4 presents several result images comparing the effect of successively applying an ever growing subset of the proposed improvement steps. We conclude with Chapter 5, which gives a short summary of the properties of our method. Finally we suggest possible directions for future research.

CHAPTER 2

Previous Work

We start this chapter with a short summary of recurring terrain-geometry data structures, then we review *level-of-detail* (LOD) management in Section 2.1, and in Section 2.2 we present a frequently encountered basic screen-space-error estimation metric that guides LOD selection at runtime. In Section 2.3 we take a closer look at several often cited terrain-rendering algorithms, focusing on how they decompose the landscape into more manageable parts (so-called patches), how they create different detail levels for each patch, and how they later select the appropriate detail level for rendering.

Since the contribution of this thesis is the improvement of the *Persistent Grid Mapping* (PGM) [32] method, we postpone the discussion of PGM to the main chapter, where we contrast the base method with our proposed method. In the chronological order PGM would appear between "Terrain rendering using GPU-based geometry clipmaps" and "Seamless patches for GPU-based terrain rendering".

Note that we only deal with convex terrains, defined on planar carrier geometry, with no caves or overhangs. At the time of this writing, this constraint is not a severe limitation, since most other terrain-rendering algorithms fall into this class as well. Further, almost no large terrain dataset takes caves or overhangs into account, simply because such datasets are predominantly acquired from satellite scans or aerial photographs, which naturally consist of a single elevation value per sample point. Still, algorithms that are able to display concave landscapes are gaining momentum lately, and are made possible by the ever-increasing computational power of today's consumer-level computer systems. Terrain-rendering methods that handle concave terrain often rely on procedurally generated datasets, thereby avoiding having to store massive amounts of data.

Besides other terrain-rendering algorithms we also touch on temporal coherence and histogram pyramids. While neither temporal coherence nor histogram pyramids are usually used in terrain-rendering algorithms, they nevertheless play an important role in our proposed method, which we present in Chapter 3.

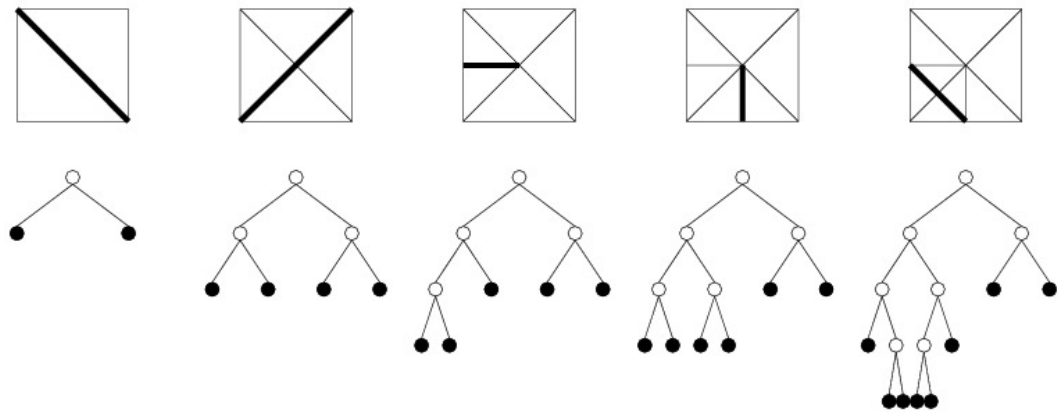


Figure 2.1: Binary tree corresponding to several triangle-bintree subdivision steps. Images courtesy of [55].

2.1 Data Structures

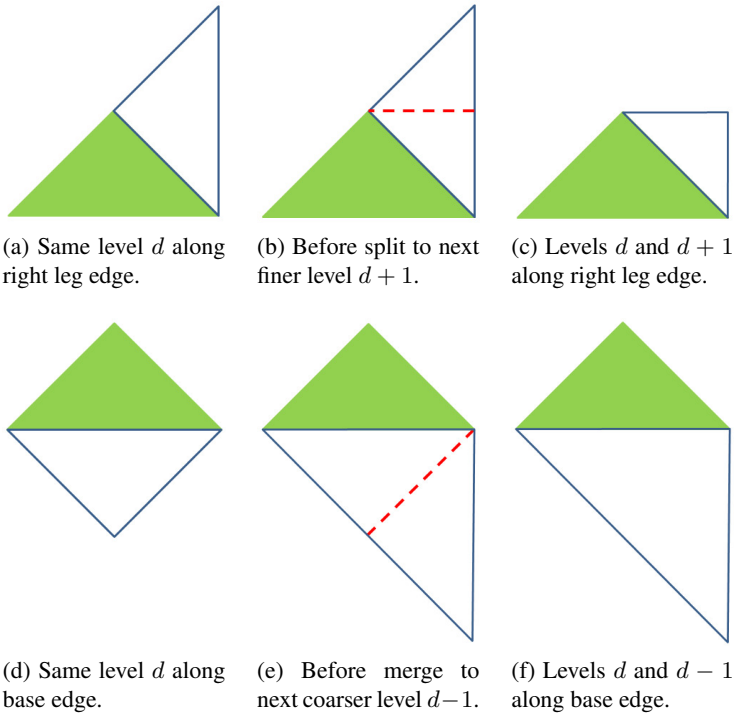
In this section we give a brief overview of data structures that are frequently encountered in the context of terrain-rendering, such as the triangle bintree, the quadtree, and the restricted quadtree.

2.1.1 The Triangle Bintree

The key geometric primitive in the triangle-bintree data structure is the isosceles right triangle. Subdivision starts with a quad that is split into two isosceles right triangles along one of its diagonals. Each of the two triangles is then split recursively into two triangles along the center of the hypotenuse, thereby forming similar triangles. Since the hypotenuse is the longest edge of the triangle, this subdivision scheme is also known as longest-edge bisection. Recursive splitting continues until a given subdivision criterion is met. In the context of terrain rendering, the subdivision criterion is usually the maximum-allowed distance of the current triangle-bintree mesh to the actual terrain dataset. The resulting leaf nodes represent the triangles that are actually rendered, and the whole tree forms a hierarchy of right triangles. Figure 2.1 shows how each node in the binary tree stands for a subdivided triangle. The regular structure of the triangle-bintree subdivision enables a direct mapping between triangle vertices and heightmap samples.

The triangle-bintree subdivision scheme creates a crack-free surface, since the method always splits two adjacent triangles at their shared hypotenuse. This edge-subdivision constraint leads to a specific configuration of subdivision levels of adjacent triangles. Triangles at level d have adjacent triangles at either the same level d or the next finer level $d + 1$ along their legs, and triangles at level d or $d - 1$ along their hypotenuse, cf. Figure 2.2.

Whenever a specific triangle in the hierarchy is refined by being split once more, further split operations may cascade toward the root node in order to maintain the edge-subdivision constraint, cf. Figure 2.3. Similarly, when two previously split triangles are merged again, the



(a) Same level d along right leg edge.

(b) Before split to next finer level $d + 1$.

(c) Levels d and $d + 1$ along right leg edge.

(d) Same level d along base edge.

(e) Before merge to next coarser level $d-1$.

(f) Levels d and $d - 1$ along base edge.

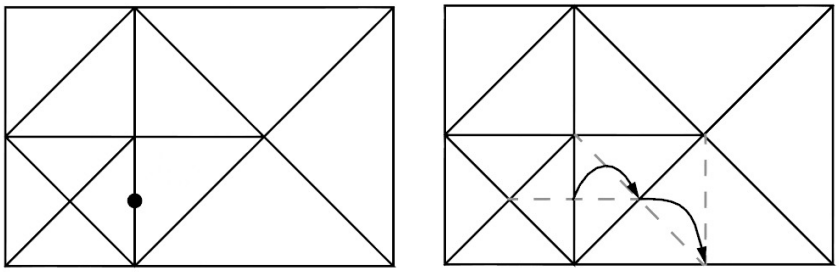
Figure 2.2: Possible adjacent subdivision levels in triangle bintrees. Along each triangle's legs there may be triangles of the same or the next finer level, as illustrated in images (a) through (c), while along the hypotenuse there may be adjacent triangles of either the same or the next coarser level, as shown in images (d) through (f).

merge operation may have to be carried out for child triangles as well, thereby forming the inverse operation of the cascaded split.

2.1.2 The Quadtree

The quadtree data structure describes a regular subdivison of a two-dimensional domain, and can therefore be directly applied to terrain-elevation data defined over the ground plane. Starting with a rectangular, typically square, region, each region is recursively split into four equal-sized, non-overlapping sub areas. Each region represents a node in the quadtree, and whenever an area is split into four sub areas, this subdivision step is represented by adding four child nodes to the node that corresponds to the just subdivided area, cf. Figure 2.4. The subdivision stops as soon as an application-specific condition is fulfilled. The leaf nodes represent the finally subdivided geometry.

Due to the regular structure of the quadtree subdivision scheme, many algorithms do not even actually store data in quadtree nodes, since the quadtree-traversal itself reveals position and size of each visited node, thereby implicitly retrieving the necessary information needed for rendering, which makes the quadtree an extremely lightweight data structure.



(a) (b)

Figure 2.3: Cascaded splits in a triangle bintree. The black dot in (a) marks the hypotenuse that is about to be split. Image (b) shows how this split propagates further up the subdivision hierarchy, i.e., toward coarser, larger triangles. Images courtesy of [43].

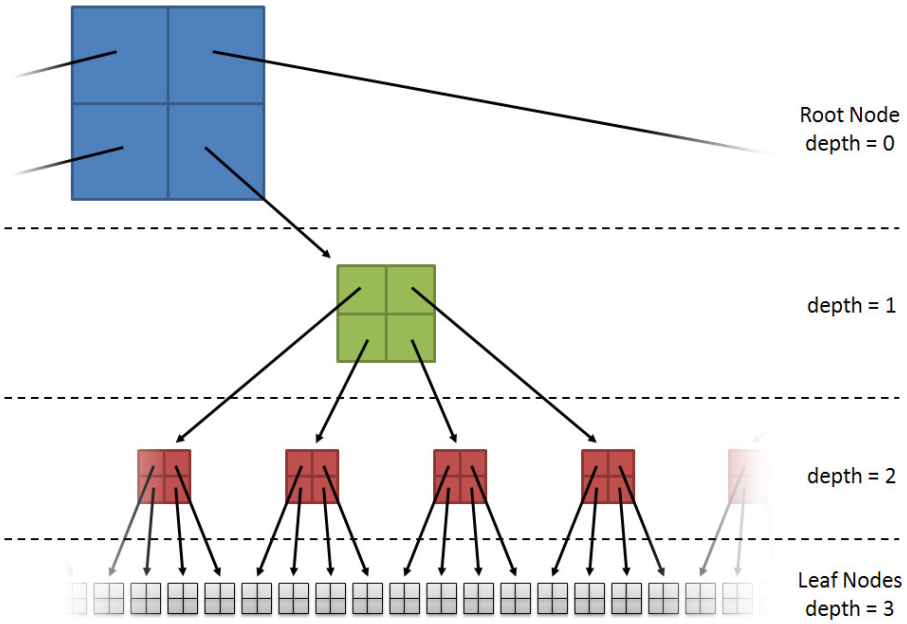


Figure 2.4: An example quadtree. Quadtree nodes are recursively split into four sub nodes, such that a node at depth $d + 1$ covers a quarter of the area that is covered by a node at depth d.

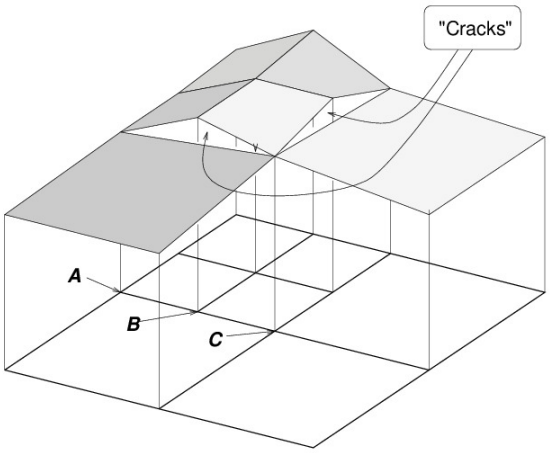


Figure 2.5: Cracks at non-shared vertices in a terrain surface that is subdivided by a quadtree. Vertices labeled A and C are present in both neighboring areas, while the non-shared vertex B only belongs to the more finely tessellated region. Vertex B gives rise to a crack in the resulting terrain surface. Image courtesy of [55].

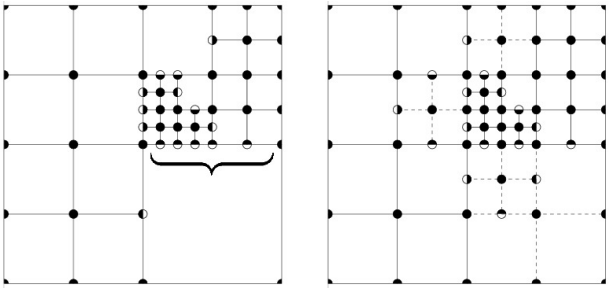
2.1.3 The Restricted Quadtree

In the just discussed quadtree data structure, the subdivision level of a node is independent of the subdivision level of its neighbors. This unrestricted subdivision of neighboring areas potentially creates non-shared vertices, so-called T-vertices, along adjacent edges as soon as neighboring nodes don't agree on their subdivision level. This means that the subdivided surface potentially exhibits cracks at T-vertices, cf. Figure 2.5.

The presence of T-vertices is an inherent property of the quadtree data structure. In theory, the resulting cracks can always be stitched by introducing new triangles, but since any two levels may meet at a certain edge, the number of possible stitching-triangle configurations explodes, which makes this form of crack stitching impractical, cf. Figure 2.6(a). To keep the number of stitching-triangle combinations manageable, the restricted-quadtree data structure introduces a new subdivision constraint – any two adjacent nodes may only differ by at most one subdivision level. This subdivision restriction makes sure that along each shared edge of two adjacent subdivision areas there is at most one T-vertex, cf. Figure 2.6(b).

Finally, Figure 2.7 shows an example watertight triangulation of restricted-quadtree nodes at different subdivision levels.

Note that when a terrain-visualization method organizes patches in a quadtree, and chooses resolution levels exclusively based on distance, then the resulting quadtree will be a restricted quadtree "by construction" for typical tile sizes and LOD-switching distances. In contrast, if the detail level of tiles depends on the heightmap, constructing a restricted quadtree requires more effort.



(a) Non-restricted quadtree. (b) Restricted quadtree.

Figure 2.6: Black circles indicate shared vertices, while black-and-white circles depict non-shared vertices. When the black part of a black-and-white circle overlaps a region, that vertex is part of this region. When the white part of a black-and-white circle overlaps a region, that vertex is not part of this region. Note that in the restricted quadtree there is at most one non-shared vertex per edge. Images courtesy of [55].

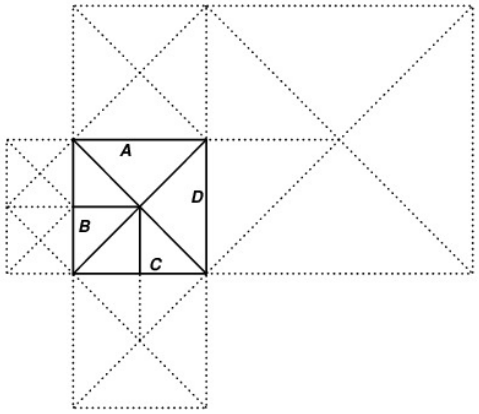


Figure 2.7: Watertight triangulation of neighboring restricted-quadtree nodes. Edges A and C are adjacent to nodes at the same subdivision level. They illustrate two valid triangulations for this case. The shared edge A shows the case where the shared edge is not split, while the shared edge C is indeed split, even though, strictly speaking, the split was not necessary. Edge B must be split, since it interfaces to a node which is subdivided one more time, while edge D must not be split due to the presence of a larger node next to it. Image courtesy of [55].

2.2 Estimating Screen-Space Error

Rendering all the visible scene geometry at full detail, regardless of the distance to the camera, obviously puts very much pressure on the whole system. To increase the performance of 3D applications, a common approach involves approximating the geometry of three-dimensional objects with gradually less triangles as the distance of objects to the camera increases. The distance-based geometry simplification is managed by *level-of-detail* (LOD) algorithms.

While replacing full-detail geometry with a coarse approximation inevitably introduces a world-space geometric error, we can exploit the properties of perspective projection to lessen the resulting screen-space error, since under perspective projection, even large world-space geometric errors map to small screen-space errors when viewed from a large-enough distance. Hence, it is not even necessary to render all parts of the scene at full detail in order to yield a faithful output image.

In this section we describe a basic metric that is commonly used to estimate the screen-space error that a world-space geometric error gives rise to at a given distance to the camera. In theory we could just as well calculate the exact screen-space error by simply transforming each vertex by the current model-view-projection matrix, but doing so would hurt performance. Since the whole idea of LOD algorithms is to reduce the load on the computer system, an exact error calculation does not meet those demands, so that a coarse, but fast to evaluate approximation of the screen-space error is preferred in practice.

The screen-space error metric we are about to present, enables us to strike a balance between rendering increasingly coarser approximations at a distance, while still maintaining the illusion of rendering the terrain surface, or any other geometry, at full detail everywhere. Since we can predict the approximate impact that a geometric error (measured in world-space units) has in screen space (measured in pixels), such a metric is an essential tool for performing unnoticeable LOD switches.

After this brief introduction, we will next derive how to estimate the screen-space error for a polygonal surface that approximates a heightfield. The heightfield is defined by the continuous function $t : \mathbb{R}^2 \mapsto \mathbb{R}$, which associates a height displacement to each ground-plane position. Let the ground plane be given by the equation $y = 0$. In the process of approximating the heightfield with a limited amount of sample points, i.e., vertices, at gradually decreasing resolution in the distance, we inevitably introduce the geometric error δ in world-space, which is given by Equation (2.1), where t' is the reconstructed height.

$$\delta(x, z) = |t'(x, z) - t(x, z)| \tag{2.1}$$

With the help of Figure 2.8, we can set up Equation (2.2). Note that an almost identical calculation for estimating the screen-space error in terms of the geometric error can be found in the chunked-LOD paper [59], and the book "Level of Detail for 3D Graphics" [36] for instance. The basic idea behind all these error-estimation calculations is to establish a relationship between the geometric error δ and the resulting screen-space error ρ by using the properties of similar triangles. The geometric error δ is the world-space height difference between the rendered surface position and the actual terrain height (see Equation (2.1)), z denotes the distance of the surface position to the camera, and last but not least, v denotes the distance of the view plane from the

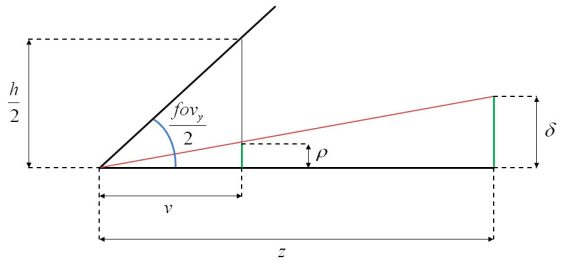


Figure 2.8: Estimating screen-space error ρ based on geometric error δ at distance z. Both the view plane's height h, and ρ are given in pixels.

camera, i.e., from the center of projection. In the equation, δ, z, and v are given in world-space units, while the screen-space error ρ is measured in pixels

$$\begin{aligned} \frac{\delta}{z} &= \frac{\rho}{v} \\ \rho &= \frac{\delta}{z} v \end{aligned} \tag{2.2}$$

Equation (2.3) shows how to calculate v, where the viewport height h is measured in pixels, fov_y is the vertical field of view, i.e., the angle between the top and bottom view-frustum planes, and is given in radians.

$$\begin{aligned} tan(\frac{fov_y}{2}) &= \frac{h/2}{v} \\ v &= \frac{h}{2tan(\frac{fov_y}{2})} \end{aligned} \tag{2.3}$$

The above equation can also be "reversed", so that, given a screen-space error tolerance τ measured in pixels, we can estimate the maximum-allowed world-space step size at a specific distance to the camera that maps to no more than τ pixels. For a fixed τ we rewrite Equation (2.2), yielding

$$\delta = \frac{\tau}{v} z \tag{2.4}$$

This interpretation comes in handy when sampling textures in the vertex shader, where screen-space derivatives aren't available yet, so that we can approximate an adequate mipmap level for the texture fetch based on δ, as shown in Equation (2.5).

$$mipmapLevel = log_2(\delta) \tag{2.5}$$

Another use case for Equation 2.4 is the adjustment of the ray-marching step size at increasing distances to the camera, as well as the calculation of a suitable spacing between slice planes through 3D datasets depending on the distance to the camera, as in Decaudin's paper on rendering volumetric billboards [19].

An interesting property of the just presented error-estimation equation is that it basically evaluates the projection of objects onto a camera-centered sphere. Therefore, the calculated screen-space error is independent of the camera's view direction, which means that when the camera only rotates while resting at a fixed position, the selected LOD level does not change either, resulting in a stable terrain surface. However, this also means that Equation (2.2) slightly underestimates the screen-space error for objects that project to the corners of the view plane.

To further reduce the runtime cost of LOD selection and screen-space error estimation in terrain-rendering algorithms, local terrain features are often ignored altogether, meaning that distances are calculated between the camera and non-displaced vertices, i.e., vertices still lying on the ground plane. This way the heightmap lookup can be skipped for each vertex, and since performing even a cheap calculation on lots of vertices can still introduce a fair amount of work, it is common to merge several neighboring vertices into a block, or patch, and calculate the camera distance to the center of the block rather than to each individual vertex in the patch.

2.3 Terrain-Rendering Methods

In this section we briefly discuss several representative terrain-rendering algorithms, roughly ordered by their publication date. The chronological ordering reveals how the evolution of consumer-level graphics boards caused an increasing number of terrain-related computations to be shifted from the CPU to the GPU. In particular, terrain-rendering algorithms started to be centered around GPU-friendly data structures, such as regular grids that can be naturally stored in textures, while earlier CPU-based techniques focused on aggressively reducing vertex count, for which irregular data structures are better suited. To increase performance, GPU-based terrain-rendering algorithms also follow the ubiquitous draw-call-batching mantra.

We implemented most of the presented algorithms in FTRAC, our small framework for comparing the different terrain-rendering algorithms, which enabled us to provide our own illustrative screenshots. We describe FTRAC in more detail in Section 4.

Before diving into specific landscape-visualization algorithms, we first want to touch upon strategies and approaches that can be found in almost all those algorithms, and introduce several common terms along the way. Terrain-rendering methods typically subdivide the two-dimensional domain of the heightfield into tiles, or patches, so that view-frustum culling and the selection of an appropriate detail level can be evaluated per patch instead of per triangle. How to exactly create and select the most appropriate detail level for each patch at runtime falls into the responsibility of LOD algorithms.

LOD algorithms can be roughly classified into view-dependent, and data-dependent approaches. View-dependent LOD techniques adjust the triangle density of each tile based on viewing parameters only – most often the mesh resolution is determined by the distance to the camera. Data-dependent LOD methods, on the other hand, take local terrain features into account, and devote more triangles to areas with high-frequency terrain features, while flat landscape regions are represented by less-densely tessellated meshes. Data-dependent LOD methods are generally combined with view-dependent LOD selection, since in their own right, data-dependent LOD methods would waste unnecessarily many triangles on distant areas.

While data-dependent LOD techniques usually create meshes that approximate the terrain surface with less triangles, they often require a preprocessing step, and impede runtime terrain modification. Further, the final triangle count per frame can be hard to predict, potentially leading to unstable frame rates. A notable side effect of employing preprocessed geometry is that it is not restricted to convex terrains, but also enables rendering terrain with caves and overhangs without increasing the rendering complexity notably. For example, BDAM [16] (see Section 2.3.1), and Chunked LOD [59] (see Section 2.3.2) are algorithms that rely on precomputed data-dependent LOD hierarchies with preprocessed geometry.

In contrast to data-dependent LOD algorithms that render preprocessed terrain-geometry patches from vertex and index buffers, view-dependent LOD algorithms are usually able to render terrain patches directly from the original dataset. Since local terrain features are ignored, the same generic regular mesh can be used across all terrain tiles, and the heightmap texture is used to displace each mesh vertex accordingly. Using the heightmap texture for mesh displacement not only makes it possible to easily change terrain heights at runtime, but also enables the GPU to access neighboring height values easily, which can be used for the dynamic evaluation of terrain normals for example. This way, normals don't need to be saved at all, which can be important if the system already runs out of video memory. Compared to data-dependent LOD algorithms, the view-dependent approach may seem inferior – after all, even flat areas are just as densely tessellated as rough areas. However, the simplicity of embarrassingly parallel algorithms is key to unleashing the full power of modern graphics hardware. Apart from the direct support of real-time terrain editing, the usage of GPU-friendly terrain data enables fractal landscapes to be synthesized on the fly on the GPU, as demonstrated by Hoppe et al. [13] in their publication about terrain rendering with the geometry clipmap data structure. Another terrain-rendering technique that employs view-dependent LOD selection is Strugar's CDLOD technique [56].

Clearly, subdividing the terrain dataset into patches of varying detail levels offers a significant performance advantage over dealing with a huge monolithic terrain mesh. This advantage does not come for free, though, as whenever two distinct detail levels meet, special care must be taken to join the neighboring tiles in a watertight manner. Otherwise, adjacent triangles at tile borders have different dimensions, and hence may create a terrain surface that contains cracks.

In the following discussion of representative landscape-visualization methods, we will focus on how each one of them generates and selects the various detail levels for the terrain-tile meshes, and how seams between different detail levels are stitched in order to create a watertight surface.

2.3.1 Early Triangle-Bintree-Based Terrain-Rendering Algorithms

In this section we review several terrain-rendering algorithms that are based on the triangle bintree, namely *Real-time Optimally Adapting Meshes* (ROAM) [22], *ROAM Using Surface Triangle Clusters* (RUSTiC) [44], and *Batched Dynamic Adaptive Meshes for High Performance Terrain Visualization* (BDAM).

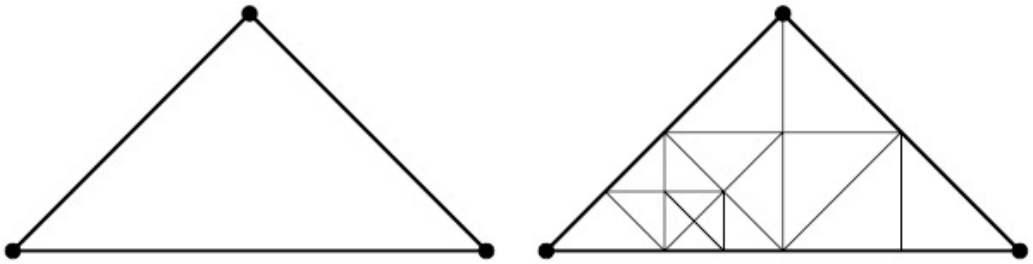
In the ROAM method, each node in the triangle bintree corresponds to a single triangle in the generated surface. ROAM uses a combination of view-dependent and data-dependent LOD calculations when reconstructing the terrain surface. The data-dependent part of the LOD selection at runtime requires a lightweight preprocessing step. The algorithm is not restricted

to convex terrains and is able to create surface approximations that exhibit a maximum screen-space error that lies below a user-defined error threshold.

The ROAM algorithm allows iterative refinement of the terrain-surface mesh, so that updates affect only a minimum subset and can be carried out efficiently. Mesh updates are triggered by view-frustum culling and by LOD adaptions, which happen in the form splitting and merging of triangles according to the rules of the bintree data structure. Splitting triangles increases the triangle count and therefore yields a better surface approximation, while merging reduces the geometric complexity. ROAM keeps two separate queues, one for splits, and one for merge updates. Based on an application-specific priority function, which is typically based on the screen-space error, each queue is sorted, and then processed so that most important geometry updates are performed first. Arranging LOD-related tasks in the two priority-sorted queues enables ROAM to guarantee a minimum frame rate by stopping to process entries in the queue once the assigned time limit is exhausted. Thus, on less capable systems the terrain might temporary expose a bigger screen space error than desired. However, once the camera stops moving, mesh refinement eventually catches up so that the user-defined screen-space error is once again satisfied.

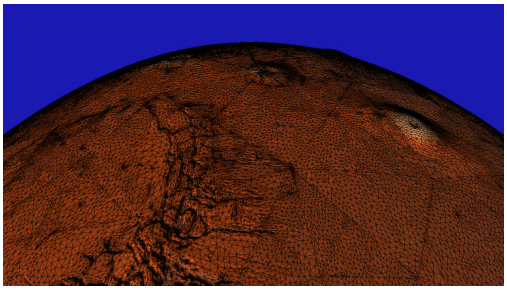
The ROAM algorithm was devised at a time when consumer-level GPUs were still in their early years, which means that all mesh updates needed to be performed on the CPU. As GPUs grew more powerful, this tight coupling between CPU and GPU became a limiting factor for the performance of ROAM, since each data exchange causes the graphics pipeline to be flushed in order to ensure that the CPU and GPU are correctly synchronized. During these pipeline stalls the graphics chip is forced to an idle state, effectively wasting cycles. Further, ROAM processes each triangle in the mesh individually, so that the algorithm doesn't scale well with increasing triangle counts. Finally, a straight forward bintree implementation tends to scatter individual bintree nodes over the whole address space over time, so that neighboring triangles in the terrain-surface mesh are not stored in adjacent memory locations. Obviously, the lacking spatial locality of vertex data leads to lots of cache misses when updating and rendering the mesh.

These observations led to an extension of ROAM called *ROAM Using Surface Triangle Clusters* (RUSTiC) [44], which manages to reduce the CPU load and increase GPU utilization. RUSTiC's key contribution the representation of bintree nodes by a cluster of triangles, getting rid of ROAM's per-triangle micro management. The triangle-cluster meshes are constructed by using ROAM-style split and merge operations. At runtime, the algorithm essentially traverses a triangle bintree in exactly the same way as ROAM does, but instead of directly rendering a ROAM bintree triangle node as a single triangle, a whole cluster of triangles is rendered instead, cf. Figure 2.9. The clustering of triangles relieves the CPU, since a whole group of triangles can be processed and rendered in a single computational step, and the bintree traversal can stop at a coarser level. Further, node updates in RUSTiC are performed by submitting lots of triangles in one batch, which reduces the number of CPU-GPU synchronisation points per frame compared to ROAM. The main challenge of RUSTiC's triangle clustering is to still create a crack-free surface reconstruction where triangles in clusters of different detail level meet. Since the triangle clusters are bintree nodes, and a bintree node interfaces to other bintree nodes that differ by at most one detail level (see Figure 2.2), a watertight terrain mesh can be generated by handling a

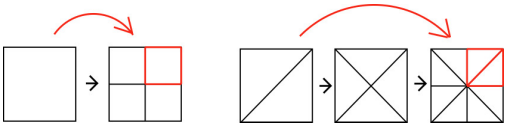


(a) (b)

Figure 2.9: The underlying ROAM node consisting of a single triangle (a), is replaced by a preprocessed cluster of triangles in RUSTiC (b). Images courtesy of [44].



(a) TIN meshes form the leaf nodes in a triangle bintree hierarchy.



(b) The texture quadtree (left) goes hand in hand with the triangle bintree for the terrain geometry (right).

Figure 2.10: BDAM data structues – TIN meshes are kept in the triangle-bintree hierarchy, while the associated textures are stored in a quadtree. Images courtesy of [16, 17].

limited number of different triangle configurations at the borders of triangle clusters that have different detail levels by making sure that all vertices along the edge borders are shared.

Finally we look at *Batched Dynamic Adaptive Meshes for High Performance Terrain Visualization* (BDAM) [16], which is another algorithm that uses a hierarchy of right triangles for LOD management. As in RUSTiC, the leaf nodes in a BDAM triangle bintree do not represent a single triangle, but rather a group of triangles. In contrast to RUSTiC's regular triangle clusters, BDAM represents bintree leaf nodes by *triangulated irregular networks* (TINs), which are generated in a preprocessing step, cf. Figure 2.10(a). BDAM uses TINs since they typically approximate the terrain surface more faithfully than regular tessellations with the same amount of triangles, as TINs do not place any constraints on vertex placement. At runtime, the triangle bintree guides the selection of the preprocessed TIN vertex buffers at the appropriate detail levels. BDAM also addresses the need for storing and accessing color data alongside the geometry, for which the algorithm employs tiled quadtrees. Figure 2.10(b) illustrates that this mapping can be established directly, since one texture-refinement step in the quadtree corresponds to two geometry-refinement steps in the triangle bintree.

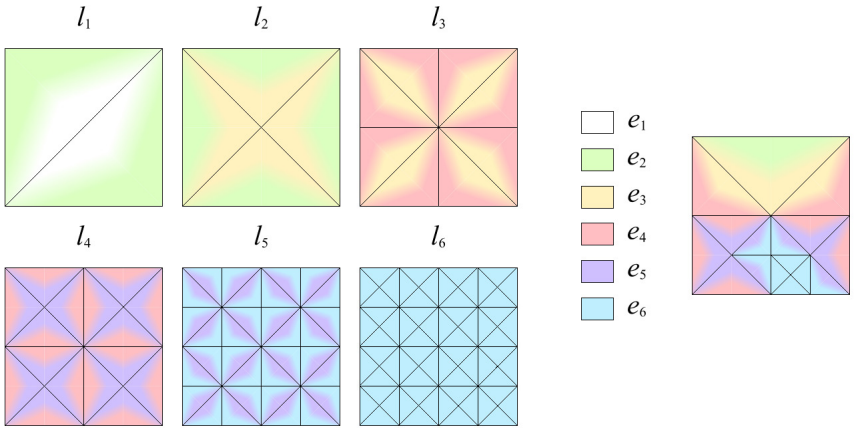


Figure 2.11: Seamless stitching of TIN patches in BDAM. To illustrate the smooth distribution of the edge-error weights across the whole TIN mesh, a color is assigned to all triangle edges based on their lengths. Note that the individual triangles in the triangle-shaped TIN patches are not shown in the figure. The smoothly interpolated triangle colors in the rightmost image show that even when nodes of different refinement levels meet, their edge-error weights always match up perfectly, creating a watertight terrain-surface mesh that distributes edge-error weights evenly. Image courtesy of [16].

BDAM depends on a preprocessing step that generates TINs for all triangular bintree nodes at all detail levels, which is a time-consuming task. In order to be able to generate all needed patches for large, real-world datasets in reasonable time, TIN creation needs to be run in parallel, but in such a way that the resulting terrain-surface mesh is still crack free. BDAM achieves this goal by introducing a simple constraint; it makes the triangulation of TIN patches depend on edge-error weights, which themselves are based on edge lengths of the governing triangle shapes that represent bintree nodes of the LOD hierarchy. Then those three edge-error weights are smoothly interpolated over the bintree-node triangle, i.e., the whole area that the TIN eventually spans. So, since TIN triangulation is controlled by smoothly blended edge-error weights and the bintree's hierarchy of right triangles guarantees that adjacent bintree nodes, and thus edge-error weights, always match up perfectly, BDAM guarantees a globally crack-free triangulation of the terrain surface even at LOD transitions, cf. Figure 2.11.

To summarize, ROAM, RUSTiC and BDAM employ the triangle bintree as their LOD management data structure. RUSTiC and BDAM improve the performance of the ROAM algorithm by representing individual bintree nodes by clusters of triangles. While ROAM transfers each individual triangle to the GPU on its own, and then issues one draw call for each triangle, RUSTiC and BDAM achieve an acceleration by submitting the aforementioned clusters of triangles to the GPU in a batched manner, which utilizes the computer system resources more efficiently. On the downside, the grouping of triangles complicates the creation of a watertight triangle surface at LOD transitions, the realization of unnoticeable LOD switches, and heightmap modification at runtime. The main difference between RUSTiC and BDAM is how bintree nodes are repre-

sented. RUSTiC uses the same regular subdivision scheme as the ROAM algorithm to group triangles into patches, while BDAM bakes the terrain geometry into TIN meshes. TINs have the advantage that they can represent the terrain surface more accurately with less triangles than meshes tessellated with regular subdivision, thereby reducing the overall triangle count. On the other hand, TINs are more costly to create than the regular bintree triangulations, and hence make dynamic modification to the terrain surface an even more challenging task in real-time applications.

2.3.2 Early Quadtree-Based Terrain-Rendering Algorithms

In this section we review landscape-visualization methods that are based on the restricted quadtree, such as Thatcher Ulrich's *Rendering Massive Terrains using Chunked Level of Detail Control* (ChunkedLOD) [59] and *GPU-Friendly High-Quality Terrain Rendering* (HQTerrain) [50].

ChunkedLOD is a terrain-rendering algorithm that organizes the terrain heightmap and colormap in a quadtree, and combines both view-dependent, and data-dependent LOD approaches. Each node in the LOD hierarchy, a so-called "chunk", represents a rectangular part of the terrain, and contains both a vertex buffer that stores the geometry and a color texture. Each chunk can be rendered with a single draw call, so that the whole terrain can be rendered with relatively little draw calls, reducing graphics-driver overhead. Hence, the basic idea of reducing CPU overhead and improving GPU utilization is the same as RUSTiC [44] and BDAM [16], which we discussed in the previous section, but this time working on a quadtree rather than on a triangle bintree. ChunkedLOD handles out-of-core rendering efficiently by only transferring data for patches that switch resolution, and those that have just come into view due to camera motion. Compared to the number of chunks rendered each frame, the number of chunks requiring an update is small.

ChunkedLOD builds the whole terrain quadtree in a preprocessing step. The chunks form a data-dependent quadtree-LOD hierarchy in which flat terrain areas are approximated by fewer triangles, while rough terrain areas yield meshes with higher triangle counts. The geometry of each chunk is formed by a series of quadtree subdivision steps, yielding a regular grid. While processing the geometry, the bounding box of each chunk is evaluated and stored alongside the other data in the quadtree node. The bounding boxes are used for fast chunk-level view-frustum culling and also guide the view-dependent LOD selection at runtime. Due to the obligatory chunk-preprocessing step, ChunkedLOD works best for static scenes, making both interactive terrain modifications, and procedural terrain generation at runtime a challenging task.

Chunks at a certain level L in the quadtree all share the same maximum geometric world-space error δ. The child nodes at level $L + 1$ halve this error, so that the following equation holds

$$\delta(L + 1) = \delta(L)/2 \tag{2.6}$$

Alongside the geometry preprocessing, chunk textures are created as well. Texture resolution is selected such that an average screen-space error of one pixel is maintained at all subdivision levels.

At runtime, the algorithm selects the detail level of visible patches in such a way that the screen-space error stays below a user-defined threshold τ. Equation (2.2), which calculates a

screen-space-error estimate, can be used test whether a chunk at a particular subdivision level already fulfills the maximum-allowed screen-space error or if further subdivision is required. We repeat Equation (2.2) here to show how it can be applied to Chunked LOD.

$$\rho = \frac{\delta}{z} v \qquad \text{(2.2 revisited)}$$

The world-space error δ is given by the subdivision level of the patch (see Equation (2.6)), the distance z between the camera and the chunk is given by the distance to the closest corner of the patch's bounding box to the camera, and v denotes the distance of the view plane from the center of projection. The resulting estimated screen-space error ρ is finally tested against the maximum-allowed screen-space error τ, and if ρ is smaller than τ, an adequate subdivision level for the chunk has been determined.

As just outlined, the selection of a chunk's resolution level is independent of the LOD of its neighbors. Therefore, the resulting terrain surface potentially contains cracks where chunks of different detail levels meet. Ulrich presents several solutions to the problem, finally settling for vertical skirts around a chunk's border. While skirts close the holes in the surface, they introduce other problems such as texture stretching and lighting artifacts. However, for a small-enough screen-space error, skirt pixels are hard to spot.

Another distracting issue associated with LOD switching is vertex popping. In contrast to algorithms such as ROAM [22], where LOD switches happen at the triangle level, in the ChunkedLOD algorithm a whole patch, i.e., an aggregate of many triangles that covers many pixels on screen, switches resolution instantly. Since many triangles are affected by the switch all at once, the LOD transition is visually even more distracting. ChunkedLOD alleviates this popping artifact by vertex morphing. Vertex morphing is implemented by assigning a morph factor to each chunk. The morph factor alters vertex heights only, and does not alter the texture coordinates of the chunk vertices. Before morphing, all vertex heights correspond to their actual detail level, and once the morph completes, the heights correspond to those of the next coarser detail level, essentially projecting the high-detail mesh onto the low-detail mesh along the vertical coordinate. By smoothly varying the morph factor between 0 and 1, a smooth LOD transition is achieved, cf. Figure 2.12.

More recently, Harald Vistnes presented an efficient and easy-to-implement landscape-rendering technique in Game Programming Gems 6 [62], that can be seen as an adaption of ChunkedLOD for modern GPUs. Vistnes makes use of the vertex-texture-fetch feature of modern GPUs, and only employs regular-grid terrain tiles, getting rid of the data-dependent aspect of ChunkedLOD, thereby removing the preprocessing step and enabling straight-forward real-time terrain editing. The whole terrain is rendered from a single, small vertex buffer that is instantiated many times with appropriate scale and offset, and as in ChunkedLOD, skirts are used to close cracks at LOD transitions.

GPU-Friendly High-Quality Terrain Rendering (HQTerrain) [50] is another take on adapting ChunkedLOD to modern GPUs. HQTerrain organizes the terrain dataset in a restricted quadtree that is precalculated. The authors of HQTerrain make the observation that triangle bintrees and restricted quadtrees share the property that once a vertex is inserted at a specific tree depth, the vertex will be present in the triangulations of all higher-detail child nodes. In such a nested hierarchy it is therefore not necessary to store all vertices at each detail level. HQTerrain exploits

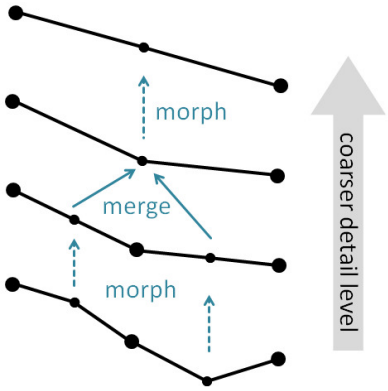
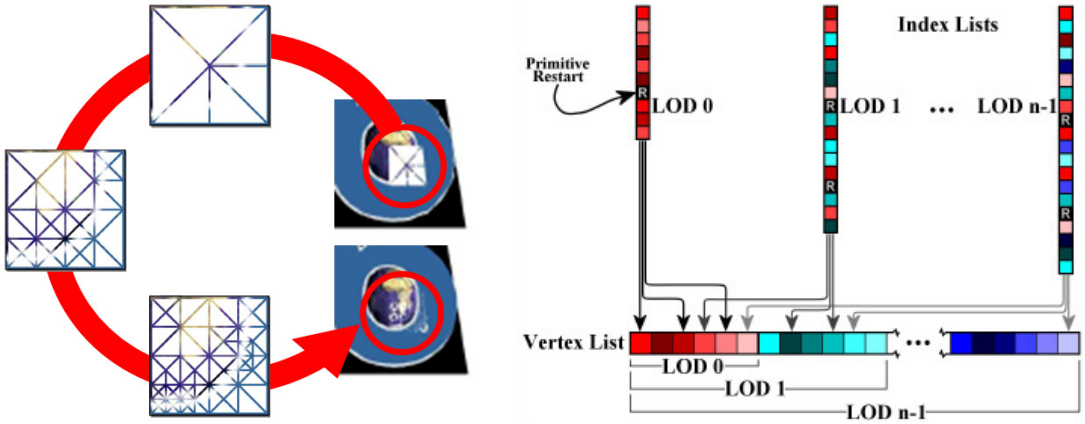


Figure 2.12: Morphing chunk vertices to avoid vertex popping during LOD switches in ChunkedLOD. The merge step corresponds to a LOD switch.



(a) (b)

Figure 2.13: Image (a) shows a HQTerrain tile as it transitions from a low-detail to a high-detail representation. Note that all vertices from low-detail representations are also present in all higher-detail meshes, forming a nested hierarchy of vertices. GPU data structures for progressive transmission of vertex and index data – vertices are uploaded incrementally, but each detail level has its own index list (b). Images courtesy of [50].

this property, and merely stores those vertices that have been added by the next subdivision step, but weren't present yet in any of the coarser detail levels that this patch belongs to, cf. Figure 2.13(a). In HQTerrain the terrain dataset is preprocessed and stored in a restricted quadtree. Index buffers of different detail levels do not form a nested hierarchy, and are therefore stored in entirety for each detail level, cf. Figure 2.13(b).

The efficient geometry storage has another advantage, since vertex data can be updated incrementally when switching to a higher detail level, resulting in reduced bandwidth requirements at runtime. Further, HQTerrain bounds graphics-memory fragmentation, since the algorithm al-

locates only one vertex buffer for all detail levels of a chunk. This buffer needs only be large enough to hold the vertices of the highest detail level, since all vertices of the coarser detail levels are only subsets. This means that when a patch transitions to a higher detail level, we can be sure that the vertex data will fit into the existing buffer, eliminating the need to (re-)allocate vertex buffers dynamically. The downside of this optimization is that the active set of all currently visible chunks takes up the same amount of memory on the GPU, even if lots of those chunks are currently rendered at a low resolution.

To eliminate vertex popping caused by LOD transitions, the vertices in the high-quality chunk are morphed to the respective height in the corresponding low-detail chunk. In contrast to ChunkedLOD, which calculates a single morph value for all vertices, the HQTerrain method assigns a different morph value to each patch vertex, which makes LOD transitions even less noticable. Based on the distance to camera, morph values are calculated for each corner of a tile's bounding box. Finally, each vertex is assigned a smoothly blended morph factor based on the vertex's relative position within the tile's bounding box.

While geomorphing eliminates cracks, T-vertices at patch borders are still present after the geomorph. To resolve this issue, appropriate zero-area triangles are calculated on the CPU, and transmitted to the GPU. These steps ensure a watertight mesh across patches. Compared to the skirts at patch borders in ChunkedLOD, the terrain is rendered with less artifacts, albeit at the cost of additional calculations.

2.3.3 Terrain Rendering Using GPU-Based Geometry Clipmaps

In this section we discuss a terrain-rendering algorithm that was first introduced by Losasso and Hoppe in their paper "Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids" [35], and later revised by Asirvatham and Hoppe in the publication *Terrain Rendering Using GPU-Based Geometry Clipmaps* (GPUGCM) [13]. These two publications show the transition of a CPU-based landscape-visualization technique to a fully GPU-based method. In the following we will focus exclusively on GPUGCM, which is one of the first terrain-rendering algorithms that exploits the vertex-texture-fetch feature of modern GPUs. Storing the heightmap in vertex textures not only simplifies the initial version of their algorithm, but also offloads most terrain-related calculations from the CPU to the GPU, which improves GPU utilization and allows the GPU to run at full speed.

As the name of the algorithm suggests, the geometry clipmap is at the heart of GPUGCM. The geometry clipmap is based on the clipmap data structure that is described by Tanner et al. in "The Clipmap: A Virtual Mipmap" [58]. A clipmap is essentially a clipped mipmap pyramid, in which each detail level is limited to a fixed size. Figure 2.14 reveals that after clipping each mipmap level to a maximum size, at some point the pyramid actually becomes a stack of mipmap-layer subsets.

Since the clipped mipmap pyramid only contains data that is relevant for the current camera position, it has moderate memory demands compared to the whole terrain dataset, so that the entire clipmap can be kept in graphics memory. When the camera position changes, the clipped areas are updated in order to maintain the illusion that the whole dataset is stored on the graphics card. Hence, the clipmap manages to decouple the storage requirements of the entire terrain dataset from the memory requirements of the actually visible subset. To foster fast incremental

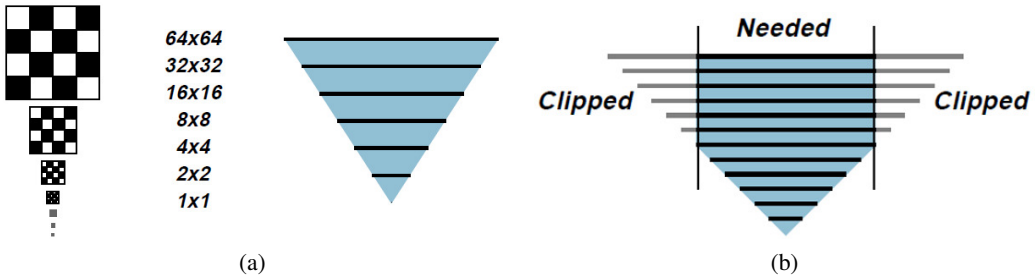


Figure 2.14: Mipmap pyramid (a) and clipped mipmap pyramid (b). Images courtesy of [58].

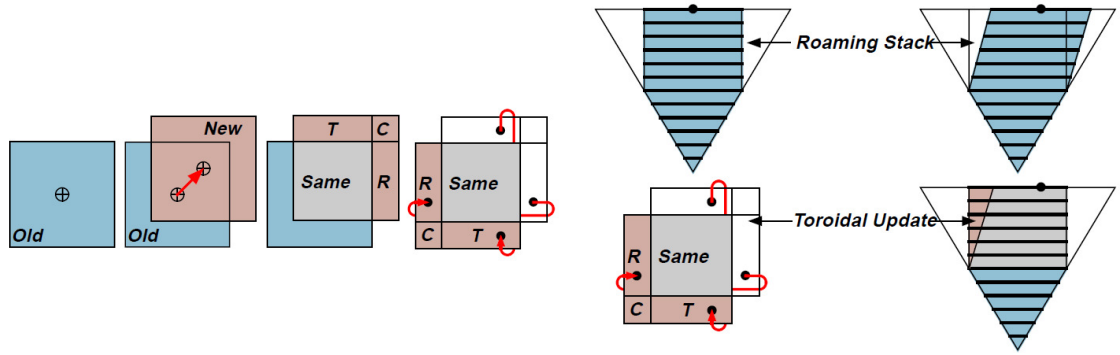
clipmap updates, the updates are performed toroidally, i.e., with two-dimensional wrap-around addressing. Otherwise, blocks of already uploaded data would have to be moved first to make room for the new data before actually bringing it in, cf. Figure 2.15. Note that coarse detail levels are updated less often than fine detail levels, with the update frequency being halved whenever moving from one detail level to the next-coarser one.

The geometry clipmap essentially applies the clipmap concept to meshes over a heightmap, and defines a vertex layout that enables efficient crack-free rendering of the resulting displaced terrain mesh. GPUGCM represents the visible part of the terrain with nested rectangular regular-grid meshes, which consist of the same number of samples at each detail level. The area that a specific mesh covers in world space is four times the area that its next-higher detail level covers, a property that the (geometry) clipmap inherits from the underlying mipmap, cf. Figure 2.16(a). All levels except for the innermost, highest-detail level, have the shape of hollow rectangular rings that are centered about the viewer. The highest clipmap level fills the hole that is left by the hollow outer rings, cf. Figure 2.16(b) and (c).

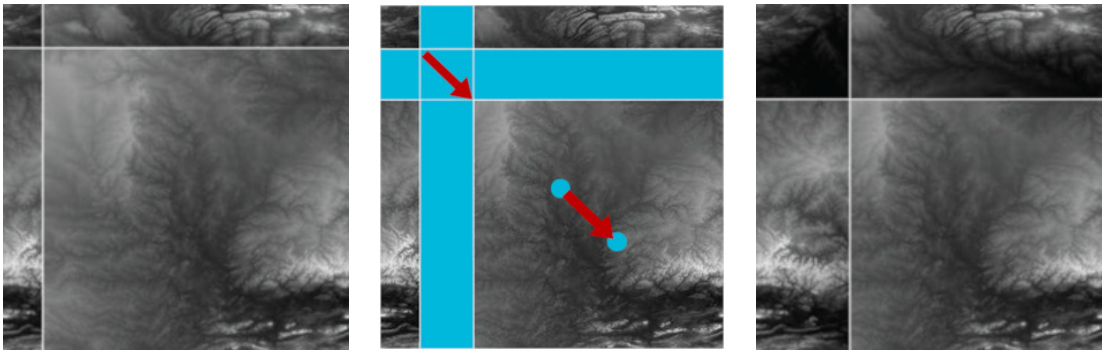
Figure 2.17 shows the regular grids that make up one hollow rectangular ring in GPUGCM. Note that when employing instanced rendering, the same small set of vertex buffers can represent hollow rings at all scales. The vertices in the regular grids are all aligned to the heightmap samples, so that the terrain height can be sampled with nearest-neighbor filtering in the vertex shader, which may be a performance advantage over bilinear filtering, since the latter requires the hardware to actually fetch four texels from the texture for each heightmap sample instead of only one texel when using nearest-neighbor filtering.

At the outer borders of each hollow ring, where two different resolutions meet, care must be taken to render the terrain surface without cracks. To achieve a smooth transition from one detail level to the next, vertex heights at the outer perimeter of each ring are gradually morphed to the respective heights at the next-coarser mipmap level, so that the fully-morphed surface heights match those of the next coarser ring. While the geomorphing makes sure that terrain-surface heights at ring borders match, the resulting surface still contains T-vertices there, so that small numerical instabilities can lead to pixel-sized rasterization artifacts. GPUGCM removes T-vertices by rendering zero-area triangles around the outer perimeter of each hollow ring. The transition region, the blending effect, and the zero-area triangles are illustrated in Figure 2.18.

Since GPUGCM does not require any heightmap preprocessing, the algorithm supports heightmap modifications at runtime, and the authors even demonstrated real-time terrain synthe-



(a) (b)



(c) Elevation map before update (d) Viewer motion in red and update region in cyan (e) Elevation map after update

Figure 2.15: The top row shows the effect of toroidal updates on the clipmap stack; T, C, and R stand for top, corner, and right, respectively. The bottom row is an example of a toroidal update of a heightmap image upon camera movement. Images courtesy of [13, 58].

(a) (b) (c)

Figure 2.16: The geometry clipmap. Image (a) reveals that the geometry clipmap is a subset of the full mipmap pyramid. As with mipmaps, the coarsest geometry clipmap level covers the largest area, the finest level the smallest. Images (b) and (c) show the innermost non-hollow mesh, and the nested rectangular outer rings from a top view and a perspective view. In image (c) the mesh is displaced according to a heightmap. Images courtesy of [13].

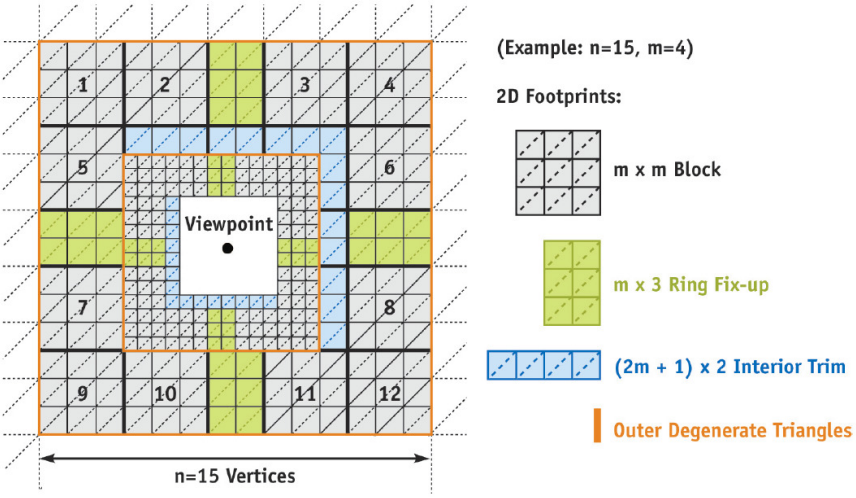
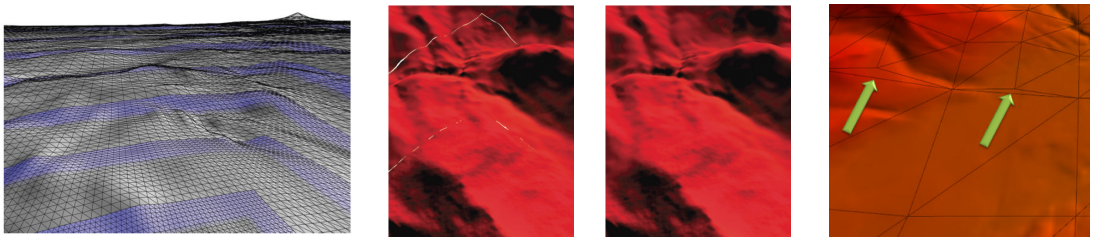


Figure 2.17: GPUGCM – a small set of vertex buffers suffices to render the hollow rings at any scale. The geometry is mostly made up of 12 $m \times m$ blocks. Image courtesy of [13].



(a) Highlighted LOD transition regions

(b) Without LOD blending and zero-area triangles

(c) With LOD blending

(d) Zero-area triangles revealed when LOD transition is disabled

Figure 2.18: Handling LOD transitions in GPUGCM. Image (a) highlights geomorphing blending weights in blue. Note that geomorphing affects only the outermost vertices of a ring. Images (b) and (c) show that the terrain surface exhibits cracks in the absence of the vertex-height blending and zero-area triangles, and how geomorphing achieves a smooth LOD transition. Image (d) reveals zero-area triangles at the outer perimeter of a ring by disabling LOD blending. Images (a) through (c) courtesy of [13].

sis. This is possible because the algorithm creates a purely view-dependent LOD approximation of the terrain dataset, and as such utilizes triangulations for the "worst case" terrain, i.e., an extremely rough terrain , everywhere. Further, this means that any terrain dataset yields constant triangle counts, and thus a predictable and stable frame rate.

Just as clipmaps, geometry clipmaps are well suited for out-of-core rendering, since the amount of data that needs to be streamed from main memory to video memory each frame is bounded. GPUGCM exploits this property for load balancing by letting the incremental updates of the inner high-detail rings fall behind if the system fails to handle the updates in time. This leads to a slightly degraded detail level around the viewer, but framerate stuttering is avoided. Once the camera slows down or stops moving, the updates of the innermost levels can catch up, thereby progressively refining the nearby terrain with a slight delay.

The basic GPUGCM technique has been extended to handle spherical [18] and even arbitrary [26] carrier geometry. While the former continues to essentially store a single elevation value per vertex in the geometry clipmap, the latter stores full 3D positions. Sven Forstmann et. al [25] took the two-dimensional nested rectangular grids to the third dimension, where they become nested clip boxes, to render volumetric landscapes with caves and overhangs in realtime.

2.3.4 Seamless Patches for GPU-Based Terrain Rendering

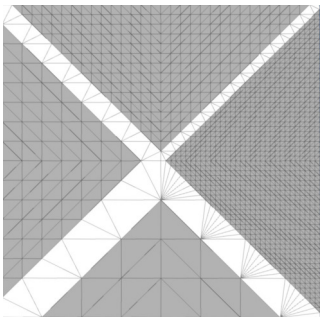
In this section we investigate the "Seamless Patches for GPU-Based Terrain Rendering" algorithm [31]. The novel idea that Livny et al. present is that LOD transitions are handled within a patch instead of at its borders.

As in many of the algorithms that we have discussed so far, a restricted quadtree is used to subdivide the heightfield and manage the different resolution levels of the resulting rectangular patches. Each patch is made up of four *triangular tiles* (triles), where the longest edge of each triangular tile faces to the outside of the patch, forming the patch's border. The triles are stitched together by four slim diagonal strips, cf. Figure 2.19(a). A patch adapts to the desired detail level by adjusting the detail level of its borders, which is realized by switching to the desired discrete LOD of the trile that forms the patch border. The patch-border detail levels can be set independently of each other, as the stitching strips take care of connecting the four triles within a patch in a crack-free, watertight manner. Trile vertices across all detail levels are always aligned to heightmap texels at the corresponding mipmap level.

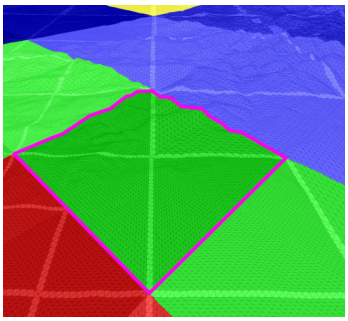
Further, each patch may calculate its LOD level independent of its neighboring patches by means of Equation (2.7), which basically corresponds to the screen-space error estimation equations from Section 2.2. In Equation (2.7), l stands for the patch's edge length (which depends solely on the node's depth in the quadtree), d denotes the distance of the view point to the edge, and ρ is a user-defined precision factor – larger values for ρ favor subdivison, and hence lead to finer detail.

$$\epsilon = \rho \frac{l}{d} \tag{2.7}$$

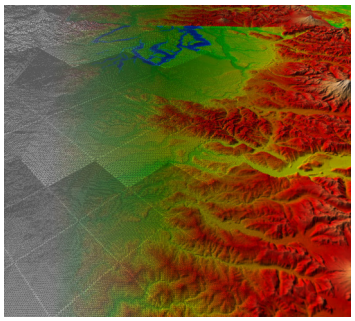
The equation's result ϵ represents the subdivision criterion, i.e., ϵ controls the subdivision. If ϵ is larger than 1 on any patch edge, the patch is further subdivided, otherwise the edge resolution is calculated as $\epsilon * R_{max}$, where R_{max} is the highest available mesh resolution for a trile. Fig-



(a) A patch consists of four triles (gray) and four LOD-transition stitching strips forming an "X" (white)



(b) Joining patches at different detail levels in a watertight manner. One patch is highlighted in magenta.



(c) Aerial view, blended with wireframe overlay

Figure 2.19: Image (a) shows one example patch and its four *triangular tiles* (triles). The triles are all at different resolutions, and are connected seamlessly by four stitching strips. The discrete detail levels of a trile are triangulated in such a way that the mesh vertices align with the regular grid of the color and height texture at the corresponding mipmap level. Image (b) shows color-coded patches that correspond to quadtree nodes – hue is assigned according to the node's depth in the quadtree, and patches of the same quadtree level can be distinguished by slight adjustments to their brightness. Stitching strips are rendered brighter than triles. The patch that is highlighted in magenta currently connects to a finer and a coarser detail level. In order to seamlessly connect to the coarser patches, the trile on the top right needs to switch to a lower resolution. Finally, image (c) shows an aerial view of of the Puget Sound dataset. To better see the underlying mesh, a wireframe overlay is blended in on one half.

ures 2.19(b) and (c) show how the method connects patches of varying LOD levels seamlessly to form a crack-free terrain surface.

As mentioned in the previous paragraph, the rectangular patches correspond to quadtree nodes. Since the position of a node in the quadtree implicitly determines the world-space position and scale of the patch mesh, the corresponding node merely needs to store the resolution levels of the four patch borders, making the LOD management data structure extremely lightweight. During the quadtree traversal, which starts at the quadtree root, i.e., the coarsest detail level of the terrain dataset, Equation (2.7) is evaluated for the four edges of each visible patch. The equation ensures that patches agree upon their edge resolution, resulting in a seamless mesh without cracks by construction. A proof of this claim can be found in Livny et al. [31].

To render the terrain efficiently, the algorithm caches all discrete LODs of one trile mesh along with all LOD-transition combinations for one stitching strip in video ram at startup. Since vertex-height displacement happens in the vertex shader, the cached meshes only store two-dimensional footprint coordinates. At runtime, each patch can be rendered by translating, scaling and rotating the cached trile- and stitching-strip geometry. As mentioned at the beginning, the terrain vertices are aligned to heightmap texels, so that vertex texture fetch can be performed with

nearest-neighbor filtering without sacrificing performance. On some graphics cards this gives a performance benefit over bilinear filtering, as already noted in our discussion about GPUGCM in Section 2.3.3.

2.3.5 The Frostbite™ Terrain-Rendering Method

In this section we discuss how the Frostbite™ engine [12, 64] from DICE™ renders terrain in games such as the renowned Battlefield series of first-person shooters. The Frostbite™ terrain subsystem uses a quadtree structure to subdivide the two-dimensional heightfield domain. Each node in the restricted quadtree has a minimum and maximum height associated with it, effectively creating an *axis-aligned bounding box* (AABB) per node. The AABB of each node is then used for view-frustum culling and LOD computations. Note that in general the AABBs are calculated in a preprocessing step, which however does not rule out local terrain-height modifications at runtime, since recalculating the AABBs for a small number of terrain tiles can be carried out with little overhead.

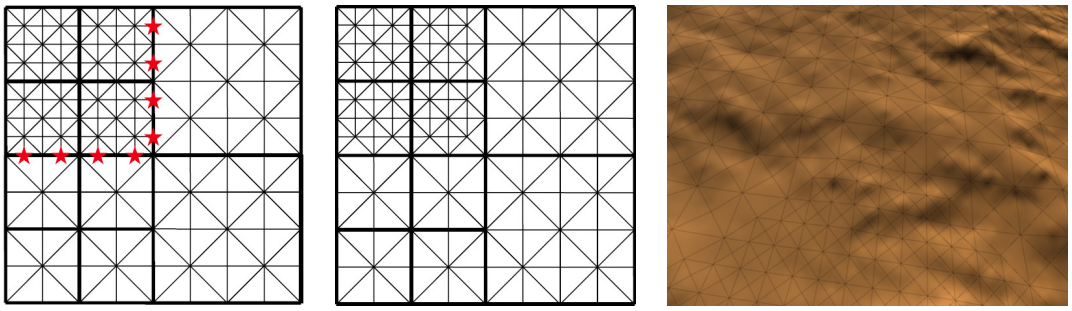
Each quadtree node is represented by a 33×33 regular-grid mesh. When the graphics hardware supports sampling textures in the vertex shader, a single 33×33 vertex buffer suffices to render the whole terrain. This is possible since the only difference between any two patches is their world-space position and potentially their scale, both of which can be adjusted by applying a suitable transformation in the vertex shader. The vertices in the patch only store two-dimensional coordinates, since the third coordinate, i.e., the height displacement, is sampled from the heightmap on the fly.

On platforms that do not support vertex-texture fetch, or on which reading from a texture in the vertex shader is prohibitively slow, a pool of vertex buffers is managed by the application. Whenever the detail level of a patch changes or when new patches come into view, buffers that have become obsolete are filled with data on the CPU and re-uploaded to the GPU. Since heightmap sampling happens on the CPU, the patch vertices now consist of three-dimensional coordinates.

As with other quadtree-based algorithms, the world-space position, scale and resolution of each node are given implicitly by their position in the tree and need not be stored in the quadtree node itself. Quadtree nodes essentially only store the AABB, plus a reference to the concrete vertex buffer when height displacement does not happen on the fly in the vertex shader. By keeping vertices at all resolution levels aligned with the heightmap texels, the heightmap can be sampled with a nearest-neighbor filter without a loss in reconstruction accuracy. Compared to bilinear filtering, nearest-neighbor filtering may perform better on some systems, as already noted in our discussion about GPUGCM in Section 2.3.3.

As many other previously discussed terrain-rendering algorithm, LOD calculations are purely based on camera distance, i.e., local terrain features do not influence the detail level of a patch at all, so that flat areas are rendered with the same amount of triangles as rough parts of the heightfield. Using the same regular grid everywhere not only simplifies the algorithm, but also maps well to current GPUs. Besides, this approach directly supports editing the terrain at runtime.

A straight-forward implementation of the algorithm as described so far creates T-vertices(T-junctions) at adjacent patches of different detail levels, cf. Figure 2.20(a), which means that the reconstructed terrain surface may contain cracks. Even if the heights of the adjacent triangles



(a) Red stars mark T-junctions (b) T-junctions removed (c) LOD transition in action

Figure 2.20: T-junctions emerge when patches at different resolutions meet (a). T-junctions are removed by deleting T-vertices at patch borders, which leads to the replacement of two triangles by one new bigger triangle that matches the tessellation of the next-coarser patch detail level at the outer border of the patch, thereby forming a crack-free transition between quadtree nodes (b). A crack-free LOD transition on a sample dataset with wireframe overlay (c). Images (a) and (b) courtesy of [12].

did always match, rasterization would nevertheless most likely produce pixel-sized holes at a small number of fragments along these edges due to numerical imprecision. These rasterization artifacts are emphasized when the camera moves over the terrain, leading to pixel flickering along edges where the background shines through the terrain surface.

The Frostbite™ engine removes T-vertices efficiently by forcing adjacent quadtree nodes to differ by at most one detail level, i.e., by using a restricted quadtree (see Section 2.1.3), and by using one of nine different index buffers at LOD transitions, cf. Figure 2.21. Each of these index buffers triangulates the patch borders at either full detail, or with all T-vertices removed. When a T-vertex is removed, two triangles are merged into one larger triangle, such that the triangles at patch borders coincide with the triangles of the next-coarser patch resolution, thereby removing the aforementioned cracks, cf. Figure 2.20(b). Figure 2.20(c) shows a crack-free LOD transition on an actual terrain dataset. Note how two adjacent patches share all vertices along their border edges even at LOD transitions, thereby creating a seamless transition, free of cracks and T-vertices. When adapting the patch topology at patch borders for a seamless LOD transition, Frostbite™ only changes the index buffer as needed – the associated vertex buffer itself remains unchanged.

Judging from the presentation about Battlefield 3's terrain system at GDC 2012 [64], apart from adding support for tessellation shaders on DirectX 11 and OpenGL 4 class graphics cards, the basic terrain-tile tessellation, and LOD-transition handling has basically not changed since their SIGGRAPH talk in 2007 [12]. In fact, hardware tessellation works totally orthogonal to the base algorithm, and enables rendering high-frequency detail more realistically. Further research on incorporating hardware tessellation into the Frostbite™ engine has been conducted by Albert Cervin in his master thesis [15].

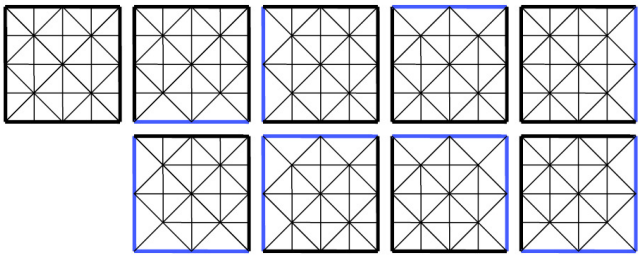


Figure 2.21: The 9 permutations for T-junction-free LOD transitions. Edges that are adjacent to patches of the next coarser detail level are displayed in blue, while black edges have neighbor patches at the same resolution. Image courtesy of [12].

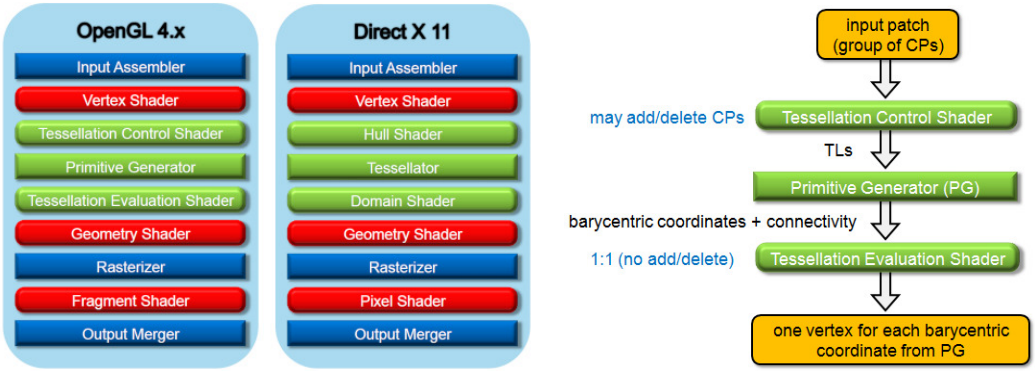
2.3.6 Brute-Force Hardware Tessellation

Hardware tessellation was introduced with the OpenGL 4.x and DirectX 11 Graphics APIs, and marked a paradigm shift in the evolution of consumer-level GPUs. Hardware tessellation enabled efficient dynamic surface refinement for the first time, making effects such as adaptive silhouette refinement and displacement mapping on lots of scene objects possible out of the box. In fairness, the *Geometry Shader* (GS) stage, which arrived one graphics-card generation earlier with OpenGL 3.2 and Direct X10 , already allowed the creation of new vertices entirely on the GPU (consult the GS section of the DirectX 10 online documentation [3]). However, using the GS for general triangle subdivision to realize the LOD hierarchy in a terrain-rendering algorithm was hindered by two factors. First of all, the number of vertices that can be created for each input primitive is fairly limited, secondly performance is severely impacted when many new vertices are created on the fly. Hence it becomes clear that the GS is not well suited for general GPU-based vertex-stream magnification [4, Chapter 4.6], which is also why we will not cover GS-based mesh refinement here. Subdividing vast meshes dynamically on the GPU is a task that the hardware-tessellation unit is better suited for.

In the following we will give a quick overview of how hardware tessellation works, so that we can then briefly sketch a simple terrain-rendering algorithm that exploits hardware tessellation.

Hardware tessellation comprises the *Tessellation Control Shader* (TCS), the *Primitive Generator* (PG), and the *Tessellation Evaluation Shader* (TES), of which only the TCS and TES are programmable. Figure 2.22(a) shows the graphics pipeline with the three newly introduced hardware-tessellation stages as modeled by the two popular graphics APIs OpenGL [52] and DirectX (we will stick with the OpenGL terminology for the rest of this section). Figure 2.22(b) reveals the high-level dataflow through the individual hardware-tessellation stages.

To control the tessellation of a patch, the TCS exports the inner and outer tessellation levels. Figure 2.23 illustrates how a triangle is subdivided based on two different inner tessellation levels. The inner tessellation level basically controls how often the triangle is “nested” within itself. The outer tessellation levels then control how many times to subdivide each edge of the original input primitive. When tessellating a triangle, we therefore need to provide one inner and three outer tessellation levels. Besides triangles, the PG can operate on quads and isolines, but we do



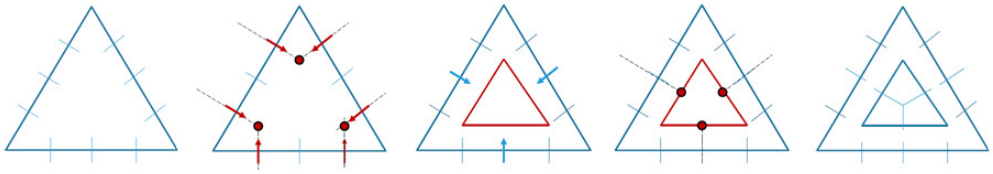
(a) (b)

Figure 2.22: Overview of hardware tessellation. Image (a) shows the hardware-tessellation stages in green in the OpenGL 4.x and DirectX 11 graphics pipelines. Round rectangles represent programmable stages, while rectangular boxes represent fixed-function stages. Image (b) shows the dataflow through the three tessellation stages in more detail. The *Tessellation Control Shader* (TCS) is fed with a number of *control points* (CPs) which make up a patch . The TCS configures the *Primitive Generator* (PG) by writing the inner- and outer- *tessellation levels* (TLs), the TES tells the PG to operate on triangles. The PG then subdivides the patch domain. Finally, the *Tessellation Evaluation Shader* (TES) maps the relative coordinates that are calculated in the PG to an actual vertex position.

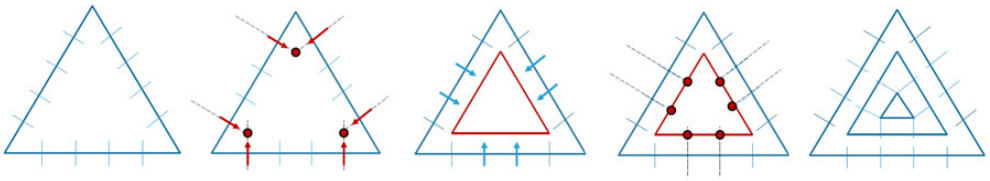
not pursue these other abstract patch types any further here. Note that, somewhat anachronistically with regards to the position of the three tessellation stages in the graphics pipeline, it is actually the TES that defines the type of abstract patch that will be tessellated by the PG [10]. The PG doesn't actually subdivide a patch based on its vertex positions, but rather uses the tessellation levels to generate a "subdivision template" in the patch's normalized parameter space. For example, when tessellating a triangle input patch, the PG generates barycentric coordinates for each output vertex. Finally, the TES is responsible for mapping the normalized subdivision coordinates (the output from the PG) to actual vertex positions, much like the vertex shader does when neither a geometry shader, nor hardware tessellation are active.

Hardware tessellation can be put to work in a brute-force terrain-visualization system by rendering a set of equal-sized, viewer-centered patches. The desired view distance determines the number of patches that need to be rendered. Coarse view-frustum culling at the patch level is performed on the CPU.

Patches represent a terrain tile, and are made up of only four vertices, which form a rectangle that spans the entire area of the terrain tile. Each patch border is assigned a subdivision factor based on the distance of that edge to the camera. The subdivision factor is interpreted as the outer tessellation level of the patch. Since the same outer tessellation level is chosen for shared edges of two tiles, a watertight terrain surface is guaranteed by construction. Similarly to the outer tessellation level, the inner tessellation level is also calculated based on the distance between the camera and the tile.



(a) Inner tessellation level is 4



(b) Inner tessellation level is 5

Figure 2.23: Triangle subdivision based on the inner tessellation factor. The inner tessellation factor determines into how many pieces the triangle edges are subdivided. Then, lines that pass through the outermost subdivision points along the edges and that are orthogonal to the edge are intersected, forming a new, smaller nested triangle. Now the previously second to outermost points of the initial triangle become the outermost points of the first-level nested triangle. The process continues until there are only two or even just one subdivision point(s) left on each edge. The first and second row illustrate the process for an inner tessellation level of 4 and 5, respectively. The first column shows the input triangle, the last column visualizes the result.

To summarize, apart from the lightweight view-frustum culling step, this simplistic view-dependent terrain-rendering technique runs entirely on the GPU. Figure 2.24 shows a screenshot of the outlined method in action.

More advanced terrain-visualization techniques that are based on hardware tessellation, were presented by Cantley [14], Yusov [66], and Cervin [15], to name just a few.

2.3.7 GPU Ray-Casting for Scalable Terrain Rendering

At ever-increasing screen resolutions, maintaining a pixel-sized screen-space error becomes a challenge, since increasingly higher triangle counts need to be displayed. It is actually not only the increased triangle count that slows rendering down, but rather the fact that those triangles are small. Drawing many small triangles not only reduces z-cull efficiency, as mentioned in GPU programming guides for several NVIDIA chip generations [4, 5, Chapter 3.6.3], but also puts additional stress on the rasterizer [24]. A viable alternative to rasterizing small triangles is to perform first-hit ray casting on the GPU, which, in the context of terrain rendering, is typically implemented in the form of ray marching. In short, raymarching describes the process of iteratively sampling the heightmap at positions along rays that emanate at the camera and pierce one screen pixel after the other. The ray-terrain intersections are found by comparing the ray's height

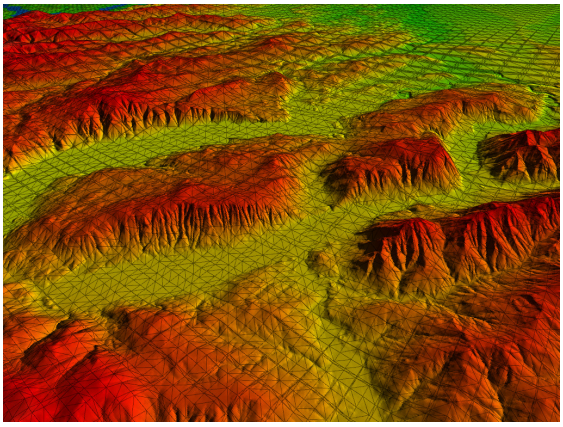


Figure 2.24: Brute-force hardware-tessellated terrain in action, rendered with wireframe overlay.

over the ground plane against the actual terrain height at the current sampling position. By writing not only the terrain color at the intersection point to the framebuffer, but also the depth of the intersection position to the z-buffer, the ray-marched terrain integrates correctly with any rasterized object. GPU-based ray marching is typically accomplished by rendering a screen-sized quad and letting the fragment shader perform the iterative search for the ray-terrain intersections, so that both the vertex processor and the rasterizer are basically left idle. One big advantage of terrain ray casting regarding performance tuning is that while we can't do anything about the performance limits of rasterization from a software application, we do have room for optimizing the ray-casting process by speeding up the search for the ray-terrain intersections. So even though brute-force ray-marching isn't exactly fast either – the main bottleneck being the large amount of texture fetches per fragment – researchers have demonstrated that, when properly used, terrain raycasting can in fact be faster than traditional terrain-surface triangle rasterization. In this section we will give an overview of one such technique called *GPU Ray-Casting for Scalable Terrain Rendering* [20] (GPURayCast), as well as the follow-up paper *GPU-Aware Hybrid Terrain Rendering* [21], in which the authors describe that switching between rasterization and ray casting on a per-tile basis, depending on factors such as terrain roughness and distance of a tile to the camera, can increase overall performance.

GPURayCast divides the heightfield domain into tiles that are stored in a quadtree. Additionally, each tile maintains a maximum mipmap for its corresponding section of the heightfield. The maximum-mipmap pyramid of the heightmap texture is equivalent to a hierarchy of *axis-aligned bounding boxes* (AABBs), and is an essential acceleration data structure in the GPURayCast algorithm through which empty-space skipping is enabled. The maximum-mipmap pyramid is generated in a preprocessing step, and can be calculated just as a normal mipmap pyramid, but instead of averaging the four texels of the next higher level, the maximum operator is used when calculating each successive pyramid level, cf. Figure 2.25. Note that since the heightmap texture and the maximum-mipmap texture are identical at mipmap level 0, that mipmap level need not be stored in the maximum-mipmap pyramid, thereby reducing the additional memory requirements. The apex of the maximum-mipmap pyramid is the maximum value that appears

(a) 16 texels at mipmap level i

(b) 4 texels at mipmap level $i + 1$ (average of source texels)

(c) 4 texels at mipmap level $i+1$ (maximum of source texels)

Figure 2.25: Going from one mipmap level (a) to the next coarser one in a normal mipmap pyramid (b) and a maximump mipmap pyramid (c). The blue region highlights which source texels from mipmap level i contribute to the corresponding value at mipmap level $i + 1$.

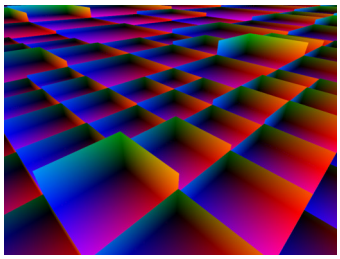


Figure 2.26: AABB backfaces. Colors encode coordinates of surface positions within a tile. The Red and Blue component of the RGB color triple represent the local offset of the position within the tile's 2D footprint, while the intensity of the Green component corresponds to the fraction of the maximum terrain height.

in mipmap level 0. Since the maximum-mipmap pyramid of individual tiles can be evaluated quickly, terrain modifications at runtime are well doable.

Rendering in GPURayCast starts with traversing the tile quadtree, determining visible tiles at an appropriate detail level based on their distance to the camera. The AABB back faces of each visible tile are then rendered front-to-back using standard rasterization, where tile position and scale are given implicitly by the position of the node in the quadtree. The colors of the back faces encode the coordinates of the surface position within a tile, cf. Figure 2.26. Note that the figure only serves illustrative purposes, since in reality we do not actually render the colored back faces to any onscreen or offscreen render target. Rather, the back-face rasterization kicks off the ray marching process for that tile.

The rasterized back-face fragments represent the points at which rays exit the tile, so we only need to calculate the ray-entry points for the tile, which we do by evaluating a simple ray-box intersection. Note that rasterizing tile-AABB back faces instead of front faces has the advantage that it also works when the camera is inside a tile's AABB. If front faces were used,

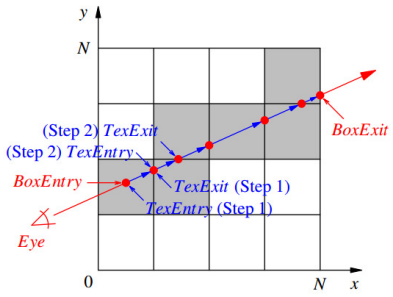
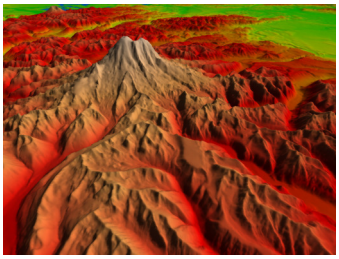
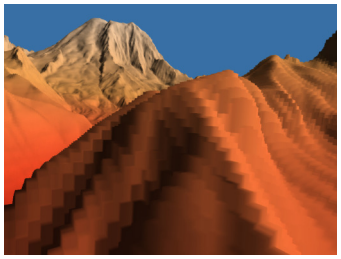


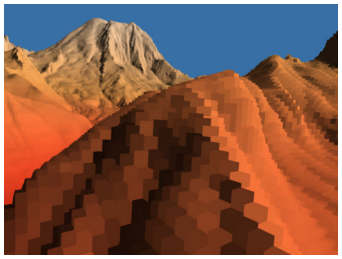
Figure 2.27: DDA ray marching makes sure that no heightmap samples along a ray are missed. The first sampling position (BoxEntry) seems to not be aligned with the two-dimensional raster grid of the heightmap texture – this is due to the ray entering the tile's AABB from the top. The ray exists through the surface on the right side of the tile (BoxExit). Image courtesy of [20]



(a) Distant view.



(b) Close-up with bilinear color interpolation.



(c) Close-up with nearest-neighbor color interpolation.

Figure 2.28: Image (a) shows a distant view of Mt.Rainier ($8K \times 8K$ Puget Sound dataset). Images (b) and (c) show a zoomed-in view of the same dataset, revealing individual heightmap samples, where (b) uses bilinear color interpolation, and (c) uses nearest neighbor colormap filtering, respectively.

they would be culled for the tile that the camera is in, and no fragments would be created to get the ray-marching process started.

By performing ray-terrain intersection tests only between the ray-entry and ray-exit points, we can be sure that we only perform work that is relevant for this specific tile. As noted in the beginning, ray-terrain intersection tests are determined by iteratively sampling the heightmap at consecutive positions along the ray, starting at the tile's ray-entry point. GPURayCast employs a variant of the *digital-differential-analyzer* (DDA) algorithm (see Figure 2.27), and therefore never misses a heightmap sample, creating a perfectly stable reconstruction of the terrain. Figure 2.28 shows close-ups of the resulting reconstructed terrain.

GPURayCast speeds up raymarching by reducing the number of sampling positions that need to be visited between the ray-entry and ray-exit positions, i.e., by employing empty-space skipping. This is where the maximum-mipmap hierarchy mentioned earlier comes into play, and in order to exploit it, raymarching starts at the second-to-coarsest mipmap level in the maximum-mipmap hierarchy, i.e., the mipmap level at which the tile is represented by only 2×2 texels,

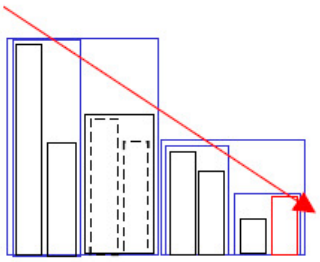


Figure 2.29: Accelerating the ray-terrain intersection test with hierarchical raymarching. Dotted lines depict texels that don't need to be tested for ray-terrain intersections. The ray hits the landscape surface at the red texel. Image courtesy of [63].

since a texel T in the maximum-mipmap pyramid forms a hull around all texels in higher-detail mipmaps that fall within T's footprint. Due to this fact, it suffices to first test for an intersection of a ray with the maximum-mipmap height T, and if the ray doesn't intersect the terrain at T, the ray can't possibly intersect any of the higher-resolution texels contained within T's footprint. All these finer-detail texels can thus be safely skipped. If, however, the ray intersects the terrain at T, we don't yet know if the ray actually hit the terrain, so we continue searching for a ray-terrain intersection within T's area by setting the current mipmap level to the next-finer level. If we reach the highest-detail mipmap level at this point, we are done, since we have found an intersection with the terrain, otherwise we continue the ray-marching loop. Note that when raymarching within a tile, the mipmap level may not only be adjusted toward the higher detail level, but may also be switched back to a coarser level. For example, when raymarching at mipmap level i and no intersection is found at the current sampling position, and this position happens to lie at the border of the texel grid of the next-coarser mipmap level $i+$, DDA traversal will continue at the coarser mipmap level. Of course the DDA step size is kept in sync with the texel size whenever the mipmap level is adjusted.

We see that the ray-marching loop either finds an intersection with the terrain at the finest mipmap level, or leaves the current tile's AABB. Figure 2.29 illustrates how this hierarchical ray-casting algorithm accelerates the ray-terrain intersection by only testing for intersections where absolutely necessary.

Now that we have looked at how GPURayCast speeds up intra-tile raymarching, it is time to talk about how the algorithm exploits inter-tile occlusions to further speed up rendering. Current graphics hardware performs several optimizations, such as early-z rejection, "under the hood". By rendering the tiles front-to-back, GPURayCast takes advantage of inter-tile occlusions to speed up the technique by simply not generating fragments in the AABB-rasterization step, and thus skip the ray-cast loop for these fragments altogether, when an intersection closer to the camera has already been found along the fragment's corresponding ray. However, performing this check in a GPU-friendly way is more involved than it may seem at first sight. As stated in several NVIDIA GPU Programming Guides [4, 5], certain operations disable the early-z optimization. One such performance-hindering operation is writing a custom depth value in the fragment shader. Unfortunately, writing the depth value for a ray-terrain intersection points is

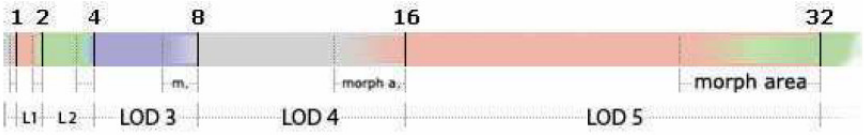
needed to establish a proper z-buffer for the terrain. If the depth of intersection point was not written to the z-buffer, the z-buffer would contain the depths of the AABB back faces, which are typically farther away than the true intersection points. While we can't trick the hardware into still using the optimized hardware path after messing with the depth value in the fragment shader, GPURayCast instead emulates early-z rejection by using a separate floating-point render target to store intermediate depth values. Note that the GPU z-buffer can only be written to, but not read from in the fragment shader, which is why we need the additional custom depth-render target in the first place. Before the actual ray-marching is performed, a per-fragment test checks the value of this custom depth texture against the current fragment's depth, and if the stored depth is smaller, we know that an intersection closer to the camera has already been found, so that the ray-cast loop can be skipped safely. Whenever a ray-terrain intersection point is determined, its depth is not only written to the z-buffer, but also to the custom depth render target to keep the custom depth texture in sync with the hardware depth buffer.

2.3.8 Continuous Distance-Dependent Level of Detail for Rendering Heightmaps

The final terrain-rendering algorithm that we discuss in the context of our previous-work analysis is *Continuous Distance-Dependent Level of Detail for Rendering Heightmaps* [56] (CDLOD) by Filip Strugar. As many other previously presented terrain-rendering algorithms, CDLOD deals with LOD management and view-frustum culling by subdividing the terrain area into rectangular tiles, which are kept in a quadtree. Each terrain tile additionally stores the minimum and maximum terrain height that occurs within the part of the heightmap that is covered by the tile. These minimum and maximum height displacements are determined in a preprocessing step, and enable us to define an AABB for each tile, which, as we shall see in a moment, will be put to work not only for view-frustum culling, but also for smoothly morphing between detail levels.

All quadtree nodes are represented by the same regular grid that is stored in a vertex buffer in video memory. The grid's dimensions are $2^m \times 2^m, m \in \mathbb{N}$, where typically $4 \leqslant m \leqslant 7$. By properly scaling and translating this vertex buffer, the single mesh can represent any node in the quadtree hierarchy. As all terrain tiles are represented by the same vertex buffer, CDLOD performs height displacement by reading the heightmap at each vertex position in the vertex shader to reconstruct the surface of the landscape.

For selecting the right resolution level of a tile, CDLOD precalculates LOD ranges, i.e., the range between the two distances to the camera at which the detail level starts and ends. The author proposes that the range that a detail level covers should be twice the range that the next coarser detail level covers, with the farthest distance being equal to the view distance, cf. Figure 2.30. He argues that, when taking into account that the area that a quadtree node covers is four times the area that each child of that quadtree node covers, this yields a near constant triangle density on screen under perspective projection. In the figure, we notice two things. First, LOD ranges with increasing numbers denote distances that are farther away from the camera, and, more importantly, we see that when the distance to the camera approaches the end of one detail level, or equivalently, the start of the next coarser detail level, a morphing area is entered. Typically, the morph area spans the last $15 - 30\%$ of each LOD range. Within the morphing area a morph factor is linearly interpolated between 0 (no morphing applied yet) and 1 (fully morphed to the next coarser detail level). The morph factor smoothly transforms the



(a) Precomputed ranges for LOD switches.

Figure 2.30: Precomputed distance ranges for LOD switches, and morph areas. The ranges are chosen such that triangle density in screen space is approximately constant over all detail levels. Image courtesy of [56].

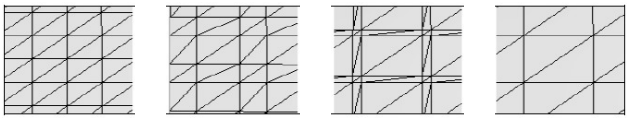
full-detail mesh into the mesh of the next coarser detail level, so that by the time the detail level is switched, the LOD change remains unnoticed.

At runtime, CDLOD traverses the quadtree starting at the root node, i.e., from the coarsest terrain representation. For each visited node, the AABB of the associated tile is compared to all spheres around the viewer, having radii from the LOD-ranges array mentioned previously, starting at the largest radius. Remember that traversing the quadtree implicitly determines the two-dimensional node position as well as the scale, and together with the minimum and maximum terrain height stored in the node, we can define an AABB for the terrain tile. This AABB is used in the following intersection tests. If a tile is completely inside the shape defined by Equation (2.8), i.e, in the area that lies inside the sphere with a radius that correspond to the LOD range for LOD $i+1$, but outside of the sphere for the next finer LOD, then the subdivision stops and the tile is rendered at LOD $i+1$.

$$sphere_{radius(LOD(i+1))} - sphere_{radius(LOD(i))} \tag{2.8}$$

Otherwise the corners of the tile's AABB lie either in LOD range $i+1$ or closer to the camera than LOD range i. The tile is then subdivided into four equal-sized children according to the rules of quadtree subdivision. In the case that a child's AABB still intersects the sphere having a radius equal to LOD range $i+1$, and the other corners do not intersect the next closer LOD-range sphere, that child is rendered at LOD $i+1$. Otherwise, the traversal carries on recursively until all quadtree nodes are properly subdivided according to their euclidean distance to the camera. Note that view-frustum culling is a byproduct of the quadtree traversal, since once a tile's AABB is touched, that AABB can be readily tested against the view-frustum, so that the tile can be discarded if it lies completely outside.

Interpolation between LOD levels is performed by morphing the high-detail mesh into a low-detail mesh. Once morphing is done, blocks of four quads (made from four groups of triangles) are collapsed into one quad (made from one triangle pair) that covers the same area. Obviously, this can only be achieved when morphing not only changes vertex heights, i.e., the y coordinate, but also the 2D vertex coordinates, i.e., the x- and z-coordinates, cf. Figure 2.31. Note that most other terrain-rendering algorithms that employ vertex morphing merely adjust vertex heights. The morphing in CDLOD actually looks very similar to the transitions between tessellation levels when using hardware tessellation (see Section 2.3.6), without requiring graphics hardware with hardware-tessellation support.



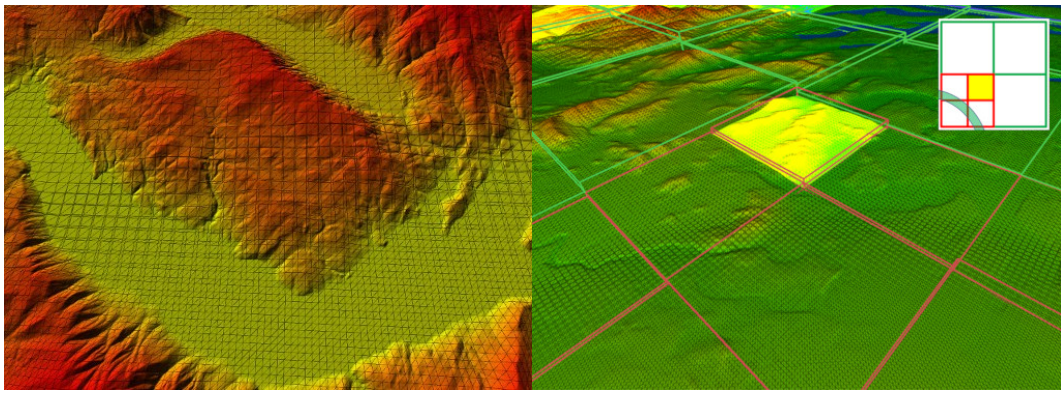
(a) $k = 0$ (b) $k = 0.2$ (c) $k = 0.6$ (d) $k = 1$

Figure 2.31: Morphing from a high-resolution mesh to the next-coarser resolution, illustrated for various morph factors k. Fully morphed vertices line up perfectly with the non-morphed vertices of tiles at the next-coarser resolution level, which ensures unnoticeable LOD switching. Note that when a tile is fully morphed to the coarser resolution, most triangles in the mesh degenerate to zero-area triangles, thus the tile can then be rendered more efficiently by using a different index buffer which skips the otherwise degenerate triangles. This coarse-mesh index buffer uses only every second vertex to form triangles. Images courtesy of [56].

An important property of CDLOD is that euclidean 3D distances to the camera guide the LOD selection, which achieves uniform triangle density for all distances everywhere on the screen, and enables vertex-popping-free LOD transitions under all viewing conditions. In contrast, for example GPUGCM [13](see Section 2.3.3) only compares distances between 2D footprints on the ground plane. While the finest detail levels are successively dropped when the camera is gaining altitude, this causes slight vertex-popping artifacts. When comparing CDLOD to the previously discussed quadtree-based landscape-visualization techniques, morphing is a lot smoother, since it is applied per vertex instead of only per tile (as in ChunkedLOD [59], see Section 2.3.2) or not at all (as in Frostbite™ [12, 64], see Section 2.3.5). Figure 2.32(a) shows smooth LOD transitions in action on the Puget Sound dataset.

In Figure 2.32(b), several of the four sub tiles of a quadtree node are already morphed to the next-lower detail level completely, while others are still in a transition state. Since not all four sub nodes are fully transformed to the next coarser LOD, the four nodes can't be merged to the next-coarser parent quadtree node just yet. CDLOD nevertheless speeds up rendering by switching to low-resolution meshes for the fully morphed sub nodes. The low-resolution meshes actually use the same vertex buffer, and alter solely the index buffer. This low-resolution-mesh index buffer connects only every second vertex, so that only a quarter of the triangles are drawn. Hence, the low-resolution mesh is identical to the fully morphed high-resolution mesh, but can be rendered more efficiently.

To summarize, CDLOD creates a watertight, crack-free mesh, without having to resort to additional geometry at tile borders such as skirts in the Chunked LOD algorithm (see Section 2.3.2), or GPUGCM's degenerate triangles at LOD transitions (see Section 2.3.3). CDLOD does not need to explicitly handle T-vertices, thanks to its vertex-morphing technique, which not only interpolates the vertex heights of two adjacent detail levels, but also alters the two-dimensional vertex-footprint positions, so that fully morphed tiles are identical to tiles at the next coarser LOD level. Hence, when the detail level changes, there are absolutely no vertex-popping artifacts. In contrast, for example Frostbite (see Section 2.3.5) removes T-vertices at any tile border by switching between several different index buffers, which always causes a slight terrain-geometry change that can potentially be spotted by the user. Further, CDLOD's



(a) (b)

Figure 2.32: Image (a) shows how a higher detail level is morphed smoothly to the next coarser detail level. Image (b) shows an opportunity for reducing the triangle count without degrading terrain-reconstruction quality. The edges of each tile's AABB are color coded according to their quadtree depth, yellowish bright tiles accentuate quadtree nodes which have already fully morphed to the next coarser level. The inlay in the upper right corner shows a color-coded bird's eye view of a part of the current quadtree subdivision, indicating a morph area with a transparent green arch. For the view shown in image (b) we see that we can't merge all four sub tiles to form the next-coarser quadtree node, since not all sub tiles have finished their LOD transition yet. Still, those sub tiles that have fully morphed to the next coarser level can be represented by an equivalent low-resolution mesh which contains only a quarter of the original mesh's triangles.

LOD switching and morphing is based on the three-dimensional euclidean distance of a vertex from the camera, making it work in a stable fashion for all possible viewing conditions, without having to handle any special cases.

2.4 Temporal Coherence

In many computer-graphics applications, the similarity between consecutive frames, i.e., their temporal coherence, is typically high. Therefore, rendering can be sped up by reusing data from previously rasterized frames instead of calculating that very same data for each pixel of the current frame anew, even more so in the case of expensive per-fragment calculations. Besides the performance benefit, temporal coherence also helps reduce temporal aliasing, which manifests itself as flickering.

The straight-forward approach for accessing data from the past is to simply store each elapsed frame, or at least a limited number of elapsed frames. Keeping several elapsed frames in memory, however, is wasteful and impracticable. Scherzer et al. [48] present a technique that requires only one screen-sized memory area – the so-called history buffer, or equivalently reprojection cache [41, 46]. When using a history buffer in practice, the history buffer actually needs to be double buffered, since the history buffer is both read from and written to in the same

pass – an operation that leads to undefined results in current graphics application programming interfaces.

When representing the history buffer with two textures, we can circumvent this issue by using one of the textures exclusively for reading, and the other one only for writing. After each frame, the roles of the two textures are swapped – the texture that was written to in the previous frame becomes the new source texture to read from, and the texture that was previously read from acts as the new render target that is written to. This approach is known as "ping ponging".

The history buffer is updated incrementally once per frame according to Equation (2.9), which is evaluated for each screen pixel.

$$H_t(x,y) = w_t(x,y)\,F_t(x,y) + (1 - w_t(x,y))\,H_{t-1}(x,y) \tag{2.9}$$

In Equation (2.9), H_t refers to the history buffer at time t, F_t denotes the newly rasterized frame at time t, and w is the blend weight, where $0 < w_t(x,y) \leq 1$. The blend weight determines the influence of the current frame and the history buffer, respectively. Although the blend weight is defined as a function that potentially varies per frame and fragment position, it is typically chosen to be a globally exposed user-adjustable parameter, since a single parameter is easier and more intuitive to tweak. Equation (2.9) actually describes exponential smoothing, and as such contains a recursion which potentially captures an infinite history for a pixel, where the influence of elapsed frames falls off exponentially.

The direct approach to access data from the past is to transform the previous frame into the current frame. Unfortunately, this forward transformation has several shortcomings, which stem from the fact that the transformed pixels from the past frame hardly ever align perfectly with the current frame's pixel grid. Firstly, the forward transform may lead to gaps in the current frame, because some pixels in the current frame may never be hit by a transformed pixel. Secondly, some pixels in the current frame may be hit by multiple transformed pixels, introducing potential indeterminism and flickering if the order in which pixels in the current frame are overwritten varies from frame to frame. For this reason, Scherzer et al. [48] recommend using a reverse history-buffer lookup instead. For the reverse lookup, they transform pixel positions from the current frame back into the previous frame. The reprojected position is then used for accessing the reprojection cache.

While the reverse lookup avoids pixel gaps and pixels being written to more than once, fragment centers from the current frame are not guaranteed to map perfectly to texel centers in the history buffer either, which is to some extent addressed by bilinearly interpolated history-buffer fetches, though. Regrettably, bilinear filtering introduces an unwanted incremental blurring over time. The blurring can be easily explained by looking at Equation (2.9) again – the given feedback loop reveals that small errors will be accumulated into the history buffer when holding on to reprojected values for an extended number of frames. To lessen the blurring, the history buffer is invalidated for randomly chosen fragments at random points in time, which forces the history buffer to be partially refreshed with newly rendered fragments. An important implementation detail is to not refresh individual fragments in a random manner, but rather refresh blocks of fragments, since forcing the fragment shader to take separate branches for fragments within a block has a negative impact on performance. This happens because GPUs typically process fragments at the granularity of small blocks instead of on a per-fragment basis. When the flow

of execution within such a fragment block diverges, the GPU falls back to predication for all fragments in the block, which means that both sides of a branch are evaluated all the time. At the end of the branch one of the results is discarded, depending on the evaluated branch condition [27]. In the case of history-buffer refreshing this means that if just one of the fragments in a block takes a different branch due to a random refresh, all the branches in the block will be forced to evaluate both the history-buffer lookup and the full shading model, thereby squashing the theoretical performance improvements. An alternative approach to the just described forced partial history-buffer refreshing is presented by Valient [61]. He employs the back-and-forth error compensation and correction method by Selle et al. [53] to tackle the numerical diffusion problem.

Reprojecting a fragment from the current frame into the previous frame is described by Equation (2.10), where p_t and p_{t-1} are the homogeneous *normalized device coordinates* (NDC) coordinates of a fragment in frame t and frame $(t-1)$, respectively, and V and P are the view- and projection matrices of the respective frames.

$$p_{t-1} = P_{t-1} * V_{t-1} * V_t^{-1} * P_t^{-1} * p_t \tag{2.10}$$

While this equation maps the current fragment's position into the previous frame, we cannot be sure that the reprojection cache contains valid information there. Firstly, camera movement can cause parts of the history buffer to become obsolete, secondly, surface positions may have been occluded in the previous frames, cf. Figure 2.33. The shown camera movement causes the right border of the previous frame to become obsolete. Obviously, it doesn't make sense to reuse those parts of the history buffer, since they no longer contribute to the current frame. At the same time, the current frame exhibits "new" information on its left border and when an occluder no longer blocks the line of sight – in this context "new" refers to parts of the scene that have not been seen by the main camera in the previous frame. Thus, whenever new surface points come into view, they lead to an unavoidable reprojection-cache miss, cf. Figure 2.33. In the next two paragraphs, we show two possible checks to determine if a reprojected fragment results in a reprojection-cache hit or a reprojection-cache miss.

Reprojected positions from the current frame that lie outside the previous frame's view frustum can be quickly rejected with the following test. Let $p_{t-1} := \begin{pmatrix} x, & y, & z, & w \end{pmatrix}^T$ denote the reprojected position, then p_{t-1} is within the previous view frustum if and only if the relation in Equation (2.11) is fulfilled.

$$\begin{pmatrix} -w \\ -w \\ -w \end{pmatrix} \leq \begin{pmatrix} x \\ y \\ z \end{pmatrix} \leq \begin{pmatrix} +w \\ +w \\ +w \end{pmatrix} \tag{2.11}$$

If a fragment passes the above test, a second check is performed to verify that the fragment had not been occluded by closer geometry in the previous frame. To that end, the calculated depth of the reprojected fragment is compared to the depth that is fetched from the history buffer at the reprojected fragment's 2D coordinates. If the depth difference exceeds a certain threshold, we conclude that the current fragment does not have a valid history, and must therefore be treated as a new fragment, cf. Figure 2.34. In practice, the history buffer often stores the world-space distance of the surface position to the camera, since it is easier to compare differences between linear depth values instead of differences between two z-Buffer values that result from

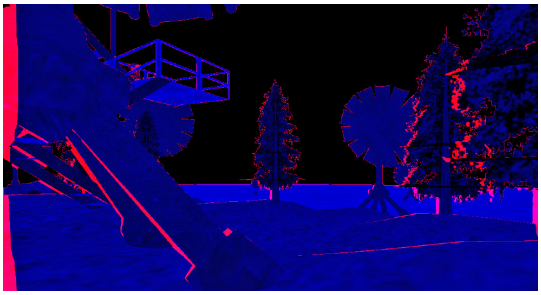


Figure 2.33: Determining which fragments have a valid history in the reprojection cache when the camera performs a strafe movement to the left. New fragments, i.e., parts of the scene which have just come into view in the current frame, are shown in red. These fragments don't have a valid history, since they were previously either outside the view frustum or occluded. Since the history buffer contains no data for those fragments, they must be calculated anew. In contrast, blue fragments do have a valid history, and so for them the potentially time-consuming fragment-level shading calculations can be replaced by a fast history-buffer lookup. Image courtesy of [48].

rasterization. For example, the standard projection matrix in OpenGL [54, p. 834] produces fragment depths that are proportional to the reciprocal of the distance of a surface position to the camera, and therefore nonlinear.

The following list itemizes several application areas that can benefit from the application of temporal coherence methods.

- Motion Blur and Depth of Field
  Nehab et al. [41] point out that both motion blur and depth of field effects can be accomplished by combining several images resulting from rendering the scene many times from different points of view or at slightly different times in each frame. Each of those individual images exhibits high spatial and temporal coherence, enabling the application to skip expensive shading calculations on all but the first of the many consecutive passes. The remaining passes of the current frame are colored by reprojection into the first image.

- Stereoscopic Rendering
  Lately, headset-based *virtual-reality* (VR) applications have reached a wide audience. The big challenge that developers of those applications face is that the scene must theoretically be rendered twice – once for each eye. Nehab et al. [41] exploit the similarity of the two resulting images. They render the scene with full surface shading only for one eye, and skip most of the shading calculations for the second eye by reprojecting the first, fully rendered frame onto the second frame instead.

- Shadow Mapping
  Scherzer et al. [48] describe how temporal coherence can be applied to shadow mapping in order to yield high-quality shadows. They observe that while shadows resulting from a single low-resolution shadow map lead to low-quality shadows, some pixels are actually

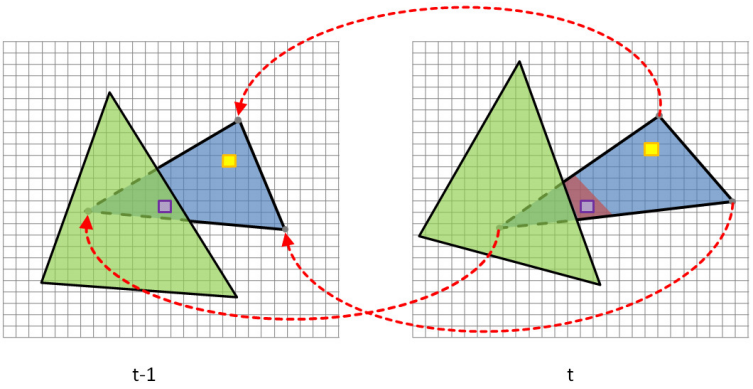


Figure 2.34: Mapping pixels from the current frame t to the previous frame $t-1$. Both the yellow and the purple fragment in frame t map to a position within the view frustum of the previous frame $t-1$, and yet only the yellow fragment has a corresponding valid entry in the previous frame. The purple fragment's reprojected depth differs from the stored depth in the reprojection cache, since that fragment's corresponding surface position was occluded in the previous frame, hence, access to the history buffer results in a cache miss. Note that in this example the reprojected fragments on the left align perfectly with the pixel grid. In general, this is hardly ever the case, so that we apply bilinear filtering when fetching data from the history buffer.

shadowed flawlessly. The pixels that are shadowed correctly are those that correspond to a surface position in the scene that happens to project exactly to the center of a shadow map texel. All other pixels sample the shadow map inaccurately, thus causing jaggy shadow silhouettes and flickering artifacts under camera motion. By randomly jittering the shadow camera a bit each frame, the set of pixels that are calculated exactly varies from frame to frame. They accumulate the best shadow results from all previous rasterizations in the history buffer, so that the shadows iteratively converge to a pixel-perfect solution. The metric used to assess the quality of a shadow sample is based on the distance of the shadow-sample position to its shadow-map-texel center. This confidence measure determines the weight in the exponential smoothing Equation (2.9). Their proposed method works independently of the specific shadow-mapping algorithm employed, and benefits from any improvement that manages to redistribute shadow-map texels in such a way that, depending on the surface orientations in the scene and the light and camera setup, a near one-to-one correspondence between screen pixels and shadow-map texels is established.

- Screen-Space Ambient Occlusion
  The quality of screen-space ambient occlusion depends on the number of samples taken to estimate an occlusion factor for each pixel, however, simply increasing the sample count may be too costly for some applications. By exploiting spatio-temporal coherence, those additional lookups to estimate the occlusion factor can be spread across several frames. The individual results are then accumulated in the history to again increase image quality incrementally over time. The briefly sketched amortized sampling method is described in

more detail by Mattausch et al. [39].

- Temporal Interlaced Rendering and Upsampling
  Valient [61] presents a way to reconstruct output frames at 1080p in "Killzone Shadow Fall" by rendering half-resolution frames at 960×1080 only. The TC-assisted upsampling relies on the two most recent half frames and a full-resolution history buffer. Valient's method was later picked up and further refined in "Rainbow Six Siege" [38].

2.4.1 Bidirectional Temporal Coherence

The temporal-coherence method described so far reuses information from the past to speed up rendering of new frames. *Bidirectional temporal coherence* (BidirTC) is an extension of the temporal-coherence algorithm, which reuses information from both the past and the future. BidirTC was introduced by Yang et al. [65].

In order to distinguish bidirectional temporal coherence from the temporal-coherence technique described in the beginning of this section, we refer to the already introduced method as unidirectional, or accumulative temporal coherence. Both terms seem fit: we have seen that the previously described temporal-coherence technique performs reprojection in one direction only, namely into the past. Furthermore the update of the history buffer according to the exponential-smoothing equation (2.9) accumulates weighted influences of a potentially infinite amount of past frames. In the following we will use both terms interchangeably, sometimes preferring one over the other in order to emphasize either the unidirectional or the accumulative aspect.

As just stated, while unidirectional temporal coherence only reuses information from the past, bidirectional temporal coherence references information from both the past and the future to deduce the current "intermediate" frame. The frames that are rendered "normally" are called *intra frames* (I-frames). The interpolated images between two I-frames are referred to as *bidirectionally predicted frames* (B-frames). They are reconstructed from the pair of I-frames that they are enclosed by – their past- and future I-frame. The terms I-frame and B-frame are borrowed from the field of video encoding and decoding. While in compressed videos all frames are already predetermined, we clearly can't predict the future in interactive real-time applications. Instead, BidirTC redefines the notion of what "past" and "future" mean. The future I-frame in BidirTC is actually the most recently calculated I-frame. Once a new I-frame is available it becomes the new future I-frame. The I-frame that acted as the future I-frame up to now becomes the new past I-frame.

The image that is actually presented to the user is either I-frame F_t at time t, or an interpolated B-frame at the $n-1$ intermediate points in time, i.e., at times $t + \frac{1}{n}$ to $t + 1 - \frac{1}{n}$. In the rest of this thesis, we will assume that n is equal to 4, so that the sequence of displayed frames on screen will be one I-frame, followed by three interpolated B-frames. The chronology of the alternation of I-frames and B-frames is illustrated in Figure 2.35.

Yang et al. introduced two approaches to utilize bidirectional scene reprojection: scene-assisted and image-based BidirTC. Figure 2.36 juxtaposes the two approaches visually.

In scene-assisted BidirTC, the scene geometry is rendered for both I-frames and B-frames. I-frames are rendered with full shading, while only depth values are rendered for B-frames. The color values of a B-frame are then reconstructed from the adjacent I-frames by temporal

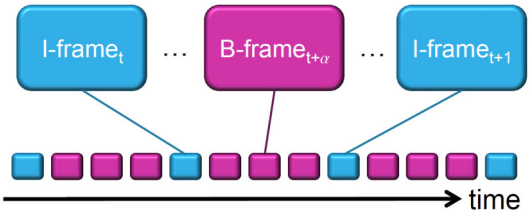


Figure 2.35: I-frames and B-frames arranged on a timeline. B-frames are calculated from two neighboring I-frames according to the interpolation factor α. In this illustration, α increases in increments of 0.25 time units, i.e., $\alpha \in \{\ 0,\ 0.25,\ 0.5,\ 0.75\ \}$. When α is 0.75, and it is time to increment α another time, the past I-frame is quashed, the future I-frame becomes the past I-frame, and the newly rendered I-frame becomes the future I-frame. After this I-frame update, α is reset to 0, and the whole cycle starts over again. Note that when α is 0, an I-frame is directly rendered "as is", otherwise a B-frame is reconstructed from the two closest I-frames.

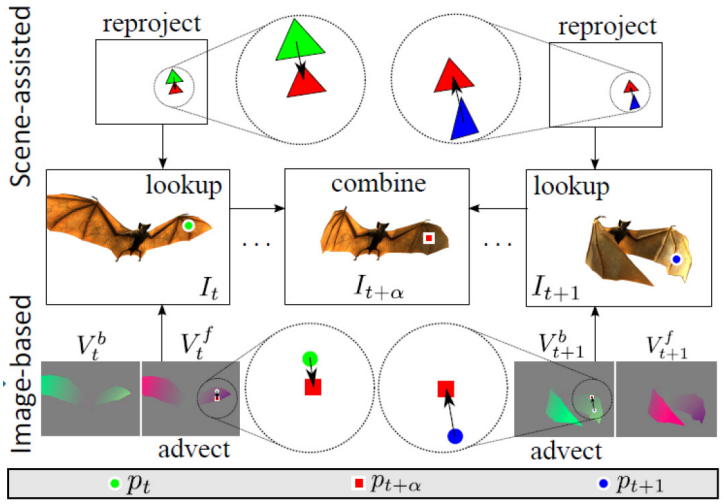


Figure 2.36: Two basic approaches for exploiting bidirectional temporal coherence. Starting from the red triangle in the scene-assisted approach at the top, or, the red fragment in the image-based approach at the bottom, we find the associated information from the past (green, left side) and from the future (blue, right side). The center image shows the interpolated intermediate frame. Image courtesy of [65].

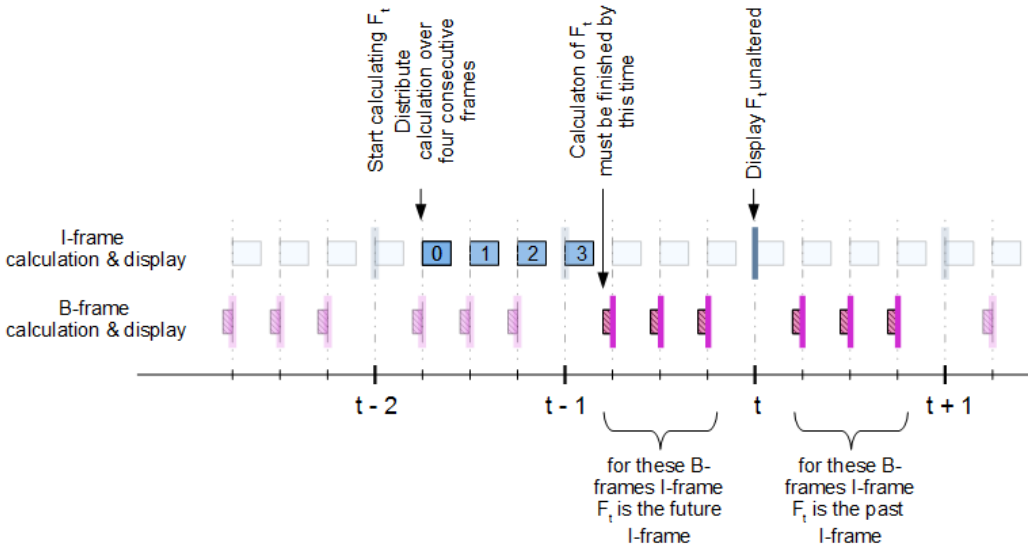
reprojection into the past- and future I-frames. The reprojection works in exactly the same way as in unidirectional temporal coherence. Compared to unidirectional TC the reprojection-cache hit rate of scene-assisted BidirTC is typically higher, though, since even if a lookup in one I-frame results in a reprojection-cache miss, the lookup for the same position often results in a reprojection-cache hit in the opposing I-frame. As scene-assisted BidirTC is essentially unidirectional TC "applied twice", we will not describe the scene-assisted flavor of BidirTC in any greater detail in the remainder of this section.

In image-based BidirTC, B-frames are reconstructed from I-frames only, without rendering any scene geometry at all. For this to work, the I-frames are augmented by three-dimensional optical-flow fields, which guide an image-space reconstruction algorithm. These scene-flow fields consist of the velocity vectors which map a surface position from I-frame$_t$ into I-frame$_{t-1}$, and I-frame$_{t+1}$, respectively. Put differently, adding the forward velocity vector to a pixel position in I-frame$_t$ tells us where that pixel will be in I-frame$_{t+1}$, while adding the backward velocity vector to a pixel in I-frame$_{t+1}$ results in that pixel's position in I-frame$_t$. Because the scene geometry is always "normally" rasterized for I-frames, this mapping is exact.

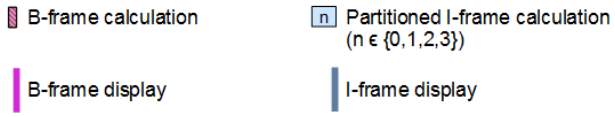
Inn contrast to image-based BidirTC, scene-assisted BidirTC works in a more "traditional" manner – after all, each B-frame has a rasterized depth buffer, hence each B-frame pixel can be directly transformed into the two adjacent I-frames with the already discussed reprojection method that is also employed in unidirectional temporal coherence. While straight-forward to implement, scene-assisted BidirTC does not allow for as pronounced performance improvements as image-based BidirTC. The massive performance improvement of image-based BidirTC over the scene-assisted variant comes from the fact that firstly, the scene geometry is rendered for I-frames only, and secondly, that I-frame rendering can furthermore be distributed over 4 frames, since only every 4^{th} displayed frame is actually an I-frame. This way the rendering load is evenly distributed over several frames. Without this distribution I-frames would put heavy stress on the geometry-processing resources of the system, while B-frames would leave the geometry processor(s) practically idle. Furthermore, compared to I-frame rendering, B-frame interpolation is cheap. The reconstruction of B-frames is performed purely in image space, utilizing the flow fields that are associated with the neighboring I-frames. Hence, performance gains in image-based BidirTC are most noticeable for scenes with high polygon counts as well as for scenes that require expensive shading calculations for many pixels in each frame.

Figure 2.37 shows a timeline with the exact timing requirements of the amortized generation and display of one I-frame F_t, i.e., the I-frame at time t, as well as the points in time at which the calculation and display of those interleaved B-frames, which depend on F_t, takes place. While the distributed computation of I-frames improves the frame rate, it also comes at a price. In order for the split rendering to finally form a consistent frame, the user input, as well as the scene state, must be kept constant over these four frames, leading to a slightly increased lag.

Implementing image-based BidirTC is not as straight forward as scene-assisted BidirTC, since for the intermediate B-frames there is no depth value, and thus no exact 3D position to reproject into the neighboring I-frames. Yang et al. [65] show how the reprojection can nevertheless be accomplished by creating the already mentioned optical flow fields alongside color- and depth fragment data of each I-frame. While calculating the color- and depth buffer for an I-frame, we determine two additional 3D vector fields for both the forward and the backward



(a) Timing of I-frame F_t and the B-frames that depend on it.



(b) Symbols used in (a)

Figure 2.37: Partitioned calculation and display of F_t (the I-frame at time t) as well as the B-frames that depend on it. The computation of F_t is carried out in the interval $[t-2+\frac{1}{n}; t-1+\frac{1}{n}]$, where n denotes the number of frames over which I-frame calculations are distributed. Here n equals 4. The calculation of F_t must be finished before $t-1+\frac{1}{n}$, since the B-frames that are displayed in the interval $[t-1; t+1]$ depend on F_t. As soon as I-frame F_t is calculated, it is used as the future I-frame for the B-frame $B_{t-1+\frac{1}{n}}$ and is last used when reconstructing B-frame $B_{t+1-\frac{1}{n}}$. As can be seen in image (a), F_t acts as the future I-frame for the $n-1$ B-frames $B_{t-1+\frac{1}{n}}$ to $B_{t-\frac{1}{n}}$ and as the past I-frame for the $n-1$ B-frames $B_{t+\frac{1}{n}}$ to $B_{t+1-\frac{1}{n}}$.
Regarding the flow fields, we see that for the B-frames in the interval $[t-1; t]$, we need the forward flow field V^f_{t-1} and the backward flow field V^b_t, while for the B-frames in the interval $[t; t+1]$, we need the forward flow field V^f_t and the backward flow field V^b_{t+1}. Focusing on I-frame F_t, this means that V^b_t must be available at the same time at which F_t's color- and depth buffer are calculated, while V^f_t is not needed until B-frame $B_{t+\frac{1}{n}}$ is rendered. Note that at this time the rendering of I-frame F_{t+1} is already finished, which is absolutely necessary for the forward flow field V^f_t to be calculated, as the I-frame at the next point in time must be known. After all, a forward flow field defines a mapping of an I-frame into its adjacent future neighbor I-frame.

flow of the rendered scene. Let the transformation $\pi_{t\to t'}(\overline{p})$ denote the mapping of the clip-space surface point $\overline{p} = (p.x, p.y, zBuffer\,(p.xy))$ at time t into the clip space of time t'. This transformation is easily determined with the help of reprojection, just as is employed in unidirectional temporal coherence (see Equation (2.10)). The three-dimensional forward- and backward flow fields, respectively, are given by

$$V_t^f(p) = \pi_{t\to t+1}(\overline{p}_t) - \overline{p}_t \tag{2.12}$$

$$V_{t+1}^b(p) = \pi_{t+1\to t}(\overline{p}_{t+1}) - \overline{p}_{t+1} \tag{2.13}$$

We assume that each B-frame coordinate $p_{t+\alpha}$ is related to coordinate p_t in the past I-frame by a fractional linear displacement along the forward flow field, and to coordinate p_{t+1} in the future I-frame by an according displacement along the backward flow field, respectively, using α as the interpolation factor. Using Equations (2.14a) and (2.14b) it becomes straight forward to determine the interpolated fragment $p_{t+\alpha}$ that corresponds to fragments p_t and p_{t+1}, respectively.

$$p_{t+\alpha} = p_t + \alpha V_t^f(p_t).xy \tag{2.14a}$$

$$p_{t+\alpha} = p_{t+1} + (1-\alpha)\, V_{t+1}^b(p_{t+1}).xy \tag{2.14b}$$

In image-based BidirTC we are interested in the inverse of the mapping defined by Equations (2.14a) and (2.14b), though, which means that for the B-frame fragment $p_{t+\alpha}$ we wish to find the corresponding fragments p_t and p_{t+1} in the past- and future I-frames, respectively, cf. Figure 2.38(a). Unfortunately, the inverse mapping can't be expressed in a closed equation – for example, to express p_t in terms of $p_{t+\alpha}$, we can subtract $\alpha V_t^f(p_t).xy$ from both sides of Equation (2.14a). After this transformation step we clearly see the circular dependency of p_t on itself, because in order to evaluate $\alpha V_t^f(p_t).xy$, we already need p_t. All hope is not lost, though, since with the help of the flow fields, intermediate B-frames can still be derived from adjacent I-frames by employing an iterative greedy image-space search. This search algorithm is performed once using the forward flow field according to Equation (2.15a), and once using the backward flow field according to Equation (2.15b)

$$p_{t,i} = p_{t+\alpha} - d_i^f \qquad \text{where} \quad d_i^f = \alpha V_t^f(p_{t,i-1}).xy \tag{2.15a}$$

$$p_{t+1,i} = p_{t+\alpha} - d_i^b \qquad \text{where} \quad d_i^b = (1-\alpha) V_{t+1}^b(p_{t+1,i-1}).xy \tag{2.15b}$$

For example, to find the matching fragment p_t from the past I-frame I_t for the B-frame fragment $p_{t+\alpha}$, we fetch the flow-field vector at the current fragment position $d_1^f = \alpha V_t^f(p_{t+\alpha})_{xy}$, and calculate a new fragment position $p_{t,1} = p_{t+\alpha} - d_1^f$. This concludes the first iteration of the search. Next we read the flow field again, but this time at the new position $p_{t,1}$, yielding $d_2^f = \alpha V_t^f(p_{t,1})_{xy}$. We now calculate yet another estimate position $p_{t,2} = p_{t+\alpha} - d_2^f$, and so on. After m steps the search is terminated. Yang et al. [65] point out that in practice we arrive at a reasonably exact result within a mere $m = 3$ iterations.

Once $p_{t,m}$ is located, we use the forward flow field to estimate the depth value $z^f = Z_t(p_{t,m}) + \alpha V_t^f(p_{t,m}).z$, where Z_t denotes the z buffer of frame t. We see that finding the

predicted depth is completely analogous to locating the corresponding 2D position in the error-estimation equation.

We perform the same search to find the corresponding fragment in the future I-frame I_{t+1}, only this time we use the backward flow field from the future frame V^b_{t+1}, and the interpolation factor $1-\alpha$. In both cases, the initial position for both the forward and backward iterative searches, respectively, is given by $p_{t,0} = p_{t+1,0} = p_{t+\alpha}$. Figure 2.38 illustrates three iterations of the search algorithm for the forward direction, i.e., when looking for the past I-frame fragment that gives rise to the current B-frame fragment when going forward in time (along the forward optical-flow field). Once we have found the two-dimensional point after m iterations, we determine the depth by Equation (2.16a) for the forward-search result, and Equation (2.16b) for the backward-search result, respectively, where Z_t is the clip-space depth in frame t.

$$z^f = Z_t(p_{t,m}) + \alpha V^f_t(p_{t,m}).z \tag{2.16a}$$

$$z^b = Z_{t+1}(p_{t+1,m}) + (1-\alpha) V^b_{t+1}(p_{t+1,m}).z \tag{2.16b}$$

Yang et al. also introduce a measure for estimating the screen-space error according to Equation (2.17a) and Equation (2.17b), where e^f is the screen-space error that the resulting point from the forward search exhibits, and e^b describes the screen-space error for the backward-search result. The error corresponds to the euclidean distance between the current fragment's position $p_{t+\alpha}$ and the reconstructed position, which, in the case of e^f, is $p_{t,m} + \alpha V^f_t(p_{t,m}).xy$.

$$e^f = \|(p_{t,m} + \alpha V^f_t(p_{t,m}).xy) - p_{t+\alpha}\| \tag{2.17a}$$

$$e^b = \|(p_{t+1,m} + (1-\alpha) V^b_{t+1}(p_{t+1,m}).xy) - p_{t+\alpha}\| \tag{2.17b}$$

Once we have a result from the iterative forward and backward searches, we can determine the interpolated color for each B-frame pixel. The first step is to test the two screen-space errors e^f and e^b against a user-defined threshold.

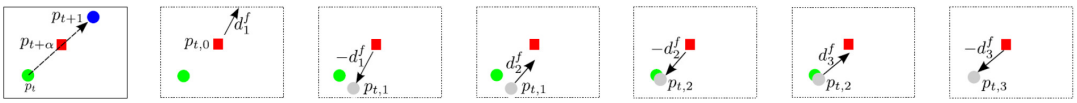
1. If both screen-space errors are smaller than this threshold and have similar depths, i.e., if the difference between the depth values satisfies $|z^f - z^b| < T_{\Delta z}$, we assume that the two positions refer to the same surface position. In this case, we could just blend the two fragment colors from the past and future I-frame, using α as the interpolation factor to infer the resulting B-frame pixel color as follows

   $$B_{t+\alpha}(p_{t+\alpha}).rgba = (1-\alpha) I_t(p_{t,m}).rgba + \alpha I_{t+1}(p_{t+1,m}).rgba \tag{2.18}$$

   However, Yang et al. note that this introduces unwanted blurring, since the two surface points $p_{t,m}$ and $p_{t+1,m}$ are hardly ever identical. It is preferable to chose the point with the smaller screen-space error, project that point into the temporally opposing I-frame, and then interpolate the colors of those two points. So, if $e^f < e^b$, we calculate the B-frame pixel colors according to Equation (2.19), and according to Equation (2.20) otherwise.

   $$B_{t+\alpha}(p_{t+\alpha}).rgba = (1-\alpha) I_t(p_{t,m}).rgba + \alpha I_{t+1}(p_{t+1,m} + V^f_t(p_{t,m}).xy).rgba \tag{2.19}$$

   $$B_{t+\alpha}(p_{t+\alpha}).rgba = (1-\alpha) I_t(p_{t+1,m} + V^b_{t+1}(p_{t+1,m}).xy).rgba + \alpha I_{t+1}(p_{t+1,m}).rgba \tag{2.20}$$



(a) Desired Point p_t (b) Retrieve d_1^f (c) 1^{st} iteration (d) Retrieve d_2^f (e) 2^{nd} iteration (f) Retrieve d_3^f (g) 3^{rd} iteration

Figure 2.38: The first three iterations of the greedy image-space search algorithm, which performs the reprojection into neighboring I-frames when no depth information is available in intermediate B-frames in image-based bidirectional temporal coherence. In the example we show the forward search, i.e., the search for the past I-frame fragment which gives rise to the B-frame fragment when stepping along the forward flow field, when the flow-field vector is scaled by α. Images courtesy of [65].

Of course, if the point that is projected into the opposing I-frame is occluded in that I-frame, we use the color from the I-frame pixel with the smaller screen-space error only. As noted before, with the help of the optical flow fields, the projection of a point from one I-frame into the opposing I-frame can be performed exactly.

2. If both points satisfy the desired screen-space-error threshold, but have different depths, we keep only the closer of the two points and discard the other one. The point that we keep is still projected into the opposing I-frame, so that we then proceed just as in step 1.

3. In the unlikely event that both positions exhibit screen-space errors that exceed the tolerated threshold, the procedure described in step 1 is applied to both points.

2.4.1.1 Improving B-frame Interpolation: Dual Initialization

Unfortunately, at the silhouette of objects that are both rotating and translating, the just presented basic greedy search algorithm may cause the current iteration's search position to fall off the object, so that the flow-field vectors for the next search steps are no longer related to the current object. Due to the accumulative nature of the B-frame interpolation, as soon as one flow vector that does not belong to the currently reconstructed object is introduced into the calculation, the final result will be useless. If the opposite search also falls off the object, or the current pixel position is not visible at all in the other I-frame, the quality of the reconstruction suffers badly. Yang et al. [65] propose two improvements to their basic search algorithm that help hide the otherwise visible artifacts. Both search refinements are based on the idea that their greedy image-based search algorithm performs better if it is supplied with good start positions. We will look at their first improvement, namely dual initialization in this subsection, and their second improvement, namely latest-frame initialization, in the next subsection.

As just mentioned, dual initialization alters the starting positions for the forward- and backward greedy searches when reconstructing B-frames. This strategy prevents the search from missing the object in the first iteration, and therefore leads to B-frames being inferred from their two adjacent I-frames more accurately. Other than the altered starting positions, dual initialization uses the same greedy search algorithm that we just described. The start position $p'_{t,0}$ for

the temporal forward search, and $p'_{t+1,0}$ for the backward search, are given by Equation (2.21), where α is the interpolation factor between the two I-frames, t and $t+1$ denote the frame number of the past- and future I-frame, and V^b_{t+1} and V^f_t are the backward- and forward flow fields, respectively.

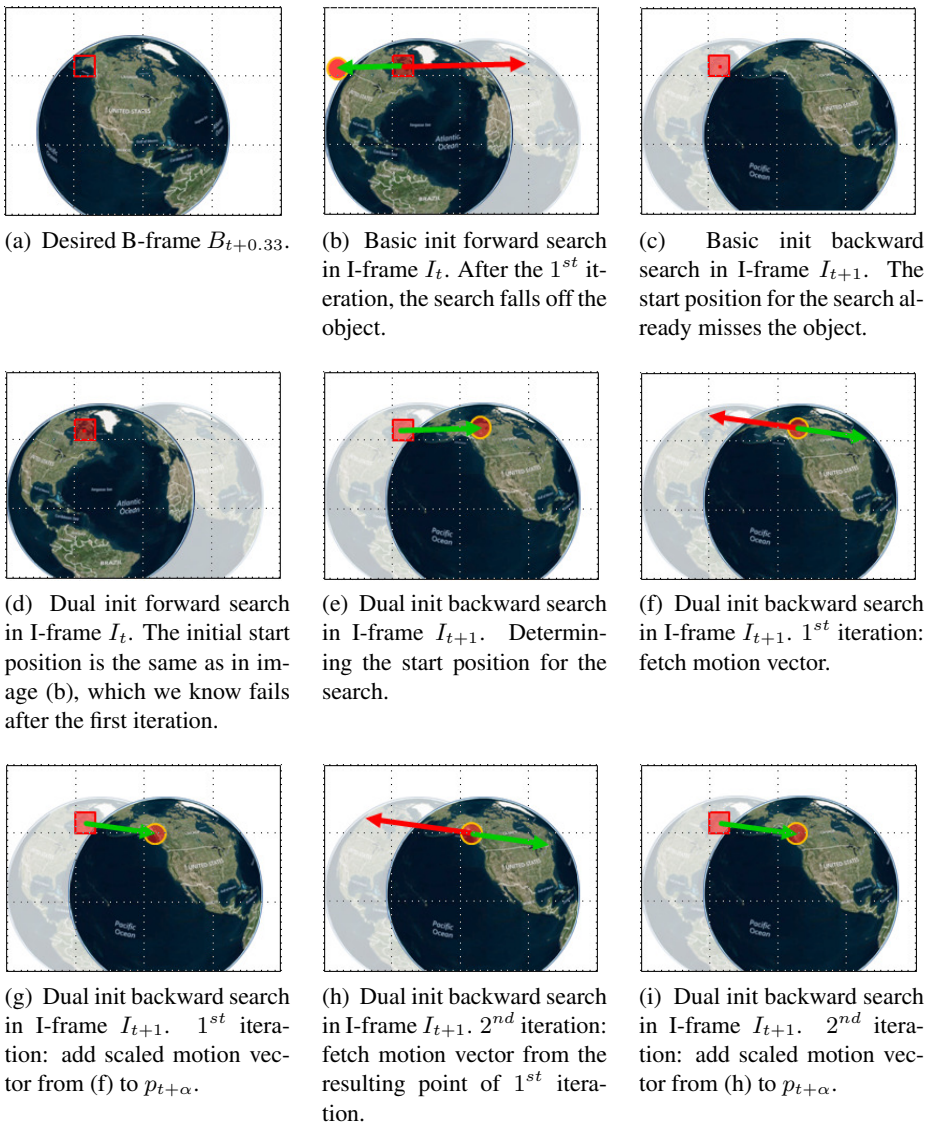
$$p'_{t,0} = p_{t+\alpha} + e^b_0 \qquad \text{where} \quad e^b_0 = \alpha V^b_{t+1}(p_{t+\alpha}).xy \tag{2.21a}$$

$$p'_{t+1,0} = p_{t+\alpha} + e^f_0 \qquad \text{where} \quad e^f_0 = (1-\alpha) V^f_t(p_{t+\alpha}).xy \tag{2.21b}$$

Figure 2.39 compares the two strategies for initializing the start positions for the iterative image-space search algorithm for B-frame reconstruction, where an earth model rotates roughly around its north-south axis and additionally travels from the left to the right screen border. Note that in this example the flow fields contain zero vectors for background pixels. Image (a) shows B-frame $B_{t+\alpha}$ (with $\alpha = 0.33$) that we want to reconstruct. The currently observed pixel is marked with a red quad. The transparent parts in images (b) to (i) in each frame show the content of the temporally neighboring I-frame so that we can more explicitly see how the optical flow vectors are formed.

Images (b) and (c) in Figure 2.39 show how the basic initialization $p_{t,0} = p_{t+1,0} = p_{t+0.33}$ falls off the surface of a moving object quickly. In the backward search pictured in Figure 2.39(c) this happens immediately after the initialization of $p_{t+1,0}$. In the temporal forward search (Figure 2.39(b)) we are lucky, since the initial search position $p_{t,0}$ is still on the object surface. The corresponding forward flow vector $V^f_t(p_{t+0.33})$ is depicted by a red arrow, the green arrow is the scaled and inverted forward-flow vector $-d^f_1 = -0.33 V^f_t(p_{t+0.33})$. The green arrow points to the position at which to fetch the next optical flow vector. This next fetch position is depicted by the red circle, which is no longer on the object surface. To summarize, images (b) and (c) in Figure 2.39 show that both the forward and the backward search falls off the object surface when we start both searches from $p_{t+0.33}$, i.e., when we use basic initialization.

Images (d) to (i) in Figure 2.39 show the greedy image-space search algorithm when the start positions are set according to the dual-initialization improvement. With this initialization strategy, the first velocity-vector fetch position is governed by the velocity vector from the temporally opposing I-frame, i.e., $p'_{t,0} = p_{t+\alpha} + \alpha V^b_{t+1}(p_{t+\alpha}).xy$ for the forward search, and $p'_{t+1,0} = p_{t+\alpha} + (1-\alpha) V^f_t(p_{t+\alpha}).xy$ for the backward search. For the forward search with dual initialization this means that we add the flow vector from the future I-frame (the position from which to fetch the vector can also be seen in image (c)), which is the zero vector in this case, as illustrated in (d). Since dual initialization only changes the start position, we actually perform the same forward search as with basic initialization, so that we do not gain anything from dual initialization. For the backward search, however, the first position for the optical-flow vector fetch does lie on the object's surface, as shown in image (e). Once again, the red quad in image (e) depicts $p_{t+\alpha}$, the green arrow points to the next position from which to read the flow-vector, and this position is marked with a red circle. In image (f) the red arrow represents the backward flow vector, while the green arrow is the scaled and inverted vector that takes part in determining the next flow-vector fetch position, as can be seen in image (g). Images (h) and (i) show yet another iteration of the search algorithm, and we see that finally the interpolated position from which to read the color from for $p_{t+\alpha}$ is reasonably close to the true position.



(a) Desired B-frame $B_{t+0.33}$.

(b) Basic init forward search in I-frame I_t. After the 1^{st} iteration, the search falls off the object.

(c) Basic init backward search in I-frame I_{t+1}. The start position for the search already misses the object.

(d) Dual init forward search in I-frame I_t. The initial start position is the same as in image (b), which we know fails after the first iteration.

(e) Dual init backward search in I-frame I_{t+1}. Determining the start position for the search.

(f) Dual init backward search in I-frame I_{t+1}. 1^{st} iteration: fetch motion vector.

(g) Dual init backward search in I-frame I_{t+1}. 1^{st} iteration: add scaled motion vector from (f) to $p_{t+\alpha}$.

(h) Dual init backward search in I-frame I_{t+1}. 2^{nd} iteration: fetch motion vector from the resulting point of 1^{st} iteration.

(i) Dual init backward search in I-frame I_{t+1}. 2^{nd} iteration: add scaled motion vector from (h) to $p_{t+\alpha}$.

Figure 2.39: Iterative search with basic- and dual initialization start positions. Transparent pixels indicate the content of the opposing I-frame to more directly see how the motion vectors are calculated. The motion vectors V_t^f and V_{t+1}^b are shown as red arrows. We use green arrows to depict the scaled motion vectors $-d_i^f$ and $-d_i^b$, where the subscript i indicates the current greedy-search iteration. The red square shows the pixel $p_{t+\alpha}$ that is currently reconstructed. Note that once we fall off the to-be-reconstructed object as in images (b) and (c), we read unrelated optical-flow-field vectors, thus rendering the search result useless. Image (d) shows a case in which dual initialization degenerates to basic initialization, and hence doesn't improve the search result. Images (e) to (i) illustrate how dual initialization finds a position that is reasonably close to pixel $p_{t+\alpha}$ in I-frame I_{t+1} after only 2 iterations. Earth-model screenshots courtesy of `https://www.webglearth.com`.

2.4.1.2 Improving B-frame Interpolation: Latest-frame Initialization

While dual initialization performs significantly better than the basic initialization, it is nevertheless not perfect. For this reason, Yang et al. [65] present a third search-initialization strategy, namely latest-frame initialization. Latest-frame initialization bases the initial search position for the reconstruction of the current B-frame on the position that was found by the image-based reconstruction algorithm in the previous B-frame. Thus, latest-frame initialization is yet another way to exploit the high degree of coherence between temporally adjacent B-frames. This effectively causes the number of iterations allotted to the B-frame reconstruction to gradually increase as the current B-frame approaches its future I-frame, without actually performing more iterations per frame. Hence, for a given pair of past- and future I-frames, all but the first B-frame benefit from this improvement.

Let $d_i'^f$ and $d_i'^b$ be the offsets for the forward and backward search, respectively, computed in the previous B-frame at time $t + \alpha'$, then the initialization for the forward and backward search for the next B-frame at time $t + \alpha$ becomes

$$p'_{t,0} = p_{t+\alpha} - d_0^f \qquad \text{where} \quad d_0^f = \frac{\alpha}{\alpha'} d_i'^f \tag{2.22a}$$

$$p'_{t+1,0} = p_{t+\alpha} - d_0^b \qquad \text{where} \quad d_0^b = \frac{1-\alpha}{1-\alpha'} d_i'^b \tag{2.22b}$$

Figure 2.40 once again compares the basic search to the dual-, and latest-frame initialization techniques, and gives clear evidence that the reconstructed B-frames benefit from these modifications. What immediately stands out in Figure 2.40(d) is that the earth model appears squashed when employing only the basic reconstruction algorithm. This happens because the iterative search positions for pixels along the border of moving and rotating objects tend to fall off the object, so that instead of sampling the object surface, we end up sampling the background, thereby incorrectly coloring parts of the object in question with the background color, making the object appear smaller than it really is, cf. Figure 2.39. Still, even the basic method outperforms the naive linear blending of the past- and future I-frame, which is plagued by ghosting artifacts as shown in Figure 2.40(c).

Figures 2.40(e) and (f) show that the dual- and latest-frame initialization modifications yield acceptable reconstructions. Especially the results achieved with latest-frame initialization are almost indistinguishable from the reference solution. The catch with latest-frame initialization is that we need to keep yet another screen-sized buffer in which we store the offset vectors $d_i'^f$ and $d_i'^b$ from the previous B-frame. This means that the algorithm demands more video memory and has still higher bandwidth requirements. Note that in any case, even with dual- and latest-frame initialization, Yang et al. [65] always perform the basic search alongside the improved elaborate ones, and then use the result with the lowest reprojection error.

2.4.1.3 An Alternative B-frame Interpolation Technique

We explore an alternative B-frame reconstruction improvement that is well suited for our specific use case in which we deal with only one static terrain model, i.e., with a mesh that never changes

(a) Overview

(b) Reference (c) Linear Blending (d) Basic Init (e) Dual Init (f) Dual Init + Latest frame init

Figure 2.40: Images courtesy of Yang et al. [65].

shape and neither moves nor rotates or scales. Before we describe our method in detail in Section 3.6.4, we briefly outline the basic idea behind it. Our B-frame reconstruction algorithm first combines the result from the image-space search with basic initialization and dual initialization, and then augments the result with splatted points that we scatter into the B-frame from the neighboring I-frames. Scattering is only performed for pixels at depth discontinuities in the two temporally adjacent I-frames, since we know that B-frame reconstruction errors happen mostly in these areas. We will present an efficient GPU-based implementation for scattering only this specific subset of I-frame samples, that employs the histogram-pyramid data structure [23]. We also give a short overview of the histogram pyramid in Section 2.5.

2.4.1.4 Rendering Future I-frames

As promised at the beginning of this section, now that we have an idea of how *bidirectional temporal coherence* (BidirTC) works, we revisit the seemingly impossible task of rendering a frame that lies in the future in interactive applications. Since we can't predict user input or other events that aren't under the control of our application, rendering the future I-frame can only be achieved by modifying the notion of what "future" means. To that end we simply move the present tense slightly into the past. We can then interpret the most recently rendered I-frame as the future I-frame, the second to most recent I-frame as the past I-frame, and the currently reconstructed B-frame as the current frame (if the interpolation factor α is 0, the currently rendered B-frame is actually an I-frame). While we present the reconstructed B-frames on the screen, we already prepare the next "future" I-frame. Of course this interpretation of time introduces an additional delay when reacting to user input, since with this scheme the frame that actually consumes the input lags behind by at least the amount of time that lies between two I-frames. To estimate the negative effect that this delay has, Yang et al. [65] conducted a user study in which I-frames are calculated at a rate of 15Hz (this means that user input is also processed at 15Hz), while B-frames are displayed at a rate of 60Hz. Their study showed that relative to "standard" 15Hz rendering, the users' objective performance impression did not suffer when compared to

BidirTC – in fact, the participants preferred the "temporally upsampled" BidirTC rendering over plain 15Hz rendering. BidirTC could however not keep up with the experience of "true" 60Hz rendering and input processing.

BidirTC uses three history buffers. Two of the three buffers are assigned the role of the past- and future I-frame, while the upcoming future I-frame is rendered to the third buffer. A big advantage of this setup is that rendering to the third buffer can be spread across several frames. For example, when the B-frame interpolation factor α is incremented by 0.25 per frame, rendering of the next future I-frame can be spread across four frames, since for those four frames the third buffer does not participate in the B-frame reconstruction process. Once the interpolation factor α reaches the value 1, rendering to this third buffer must be completed and the buffer eventually becomes the next future I-frame, while the current future I-frame becomes the new past I-frame, and the current past I-frame becomes the next temporary offscreen buffer. After this cyclic I-frame-buffer swapping is done, α is reset to 0 and the whole process starts all over.

2.4.1.5 A Short Comparison of Unidirectional- and Bidirectional Temporal Coherence

To summarize, one major advantage of bidirectional temporal coherence over "unidirectional" temporal coherence is that occlusions that are present in the past I-frame are often no longer present in the future I-frame and vice versa, thereby improving the reprojection-cache hit ratio. This is true for both scene-assisted and image-based BidirTC.

Furthermore, image-based BidirTC even enables the distribution of I-frame rendering over several frames, so that either performance can be increased significantly without degrading image quality, or image quality can be improved without degrading performance. Regarding image quality, we must add that image-based BidirTC sometimes fails to reconstruct B-frames accurately. The resulting subtle artifacts are mostly restricted to image regions that contain depth discontinuities, like for example at object borders – even more so, if objects are rotating and moving quickly relative to the camera. The inventors of BidirTC presented several strategies to cope with these kinds of artifacts, namely dual initialization and latest-frame initialization. Certain shapes just can't be reconstructed correctly – those can be skipped when reconstructing the scene with BidirTC, and later be rendered "traditionally" onto the reconstructed B-frame.

Another drawback of image-based BidirTC are the increased memory and bandwidth requirements for the additional I-frame buffers. When I-frame rendering is distributed over several frames in image-based BidirTC, user input is delayed as well. A user study conducted by Yang et al. [65] indicates that the perceived delay is not disruptive though. Also note that even though I-frame rendering itself effectively happens at a lower frame rate, this remains mostly unnoticed, since the image-based BidirTC method displays interpolated intermediate B-frames at a higher frame rate.

2.5 Histogram Pyramids

The histogram pyramid is a hierarchical data structure that is intended for stream compaction and stream expansion [23, 67]. The data structure is well suited for algorithms that run on the GPU, since historgram pyramids can be stored directly as two-dimensional texture images, where each

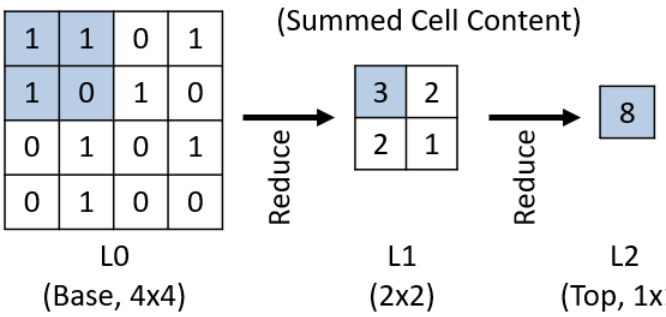


Figure 2.41: Histogram-pyramid creation. The input image encodes where to generate an element, and how many elements to generate. The "reduce" operation is then iteratively applied to 2×2 texel blocks at each successive pyramid level, starting at the pyramid's base. In this example, the top-left block is marked in blue so that we can see how it contributes to the final value across all histogram-pyramid levels.

level of the histogram pyramid is represented by a mipmap level of the underlying texture. Just as with "normal" mipmapped textures, when talking about any two adjacent pyramid levels, we say that the one which is closer to the pyramid's base is the high detail level (since it represents the stored data at a higher resolution), while the level closer to the histogram pyramid's apex is the coarse detail level (since it represents the same data with less texels). Histogram pyramids have been shown to perform well when applied to feature detection in videos, marching cube iso-surface extraction [23], and point-list generation on the GPU [67].

Figure 2.41 illustrates the process of building a histogram pyramid. The input image, which corresponds to the histogram pyramid's base level, encodes how many output elements to generate at each position, and hence can be interpreted as the stream that is to be processed. Starting at the base level, we work our way to the pyramid's apex one level a time. Each pyramid level is processed in blocks of 2×2 texels. To generate an entry in the next-coarser pyramid level, we calculate the sum of the entries in the current level's corresponding block, so that the single entry at the pyramid's apex stores the sum of all entries from the pyramid's base. Hence, the pyramid's apex tells us how many elements will be in the output stream for a specific input stream. A notable implementation detail at this point is, that the conceptual blocks of two by two entries each, can be stored in a single four-channel RGBA texel instead of four separate single-channel texels, which not only minimizes the amount of texture fetches when generating and later accessing the histogram pyramid, respectively, but also increases the number of elements that can be processed by a factor of four, since current GPUs impose a restriction on the maximum allowed texture dimension, no matter how many channels are stored per texel.

Now that we know how to create the histogram-pyramid data structure, let's turn our attention to using it. We have just seen that the single texel in the coarsest mipmap level of the histogram-pyramid texture reveals the number of elements in the output stream. Next we need to associate each element in the output stream to the element in the input stream that actually gave rise to that output element. This mapping from output elements to input elements can be efficiently deduced from the histogram pyramid itself. Since we know how many output elements

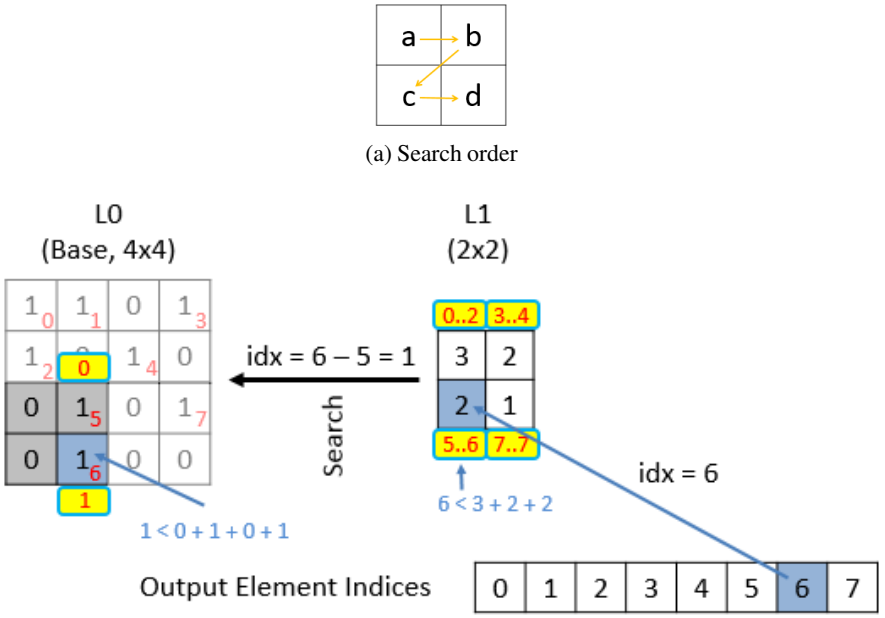
to generate, we can assign an index to each output element. This output element index acts as a search index to find the associated input element.

The search starts at the first level below the histogram pyramid's apex and proceeds toward the pyramid's base level. Note that we can always safely skip the pyramid's topmost level in our search, since any output element created from the underlying input stream must lie below the single entry at the histogram pyramid's apex. At each histogram-pyramid level we access a single block of 2×2 entries. By construction, each entry in this block represents the sum of all elements below that quadrant. We make use of this property to properly navigate through all pyramid levels according to the current search index. At each level we compare the search index to the four entries in a block, starting with the upper-left texel. If the search index is smaller than the value stored there, the current output element was created by one of the elements in the upper-left quadrant, so that the search continues there at the next-lower level. If the search index is bigger, though, it is next compared to the sum of the upper-left and upper-right entries. If the search index is smaller than this sum, the search proceeds in the upper-right quadrant at the next-lower level. If the search index is still bigger, we calculate the sum of the upper-left, upper-right and lower-left block entries, and compare this sum to the current element's index. If the current output element's index is smaller than the sum, we branch to the lower-left quadrant at the next-finer pyramid level, otherwise we proceed with the lower-right block. Whenever stepping down to the next pyramid level, we adjust the search index by subtracting the sum of all the previously tested entries, i.e., those entries for which the comparison failed, from the current search index. If the first block entry is already smaller than the search index, there is no need to adjust the search index. The last entry at the pyramid's base level that passes the "index test" is the searched-for input element that corresponds to the output element (and thus the search index). Figure 2.42 shows an example traversal, which should shed some light on how the just described search process works.

Note that at most four elements are accessed per histogram-pyramid level in the presented search algorithm. Hence, the search runs in $O(logN)$, where N is given by Equation (2.23).

$$N = max(PyramidBaseLevelWidth, PyramidBaseLevelHeight) \tag{2.23}$$

The search can be carried out in parallel for all output indices. This property is very important for an efficient implementation on the GPU. Moreover, in Figure 2.42 we see that the search algorithm does not fetch data from random positions, but rather follows a predictable pattern, always accessing neighboring entries, not only within a pyramid level, but also between any two neighboring pyramid levels (the reads are "vertically" aligned across all pyramid levels). Since we store the whole histogram pyramid as a texture, the pyramid levels are actually the mipmap levels of the texture, so that we can hope for close-to-optimal usage of the underlying hardware, as the access patterns performed by the search algorithm correspond to those performed by trilinear filtering – an operation that GPUs are optimized for.



(a) Search order

(b) Traversal

Figure 2.42: Histogram pyramid traversal for the pyramid from Figure 2.41. Image (a) shows the search order $a \rightarrow b \rightarrow c \rightarrow d$ in each block of 2×2 entries. The search order is the same at all histogram-pyramid levels. Image (b) illustrates the search for the input element that gave rise to the seventh output element, which has index 6. Each non-zero input entry at the base level is annotated with its corresponding index in the output stream. Additionally, the indices that each quadrant covers are written above/below each entry in red on yellow background (a range of indices is written as "start index .. end index"). For each block we compare the search index to a, $a+b$, $a+b+c$, and $a+b+c+d$ (following the search order from image (a)), until the search index is smaller than the respective sum for the first time. The element for which this condition is satisfied is the searched-for element in that block and determines the quadrant into which to step down on the next pyramid level. The block search is repeatedly applied on each pyramid level until the base level is reached, in which case the currently found element represents the final search result. For example, looking at the decisions at level L1, we first compare the search index to entry a, i.e., $6 < 3$, which is false. Hence, we investigate the second of the four entries in the current block, which means that we evaluate $6 < 3 + 2$, which is still false, so we perform the next comparison, which now considers the first three block entries, i.e., $6 < 3 + 2 + 2$. Since this comparison is true, the block entry c determines which quadrant to visit at the next pyramid level. Whenever we transition to the next histogram-pyramid level, we subtract the sum of the previously tested entries from the current search index and use the resulting number as the search index at that level. Hence, at level L1, we subtract 5 from the current search index. This means that the search index at level L0 is 1. Performing the same tests as before, we find that this time block entry d fulfills our check, and since L0 is the base level, entry d already identifies the input element which gave rise to the output element with index 6, at which point the search finishes successfully.

CHAPTER 3

Improved Persistent Grid Mapping

In this chapter we review *Persistent Grid Mapping* (PGM) [32] and discuss its properties and weaknesses. Speaking of the algorithm's deficiencies, PGM is particularly prone to aliasing artifacts, which stem from undersampling of the heightfield. Further, when the camera moves, so-called *(vertex-) swimming artifacts* occur, because the projected grid-vertex positions do not snap to heightmap texels, but rather depend solely on the current camera parameters. Due to this, sampling positions slide freely over the ground plane as the camera moves, and terrain reconstructions of identical heightmap areas tend to vary slightly between successive frames, giving the impression of a flabby terrain surface. While the most obvious way to battle the heightmap undersampling is to increase the persistent grid's resolution, this brute-force approach is impractical. Instead, we propose four improvements to the basic PGM algorithm, which aim at achieving more faithful heightmap sampling without increasing the vertex count of the persistent grid. These improvements are mutually orthogonal, i.e., they all improve the basic PGM algorithm in their own right, independently of each other. Before explaining each improvement in detail later in this chapter, we give a concise overview right away.

1. *Tailor the projected grid* to the camera's view frustum (see Section 3.3). Tailoring shrinks the ground-plane area that the persistent grid needs to cover. The reduced sampling intervals imply more faithful heightmap sampling.

2. Perform *importance-driven warping of the persistent grid on the GPU* (see Section 3.4). Without the warping step, projected grid cells (see Figure 3.1 for the definition of a grid cell) tend to be stretched non-uniformly on the ground plane, especially as the projected cell's distance to the camera increases. This deteriorates terrain-surface reconstruction fidelity in the distance. At the same time the grid projection usually produces an overly dense tessellation up-close. By redistributing grid-vertices we make better use of the available resources.

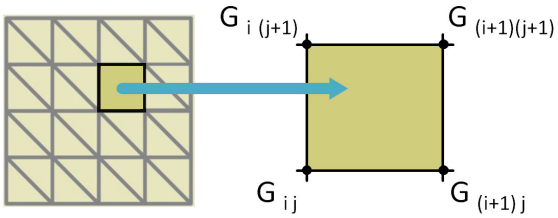


Figure 3.1: A mesh cell inside the persistent grid is the rectangular area spanned by vertices G_{ij}, $G_{(i+1)j}$, $G_{(i+1)(j+1)}$ and $G_{i(j+1)}$.

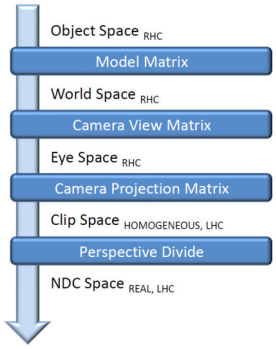


Figure 3.2: The coordinate spaces used throughout the text. *RHC* stands for right-handed coordinate frame, and *LHC* for left-handed coordinate frame, respectively.

3. *Search locally for terrain edges* between two sampling positions (see Section 3.5). Moving the farther sampling position toward the camera onto the missed terrain edge helps improve image stability.

4. Exploit *temporal coherence* [46,47,65] between consecutive frames (see Section 3.6). By making use of the already available data from previous frames we can lessen the frame-to-frame terrain-reconstruction differences, which manifest themselves in the form of swimming artifacts.

In the following, we first introduce a common vocabulary, then we describe the basic persistent-grid-mapping algorithm, and after that we shortly discuss several ways to render the persistent grid, and for that matter any regular grid, efficiently. Finally we explain our four contributions, which together improve the basic persistent-grid-mapping algorithm.

3.1 Conventions

We use right-handed camera coordinate frames throughout the text, which means that the camera's view direction is along the negative z-axis of the camera's coordinate system. Figure 3.2 shows the coordinate spaces which we will repeatedly refer to in this section. We write v^T to denote vector v's corresponding transposed vector. The coordinates of a four-dimensional vector

v are denoted by $v.x$, $v.y$, $v.z$, and $v.w$. Similar to common high-level shading languages such as GLSL, we also use swizzling. For example, the expression $v.xz$ represents a two-dimensional vector which has $v.x$ as its first coordinate, and $v.z$ as its second coordinate. When referring to vertices in a rectangular grid, we use subscripts to denote their "coordinate" within the mesh, i.e., v_{ij} is the vertex at the coordinate (i, j).

The ground plane is given by the equation $y = 0$. Displacing a vertex that lies on the ground plane at position $(x, 0, z)^T$ means adjusting its y-coordinate according to the respective heightmap value $h = heightmap(x, z)$, so that the displaced vertex has coordinates $(x, h, z)^T$. The terrain volume is the space between the minimum terrain height $minH = 0$, and the maximum terrain height $maxH$. Hence, the terrain volume's equation is given by $0 <= y <= maxH$.

3.2 Review of Persistent Grid Mapping

Persistent Grid Mapping is a terrain-rendering algorithm for heightmap-based landscapes [32, 33], which builds upon the same concept as Claes Johanson's algorithm for real-time water rendering [30]. The main idea of PGM is to sample the terrain in a way that corresponds to the detail required by the current screen resolution. Persistent Grid Mapping can be outlined as follows:

1. Define a regular planar mesh in screen space, the so-called *persistent grid*, cf. Figure 3.3(a).

2. Construct a second, auxiliary camera based on the "main" camera each frame. This camera allows handling some artifacts (see Section 3.2.2). In simple cases it is identical to the main camera.

3. Using the auxiliary camera's inverse transformation pipeline, re-project each clip-space grid vertex G_{ij}, onto the ground plane, yielding G_{ij}'s corresponding world-space position P_{ij}, cf. Figure 3.3(b).

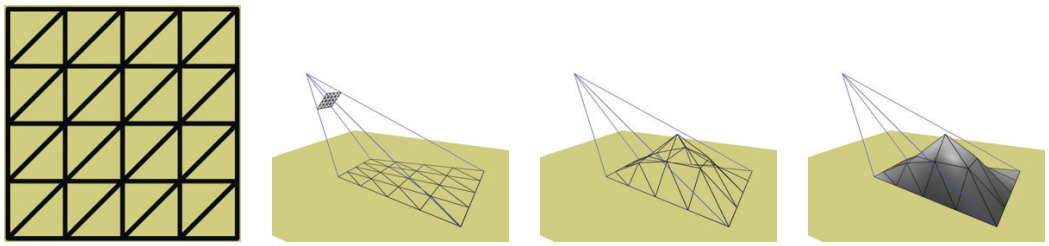
4. Evaluate the displacement function at position $P_{ij}.xz$. The displacement function is defined by a digital elevation model, typically stored in a heightmap. The sampled value is given by $h = heightmap(P_{ij}.xz)$.

5. Displace P_{ij} by h, yielding the final world-space position $P'_{ij} = (P_{ij}.x, h, P_{ij}.z, 1)^T$, cf. Figure 3.3(c).

6. Using the main camera's transformation matrices, render the displaced grid vertex P'_{ij}, cf. Figure 3.3(d).

3.2.1 Mesh Representations for the Persistent Grid

An important implementation detail of the PGM method is that the persistent grid is stored in video memory. The vertices that make up the grid are uploaded to the GPU only once at program



(a) Persistent grid before projection onto the ground plane, still on the projector's viewplane.

(b) Persistent grid after projection onto the ground plane.

(c) Displaced grid.

(d) Rendered grid.

Figure 3.3: Basics steps of the projected grid method. Images courtesy of [34].

startup. In combination with accessing the heightmap texture directly in the vertex shader, by making use of the vertex-texture-fetch feature of GPUs, no further terrain-related mesh data needs to be transmitted over the system bus at runtime.

The regular-grid mesh is a key terrain-surface building block in terrain-rendering algorithms. Thus it comes as no surprise that rendering this class of meshes as quickly as possible is of particular interest. In the following we take a look at two different internal vertex-data layouts for regular meshes that are tailored for efficient rendering on GPUs. Besides the performance aspect, we also investigate alternative persistent-grid tessellations that lead to increased shading quality and smaller surface-approximation errors.

For example, Schneider et al. [50] use triangle fans as the basic building block for terrain tiles. All the triangle fans in a terrain tile can be rendered with a single draw call by adding restart indices to the index buffer whenever transitioning from one triangle-fan block to the next. To improve cache efficiency, the triangle-fan blocks are not rendered in scanline order, but rather according to a space-filling curve (a so-called π-order traversal). Figure 3.4, shows the sequence in which the triangle-fan blocks within each terrain tile are drawn. The π-order traversal ensures that adjacent blocks are rendered one after another whenever possible, thereby favoring spatial coherence, cf. Figure 3.5. This increases the probability that vertices that are shared between triangle-fan blocks are still in the GPU's post-transform cache [8], and therefore need not be transformed again. In contrast, a scanline-oriented traversal tends to introduce cache misses at the beginning of each new row of triangles, and even within a scanline, post-transform cache hits are mostly restricted to vertices on the left- and right borders of each triangle-fan block.

Another approach is to render regular grids with short triangle strips, as described by Hugues Hoppe in his work on transparently maximizing the efficiency of vertex caching [29]. The technique once again aims at increasing the GPU's vertex-cache hit rate, which is achieved by limiting the triangle-strip length in each grid row. Due to these relatively short runs of vertices within each row, the method can keep using scanline order, since by the time the triangles in the next grid row are processed, the vertices from the previous row are still available in the post-transform cache. The number of triangles to render per row needs to be balanced, since rendering too few triangles doesn't fully utilize available slots in the GPU vertex cache, while

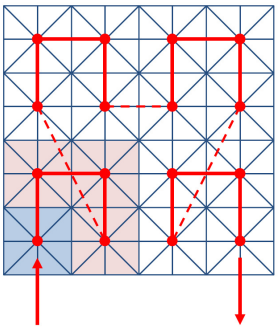


Figure 3.4: π-order traversal to render individual triangle fans in a cache friendly way. In this example the grid instance is made up of 4 triangle fans along each dimension. A single triangle fan is highlighted in blue in the lower left corner. A sub-block is shown in red.

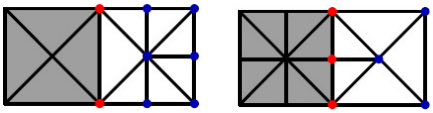


Figure 3.5: Worst-case and best-case vertex-cache reuse ratios among neighboring triangle fans. The left image shows the worst-case cache reuse – of the 8 vertices in the white fan, 2 vertices from the previously rendered gray fan can be reused, since they are most likely still in the GPU's post-transform cache. The right image shows the best-case cache reuse – of the 6 vertices in the white fan, 3 vertices from the previously rendered gray fan can be reused. The red dots show the vertices that create vertex-cache hits, the blue dots represent vertex-cache misses. Images courtesy of [50].

rendering too many triangles can lead to vertices being evicted from the cache by the time they are accessed again when processing the triangles in the next row, so that the vertex-cache miss rate starts increasing once a certain triangle-count threshold is exceeded. Hugues Hoppe suggests rendering around 20 triangles per grid row. Even though the mesh is rendered as a sequence of relatively short triangle strips, the entire grid can still be rendered with a single draw call by inserting restart indices between any two neighboring triangle-strip columns, cf. Figure 3.6. In our proof-of-concept implementation, we chose to follow Hoppe's approach, and render the persistent grid with short vertex-cache-friendly triangle strips.

As we have seen in Section 2.3, most landscape-visualization algorithms use several, relatively small, tiles to render a much larger terrain surface. Due to the constricted size of the tile meshes, 16-bit index buffers are more than enough for rendering them, which helps reduce the bandwidth requirements at runtime. Due to the regular pattern of vertices within the meshes, we can even deduce the vertex position from the index value. For example, OpenGL makes the vertex index available through the built-in vertex-shader input variable *gl_VertexID* [11] , so that we can write

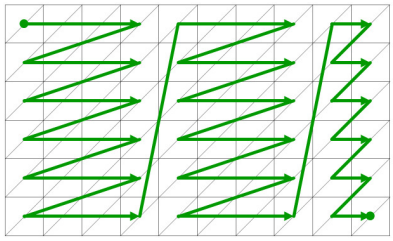
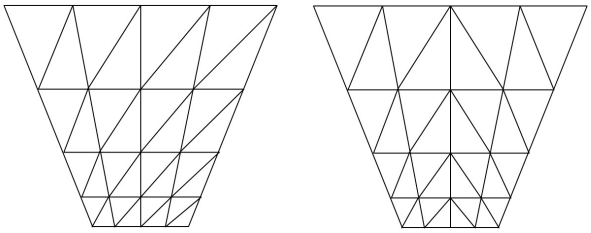


Figure 3.6: Short triangle strips help maximize the efficiency of vertex caching. The whole grid can still be rendered with a single draw call by inserting restart indices between any two triangle-strip columns. The green arrows illustrate the order in which the triangles in the mesh are rendered.



(a) (b)

Figure 3.7: The left image shows the standard regular grid after projection onto the ground plane. Since the grid has a trapezoidal shape, the triangles in the left and right half have different shapes. The second image shows that switching the grid-cell diagonals in the right half of the grid creates an evenly tessellated persistent-grid mesh.

```
ivec2 vertexPos2D = ivec2(mod(gl_VertexID, meshDim.x), floor(gl_VertexID/meshDim.x))
```

in GLSL, where *meshDim.x* stores the number of vertices in each row, and *vertexPos2D* is the two-dimensional coordinate of the vertex that corresponds to index *gl_VertexID*. This implies that theoretically, we don't even need to explicitly store vertex positions. Unfortunately, we can't render empty vertex arrays in OpenGL, but we can at least limit the vertex buffer to store 8 bit dummy values only, cutting down further on memory requirements, and, more importantly, bandwidth requirements.

Following the performance-oriented discussion, we now shift our attention toward minimizing surface-approximation errors by using different grid tessellations. For instance, when looking at the trapezoidal shape of the persistent grid after projection onto the ground plane, we notice that the triangles in the left half of the grid have a different shape than the triangles in the right half. Since in an orthogonal grid each cell consists of two right triangles, we can simply switch the diagonal of all grid cells in the right half of the grid to create a symmetric tessellation for the whole grid, cf. Figure 3.7. In our prototype we render a symmetric grid just as the one illustrated in Figure 3.7(b).

While Livny et al. [32] illustrate their PGM method with an orthogonal grid, they note

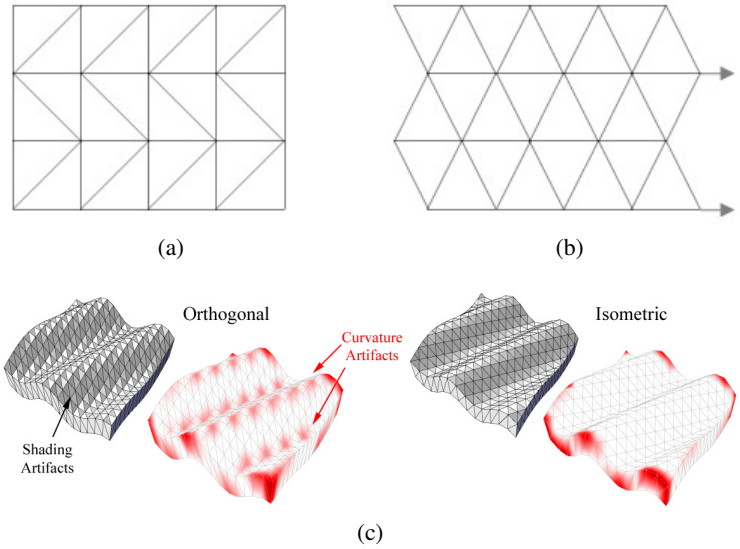


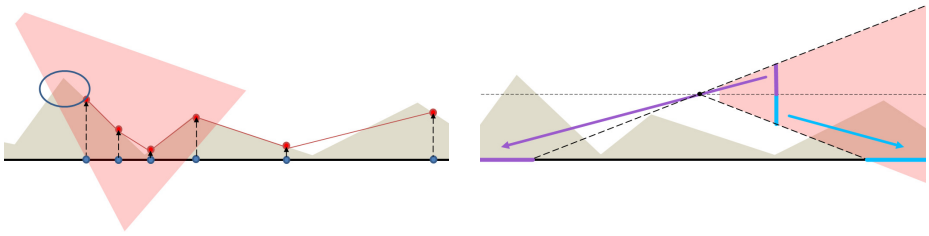
Figure 3.8: Image (a) depicts an orthogonal grid. Image (b) shows how shifting all vertices in odd rows horizontally by half the grid-cell size results in a diamond tessellation. In image (c) we see an orthogonal grid on the left and a grid with a diamond tessellation built from equilateral triangles on the right. The special isometric tessellation on the right reduces shading artifacts of the reconstructed heightfield patch. Image (c) courtesy of McGuire and Sibley [40].

that PGM works just as well with any other tessellation for the persistent grid. The authors of the chapter "Procedural Ocean Effects" in ShaderX^6 [57] pick up this remark and use a diamond tessellation for the persistent grid, which they claim lessens sampling errors compared to a rectangular tessellation. They create the diamond tessellation from an orthogonal grid (see Figure 3.8(a)) by shifting all vertices in odd rows by half a cell to the right (see Figure 3.8(b)). A possible justification for their claim that the diamond tessellation gives better surface-reconstruction results is that the adjusted grid is essentially the same as the one used for terrain patches in "A Heightfield on an Isometric Grid" [40] by Morgan McGuire and Peter Sibley. They show that this equilateral-triangle grid minimizes the shading artifacts that are brought forth when using ordinary orthogonal grids, i.e., grids made up of right triangles.

3.2.2 The Auxiliary Camera

In the overview of Persistent Grid Mapping, we have seen that the algorithm not only uses a "normal" camera through which we view the scene, but also a second, auxiliary, camera (see Section 3.2). We devote this section to the construction of this additional camera, as it plays a key role in the correct operation of PGM.

In order to see why an additional camera is needed in the first place, let us briefly assume that there is only a single camera – the camera which is used to render the scene. Imagine that the camera is placed at low altitude just in front of a mountain, while looking toward the horizon.



(a) Missing part of the terrain underneath the camera (indicated by the blue circle) when creating the surface approximation. Sampling points on the ground plane are visualized by blue dots, which after displacement by the fetched heightmap value create a terrain surface approximation, shown by connected red dots.

(b) Backfiring. The persistent grid is incorrectly split into two parts. The upper part of the grid (highlighted in purple) samples irrelevant parts of the ground plane behind the camera, leaving us with only the lower part of the grid (depicted in blue) sampling the ground plane in front of the camera.

Figure 3.9: Problems of the projected grid method when the main camera is used for the persistent grid's projection. In (a) we see that part of the terrain is missed. This happens because the main camera does not see the part of the ground plane that gives rise to a terrain section that is actually visible. So, while the ground plane itself is not in the main camera's view frustum, the terrain above it may well be, due to displacement. Image (b) illustrates that any view ray of the main camera that has a y-coordinate larger than or equal to zero will either have an intersection with the ground plane behind the camera, or no intersection at all.

If we now project the persistent grid from the camera's view plane into the scene, the grid will not cover the ground-plane area just underneath the camera. This means that this part of the heightmap will not be sampled, and thus none of the associated terrain will be rendered at all, resulting in an output image that misses the mountain in the camera's vicinity, cf. Figure 3.9(a). Furthermore, since the camera's position acts as the center of projection when mapping the persistent grid onto the ground plane, any view ray v with $v.y = 0$ will not even intersect the ground plane, while a ray with $v.y > 0$ will yield an intersection with the ground plane that is actually behind the camera, which is known as backfiring, cf. Figure 3.9(b).

These problems can be solved by introducing an additional camera. This auxiliary camera aims in the same direction as the main camera, but in such a way that the ground plane is always in full view from the auxiliary camera. Therefore, projecting the grid from the auxiliary camera's viewplane always yields correct intersections with the ground plane in front of the camera, without missing potentially visible terrain underneath or close to the viewer. If we rendered the persistent grid from this second camera before and after projection onto the ground plane (without performing height displacement), only the grid vertices' depth in the auxiliary camera's clip space would change, whereas the x and y-coordinates of the grid vertices on the auxiliary camera's view plane would remain unchanged. Hence, when using a uniformly distributed grid to start with, the grid remains tessellated in a screen-uniform manner even after projection onto the ground plane. Of course, after the vertices are displaced, the grid may be distorted so that it no

longer appears regular on screen.

At this point we have answered the question why a second camera is needed at all. Further, we have seen that the auxiliary camera essentially acts as the persistent-grid projector. To avoid confusion between the two involved cameras, we will refer to the "standard" camera through which we view the scene as the *main camera* or *viewing camera*. Unfortunately, in the literature the two cameras are not consistently named. For example, Claes Johanson [30] uses the terms "camera" for the main camera, and "projector" for the auxiliary camera, Löffler et al. [34] calls the main camera "scene camera", and refers to the auxiliary camera as "projection camera", and Mahsman [37] uses the terms "viewing camera", or just "camera" for the main camera, and "sampling camera" for the auxiliary camera. We adapt the naming scheme from the original Persistent-Grid-Mapping publication [32].

To construct the auxiliary camera, we start by cloning the main camera. The necessary adjustments that are then involved in creating the auxiliary camera will be the topic of Section 3.2.2.1 and Section 3.2.2.2.

3.2.2.1 Aim the Auxiliary Camera at the Ground Plane

If the auxiliary camera does not point at the ground plane properly, we must adjust that camera's orientation. In order to determine whether the coordinate frame needs to be adjusted, we calculate the four direction vectors that emanate from the auxiliary camera's position and end at the corners of the far plane. If any of the ray vectors' y-coordinate is larger than or equal to zero, i.e., if any viewing ray emanating from the auxiliary camera is parallel to the ground plane or points into the sky, the auxiliary camera needs to be realigned. The reorientation ensures that the auxiliary camera aims completely at the ground plane, and thus never sees the horizon, let alone the sky. Now, when using the auxiliary camera as the projector for the persistent grid, we get rid of backfiring, and the persistent grid no longer fails to sample parts of the terrain that are close to the viewer.

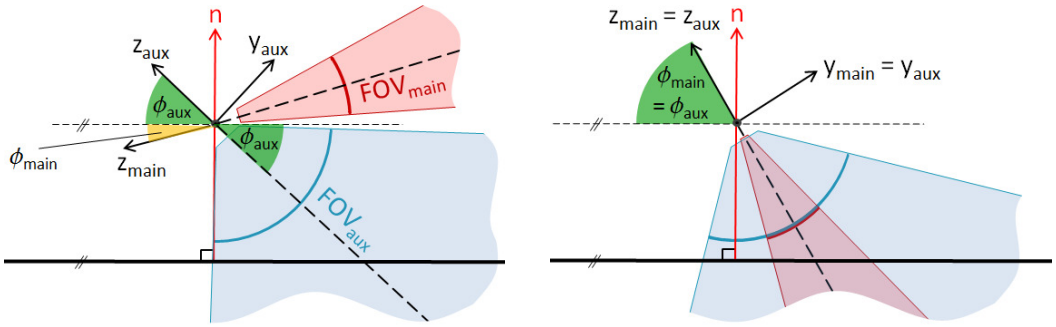
The actual reorientation of the auxiliary camera starts with projecting its x-axis x_{aux} onto the ground plane. This is achieved by setting $x_{aux}.y$ to zero. Note that if x_{aux} was orthogonal to the ground plane prior to projecting x_{aux} onto the plane, we would end up with a zero vector, which would make it impossible to build a valid coordinate frame for the camera. To detect this case, we check the length of x_{aux} after projection onto the ground plane and if it is close to zero, we first rotate the auxiliary camera's coordinate frame around z_{aux} by -90 degrees, and then proceed as before. The resulting vector x_{aux} is then normalized.

Now that x_{aux} is parallel to the ground plane and has unit length, we point the auxiliary camera downward. This is achieved by modifying z_{aux} such that it encloses the angle ϕ_{aux} with the ground plane, cf. Figure 3.10, where ϕ_{aux} is given by Equation (3.1).

$$FOV_{aux} = max(FOV_{main}, 90) \tag{3.1a}$$

$$\phi_{aux} = max(\phi_{main}, \frac{FOV_{aux}}{2} + \epsilon), \text{with } \epsilon \to 0^{+} \tag{3.1b}$$

In Equation (3.1), ϕ_{aux} and ϕ_{main} denote the angles between the ground plane and the z-axis of the respective camera, FOV_{main} is the main camera's vertical field of view, FOV_{aux} is the



(a) When the main camera is pointing upwards, ϕ_{aux} must be set to $FOV_{aux}/2 + \epsilon$, where $\epsilon \to 0^+$, to aim the auxiliary camera properly at the ground plane.

(b) When the main camera is pointing at the ground plane, ϕ_{aux} does not need to be altered and may remain equal to ϕ_{main}. Apart from differing fields of view, both cameras are identical.

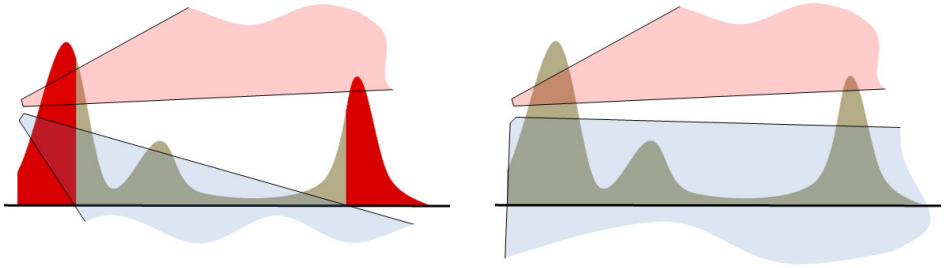
Figure 3.10: Visualization of the axes and angles involved in calculating the auxiliary camera's orientation. The main camera's view frustum is depicted in red, the auxiliary camera's view frustum is indicated in blue. The ground-plane normal is depicted by a red arrow labeled with n, while y_{aux} and z_{aux} denote the auxiliary camera's y- and z-axis, respectively. The auxiliary camera's x-axis is currently pointing toward the viewer. The main camera's z-axis is labeled with z_{main}, and ϕ_{main} is the angle between the ground plane and z_{main}. Since we use right-handed coordinate systems, the camera's view direction is along the camera frame's negative z-axis. The view vectors are indicated by dotted lines.

auxiliary camera's adjusted vertical field of view, and ϵ is a small positive value. All angles are measured in degrees. Adding ϵ precludes auxiliary-camera viewing rays that are parallel to the ground plane. The first part of Equation (3.1) makes sure that FOV_{aux} is at least 90 degrees, since otherwise parts of the surface, especially just in front of the camera, may be missed by the projected grid, even if the auxiliary camera pointed down at the ground plane properly, cf. Figure 3.11(a). Figure 3.11(b) shows that extending the auxiliary camera's field of view to FOV_{aux} fixes the holes in the terrain surface that arise otherwise.

Once x_{aux} and z_{aux} are known, the auxiliary camera's y-axis is given by the cross product of the other two axes, i.e., $y_{aux} = z_{aux} \times x_{aux}$. The newly established auxiliary camera frame finally aims at the ground plane at all times, so that the projection of the persistent grid works as desired.

3.2.2.2 Adjust the Auxiliary Camera's Position

So far, we have only modified the sampling camera's orientation and field of view, and the auxiliary camera's position is still equal to the main camera's position. When the main camera is situated at low altitudes very close to the ground plane, however, we limit the auxiliary camera's height to a small positive value. Otherwise, in the worst case, when the auxiliary camera is exactly at the ground plane's height, the persistent grid would be mapped to a single point on the ground plane.



(a) (b)

Figure 3.11: When the auxiliary camera's vertical field of view is too narrow, the grid potentially fails to sample some parts of the ground plane which give rise to terrain that should be visible (a). Here, the ground plane is properly sampled only after extending the field of view of the auxiliary camera to at least 90 degrees (b). The main camera's view frustum is depicted in red, the auxiliary camera's view frustum in blue.

Apart from this minimum-height limitation, Johanson explores several other alternatives for the placement of the auxiliary camera in his thesis [30]. His alternative placement strategies aim at optimizing the distribution of the grid vertices on the ground plane for several view directions and positions. Nevertheless, we do not consider Johanson's more elaborate techniques for translating the projector camera, since our subsequent contributions grid tailoring (Section 3.3) and grid warping (Section 3.4) address this topic in such a way that we do not profit from further processing at this stage.

3.2.3 Determining the Projected Grid Vertex Positions

To find the positions for sampling the heightmap, we simply form one ray from the auxiliary camera's position through each vertex of the persistent grid and intersect this ray with the ground plane. Let C denote the sampling camera's position, v the current ray's direction vector, and P the resulting intersection point on the ground plane, then we end up with Equation (3.2).

$$P = C + t * v \tag{3.2}$$

We can solve for the parameter t, since we know that the intersection point P must lie on the ground plane. This means that $P.y$ must be zero, which leads to Equation (3.3).

$$t = -C.y/v.y \tag{3.3}$$

Since the auxiliary camera is always aimed properly at the ground plane, every ray that emanates from the auxiliary camera's center of projection and pierces the auxiliary camera's view plane is guaranteed to hit the ground plane.

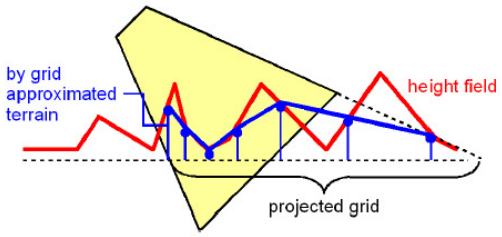


Figure 3.12: Undersampling leads to an inappropriate reconstruction of the terrain surface. Image courtesy of [34].

3.2.4 Properties of PGM

The perspective mapping of the persistent grid onto the ground plane results in a view-dependent continuous-LOD approximation of the terrain dataset. To see why this is the case, imagine the camera looking toward the horizon. Each sampling ray emanates from the auxiliary camera's center of projection and then pierces a grid vertex on the auxiliary camera's view plane. When calculating intersection points of these rays with the ground plane, as the y-coordinates of two neighboring grid vertices on the view plane increase, the distance between the corresponding sampling points on the ground plane also increases, thus leading to a natural adaptation of the sampling intervals, cf. Figure 3.3(b). Many other terrain-rendering algorithms decompose the terrain domain into patches that are culled against the view frustum, and rendered at varying detail levels (see Section 1.2). In contrast, PGM adaptively tessellates a single patch – the persistent grid – in a view-dependent way. This tremendously minimizes the otherwise necessary bookkeeping associated with selecting only the visible patches at the appropriate detail level. The many-patches approach, however, makes it possible to align the patches, and thus the terrain geometry, to the elevation data. This is not possible when employing the PGM method. Since the projected grid does not snap to the terrain data, but rather moves freely over the ground plane with the camera, PGM suffers from the aforementioned swimming artifacts. Swimming artifacts are especially disturbing on a supposedly static terrain surface. Even in the absence of camera movement and the accompanying swimming artifacts, inadequate sampling may fail to faithfully reconstruct important terrain features. Without precaution, PGM tends to undersample the terrain, especially at bigger distances, cf. Figure 3.12.

To summarize, Persistent Grid Mapping exhibits the following properties:

- Automatic continuous LOD without LOD-switching-related popping
  Due to the nature of the perspective transform, the sampling intervals on the ground plane are enlarged with increasing distance to the camera, which is exactly the idea behind level-of-detail algorithms. Unfortunately, projected grid cells scale non-uniformly and become increasingly elongated along one dimension towards the horizon. This impedes appropriate sampling of the heightmap, and makes terrain features disappear quickly in the distance, producing a flat terrain silhouette.

- Watertight terrain mesh

Terrain tessellation is guaranteed to be watertight as long as the persistent grid itself does not contain any holes, which is easy to ensure.

- Automatic view-frustum culling
  Since the camera-aligned grid is projected onto the ground plane from a camera that is based on the main camera, the mesh roughly covers only the area in the camera's view frustum.

- Low bus-bandwidth requirements and constant frame rate
  If graphics hardware supports vertex-texture fetch or render-to-vertex-buffer, the displacement of vertices can take place on the GPU. This means that the persistent grid itself needs to be uploaded to the GPU only once at startup. Since the amount of rendered vertices stays constant, the algorithm guarantees a constant frame rate.

- Swimming artifacts
  As the camera moves over the terrain, the heightmap is sampled at ever-changing positions, leading to an unstable terrain-surface approximation from frame to frame. This artifact is most noticeable in the presence of high-frequency terrain features.

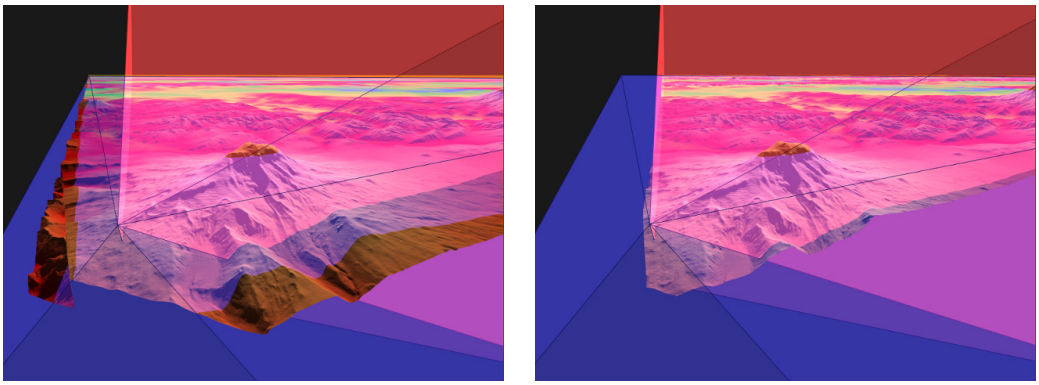
- No guaranteed maximum screen-space error
  Since PGM involves no terrain-data preprocessing, and the sampling positions are not restricted to line up with the regular texel positions of the heightmap, no guarantee can be given about the maximum deviation of the current approximation from the actual terrain dataset – neither in world space nor in screen space.

The properties just listed make the algorithm a perfect candidate for water rendering, since its shortcomings are less noticeable in that context. For example, swimming artifacts quite literally don't disturb the user's visual experience and are hard to spot, since water surfaces are not static in the first place. Further, water waves are generally rather shallow, so that the reconstruction error is hardly noticeable. The potential reconstruction artifacts in the case of camera motion, caused by the freely moving sampling locations when rendering a landscape instead of a water surface, were for example already pointed out by Hinsinger et al. [28].

3.2.5 Other PGM-based Algorithms

Apart from PGM [32, 33] itself and Johanson's thesis "Real-time water rendering: Introducing the projected grid concept" [30], the most important influence for our work is the terrain-rendering algorithm by Löffler et al. [34]. They present a PGM-based algorithm that renders the landscape within a given screen-space error threshold. While the basic PGM algorithm has a constant frame rate, their algorithm does not exhibit this property anymore, since in order to guarantee the maximum allowed screen-space error, ray casting must be performed on a varying amount of pixels. They also present a grid-warping step on the CPU which is applied prior to the grid's projection. Their warping algorithm redistributes the vertices of the persistent grid in order to improve the quality of the reconstructed terrain surface.

Schneider et al. [49] and Mahsman [37] apply the projected-grid method to planetary terrain and thereby show that PGM is not restricted to landscapes defined on a planar domain.



(a) Without tailoring: the mesh covers "everything that the sampling camera sees".

(b) With tailoring: the mesh covers "only what the main camera can potentially see".

Figure 3.13: Tailoring the projected grid to the camera's view frustum. The sampling camera's view frustum is indicated by a blue transparent surface, the main camera's view frustum is shown in red. Note that in the basic algorithm (a) unnecessarily wide areas are covered by the grid, thereby increasing the distance between adjacent samples. The improved method (b) manages to redistribute samples to lie mainly in the actually visible part. The sampling density is therefore increased where it is needed most. As can be seen in (b), tailoring helps the most in areas close to the camera when looking toward the horizon.

3.3 Tailoring the Persistent Grid

PGM samples the heightmap from an auxiliary camera, which, as we have seen in Section 3.2.2, may deviate from the viewing camera, especially when the camera is looking toward the horizon, cf. Figure 3.13. Due to the differing view frusta of the two cameras, the projected grid potentially covers a conservatively large area, especially in the viewer's vicinity, so that a certain percentage of the reconstructed terrain will never be visible, thereby wasting precious grid sampling points. This observation motivated our first modification to the PGM method, the tailoring of the persistent grid, which we present in this section.

The goal of grid tailoring is to increase the vertex efficiency – the fraction of grid vertices actually ending up in the main camera's view frustum. In short, grid tailoring is realized by first determining the minimal area on the ground plane that the grid needs to cover, and to then incorporate this information into the grid-projection step by adapting the grid-vertex positions on the auxiliary camera's view plane. Johanson already presented a limited grid-tailoring method in his Master's thesis [30]. He also shrinks the persistent grid before projecting it onto the ground plane, but while our method is free to transform the grid into a general polygon, Johanson uses scaling along the grid's x- and y-axes only. Hence, the grid-transformation that he presents still maps the persistent grid to a trapezoidal shape on the ground plane. Due to this restriction, the shrunk grid generally still covers an unnecessarily large area on the ground plane. In the following, we refer to Johanson's grid-shrinking technique as *bounding-box tailoring*.

We now describe our less-restricted grid-scaling transformation, which shrinks the persistent

grid more aggressively than Johanson's basic bounding-box tailoring. Our method is built upon the following two observations. Firstly, the terrain that the main camera potentially sees lies within the intersection of the terrain volume and the main camera's view frustum. We refer to the resulting intersection volume as V_{main}. Secondly, the terrain that PGM reconstructs is defined by that part of the terrain volume which is confined by the intersection of the sampling camera's view frustum with the ground plane. We refer to the resulting intersection volume as V_{aux}. Our tailoring step makes sure that V_{aux} is equal to V_{main}, so that after tailoring the persistent grid covers only that part on the ground plane which gives rise to terrain that can be potentially seen through the main camera. Any vertex not lying inside the tailored ground-plane area does not end up in the main camera's view frustum, regardless of the specific height displacement, and would thus be wasted. This minimal area on the ground plane that the persistent grid needs to cover is a planar polygon which can be determined as follows:

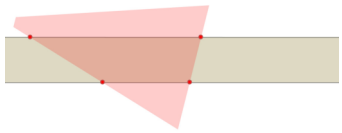
1. Intersect the main camera's view frustum with the terrain volume, yielding the volume V_{main}, cf. Figure 3.14(a).

2. Project the resulting intersection-shape's corner points onto the ground plane by setting their y-coordinate to 0, cf. Figure 3.14(b).

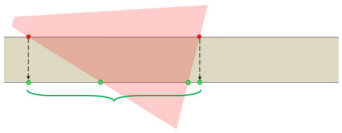
3. Determine the two-dimensional convex hull of the intersection points on the ground plane. The polygon formed by the convex hull is the desired minimal area.

Now that we know which area on the ground plane the persistent grid should ideally cover, we explain how to restrict the grid to be mapped solely onto this area. In the basic PGM-projection step, the persistent grid's two-dimensional domain matches the auxiliary camera's view plane, i.e., as we vary the grid coordinates within the range $[0; 1]\times[0; 1]$, the grid completely covers the view plane, which causes the rectangular grid to be mapped to a trapezoidal shape on the ground plane, cf. Figure 3.14(d). Further, the projection of the persistent grid from the auxiliary camera's view plane onto the ground plane is invertible. Mapping subsets of the covered ground-plane area back to the auxiliary camera's view plane is accomplished by simply rendering the area in question (without applying any displacement) using the auxiliary camera's model-view-projection matrix.

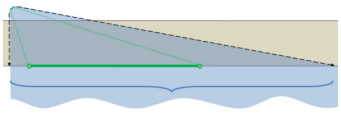
The bijective grid-projection mapping is the key to incorporating the grid-shrinking into the grid-projection step: rendering the convex-hull polygon from the auxiliary camera's point of view gives the area that the convex-hull polygon covers on the auxiliary camera's view plane. Typically, the rasterized convex-hull polygon will not cover the whole view plane, cf. Figure 3.15(a). Hence, we can increase the grid's vertex efficiency by scaling the grid (still in the sampling camera's clip space) such that all vertices fit within the convex-hull area just rasterized. In Figure 3.15(a), we further indicate the ineffectiveness of Johanson's bounding-box tailoring – for illustrative purposes, we apply it only along the y-coordinate of the persistent grid. We see that due to the y-coordinate bounding-box tailoring, the uppermost and lowermost convex hull vertices already touch the view plane's top and bottom borders, respectively, and that there is yet lots of area on the view plane that corresponds to ground-plane coordinates which do not contribute to the final image, but which the grid will sample nonetheless (shown in black). To restrict the grid to lie within the reduced area, we introduce an indirection step which alters the



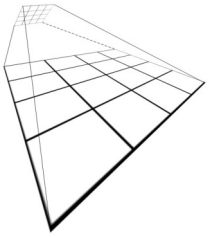
(a) Intersecting the main camera's view frustum with the terrain volume.



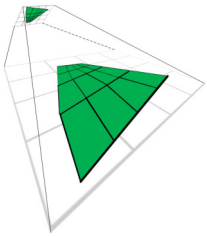
(b) Intersection points after projection onto ground plane – resulting convex hull indicated by the green parenthesis.



(c) Without tailoring, the auxiliary camera projects the persistent grid on an unnecessarily large area on the ground plane (blue parenthesis).



(d) 3D view of persistent grid covering all of the auxiliary camera's view plane, and the resulting area that is covered on the ground plane after projection.

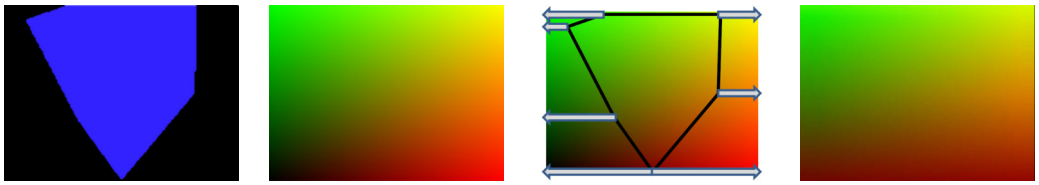


(e) 3D view of persistent grid after tailoring – tailored grid shown in green.

Figure 3.14: Tailoring the persistent grid to the main camera's view frustum. The main camera is indicated by a red view frustum, the sampling camera by a blue one, and in (a) through (c) the terrain volume is denoted by a beige rectangle. Further, images (b) and (c) illustrate that typically, the region of the heightmap that corresponds to the visible part of the terrain surface is smaller than the region that the persistent grid covers. Images (d) and (e) show how the grid vertices are less spread out when our tailoring algorithm is applied, hence resulting in more accurate heightmap sampling.

grid-vertex coordinates on the auxiliary camera's view plane, i.e., before projecting the vertex onto the ground plane. This can be implemented on today's graphics hardware efficiently by creating a redirection texture – accessing this texture with the unaltered vertex coordinates that are stored in the persistent grid will give the tailored position of each grid vertex. This way, even when performing grid tailoring, the persistent grid itself still remains static. We use the natural correspondence between 2D positions on the view plane, which are in the range $[0;1]\times[0;1]$, and the red and green color channels in the redirection texture, cf. Figure 3.15(b). For example,the RGB values for black $(0,0,0)$, red $(1,0,0)$, green $(0,1,0)$, and yellow $(1,1,0)$ correspond to the positions of the four corners of the view-plane $(0,0)$, $(1,0)$, $(0,1)$, and $(1,1)$, respectively. Therefore, the colors that are contained within the view-plane range that the convex hull covers represent the coordinates that we want the grid-vertex positions to vary between.

To create the redirection texture, we spread the convex-hull polygon so that it covers the whole view plane, as is illustrated in Figure 3.15(c). When rendering the scaled-up convex-hull polygon to the redirection texture, we use the polygon's initial vertex positions on the view plane as the polygon's vertex colors, thus covering all of the view plane in the smoothly interpolated



(a) (b) (c) (d)

Figure 3.15: Use of a redirection image in the tailoring step. Image (a) shows the convex-hull polygon – rendered in blue – as seen from the auxiliary camera. Image (b) illustrates the correspondence between colors and positions. Image (c) indicates how the convex-hull polygon is scaled up so that it covers the whole view plane. In the final redirection image (d) the whole view plane is dyed with smoothly interpolated colors that were originally contained within the convex-hull polygon only. By fetching the colors from this redirection texture, using the unaltered grid-vertex positions as texture coordinates, and interpreting the fetched colors as the new grid-vertex positions, the persistent grid is ultimately projected exactly onto the convex hull area on the ground plane.

colors corresponding to the convex-hull polygon's position range, cf. Figure 3.15(d).

Stretching the convex-hull polygon over the whole view plane is performed by first finding the vertices with the smallest and largest y-coordinate on the view plane, respectively. Then we split the polygon at these points, basically dividing its outline into a left and right polyline. If there are two vertices at the same maximum or minimum height, we assign the left vertex to the left polyline and the right vertex to the right polyline. Otherwise, we duplicate the vertex in question so that we still have one unique vertex for each of the two polylines. Referring to Figure 3.15(c) once more, there are, for example, two distinct uppermost vertices but only a single lowermost vertex which is yet to be duplicated. Consecutively, we move all convex-hull vertices that belong to the left polyline to the left view plane border by setting their x-coordinates to 0, and the vertices in the right polyline to the right border, by setting their x-coordinates to 1. Next we calculate the stretched y-coordinate of each grid vertex v by Equation (3.4), where $v_{y_{min}}$ is the convex-hull vertex with the smallest y-coordinate, and $v_{y_{max}}$ is the convex-hull vertex with the largest y-coordinate.

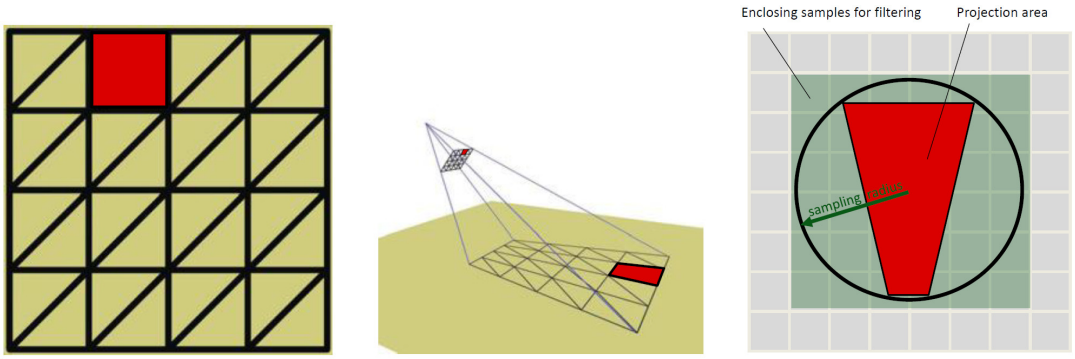
$$v_{stretched}.y = (v.y - v_{y_{min}}.y) \,/\, (v_{y_{max}}.y - v_{y_{min}}.y) \tag{3.4}$$

In the actual implementation, we saw that it suffices for the redirection texture to be considerably smaller than the grid's resolution if we enable bilinear filtering when accessing the texture. This reduces both memory- and bandwidth requirements while the effectiveness of the tailoring step is hardly affected in a negative way. To give a concrete example, we use a 32×32 texture for a 240×1050 persistent grid. Further, we don't need to store the redirection coordinates with 32-bit single-precision floating-point numbers – 16-bit half-precision floating-point numbers proved to provide enough precision, and while we currently store absolute redirection coordinates, we could easily switch to storing relative offsets only.

3.4 Warping the Persistent Grid

The perspective mapping of the persistent grid onto the ground plane results in a view-dependent continuous LOD approximation of the terrain dataset. To analyze the behavior of this mapping, we look at individual cells in the persistent grid. A cell is the rectangular area spanned by the four adjacent grid vertices G_{ij}, $G_{(i+1)j}$, $G_{(i+1)(j+1)}$ and $G_{i(j+1)}$, with $i, j \in \mathbb{N}$ and $0 \leqslant i < m, 0 \leqslant j < n$, where m and n are the number of vertices along the persistent grid's x-dimension and y-dimension, respectively, as illustrated in Figure 3.1. Under the projection, rectangular grid cells on the auxiliary camera's view plane are transformed to trapezoids on the ground plane. As the distance to the camera increases, cells are increasingly elongated on the ground plane, cf. Figure 3.16(b). While the cell-stretching property is desired and is in fact responsible for the automatic LOD adaption of the persistent-grid mesh, cell stretching is often too strong in the distance, leading to projected cells spanning many heightmap samples along the projected cell's y-dimension, while only covering a few heightmap samples along the projected cell's x-dimension. Since a distant projected cell's width and height vary significantly, it is difficult to choose an adequate mipmap level when sampling the heightmap texture, cf. Figure 3.16(c). While we would like to choose a high-detail mipmap level based on the shorter cell dimension, we are forced to accommodate the larger cell dimension, which means that we must sample the heightmap at a coarse mipmap level – otherwise we would violate the Nyquist sampling criterion. While using the higher-detail mipmap level may even result in better-looking screenshots (see Figure 3.17(c)), in animations we clearly see severe vertex swimming due to the heightmap undersampling. Choosing the coarser mipmap level, however, makes high-frequency detail disappear fast as the distance to the camera increases, leading to an almost flat horizon, cf. Figure 3.17(b). At the same time as cells that project to distant locations on the ground plane are strongly elongated, cells at the bottom of the grid in projector space, i.e., cells closest to the main camera after projection, tend to cover the ground plane unnecessarily densely. A similar observation was already made by Hinsinger et al. in their paper "Interactive Animation of Ocean Waves" [28]. Nevertheless, the authors merely describe the effect of redistributing vertices on the sampling camera's view plane without presenting an algorithm for this grid-vertex redistribution at runtime. We can thus relocate grid vertices from densely covered areas to sparsely covered areas, to lessen the above-mentioned mipmap-level disparity, while not harming the reconstruction quality of the terrain surface up close. Figure 3.17(d) illustrates how the grid warping improves the fidelity of the terrain reconstruction in the distance. In any case, the direction of maximum cell stretching always corresponds to the y-axis in projector space, so that we can focus our attention on adjusting grid vertex positions on the auxiliary camera's view plane along the y-axis only. The projected camera-space y-axis of the auxiliary camera is the axis of anisotropy. In this section we present our warping technique, which efficiently redistributes grid vertices from densely covered parts on the ground plane to sparsely covered areas along the axis of anisotropy, so that the persistent grid is stretched less non-uniformly on the ground plane, and projected cells end up having a near-square footprint on the ground plane, cf. Figure 3.18.

When sampling the heightmap at a projected grid-vertex position, we must select an appropriate mipmap level in order to prevent aliasing artifacts. As cells get stretched a lot, there is a mismatch between the mipmap level that corresponds to the shorter edge and the longer edge,



(a) Rectangular cell in projector space.

(b) Cell after projection onto ground plane.

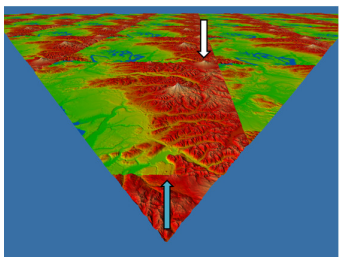
(c) Projected cell footprint and required filter radius.

Figure 3.16: The projection of the grid maps rectangular and near-square cells (a) to elongated trapezoids (b). When selecting the mipmap level for accessing the heightmap texture, we must take the 2D footprint of the projected cell (c) into account to avoid aliasing artifacts. Since the cell is stretched non-uniformly by now, there is a strong mismatch between the desired filter widths along the two dimensions of the cell. To avoid undersampling, we must accommodate the larger of the two, which makes the high-frequency detail disappear quickly as the terrain approaches the horizon. Images courtesy of [34].

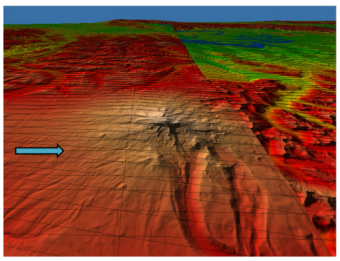
respectively. To satisfy the Nyquist criterion, we choose the coarser mipmap level. Hence, the basic PGM algorithm fails to preserve high-frequency terrain features, making the terrain appear unnaturally flat toward the horizon.

Before describing our grid warping in more detail, we shortly explore two alternatives to better sample distant areas on the ground plane without increasing the vertex count of the persistent grid: anisotropic filtering and a persistent grid tessellation tweaked to counteract the worst-case cell stretching. While anisotropic filtering addresses this very problem – the proper filtering of non-square sampling footprints – it requires taking many heightmap samples along the axis of anisotropy, making it costly performance-wise. Further, for anisotropic filtering to work automatically at the hardware level, the screen-space derivates are needed. This means that anisotropic filtering is not available when we sample the heightmap in the vertex stage of the graphics pipeline. Of course, anisotropic filtering can be emulated by manually performing multiple texture fetches in the vertex shader, but this degrades performance even more. The second option to more appropriately sample the heightmap in the distance is to create a static non-uniformly distributed grid with increasing vertex density toward the top of the grid, in order to mitigate the cell stretching even when aiming the main camera at the horizon. The advantage of this approach is that there is no runtime overhead. The static nature is, however, also its big disadvantage, since in cases where the main camera looks down onto the ground plane, the grid should actually stay uniformly tessellated. Obviously, a more flexible, adaptive approach is needed, so that grid vertices are redistributed exactly as needed.

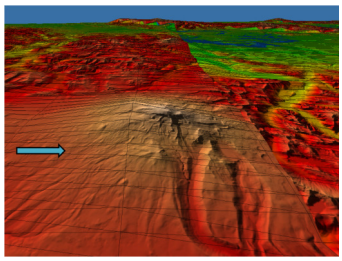
While the two alternative techniques get the job done, they both have prohibitive drawbacks. Anisotropic filtering is too costly, and the static non-uniformly-distributed grid fails to work well



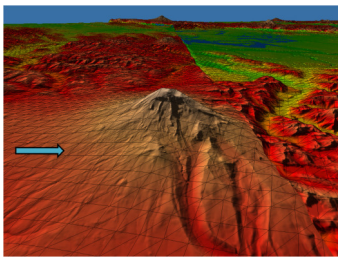
(a) Meta view.



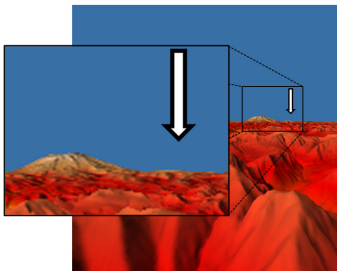
(b) Without warping, close-up of distant mountain.



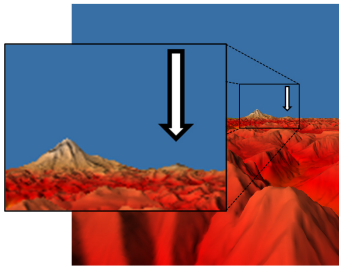
(c) Without warping (using higher-detail mipmap level), close-up of distant mountain.



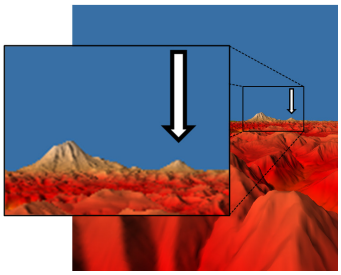
(d) With warping, close-up of distant mountain.



(e) Without warping, as seen from main camera.

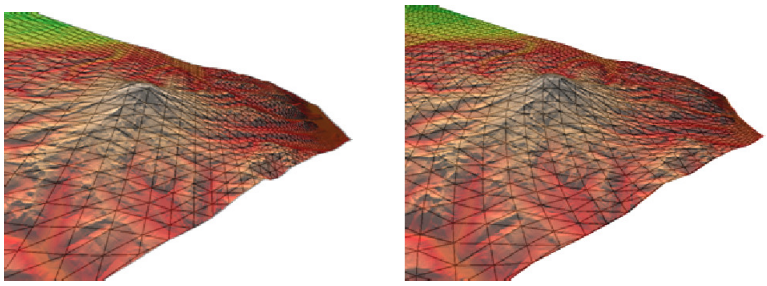


(f) Without warping (using higher-detail mipmap level), as seen from main camera.



(g) With warping, as seen from main camera.

Figure 3.17: Terrain surface without and with importance-driven warping of the persistent grid, respectively. Image (a) shows a meta view of the terrain, highlighting a distant mountain with a white arrow. The blue arrow points in the viewing direction. Images (b), (c) and (d) show close-ups of the distant hill. To see the underlying tessellation, a wireframe overlay is rendered on top of the terrain in images (b), (c) and (d). Images (b) and (c) reveal how strongly stretched the projected persistent-grid cells already are at this distance to the main camera when no importance-driven warping is performed. In image (c) and (f), we use the shorter of the two cell dimensions to determine the mipmap level, which seems to give better results in screenshots, but as soon as the camera moves, this setting leads to strong vertex swimming. Images (e), (f) and (g) show the corresponding views from the main camera with an inlay that displays a zoomed part of the view-plane area around the distant mountain. We see that without warping the mountain is filtered out almost completely, while with warping in place, terrain features toward the horizon are reconstructed more faithfully.



(a) Projection of screen-space uniform grid

(b) Projection of warped grid

Figure 3.18: The difference between projecting a screen-space uniform grid (a) and an importance-driven warped grid (b). Note how grid cells close to the camera are slightly widened, while stretched cells at increasing distances from the camera are shrunk. Images courtesy of [34].

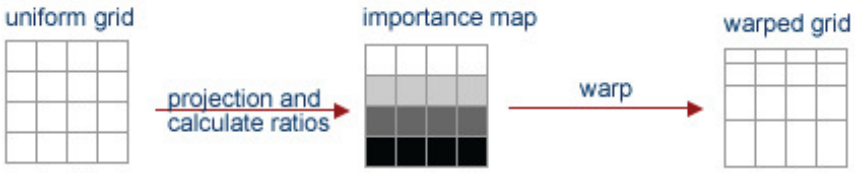


Figure 3.19: The basic steps involved in the grid warping algorithm. Image courtesy of [34].

across all possible viewing directions. Instead, we perform an importance-steered distortion step which redistributes grid vertices on a per-frame basis. In fact, this idea has been applied to PGM before by Löffler et al. [34], cf. Figure 3.19. In their work, warping is performed on the CPU. We present a different warping algorithm that runs on the GPU.

Just as in Löffler's importance-driven warping step [34], we restrict the warping to happen only along the y-coordinate on the auxiliary camera's view plane, since the view-plane y-axis corresponds to the axis of anisotropy, the axis on the ground plane along which cells are stretched. We also pick up Löffler's definition of importance, which attributes a large importance to strongly stretched grid cells and a small importance to weakly stretched cells. This importance metric lends itself to efficiently identifying those areas of the persistent grid where we need to increase the sampling frequency, and where we may lower it, respectively. The importance $I(v)$ of a mesh vertex v is given by the aspect ratio of the projected grid cell that has v as its lower-left vertex, which leads to the following equation

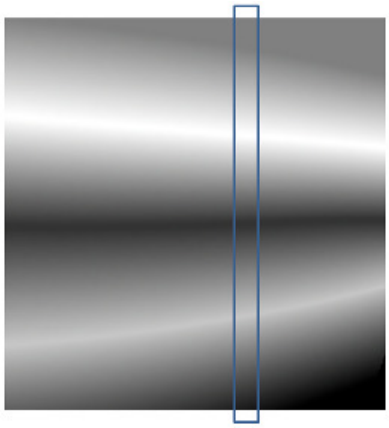
$$I(v) = projectedCell_{height}/projectedCell_{width} \tag{3.5}$$

Since after projection the persistent-grid cells are trapezoids, cf. Figure 3.16, we do not calculate the exact cell dimensions on the ground plane, but merely approximate the projected cell's width and height by taking distances between the current projected vertex position and the projected position of the immediate upper-right neighboring grid vertex.

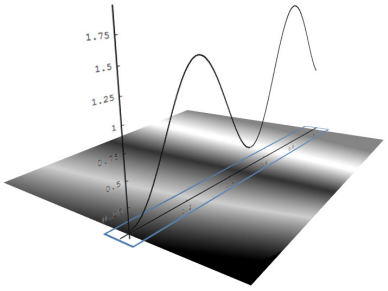
In order to distort the persistent grid along vertical lines, we create an intermediate importance image which we align to the auxiliary camera's view plane. Let the importance image have the same dimensions as the persistent grid, so that an importance image having $m \times n$ pixels corresponds to a persistent grid consisting of $m \times n$ vertices. We let the importance map cover the whole view plane of the auxiliary camera, so that all the pixels of the importance image fall within the range $[0, 1] \times [0, 1]$. Further, remember that we write G_{ij} for a grid vertex at coordinates (i, j) within the mesh. We evaluate Equation (3.5) for each vertex G_{ij} in the persistent grid, and then assign the calculated importance at this vertex to the pixel at the corresponding coordinate (i, j) in the importance image, cf. Figure 3.20(a). Next we turn our attention to vertical lines in the importance image, which are functions themselves, cf. Figure 3.20(b). Let $L_{vline}(y)$ designate a vertical line at a fixed x-coordinate in the importance image. Integrating along such a vertical line, i.e., evaluating $\int_0^1 L_{vline}(y)\,\mathrm{d}y$, gives the area A_{vline} under this line. We define $L_{vline}(y)$ to be 0 outside of the importance image's domain, so that $\int_{-\infty}^{\infty}(1/A_{vline}) * L_{vline}(y)\,\mathrm{d}y = 1$, i.e., the integral of the scaled function evaluates to 1. We can then interpret each vertical line $I_{vline}(y) = (1/A_{vline}) * L_{vline}(y)$ as a *probability density function* (PDF). Integrating a PDF yields a *cumulative distribution function* (CDF), cf. Figure 3.20(d). Since the importance function yields only positive values, its integral is strictly increasing, hence the inverse of the importance function's integral exists and is uniquely determined. Sampling the inverse of the integrated importance function at equidistantly spaced positions yields the desired redistribution of the samples according to the given input importance. In Figure 3.20(e), note how sampling the inverse CDF at equidistant positions maps to positions with varying distances – at positions of high importance the CDF is steep, and thus sampling intervals are decreased, whereas in areas of low importance the CDF increases slowly, which leads to incoming sampling intervals being enlarged.

The warping algorithm just described can be carried out efficiently on the GPU. To generate the importance function, we first render Equation (3.5) to a single-channel half-precision floating-point texture. This is achieved by rendering a viewplane-aligned quad, and interpreting its interpolated texture coordinates as if they were the persistent grid's vertex positions, so that we can calculate the projected ground-plane positions per fragment in exactly the same way as when actually projecting persistent-grid vertices. In order to correctly incorporate the tailoring step, we first transform input coordinates (i, j) according to the redirection texture, i.e. $(i, j) = redirectionTexture\,(i, j)$. Since we now have a mapping between output texels and the ground plane, we simply evaluate Equation (3.5) per fragment and write the resulting value to the output texture.

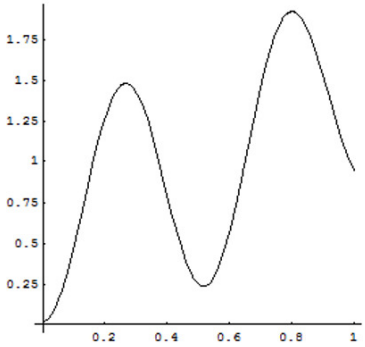
We approximate the integral of the importance function along a vertical line by employing the parallel prefix-sum algorithm on the GPU. We render the integral to a second texture that has the same resolution as the importance texture. With additive blending enabled, the importance texture is repeatedly rendered into this second texture $h - 1$ times, where h denotes the importance texture's height. In each iteration of this render loop, the y-coordinate of the quad that is textured with the importance image is incremented by one pixel. This process calculates the integrals of all vertical lines in parallel. Figure 3.21 shows the values along one vertical line, and that eventually the topmost entry gives the overall sum of all input values. The topmost row in the resulting texture thus contains the approximations of the integrals of each vertical line of



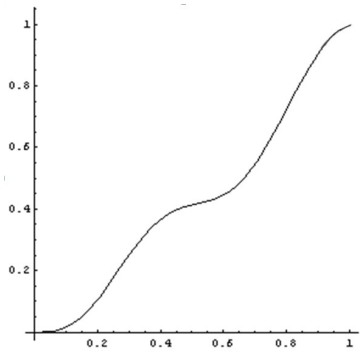
(a) Each vertical line in the normalized importance image defines a function of importances. One such function $I_{vline}(y)$ at a fixed x-coordinate is highlighted in blue.



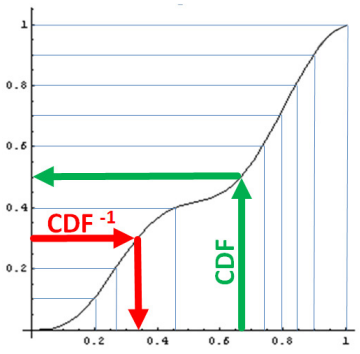
(b) 3D view of $I_{vline}(y)$.



(c) The function $I_{vline}(y)$ is a PDF.



(d) The integral of $I_{vline}(y)$ is a CDF.



(e) Evaluating CDF^{-1} using CDF.

Figure 3.20: Importance-steered redistribution of grid vertices. Image (a) shows the normalized importance image as seen on the auxiliary camera's view plane – normalization is performed by dividing each vertical-line function $L_{vline}(y)$ by the area under itself, yielding $I_{vline}(y)$. Each pixel in the importance map corresponds to the importance of the associated grid vertex. The brighter the pixel, the higher the assigned importance. Images (b) and (c) show $I_{vline}(y)$ as a function of the y-coordinate on the auxiliary camera's view plane at a fixed x-coordinate that is indicated by a blue rectangle in (a). Integrating $I_{vline}(y)$ yields the function shown in (d), which is a CDF. Image (e) illustrates how evaluating the inverse of the CDF redistributes samples according to the PDF. In (e), note how the equispaced input positions on the y-axis under CDF^{-1} accumulate on the x-axis exactly where the importance is high. Note that the importance map shown in (a) is not representative of importance images in PGM, but rather acts as an illustrative example to better show how samples are redistributed as desired by CDF^{-1}. See Figure 3.19 for a typical importance map.

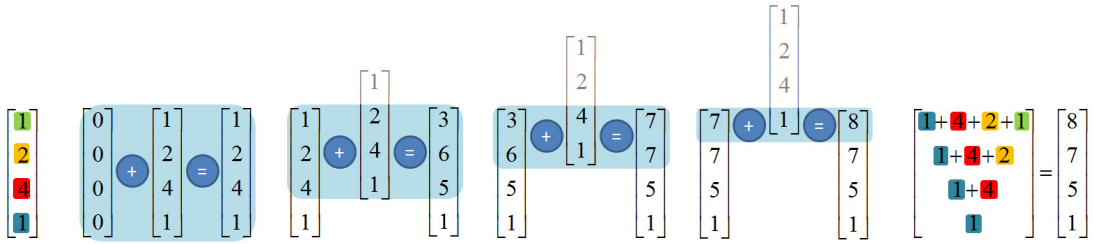


Figure 3.21: Approximating the importance texture's integral on the GPU. Image (a) represents the input importances. Image (b) sketches the intermediate steps of the iterative summation done with the parallel-prefix sum algorithm. In each summation step, the left column represents the current sum, the center column represents the textured quad holding the importances to be integrated, and the right column is the updated sum. Image (c) shows the discrete sum that approximates the integral of the input function.

importances. Since additive blending is commutative, the order in which the textured quads are rendered is irrelevant. We can exploit this property and perform the parallel prefix-sum operation efficiently on current graphics cards by issuing a single instanced draw call in which we use the built-in gl_InstanceID variable as the y-coordinate offset in the vertex shader.

Note that to normalize a vertical line $L_{vline}(y)$ in the importance map, we scale that function by $1/A_{vline}$, the reciprocal of the area under that line, which means that we need to evaluate the integral $A_{vline} = \int_0^1 L_{vline}(y)\,\mathrm{d}y$ first. Once we have determined A_{vline}, however, it does not make sense to normalize $L_{vline}(y)$ any longer, as from then on it is not referenced anymore in the grid-distortion process. All we need is the integral of $L_{vline}(y)$, and since $\int_0^1 (1/A_{vline}) * L_{vline}(y)\,\mathrm{d}y \equiv (1/A_{vline}) * \int_0^1 L_{vline}(y)\,\mathrm{d}y$, we can just as well integrate the unnormalized importance function, and apply the normalization later on. We will see shortly that we can even skip the explicit normalization of the integrated importance map altogether which saves us another render pass, and thus precious render bandwidth. However, for the sake of an easier explanation in the following paragraph, let us assume that we did perform the normalization.

With the normalized integrated importance map (the CDF) at hand, we can finally redistribute grid vertices according to the importance function. We don't need to actually precalculate CDF^{-1} from CDF, since we can use the CDF itself to look up the corresponding value of the inverse function. We explain the process on the basis of Figure 3.20(e). To determine the value of CDF^{-1} at a specific input coordinate y_{in}, we perform an iterative search. Since typically the amount of warping performed is small, we assume that CDF, and thus CDF^{-1}, is the identity function. Therefore, we choose $x_0 = y_{in}$ as the start position for the iterative search. We evaluate the CDF function at this initial position and test the value $y_0 = \mathrm{CDF}(x_0)$ against x_0. If the function value y_0 is smaller than y_{in}, we search further to the right, i.e., we set $x_1 = x_0 + s$, where s is the step size. If y_0 is larger than y_{in}, we search further to the left, i.e., we set $x_1 = x_0 - s$. We perform these calculations until after k iterations the difference between y_k and y_{in} is within a certain threshold, or the sign of the difference changes. In the former case we

set $x_{out} = y_k$. In the latter case we refine the current result with a binary search between x_k and x_{k-1}, and we assign the result of the binary search to x_{out}. The finally determined x-coordinate x_{out} is the searched-for value that the inverse CDF maps y_{in} to, i.e., $x_{out} = \mathrm{CDF}^{-1}(y_{in})$.

Now that we know how to evaluate CDF^{-1} in terms of CDF, we show how we circumvent the normalization of the integrated importance texture before using it for the lookup of its inverse, as advertised before. The topmost texel in each vertical line of the integrated-importance map stores the area A_{vline} of the importance function under that line. In each iteration of the search for the value CDF^{-1}, whenever we fetch a value from the integrated-importance texture, we first normalize the fetched value by multiplying it by $1/A_{vline}$ before carrying on with the already-described calculations.

To combine grid tailoring (see Section 3.3) with grid warping, we use the tailored grid-vertex positions as input coordinates when calculating CDF^{-1}. This means that in the just-described algorithm to evaluate CDF^{-1}, we need to alter each input coordinate (i, j) according to the redirection texture from the tailoring step, so that the coordinate becomes $(i, j) = redirectionTexture\,(i, j)$. This is the exact same coordinate transformation that was already employed when creating the importance map for an already-tailored grid.

Since it is hard to see the effect of the just-presented GPU-based warping algorithm when applied to the abstract images that occur in the PGM algorithm, in Figure 3.22 we illustrate how our warping method distorts a natural image according to a given importance function. When redistributing grid vertices of the persistent grid, we only warp along one axis, therefore Figures 3.22(b) and (c) show an example of a one-dimensional warp. Images (d) and (e) illustrate the application of the warping algorithm to both the x-axis and the y-axis. When warping an image, the equal-spaced input positions are given implicitly by the pixel-raster grid. The final color at that pixel position p is given by looking up the color in the original image at the offset position p', which is defined by evaluating the warping function at the input position p. Returning to the context of terrain rendering, the vertex grid corresponds to the pixel grid in the image to be warped.

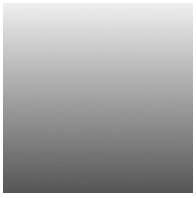
Note that besides our GPU-based warping algorithm, which is inspired by an algorithm to create non-photorealistic images in real time [51], alternative GPU-based warping methods exist. We could for example interpret the persistent grid as a spring-mass system, assign forces according to the importance function, and iteratively solve the system until an equilibrium is reached. Paul Rosen describes yet another warping algorithm [45], where warping is applied to the creation of adaptive shadow maps. His algorithm could however just as well be employed in the redistribution of persistent-grid vertices.

Maximum cell stretching happens when the camera is at low altitude while looking toward the horizon. In this worst-case scenario, we get better results when we use a non-square persistent grid with a higher resolution along its y-axis than along its x-axis to start with, since in a square grid there may not be enough vertices to work with when tackling grid stretching in the warping step. Additionally, with these camera parameters, the tailoring improvement moves the grid vertices even closer together, predominantly along the grid's x-axis, which fortifies reducing the grid's x-resolution in favor of increasing its y-resolution. We have found that a ratio of up to $1 : 5$ works well in practice. For example, the result screenshots in Section 4.3 were all created with a grid size of 240×1050. In order to lessen the performance impact of the warping step,

(a) Input image

(b) Warped image according to (d)

(c) Warped image according to (e)



(d) Input importance for (b)

(e) Input importance for (c)

Figure 3.22: Applying the proposed importance-driven warping algorithm to a photograph. Image (a) is the input image to be warped (Baboon image from `http://www.hlevkin.com/TestImages/classic.htm`). The two-dimensional importance function is given as another image of the same dimension as image (a). Brighter shades of gray mark more important areas. Images (c) and (e) show the respective result after warping. Note how the samples in the original image are redistributed according to their importance, effectively making important areas take up more space in the output image.

for a persistent grid of size 240×1050, it suffices to use 8×256 textures to store the importance map and its integral.

Analogously to the redirection texture (see Section 3.3), we store warped positions as absolute positions in a bandwidth- and memory-requirement friendly 16-bit half-precision floating-point format. Similarly to the remark at the end of the tailoring section, if numerical imprecision became a problem, we could switch the distorting texture from storing absolute positions over to relative positions without resorting to the 32-bit single-precision floating-point format.

3.5 Local Terrain-Edge Search

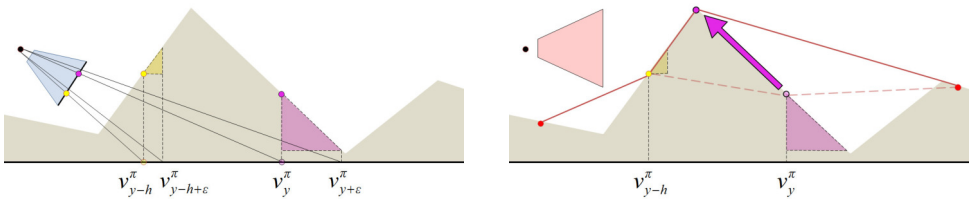
We have already seen that in Persistent Grid Mapping, the persistent grid is projected onto the ground plane according to the auxiliary camera's viewing parameters. This leads to projected persistent-grid vertices not being aligned to the heightmap texel centers on the ground plane, so that certain features in the terrain may be inadequately reconstructed. Further, since the sampling positions slide freely over the heightmap as the camera is animated, the reconstructed terrain surface exhibits vertex-swimming artifacts, i.e., the landscape looks wobbly due to small differences in the respective terrain-surface approximations from frame to frame. The swimming artifacts are most obvious at terrain edges close to the camera, which is why we limit the

local edge search to vertices within a certain distance to the camera. Firstly, this lessens the performance impact of this improvement. Secondly, since importance-driven warping (see Section 3.4) generally shifts vertices toward the horizon, the warped grid covers the ground plane in the camera's vicinity slightly less densely, so that we again profit from the local edge search the most for nearby terrain. In this section, we show how to perform a search for missed edges in the terrain dataset between two projected grid vertices, so that we can subsequently adjust vertex positions on the ground plane in order to reconstruct those terrain edges more accurately.

In the following explanation, let us once again consider grid vertices along a vertical line at a fixed x-coordinate on the auxiliary camera's view plane. Within this line, we can then refer to a vertex at position (x, y) by v_y. The adjacent vertices of persistent-grid vertex v_y along the y-axis are v_{y-h} and v_{y+h}, where h is the grid-cell height. The position that results from projecting v_y onto the ground plane is denoted by v_y^π. To estimate the terrain slope at a vertex v_y, we determine the projected coordinates of a second point $v_{y+\epsilon}$ which has a slightly bigger y-coordinate in projector space. We set the step size ϵ along the y-coordinate to $0.1 \times h$, thus $v_{y+\epsilon}$ is not actually a grid vertex. Nevertheless, we can treat this second point as a virtual grid vertex and project it just as any other real grid vertex. Eventually, the slope is calculated by the forward difference between the corresponding height values at v_y^π and $v_{y+\epsilon}^\pi$, respectively.

To perform the search for a potentially missed terrain edge between two grid vertices, for each grid vertex v_y that we project, we actually calculate four ground plane positions – the positions resulting from the projection of v_{y-h}, $v_{y-h+\epsilon}$, v_y, and $v_{y+\epsilon}$. This allows us to estimate two slopes as described in the previous paragraph, one at v_{y-h}^π, and a second one at v_y^π. If at the projected grid-vertex position v_{y-h}^π the heightmap has positive slope, while at vertex v_y^π the landscape has negative slope, we conclude that there is a terrain edge between the two vertices which has been reconstructed inadequately, cf. Figure 3.23(a). To amend the edge sampling, we move the current vertex v_y^π closer to the missed peak's position, cf. Figure 3.23(b). Since we already know that the slope between the two endpoints changes, we perform the search for the terrain edge in between them by comparing heights of consecutive equispaced intermediate points. Starting at vertex v_{y-h}^π, stepping toward vertex v_y^π will first lead to increasing heights until the edge is found, and then to decreasing heights once we cross the egde. The first time the currently fetched height is less than the height at the previous position, we know that we have just stepped over the edge. We follow up with a binary search between the last two sampling positions to improve accuracy. Finally we set v_y^π to the just-determined approximation of the edge position, and displace the vertex by the terrain height at that position.

Our local edge-search refinement effectively moves vertex v_y^π closer to the camera, thus widening the gap to the next vertex v_{y+h}^π. At first sight, it seems that our algorithm just shifts the undersampling issue from one grid cell to the next, but the specific condition that we check for tells us that prior to our position adjustment, v_y^π lies in a heightfield area of negative slope with respect to the direction of derivation. This means that the vertex is most probably on a back face and thus invisible from the main camera. Whether v_y^π is really occluded depends on the specific camera height and viewing direction, of course. For example, when the camera is at low altitude while looking toward the horizon, the vertex is most probably really invisible. When the camera is at high altitude and looking straight down, however, the vertex is most likely visible, which means that our assumption would be wrong in that case. So while in the former



(a) Inferring the existence of a terrain edge between two adjacent projected grid vertices v^{π}_{y-h} and v^{π}_{y} along a vertical line in projector space. Terrain slope is positive at v^{π}_{y-h}, and negative at v^{π}_{y}.

(b) Terrain-surface approximation before and after applying the local-edge-search improvement. Moving vertex v^{π}_{y} to the position where the terrain's slope changes, captures the previously missed edge.

Figure 3.23: Local Edge Search. By slope-based adjustment of projected grid-vertex positions, terrain edges are reconstructed more faithfully. The height of a grid cell is denoted by h, while ϵ denotes a small fraction of h. In (a) the auxiliary camera's view frustum is sketched in blue, in (b) the main camera's view frustum is drawn in red.

case the enlarged sampling interval doesn't do any harm since the main camera doesn't see that part of the terrain anyway, we may experience a negative impact on the quality of the terrain-surface approximation in the latter case. In practice, this is not too bad, however, since then the landscape is viewed from above and artifacts that stem from undersampling have a far less distracting effect as when viewed head-on. It is exactly for this reason that we do not perform the edge search between adjacent grid vertices along the grid's x-axis as well, since in this case we cannot assume that the vertex being translated was previously invisible. Therefore, adjusting vertices along the grid's x-axis could inadvertently introduce even more swimming artifacts.

The presented local edge-search method may fail to detect a terrain edge between two projected vertex positions. To give an example, we look at Figure 3.23(a) once again. If v_y was projected to a ground-plane position just a little further away from the camera, it would lie in a region of the terrain where slope is no longer negative, and therefore our technique would fail to infer the existence of a missed edge. Further, the iterative search for the edge between two vertices is not totally accurate either. Still, in practice, the method works reasonably well and generally improves image stability significantly.

3.6 Exploiting Temporal Coherence

In landscape visualizations, camera position and view direction typically do not change abruptly from one frame to the next, resulting in output images that show mostly identical subsets of the scene. Hence, terrain-rendering algorithms are suitable candidates for *temporal-coherence* (TC) methods. In this section we explain how to apply and combine both image-based *bidirectional temporal coherence* (BidirTC) and unidirectional temporal coherence [41, 46–48, 65] to lessen vertex-swimming artifacts and terrain-surface reconstruction inaccuracies of the PGM method. Furthermore, we propose a new interpolation algorithm for *bidirectionally predicted frames* (B-frames) in image-based BidirTC. In the following, we introduce the term *unidirectional temporal*

coherence (UnidirTC) to distinguish the earlier temporal-coherence method, which uses data from past frames only, from BidirTC. We use the general term *temporal coherence* (TC) to refer to both UnidirTC and image-based BidirTC.

Before explaining our image-based BidirTC improvements in detail, we start with a short review of UnidirTC (refer to Section 2.4 for a more exhaustive explanation) followed by a brief discussion of the challenges that we faced when applying UnidirTC to PGM.

3.6.1 A Quick Review of Unidirectional Temporal Coherence

As we have outlined in Section 2.4, unidirectional temporal coherence enables us to replace expensive per-fragment calculations by lookups into past frames. In order to access those past rasterizations, we need to store the rasterized data somewhere in the first place. In hardware-accelerated graphics applications, images are best stored in viewport-sized offscreen textures. While we want to access as many frames from the past as possible, storing each one explicitly is impractical. Scherzer et al. [48] present a solution by introducing the history buffer, which captures a potentially infinite number of past frames in a single offscreen texture. This is achieved by constantly updating the history buffer according to Equation (2.9). The history buffer captures data for all currently visible surface positions in a frame, where the stored data typically comprises at least the pixel's color and depth. Note that Nehab et al. [41] introduced an equivalent method to exploit similarities between frames independently at around the same time. They introduced the term reprojection cache for what Scherzer et al. call the history buffer. In the rest of this text we will use both terms interchangeably.

Apart from storing past rasterizations, we also need to know how to calculate the coordinates for the history-buffer lookup in order to take advantage of TC. In short, the coordinates for the history-buffer lookup are determined by reverting the transformation from the current frame and reapplying the transformation from the previous frame. The reprojection is described by Equation (2.10).

If the depth values of the reprojected current position and the history buffer depth at the reprojected 2D position of the current fragment are within a certain range, we have a valid reprojection, or equivalently, a reprojection-cache hit. If, on the other hand, the depth values differ significantly, we infer that the surface positions are most probably unrelated, and using the history buffer's data for the current fragment would yield wrong results. In this case we say the reprojection yields a reprojection-cache miss. Reprojection-cache misses typically happen whenever occlusions between objects change, and if new parts of the scene become visible in the current frame. Clearly, such fragments in the current frame do not have a valid history that could be reused, and so we must evaluate the shading calculations in full for such fragments.

3.6.2 Applying Unidirectional Temporal Coherence to PGM

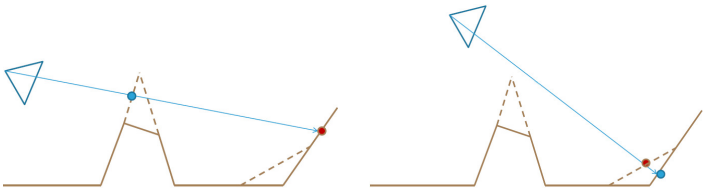
Typically, when unidirectional TC is employed, the geometry itself is rendered correctly each frame, and we merely skip surface-shading calculations whenever reprojection returns a valid entry in the history buffer. However, when applying UnidirTC to the PGM method, we must deal with slightly incorrect rasterized frames. This imperfect terrain-surface reconstruction is the major challenge when applying UnidirTC to PGM, since the inexact fragment depth values turn

the classification of reprojected fragments into reprojection-cache hits and misses, respectively, into a guessing game. Nevertheless, in the following we will show that for a static camera the PGM method can still benefit from the UnidirTC method.

In order for the terrain-surface approximation from a fixed point of view to improve gradually over the course of several frames when performing UnidirTC, we jitter the sampling camera slightly each frame, giving rise to lots of slightly different rasterizations. While each of the grid projections fails to sample all covered parts of the terrain accurately, accumulating all the imperfect results ideally yields a near-perfect terrain reconstruction. We can think of this accumulation of varying rasterizations as if it was the result of sampling the heightmap only once, albeit with a virtual persistent grid that has an extremely high triangle count. The idea of jittering PGM's sampling camera is based on the work of Scherzer et al. [48], where jittering is applied to the shadow camera in order to reduce the aliasing artifacts present in shadow mapping, eventually yielding pixel-perfect shadows.

For the current terrain-surface approximation to iteratively improve after each rasterization of the "jittered" persistent grid, we must decide which history-buffer fragments to keep and which ones to replace. We use a very simple heuristic: when points along the same line of sight are reprojected several times, we keep the closest one in the history buffer. This decision is based on the assumption that when we fail to sample the heightmap correctly, we most probably miss a terrain edge and end up seeing a more distant part of the terrain erroneously. Figure 3.24(a) illustrates how keeping the closer fragment correctly captures a terrain peak that is easily missed, while Figure 3.24(b) shows that the closer point is not always the right one. Yet, preferring the closest surface position is an effective heuristic which leads to hardly noticeable artifacts. Incidentally, the same heuristic is employed in the paper about BidirTC by Yang et al. [65], where they prefer the closer of two reprojected samples when reconstructing B-frames, as outlined in Section 2.4.1. Note that after applying the edge-search improvement described in Section 3.5, we already increase the probability that close terrain edges appear frequently in many different rasterizations, so that we generally arrive at a good approximation of the landscape surface within a small number of frames, even in the presence of high-frequency terrain features.

Note that we can only hope for an iterative image-quality refinement if disocclusion between terrain-surface points does not change, which is only the case for a static camera position (rotations are still allowed and not restricted in any way). If the camera position changes between two frames, applying the closest-fragment heuristic leads to wrong object positions and incorrect depth sorting. Figure 3.25 illustrates how violating the no-camera-translation constraint leads to one object being "smeared" over another. If we really wanted to keep accumulating the closest fragments in PGM even when the camera position changes between frames, we could perform raycasting at the reprojected surface position (at the position of the green "ghost" object, compare Figure 3.25(d)) to verify if there actually is a terrain-surface position, and if so, use the color from the ray-intersection point or from the reprojected surface position. If no ray-terrain intersection is found at the calculated position, we detect the incorrect smearing that UnidirTC would introduce, and instead use the current fragment's rasterized color and depth. Of course, we have to make sure that we do not miss thin sharp peaks during the raycast process, as raycasting is itself prone to aliasing artifacts. Alternatively, if we can't afford to perform raycasting, we may introduce a threshold for the maximum allowed distance between the original and the



(a) Preferring the closer point is correct.

(b) Preferring the closer point is incorrect.

Figure 3.24: Keeping the closest fragment from several jittered rasterizations in the history buffer when the camera doesn't move is a good, albeit not always correct choice. The solid line depicts the persistent-grid mesh in frame t, the dashed line shows the surface approximation in frame $t-1$. Further, the blue dot marks the correct surface position, while the red dot marks an incorrect position. Image (a) depicts a case in which a near peak is missed in frame t, but is correctly sampled in frame $t-1$, so that holding on to the closer position leads to a better heightfield approximation of the involved terrain region. In contrast, in image (b) the closer point is not the correct choice. Yet, when sticking to the closer point anyway, the resulting reconstruction error is hardly noticeable.

reconstructed fragment positions (compare Figure 3.25(c)), so that fragments that hardly move on the view plane could still benefit from UnidirTC. Unfortunately, if the allowed threshold is bigger than zero, smearing can still occur over a couple of frames for fragments that stay within the allowed distance threshold all the time, which is typically the case for a slowly and steadily moving camera.

In our prototype implementation of the proposed PGM improvements, we enable UnidirTC and jitter the auxiliary camera randomly each frame as long as the camera does not translate, which leads to an iterative improvement of the surface reconstruction. As soon as the camera starts moving, we turn UnidirTC off and stop jittering the auxiliary camera. With UnidirTC, the image typically converges to a stable solution within 60 to 120 frames, i.e., within 1 to 2 seconds when rendering at 60Hz. To make the refinement process less obvious, we do not immediately replace fragments in the history buffer, but rather blend the new color with the previous color. While this slows down the convergence of the iterative refinement, blending colors smoothly reduces the otherwise more prominent popping during the frames in which the refinement takes place. Figure 3.26 illustrates how this UnidirTC with smooth blending improves the accuracy of terrain-surface reconstruction over the course of 90 frames in which the camera position does not change.

While we have stated that UnidirTC improves image quality, it actually introduces a very subtle error in the resulting frame as well. Whenever the terrain projects to the screen so that "screen-space valleys" are formed, UnidirTC leads to a slight reconstruction error, while "screen-space peaks" converge to the correct solution. A screen-space peak remains sharp edged even after accumulating several rasterizations, while a screen-space valley gets incorrectly smoothed, similar to how a parabola is constructed by combining multiple, slightly-shifted straight lines, cf. Figure 3.27. By keeping those straight screen-space lines short, we

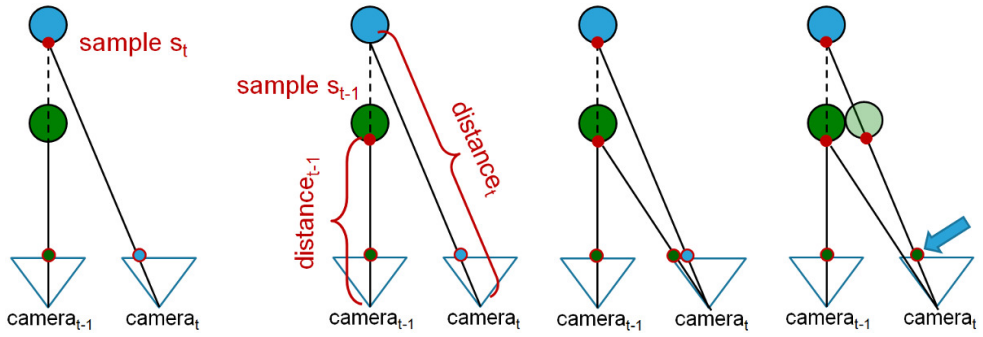
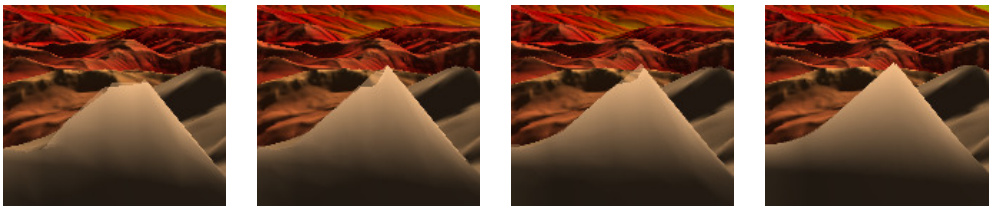
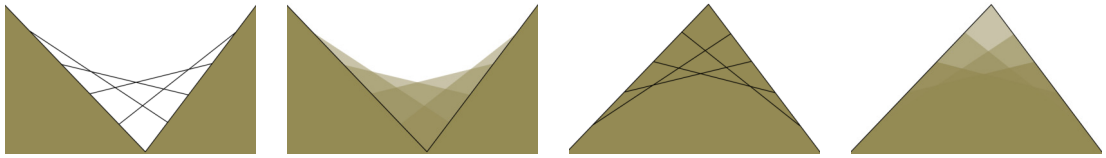


Figure 3.25: Always holding on to the closest fragment in the reprojection cache leads to wrong object positions and incorrect depth sorting when the camera position changes between two frames. Image (a) shows world-space position s_t that is visible in frame t (red dot), and the corresponding fragment on screen (cyan dot). Note that in the previous frame $t-1$, surface position s_t was occluded by a another object. Image (b) illustrates the reprojection of s_t into the previous frame; fetching the depth value in the history buffer for the corresponding fragment yields reconstructed world-space surface position s_{t-1}. The distances to the two surface positions are marked by $distance_t$, and $distance_{t-1}$, respectively. While in frame $t-1$ both world-space positions s_t and s_{t-1} project to the same fragment, this is no longer the case in frame t due to camera movement, as can be seen in image (c). If we were to use the color of the closest reconstructed surface position in frame t, we would color the highlighted fragment in green instead of in cyan, hence incorrectly "smearing" the green object over the cyan one, which is depicted in image (d).



(a) +5 frames (b) +20 frames (c) +50 frames (d) +90 frames

Figure 3.26: Refinement of the terrain surface with unidirectional temporal coherence. In each frame we slightly jitter the sampling camera in a random manner, yielding various different rasterizations of the same terrain area. While non of these rasterizations is perfect, we combine the single outcomes by keeping the closest fragment along each view ray. To lessen popping artifacts during this refinement process, we use color blending instead of resetting fragment colors abruptly, which makes the transition toward the improved terrain-surface approximation softer.



(a) (b) (c) (d)

Figure 3.27: Each line segment depicts one specific terrain approximation, where the approximation results from projecting the persistent grid from the randomly jittered sampling camera. The accumulated intermediate results may not converge to the correct solution when the closest value is preferred in the history buffer all the time. Image (a) shows a case in which a sharp terrain edge degenerates to a smooth parabola (b). Images (c) and (d) show that peaks do not exhibit this issue and converge to the correct solution after several iterations. The opacity of a fragment in images (b) and (d) roughly corresponds to the frequentness at which it is rasterized.

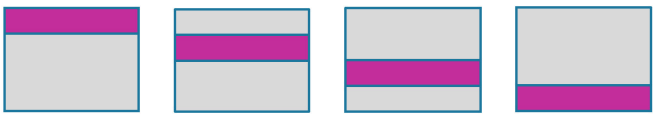
can make the effect all but disappear. We achieve this by rendering a sufficiently finely tessellated persistent-grid, and by performing grid tailoring (see Section 3.3), since grid tailoring "compresses" the projected grid especially in the camera's vicinity. With increasing camera distance this error diminishes without further ado due to perspective foreshortening.

3.6.3 Applying Image-Based Bidirectional Temporal Coherence to PGM

In the previous section we explained why we limit the application of *unidirectional temporal coherence* (UnidirTC) to cases when the camera doesn't move. Obviously, this is an extremely limiting constraint. In this section we show that by applying image-based *bidirectional temporal coherence* (BidirTC) (see Section 2.4.1), we can exploit frame-to-frame coherence even when the camera position changes between frames. Incidentally, we can easily combine the two TC methods by using B-frames from the BidirTC step as the input to UnidirTC. Note that in this section, apart from briefly summarizing (image-based) BidirTC, and how to take advantage of it in our method, we do not present any new contribution. The new contribution will be postponed to the next section in which we present a new B-frame reconstruction technique.

Remember that with image-based bidirectional scene reprojection, we take advantage of rendering the scene geometry only every n^{th} frame, where n is typically equal to 4. We call these "normally rendered" frames *intra frames* (I-frames). The three intermediate frames, so-called *bidirectionally predicted frames* (B-frames), are inferred from their temporally neighboring I-frames by an image-space search which is guided by an optical flow field. The three-dimensional flow field is calculated during I-frame rendering.

BidirTC not only enables us to perform temporal smoothing in each B-frame, we can also increase the quality of I-frames. Increasing I-frame quality is possible because we reconstruct several B-frames from the same past- and future I-frames, so that we can distribute the rendering of the next I-frame over this time interval. A straightforward way of distributing the rendering load over a couple of frames in PGM is to split the persistent grid into horizontal stripes, and render only one such stripe per frame, cf. Figure 3.28. When assuming that only every fourth



(a) t (b) t+0.25 (c) t+0.5 (d) t+0.75

Figure 3.28: Distributing the rendering of an I-frame across four frames by splitting the projected grid into four horizontal strips. The images illustrate the persistent grid in projector space, i.e., before projection onto the ground plane. In each frame, we render only the purple part of the grid. This scheme allows us to render I-frames consisting of $4 \times t$ triangles at the cost of only t triangles per frame.

frame is an I-frame, the distributed rendering means that an I-frame can theoretically contain up to four times as many triangles without increasing the per-frame triangle count. Of course, in reality, the actually observed performance gain almost certainly lies below this theoretical factor-four speedup, since reconstructing B-frames shaves some GPU cycles off the distributed scene rendering, and rendering the I-frames to offscreen buffers increases memory bandwidth requirements. Another limiting factor to be aware of is that by increasing the persistent grid's resolution, we could end up rendering many sub-pixel sized triangles, which puts more stress on the rasterizer, so that the observed slowdown could be unproportionally worse than expected. By keeping all these considerations in mind, we can still render a reasonably increased number of triangles in a distributed manner without taking a performance hit. The raised triangle count naturally increases image quality and lessens the severity of PGM's vertex-swimming artifacts. Another advantage over unidirectional temporal coherence is that whenever parts of the scene are occluded in one of the two I-frames, chances are that the very same surface point is not occluded in the other I-frame, which means that the reprojection-cache hit rate is improved.

We already mentioned that we do not need to preprocess the heightmap in any way in our method. Hence, our method directly supports dynamic heightmap modifications at runtime. When bidirectional temporal coherence is enabled, this operation needs slightly more attention, though. Heightmap editing should only be performed every fourth frame, i.e., when the rendering of a new I-frame is kicked off, due to the distributed rendering. Otherwise, the persistent grid could potentially contain holes temporarily. This becomes apparent by looking at Figure 3.28 – while some of the four horizontal persistent-grid strips would sample the heightmap before terrain editing, the remaining strips would sample the already modified heightfield. If the affected terrain lies at the intersection of two persistent-grid strips, the height values will no longer coincide.

3.6.4 Scattering I-frame samples into the current B-frame

While the image-based reconstruction algorithm proposed by Yang et al. [65] is indeed very effective – after all, B-frames are interpolated without rendering any scene geometry at all – there are cases in which their image-reconstruction algorithm fails to generate artifact-free output images. For example, at depth discontinuities, optical-scene-flow vector magnitudes differ

significantly, which leads to inaccuracies in the inferred B-frames. In the context of landscape visualization, we face depth discontinuities especially at peaks in the data set, so unfortunately, the image reconstruction is imperfect at exactly the same positions at which the PGM method has its issues, too. Yang et al. [65] tackle the image-based B-frame reconstruction problem by elaborating the search algorithm in various ways. The basic idea behind the improvements is not to alter the actual iterations of their B-frame interpolation method, but rather choose better starting positions. For the reader's convenience, we give a brief summary of the search-algorithm improvements described by Yang et al., consult Section 2.4.1 for a more detailed discussion. Their first B-frame-reconstruction-algorithm refinement, which they call "dual initialization", initializes the search positions in one I-frame by offsetting the current fragment position by the velocity vector found in the opposing I-frame, see Equation (2.21). Their second search-algorithm improvement, which they term "latest-frame initialization", initializes the starting position using the offset determined in the previous B-frame, see Equation (2.22). The problem with latest-frame initialization is that we need to store another two 2D vectors ($d_i'^f$ and $d_i'^b$) per fragment, thereby increasing both memory, and bandwidth requirements. In our method, we chose to use the basic greedy search combined with the "dual-initialization" improvement only. For the special case of static terrain geometry at hand, we propose a new B-frame reconstruction algorithm based on scattering I-frame samples into the current B-frame to improve image fidelity at depth discontinuities.

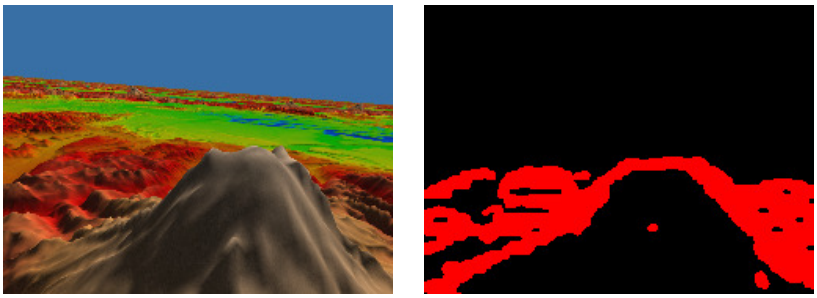
As BidirTC's image-based B-frame interpolation assumes linear motion of surface points between the neighboring I-frames, we can interpolate the camera parameters of the adjacent I-frames to yield an estimated view matrix for the B-frame. The intermediate camera position in the current B-frame is determined by a linear interpolation, while the axes are calculated by a spherical linear interpolation. Once the B-frame's camera coordinate system is established, we can scatter samples from both I-frames into the current B-frame. Since the terrain itself is static, scattering can be performed exactly based on the view- and projection matrices of the respective cameras. The optical flow vectors are not needed for the scattering process.

Scattering of an I-frame sample consists of two steps:

1. The world-space surface position that corresponds to the I-frame sample is reconstructed. This transformation can be calculated exactly, since we have a depth value for each fragment position, from which we can infer the world-space position.

2. Render a point sprite at the world-space position from step 1 with the initial I-frame fragment color, using the current B-frame's transformation matrices.

Reconstructing B-frames through scattering of I-frame samples solely is not practicable for two reasons. Firstly, rendering one point sprite per I-frame fragment hurts performance badly. Secondly, the scattering process just described does not guarantee to cover all pixels in a B-frame. Hence, it makes sense to merely refine an already reconstructed B-frame by only scattering I-frame samples at depth discontinuities. In order to find those image regions in an I-frame, we perform an edge search on a downsampled version of the I-frame. Since we store linear depth values, a standard edge-detection filter kernel can be used for this step. The downsampling not only lessens the cost of the edge-detection operation, it also creates a morphologically dilated



(a) I-frame (color) (b) I-frame samples to scatter

Figure 3.29: Which I-frame samples to scatter. The areas of depth discontinuities that are within a certain distance to the camera are found by an image-space edge-detection filter on the linear depth buffer that is part of an I-frame. For the camera view shown in image (a), the binary classification established by the edge-detection pass tells us which I-frame samples to scatter into the current B-frame (red fragments in image (b)).

edge-detection buffer, so that we automatically include fragments close to depth discontinuities as well. Since the most apparent reconstruction artifacts happen close to the camera, we further limit edge detection to depth discontinuities in the camera's vicinity, which decreases the amount of I-frame samples to scatter even more. Figure 3.29 shows the contents of the edge-detection buffer for a given camera view.

Now that we know which samples to scatter to improve B-frames that were produced by BidirTC's image-based interpolation algorithm, we need to find an efficient way to render the varying amount of point sprites at ever-changing positions. Ideally, we want to render all point sprites in a single draw call. The histogram-pyramid data structure (see Section 2.5) helps us to achieve just that. We generate the histogram pyramid from the edge-detection mask image, where a 1 marks a sample which we want to scatter, and a 0 marks a sample which we don't need to scatter. The single entry at the pyramid's apex, i.e., at the topmost level, tells us how many I-frame samples we need to scatter. Remember that we store the histogram pyramid in a texture, so that each texture mipmap level corresponds to a histogram-pyramid level. Given the index of a specific sample, the histogram-pyramid traversal from the apex toward the pyramid's base level reveals the position of the corresponding sample.

When implementing point scattering in OpenGL, we render a single vertex for each point sprite, so that we can directly use the built-in *gl_VertexID* variable as the input index for the histogram-pyramid traversal. Efficiently issuing the draw call for the point sprites is a bit more involved. In theory we just need to read back the value at the histogram-pyramid's apex, and then use that value to render the right amount of vertices. Unfortunately, this operation introduces a sync point between the CPU and GPU, so that while we transfer just a single value over the bus, performance suffers from the delay that the resulting pipeline stall causes.

We have several possibilities to omit this GPU $\rightarrow$ CPU $\rightarrow$ GPU round trip. For example, on graphics hardware supporting OpenGL 4 and later we can use indirect rendering [54, Chapter 3], which allows us to write the number of point sprites that we wish to scatter into a GPU buffer

object. This buffer object is then used by the indirect draw call, thereby avoiding the GPU-to-CPU readback altogether. On less capable graphics boards we can circumvent the performance penalty associated with the texture read back by simply rendering a fixed amount of point sprites each frame. This sounds worse than it actually is. When the estimated vertex count is too low, we simply don't cover all desired image areas with scattered point sprites. When we render too many vertices, the histogram pyramid maps all excess vertices to the same position as the last valid vertex, so that the unnecessarily rendered surplus point sprites are quickly rejected by the depth test. We have found that a good estimate for the amount of point sprites to scatter is given by $f \times numPixels$, where $f \leqslant 0.1$, and $numPixels$ denotes the number of pixels in an I-frame. For a screen resolution of 1024×768, this means that at most $80{,}000$ I-frame samples are scattered.

At this point we have seen how to perform scattering only for "relevant" I-frame samples, and how to render the point sprites efficiently on a wide range of graphics cards. To lessen the performance impact of the point scattering even further, we scatter I-frame samples only from either the past or the future I-frame, not from both.

When we experimented with improving interpolated B-frames through scattering, we noticed that the technique is most effective when the view camera translates along its x- and y-axes only, and less effective when the camera moves purely forward or backward along its positive or negative z-axis. This means that the amount of points to scatter could be chosen based on camera-movement patterns, even to the point where no point sprites are scattered at all.

3.7 Summary

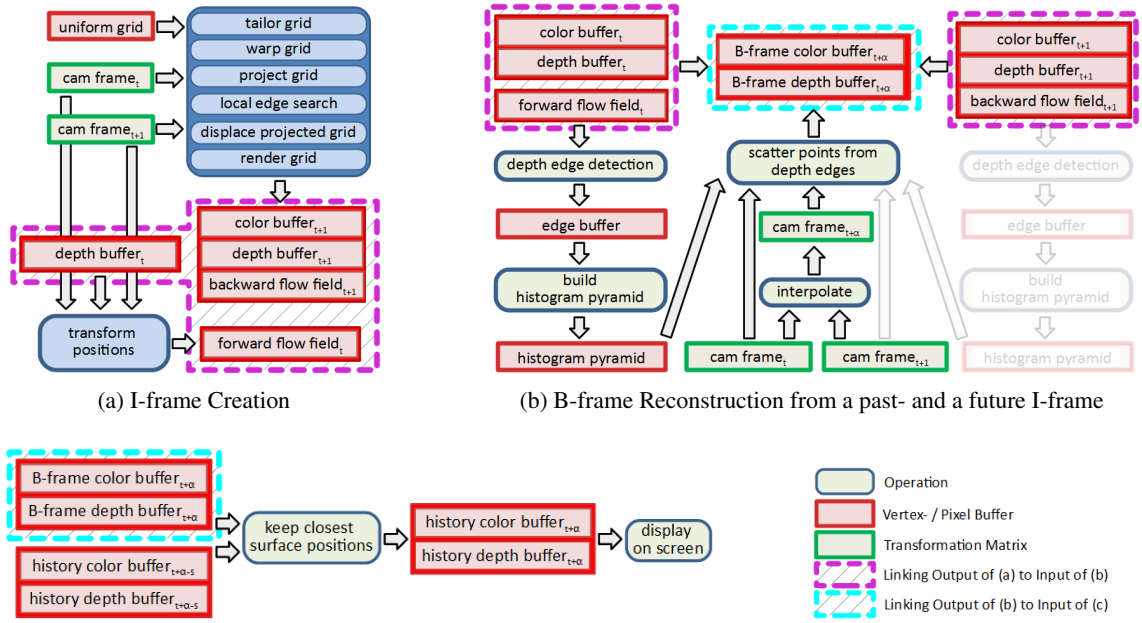
In this chapter we presented all our proposed PGM improvements. The flow charts in Figure 3.30 give a final overview of how our various building blocks interoperate, and how to put everything together.

Figure 3.30(a) shows the rendering of I-frames, which basically consists of the grid-projection phase where our tailoring (see Section 3.3), warping (see Section 3.4) and local-edge-search improvements (see Section 3.5) are applied. Further, Figure 3.30 (a) illustrates the creation of the forward and backward velocity-vector fields for the subsequent image-based bidirectional scene reprojection.

Figure 3.30(b) depicts how B-frames are interpolated from two adjacent I-frames. After a B-frame is reconstructed using an image-space search algorithm (see Section 2.4.1), the B-frame can be further refined by splatting point sprites from one of the adjacent I-frames into the current B-frame. We assign one point sprite to each pixel that lies on, or close to, a depth discontinuity in the adjacent I-frame, since especially those areas tend to be reconstructed inaccurately by the image-space search algorithm. We scatter point sprites from only one I-frame to reduce the impact that scattering has on the overall performance. Each point sprite uses the color and surface position of its corresponding I-frame pixel, and is transformed into the B-frame using the interpolated camera coordinate frame $camframe_{t+\alpha}$. The IBidirTC method assumes linear motion of surface points relative to the moving coordinate frame [65] to start with, so that we can simply interpolate the camera frames from times t and $t+1$ linearly by factor α (see Section 3.6.4 for details).

Figures 3.30 (c) illustrates how UnidirTC and image-based BidirTC can be combined. Each reconstructed B-frame represents the scene as currently seen through the camera, hence each B-frame is used as the input image to the UnidirTC technique. The history buffer that is generated by the UnidirTC method holds the image to be finally shown on screen.

Figure 3.30 also reveals that we can "mix and match" our improvements to a certain extent. This modularity enables us to trade image quality for performance by skipping some of the improvements in our pipeline. For example, we could skip UnidirTC altogether. In that case the resulting B-frame in Figure 3.30(b) would be displayed directly, and all steps shown in Figure 3.30(c) would be skipped. Another option is to not perform image-based BidirTC at all, but still employ UnidirTC. In that case the rasterized persistent grid from Figure 3.30(a) would be the input to the UnidirTC step in Figure 3.30(c), and the creation of forward- and backward motion-vector fields as well as all calculations shown in Figure 3.30(b) could be skipped. The persistent-grid rendering phase depicted in Figure 3.30(a) offers further modification opportunities by skipping any of our tailoring, warping, and local-edge-search improvements. And finally, in Figure 3.30(b), point scattering from I-frames is optional, so that the lower section of the figure (depth-edge detection, histogram-pyramid construction, ...) can be skipped.



(a) I-frame Creation

(b) B-frame Reconstruction from a past- and a future I-frame

(c) Unidirectional Temporal Coherence

(d) Symbols

Figure 3.30: Overview of our method. Image (a) illustrates the rendering of I-frames using a refined persistent grid (the tailoring-, warping-, and local-edge-search improvements are applied). I-frame rendering can be distributed over several B-frames, cf. Figure 2.37. The backward flow field is rendered alongside the color- and depth buffer of the new I-frame, while the forward flow field is calculated from the depth buffer of the previous I-frame and the camera frames at times t and $t+1$, i.e., the camera transformation matrices for the current and the next I-frame. As soon as I-frames for time t and $t+1$ have been calculated, B-frames $t+\alpha$, with $\alpha \in [0;1[$ can be interpolated (b). Resulting B-frames can be further refined by scattering samples from either the past or the future I-frame into the B-frame (the given example uses the past I-frame). We only scatter I-frame points that lie on and around depth edges into the B-frame, using a new camera frame, which is created by interpolating the camera frames that are linked to the past and future I-frame with interpolation factor α. Together, images (a) and (b) summarize the BidirTC method. Image (c) shows the accumulation of B-frames in a history buffer. The subscript s stands for the duration between two B-frames, and since we are using four intermediate frames between two I-frames, $s = 0.25$. The updated history buffer is finally displayed on screen. Image (d) explains the meanings of the shapes and colors used in the figure.

CHAPTER 4

Implementation and Results

We implemented the proposed improved PGM algorithm, as well as several of the terrain-rendering algorithms that we examined in Section 2.3, in a small application, which we call *Framework for Terrain-Rendering Algorithm Comparison* (FTRAC).

FTRAC is a minimalist collection of methods to ease proof-of-concept implementations of landscape-visualization techniques. FTRAC essentially forms a light-weight abstraction layer over OpenGL, i.e., it manages the OpenGL state and exposes shaders, vertex- and index buffers, textures and framebuffer objects. Besides the graphics-API abstraction, FTRAC takes care of window creation, user input and file I/O. The framework is written in C++ and interfaces the GPU through the OpenGL 3.3 API, except for the brute-force hardware-tessellation terrain-rendering algorithm (Section 2.3.6), which requires at least OpenGL 4.

Having all terrain-rendering algorithm implementations run in the same environment enables more meaningful performance comparisons. From all the terrain-rendering algorithms that we discussed previously, we decided to compare our method only to the basic PGM algorithm and the Frostbite™ terrain-rendering technique (see Section 2.3.5) because we found that in TRAC the Frostbite™ terrain-rendering technique is a good representative for all rendering algorithms that are not based on PGM, featuring both high-quality results and a high frame rate.

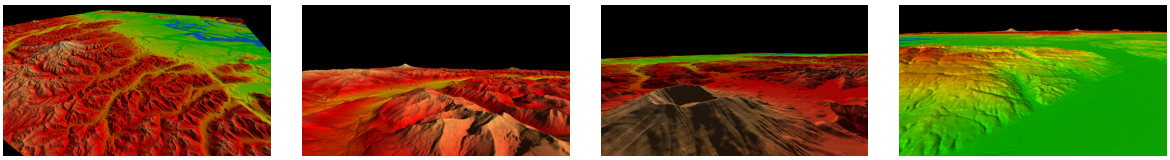
In the following, we use abbreviations to refer to the analyzed algorithms. The abbreviations are listed in Table 4.1. In particular, to discern the various variations of our modular method, we use the subscript T for tailoring (see Section 3.3), W for importance-driven grid warping (see Section 3.4), E for the local-edge-search improvement (see Section 3.5), B for image-based BidirTC (see Section 3.6.3), and U for UnidirTC (see Section 3.6.2).

While an effort was made to implement each algorithm as presented in the respective papers, features that are not investigated in this thesis were skipped, such as out-of-core terrain-data management. For example, in our framework, we fall back to using the same heightmap texture all over again by addressing it toroidally, which enables us to render an arbitrarily large terrain surface.

For the evaluation of the various terrain-rendering algorithms, we used the Puget Sound dataset at a resolution of $8K \times 8K$, a downsampled version of the $16K \times 16K$ dataset that can

PGM	Original Persistent Grid Mapping [32] (see Section 3.2).
$PGM_{<improvements>}$	Our gradual improvements to *PGM* are denoted by the following subscripts:
T	Tailoring of persistent grid (see Section 3.3)
W	Importance-driven persistent-grid warping(see Section 3.4)
E	Local-edge-search (see Section 3.5)
B	Image-based BidirTC (see Section 3.6.3)
U	UnidirTC (see Section 3.6.2)
Frostbite	The terrain-rendering method used in the Frostbite™ engine [12, 64] (see Section 2.3.5).

Table 4.1: Abbreviations used for rendering methods in the following analysis.



(a) (b) (c) (d)

Figure 4.1: Several frames along the camera path that is used for the following benchmarks.

be found in the Large Geometric Models Archive of the Georgia Institue of Technology [60]. We selected the Puget Sound dataset because it is the de-facto standard in publications about terrain-rendering algorithms, and we did not want to break with this tradition. Furthermore, using a common terrain dataset gives readers who are interested in the subject the possibility to directly compare screenshots from our method to other published results. In our implementation, the colormap is an RGB image with 8 bits per channel (GL_RGB8), and the heightmap is stored in a 16bit single-channel integer format (GL_R16).

Since we focused mainly on the geometric-LOD handling, FTRAC does not support any streaming of texture data at runtime, and because the colormap from the Puget Sound data set is prelit, FTRAC does not apply any lighting model when rendering the terrain surface.

In the remainder of this chapter, we will analyze the memory requirements, the performance and the image quality of our method. For the frame-rate and image-quality measurements, we captured and analyzed all frames of an animation in which the camera is moved along a path that starts at a high altitude above the terrain, looking down at the entire Puget Sound area. As time progresses, the camera descends to the terrain surface and visits Mt. Rainier, Mt. Adams and finally Mt. St. Helens. The entire animation takes approximately one minute, and ends with the camera hovering over a valley at a low altitude, framing the three mountains in the distance. Figure 4.1 shows several screenshots from the animation.

Texture	Count	Resolution	OpenGL Texture format
Noise	1	128×128	GL_RGBA8
Tailoring Redirection	1	32×32	GL_R8
Importance	1	$\lceil \frac{grid_w}{32} \rceil \times \lceil \frac{grid_h}{4} \rceil$	GL_R16F
Integrated Importance	1	$\lceil \frac{grid_w}{32} \rceil \times \lceil \frac{grid_h}{4} \rceil$	GL_R16F
Warping Redirection	1	$\lceil \frac{grid_w}{32} \rceil \times \lceil \frac{grid_h}{4} \rceil$	GL_RG16
UnidirTC (color)	2	$screen_w \times screen_h$	GL_RGB8
UnidirTC (depth)	2	$screen_w \times screen_h$	GL_R32F
IBidirTC (color)	3	$screen_w \times screen_h$	GL_RGB8
IBidirTC (depth)	3	$screen_w \times screen_h$	GL_DEPTH_COMPONENT32F
IBidirTC (3D optical flow)	3	$screen_w \times screen_h$	GL_RGB16F

Table 4.2: Memory footprints of the textures used in our algorithm. *Tailoring Redirection* denotes the redirection texture which realizes the tailoring of the projected grid to the main camera's view frustum (see Section 3.3), *Importance*, *Integrated Importance*, and *Warping Redirection* are responsible for the importance-driven persistent-grid warping step (see Section 3.4). The table entries *UnidirTC (color)* and *UnidirTC (depth)* refer to the textures used in the unidirectional-temporal-coherence part of our method; the *Noise* texture is utilized for randomly refreshing the contents of the history buffer so that accumulated results don't get blurry due to numerical diffusion (see Section 3.6.1). *IBidirTC (color)*, *IBidirTC (depth)*, and *IBidirTC (3D optical flow)* denote the textures that are needed for the image-based bidirectional-temporal-coherence step (see Section 2.4.1 and Section 3.6.3). The variables $grid_w$ and $grid_h$ denote the resolution of the persistent grid (in vertices), and $screen_w$ and $screen_h$ represent the screen resolution (in pixels).

4.1 Memory Requirements

In this section we break down the memory footprint of our algorithm. At this point we want to once again emphasize that apart from the terrain data itself, our method does not require any additional terrain-associated metadata.

The additional memory requirements of our algorithm stem from the involved textures, while the space needed for the vertex data, such as the persistent grid itself, is negligible. For example, storing a persistent grid made up of 240×1050 vertices, i.e., a grid with approximately 500K triangles, amounts to a mere 252KB of video memory. Further, with *image-based bidirectional temporal coherence* (IBidirTC), the persistent grid is split into four slices, and only one slice is rendered in each frame in a distributed manner, which means that we get away with storing only one slice, cf. Figure 3.28, thereby quartering the memory requirements for the persistent grid. The space consumed by textures is considerably larger and depends on both the screen resolution and the resolution of the persistent grid. Table 4.2 lists the required textures along with their dimensions and OpenGL texture formats.

Table 4.3 is created by inserting several common screen resolutions into the equations from

Screen Resolution	without TC	UnidirTC	IBidirTC	UnidirTC & IBidirTC
1024×768	126.5 KB	10.5 MB	29.3 MB	39.8 MB
1600×900		19.3 MB	53.6 MB	72.9 MB
1920×1080		27.7 MB	77.2 MB	104.9 MB
3840×2160		110.8 MB	308.6 MB	419.4 MB

Table 4.3: GPU memory footprint of our algorithm at different screen resolutions using various temporal-coherence configurations for a persistent grid consisting of 500K triangles. The table reveals that at 1080p (1920×1080) and 2160p/4K (3840×2160), TC becomes quite memory hungry, IBidirTC in particular.

Table 4.2, and hence Table 4.3 gives specific memory requirements for the textures when using a persistent grid containing roughly 500K triangles. Table 4.3 reveals that rendering at high resolutions such as 1080p (1920×1080), or even 2160p (3840×2160) takes up a significant amount of texture memory when employing both image-based bidirectional and unidirectional temporal coherence. As stated in the beginning of Section 3, our method permits skipping several of the four improvements to balance memory requirements, performance and image quality. In the case of rendering at a resolution of 4K and beyond, an application may decide to skip the IBidirTC improvement (see Section 3.6.3) in order to cut down on the required texture memory.

4.2 Performance

For the frame-rate measurements we used an x64 Windows 10 PC with a 3.0GHz Intel Core 2 Duo E8400 CPU, 3GB of main memory and an NVIDIA GeForce GTS 450 with 1GB of video memory, which is accessed through a 128-bit memory interface. The numbers presented in this section were acquired by instrumenting the source code of our rendering framework. We measure the elapsed CPU time with a performance counter that has a time resolution of approximately $1\mu s$ [9]. We use OpenGL timer queries [1] to measure the GPU time that it takes to fully complete a set of OpenGL commands. OpenGL timer queries are performed asynchronously, so that the GPU benchmark does not stall the graphics pipeline, and hence introduces little to no overhead.

Note that even when using the same computer and implementing each algorithm in the same rendering framework, it is still a challenge to do each algorithm full justice and arrive at a fair performance assessment for all of them. For example, PGM guarantees that a constant number of triangles is rendered in each frame, but for algorithms that organize terrain tiles into a quadtree for view-frustum culling, the triangle count will vary depending on which tiles are visible at which detail level, while the geometry-clipmaps method (see Section 2.3.3) even requires manual adjustment of the clipmap size to control the projected triangle size on screen. For this reason, we will compare our method to the basic PGM and the Frostbite™ methods only. As mentioned earlier, we chose *Frostbite* because we found that it is representative of the other non-PGM-based techniques. Further, by not including each and every terrain-rendering algo-

Technique	Min Frame Duration	Avg Frame Duration	Max Frame Duration
PGM	1.130	1.430	2.297
PGM_{TE}	1.207	1.449	2.245
PGM_{TWE}	1.214	1.427	2.293
PGM_{TWEB}	5.854	5.997	6.198
PGM_{TWEBU}	7.127	7.321	7.491
Frostbite	1.687	1.911	2.150

Table 4.4: Minimum, average and maximum frame duration (given in ms) measured during a one-minute long flight over Puget Sound, rendered at a resolution of 1600×900 pixels.

rithm that we implemented in TRAC in our measurements, we can focus on several variations of our method instead, arriving at a less cluttered presentation of results.

Table 4.4 depicts the minimum, average, and maximum frame durations in ms, measured for the camera-path animation when rendered to a 1600×900 window. As can be seen, PGM_{TWE}, i.e., *PGM* augmented by our tailoring, warping, and local-edge-search improvements, is only marginally slower than the basic PGM method, and still, as we will see in the following image-quality evaluation in Table 4.6, PGM_{TWE} generates higher-quality images than *PGM*. On average PGM_{TWE} is roughly 30% faster than *Frostbite* while achieving almost the same image quality. PGM_{TWEB} and PGM_{TWEBU} fall behind the other measured techniques quite a bit performance wise. Again, Table 4.6 will reveal that the two variations of our method, PGM_{TWEB} and PGM_{TWEBU}, reach the highest image quality of all the listed algorithms, justifying the slower frame rate to a certain degree.

Figure 4.2 shows the frame-time graph as benchmarked during the camera animation. The graph shows almost constant frame durations for all algorithms, except for the first few frames, during which the algorithms *PGM*, PGM_{TE}, and PGM_{TWE} render the landscape slower than in the rest of the benchmark. The "slow" frames correspond to the time in which the camera is high above the ground, looking almost straight down onto the terrain surface. We suspect that this slowdown is caused by the rasterizer. Remember that in our prototype, we purposely render a non-square persistent grid, which has more triangles along the vertical direction than along the horizontal direction (as seen from the sampling camera). This is done in order to fight the strong stretching of the persistent-grid mesh along the viewing direction from the start – even when rendering the terrain with the basic *PGM* method. However, when placing the camera high above the terrain and pointing it straight down onto the ground, this grid stretching is absent, so that the rasterizer has to deal with lots of thin, long triangles, which explains the slight performance degradation at the beginning of the benchmark. In later frames, the camera is mainly pointing toward the horizon, so that the perspective mapping of the grid onto the ground plane produces thicker, more "well-formed" triangles to be rasterized. We probably can't observe this slowdown in our benchmark when displaying the landscape with the PGM_{TWEB} and PGM_{TWEBU} algorithms, since the rasterization overlaps with other calculations that take even longer, so that the the duration variations of the rasterizer stage are effectively hidden. On our test system,

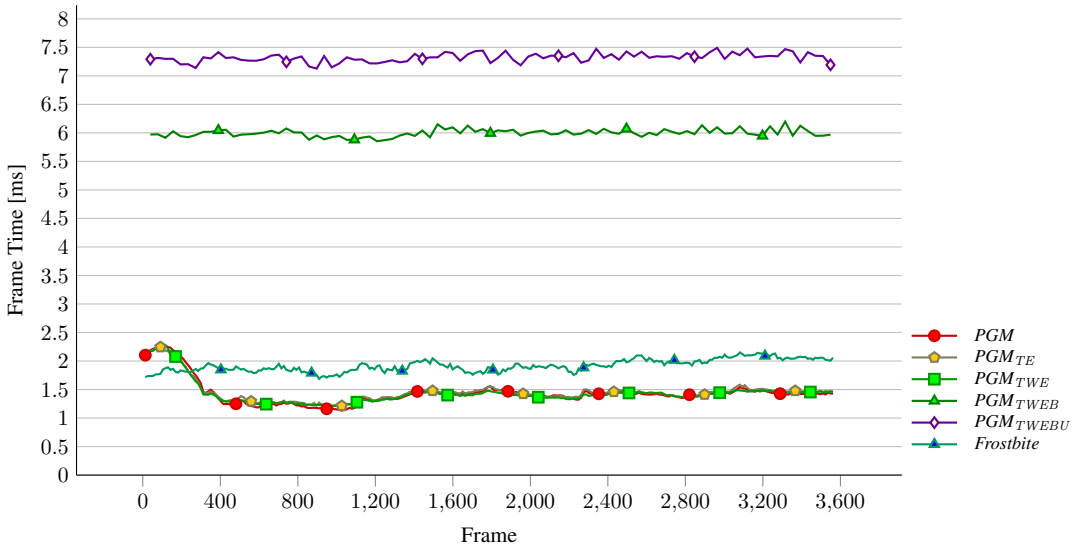


Figure 4.2: Frame durations measured during a one-minute long flight over Puget Sound, rendered at a resolution of 1600×900 pixels. See Table 4.1 for the list of abbreviations.

the frame durations for PGM_{TWEB} and PGM_{TWEBU} are primarily governed by the bandwidth requirements that come hand in hand with taking advantage of temporal coherence.

Figure 4.3 compares the performance of rendering a persistent grid containing either 500K or 2M triangles without *image-based bidirectional temporal coherence* (IBidirTC), and rendering 2M triangles in a distributed manner at only 500K triangles per frame with IBidirTC. While we only render approximately 500K triangles per frame with IBidirTC in our benchmark setup, the displayed B-frames are reconstructed from I-frames that contain 2M triangles, which justifies the interpretation that in fact 2M triangles are rendered per frame. When we apply UnidirTC, we could even argue that the accumulation of jittered terrain rasterizations in the history buffer increases the resolution of the persistent grid even more, eventually resulting in an almost pixel-perfect approximation, but we don't pursue this thought any further in this section. At a 1024×768 screen resolution, IBidirTC is indeed faster than the brute-force rendering of 2M triangles, while the perceived quality is equivalent to the brute-force method. When the screen resolution is increased, the speed of IBidirTC begins to drop to the level of the 2M-brute-force approach on our test system. This is due to the overhead of maintaining screen-sized I-frames and additionally calculating the three-dimensional flow fields, and the cost of reconstructing B-frames. Of course, with increased screen resolution these render-to-texture passes and associated texture fetches increase the algorithm's memory-bandwidth requirements, and also put more stress on the fragment processors. Therefore, increasing the screen size can become a bottleneck in our method. On our test system this especially shows, since the installed GPU has a rather narrow 128-bit memory interface. At this point IBidirTC still has the advantage of temporal smoothing that goes hand in hand with B-frame interpolation from their two adjacent I-frames, and even though the basic PGM method clearly profits from the increased triangle count, heightmap sampling is still insufficient at farther distances. On the other hand, the

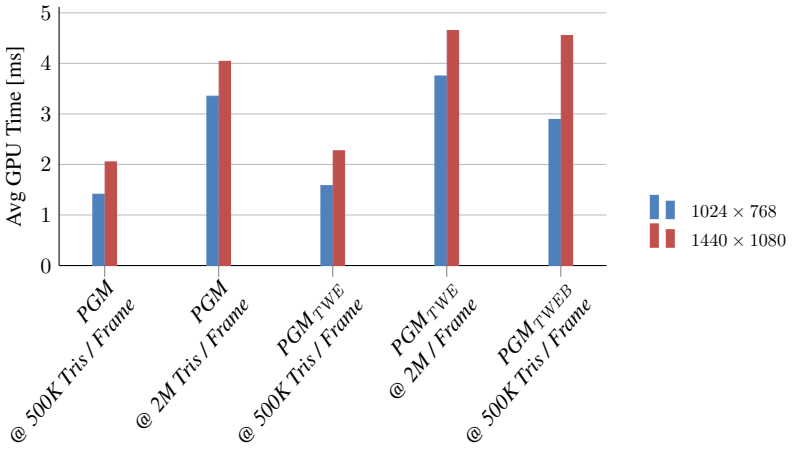


Figure 4.3: Comparison of average per-frame GPU time spent brute-force rendering 2M triangles versus the "virtual" 2M triangles achieved by rendering only 500K triangles per frame with IBidirTC at two different screen resolutions. See Table 4.1 for the list of abbreviations.

brute-force approach does not suffer from any B-frame reconstruction artifacts, does not require any additional texture memory for I-frames and B-frames, and exhibits no user-input lag. Another consideration to keep in mind is that if more complex terrain-surface shading is involved, IBidirTC will start showing its strength. By reprojecting fragments into the current B-frame, costly fragment-shader calculations are skipped, leading to a higher frame rate. As we basically only perform one texture lookup per fragment with no shading/lighting calculations at all, we do not observe this performance boost at all.

Our method leaves the CPU almost idle. A complete frame takes approximately 0.06ms of CPU time only, since apart from issuing draw calls and the obligatory OpenGL state management, the CPU merely calculates the 2D convex hull in the tailoring step (see Section 3.3), which involves just a few vertices.

Most work is done on the GPU. Table 4.5 breaks down the relative GPU frame times that major steps in our algorithm take to complete at two different screen resolutions when all of our proposed improvements are enabled. The table reveals that the tailoring and the warping step are essentially free, and still these two operations contribute noticeably to an improved terrain-surface approximation. As expected, a large part of the GPU frame time is spent actually rendering the persistent grid. This step includes applying the redirection textures to perform tailoring and warping of the grid vertices, as well as the local-terrain-edge search. With larger screen resolutions, IBidirTC and UnidirTC make up an increasing fraction of the GPU frame time.

4.3 Image Quality

In this section we analyze the image quality of *PGM*, *Frostbite*, and our method by measuring how much the generated output images differ from output images that are created by rendering

	1024×768	1440×1080
Generate redirection texture for tailoring	0.5%	0.3%
Generate redirection texture for warping	2.5%	1.5%
Render tailored and warped persistent grid with local edge search (500K triangles)	40%	35.4%
Apply IBidirTC with point scattering	37.6%	40.0%
Apply UnidirTC	19.4%	22.8%

Table 4.5: Relative GPU frame times of our improvements.

a high-resolution static terrain mesh without any LOD management.

We measure image quality by calculating the *Peak Signal-To-Noise Ratio* (PSNR) [7]) between each chosen method and the reference solution for each frame of the camera-path animation. PSNR is a frequently used measure to compare two signals. For two RGB color images I_1 and I_2 having dimensions $w \times h$ with B bits per color channel, the PSNR is calculated according to Equation (4.1), where MSE is the Mean Square Error, and MAX_I is the maximum value that can be stored per color channel [6]. The term $I_1(x, y).c$ denotes accessing color channel c (with c being either the red, green, or blue color channel) of image I_1 at position (x, y). The more similar two images are, the higher their PSNR will be. The PSNR is expressed in *decibels* (dB).

$$MAX_I = 2^B - 1 \tag{4.1a}$$

$$MSE = \frac{1}{3\,w\,h} \sum_{y=0}^{y=h-1} \sum_{x=0}^{x=w-1} \sum_{c \in \{r,g,b\}} (I_1(x, y).c - I_2(x, y).c)^2 \tag{4.1b}$$

$$PSNR = 10\,log_{10}(\frac{MAX_I^2}{MSE}) \tag{4.1c}$$

For the measurements, we render a persistent grid that consists of 240×1050 vertices when *image-based bidirectional temporal coherence* (IBidirTC) is not enabled, and 480×2100 vertices when IBidirTC is enabled – remember that in the latter case this means that we still render only 240×1050 vertices per frame. Hence, this translates to approximately 500K triangles being processed per frame in both cases. As noted previously, it is beneficial to use non-square persistent-grid dimensions, since a higher resolution along the grid's y axis battles heightmap undersampling in the distance from the get-go, even without our grid-warping improvement.

Table 4.6 lists the PSNR values that we obtained from our measurements. We immediately see that PGM delivers the worst image quality across the board. As we saw in the previous discussion about performance, PGM_{TWE} is virtually as fast as *PGM*, and yet PGM_{TWE} generates higher-quality images. In fact, in some cases, the minimum PSNR is even higher for PGM_{TWE} than for PGM_{TWEB}. This is due to B-frame reconstruction artifacts in IBidirTC. The higher average PSNR of PGM_{TWEB} is proof for these reconstruction artifacts to occur seldomly, though. Further, the maximum PSNR of PGM_{TWEB} is higher than the PSNR of PGM_{TWE}. The PSNR entries for PGM_{TWEBU} are equal to the PSNR values for PGM_{TWEB}, since the camera moves

Technique	Minimum PSNR[dB]	Average PSNR[dB]	Maximum PSNR[dB]
PGM	25.453	27.689	31.647
PGM_{TE}	28.146	29.723	31.992
PGM_{TWE}	29.220	31.246	34.745
PGM_{TWEB}	26.326	36.407	41.008
PGM_{TWEBU}	26.326	36.407	41.008
Frostbite	31.371	34.042	37.441

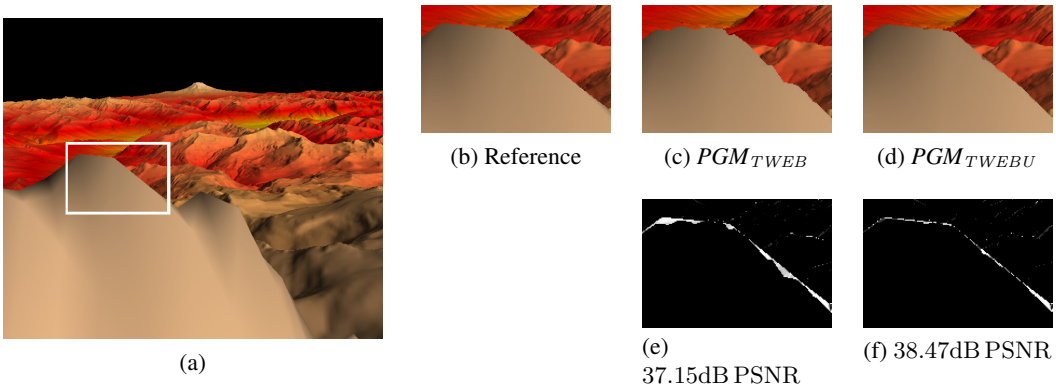
Table 4.6: Minimum, average and maximum PSNR measured during a one-minute long flight over Puget Sound for various algorithms when rendered at a resolution of 1600×900 pixels. Note that in this specific benchmark, PGM_{TWEBU} achieves the same image quality as PGM_{TWEB} since the camera position changes all the time, which disables any image-quality improvements that can otherwise be achieved when applying UnidirTC.

in each frame of the camera-path animation, which means that UnidirTC is disabled all the time, and therefore PGM_{TWEBU} "degenerates" to PGM_{TWEB} in this case. Finally, *Frostbite* positions itself somewhere between PGM_{TWE} and PGM_{TWEB}, with an advantage in the minimum PSNR column over both methods.

In our example camera animation we saw that using PGM_{TWEBU} did not yield a higher image quality than PGM_{TWEB}, since UnidirTC was effectively disabled entirely due to the constantly changing camera position. When the camera is translated between frames, disabling UnidirTC is necessary to avoid artifacts when reconstructing the final image from a combination of the current frame and the history buffer as noted in Section 3.6.2. We prepared another measurement setup to show that PGM_{TWEBU} can very well boost image quality quite a bit when the camera stops translating for even just a very short duration. For this experiment we placed the camera in front of a sharp, thin peak, which is challenging to reconstruct faithfully with PGM-based algorithms. PGM_{TWEBU} manages to generate an output image that differs less from the reference solution than PGM_{TWEB} after camera translation has stopped for approximately 30 to 60 frames. In this specific setup, PGM_{TWEBU} achieves 38.472dB PSNR, while PGM_{TWEB} accomplishes only 37.147dB PSNR. Figure 4.4 illustrates the measured image-quality difference visually. The figure clearly shows that PGM_{TWEBU} is closer to the reference solution than PGM_{TWEB}.

Figures 4.5 through 4.7 show the image-quality difference of the discussed methods compared to the reference solution (along with their PSNR) for several frames of the camera animation. The brighter a pixel in the comparison images, the bigger the difference to the reference image. Note that the comparisons are purely based on color – the pixel depth is ignored.

Figure 4.5 compares the differences of a screen region that a terrain peak is rendered to, and lists the resulting PSNR values. We can see that as more of our improvements are applied to the basic *PGM* algorithm, the image quality improves steadily. The image-quality comparison to Frostbite is especially interesting: while PGM_{TWE} falls short of *Frostbite*, PGM_{TWEB} yields better results than *Frostbite*.



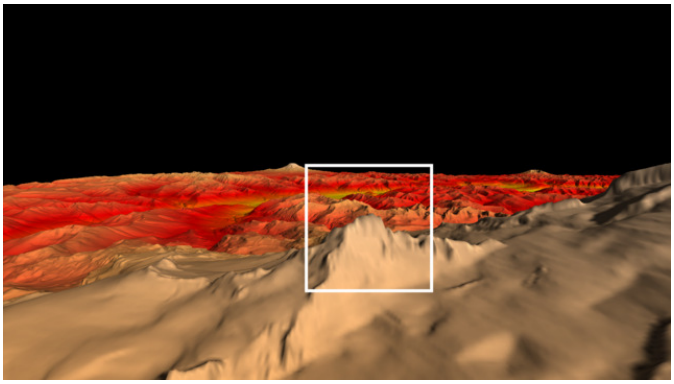
(a) (b) Reference (c) PGM_{TWEB} (d) PGM_{TWEBU} (e) 37.15dB PSNR (f) 38.47dB PSNR

Figure 4.4: Comparing the image quality of PGM_{TWEB} and PGM_{TWEBU}. The area of interest is marked with a white rectangle (a). Close up of the reference solution (b), PGM_{TWEB} (c), and PGM_{TWEBU} (d). Image (e) shows the contrast-enhanced difference between images (b) and (c), i.e., the difference between the reference solution and PGM_{TWEB}. Image (f) shows the contrast-enhanced difference between images (b) and (d), i.e., the difference between the reference solution and PGM_{TWEBU} after approximately 30 to 60 frames of no camera translation.

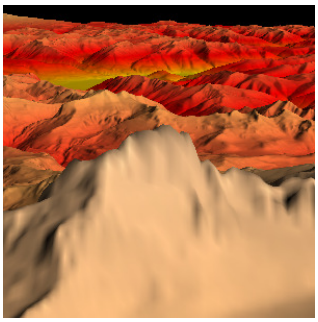
Figure 4.6 depicts a situation where PGM_{TWEB} unexpectedly exhibits slightly lower image quality than PGM_{TE}, PGM_{TWE} and *Frostbite*. Interestingly, in this specific situation PGM_{TE} even beats PGM_{TWE} quality wise, which is due to the fact that PGM_{TWE} redistributes persistent-grid vertices toward the horizon, thereby slightly decreasing the terrain-surface triangle density in the view camera's vicinity, consequently giving PGM_{TE} a moderate image-quality advantage in this case.

Apart from the just described image-quality degradation close to the camera due to the redistribution of vertices toward the horizon, the image-quality degradation of PGM_{TWEB} in this frame is caused by two additional factors, which both affect the image-based B-frame interpolation in a negative way. Firstly, the camera is panning fast in this phase of the animation, which leads to velocity vectors with large magnitudes, and secondly, the currently shown frame number 1126 is a B-frame that has maximum temporal distance to its neighboring past- and future I-frames, since $1126 \bmod 4 = 2$. Hence, the image-based B-frame reconstruction not only has to deal with large distances to find corresponding positions in the neighboring I-frames, but also large parts of a B-frame have associated positions in only one I-frame, not in both, thus losing the luxury of choosing the better of two reconstructed pixels – one from the past and one from the future I-frame – that the bidirectional-temporal-coherence algorithm usually offers. In contrast, a frame F with $F \bmod 4 = 0$ is actually an I-frame and hence not prone to this kind of B-frame reconstruction error.

We can, however, put the just described situation a bit into perspective. The fast movement of pixels in screen space essentially means that everything on screen moves very fast, so that slight reconstruction errors are hard to spot. Furthermore, post-processing effects like motion blur are very common in real-life application, which further assists in hiding these artifacts under exactly these circumstances.

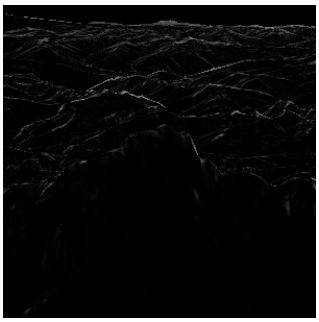


(a) Frame 1104 of Animation, White Rectangle marks Area of Interest



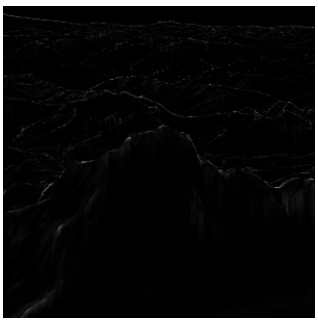
(b) Reference

(c) *PGM*, 20.34db PSNR



(d) PGM_{TE}, 23.86db PSNR

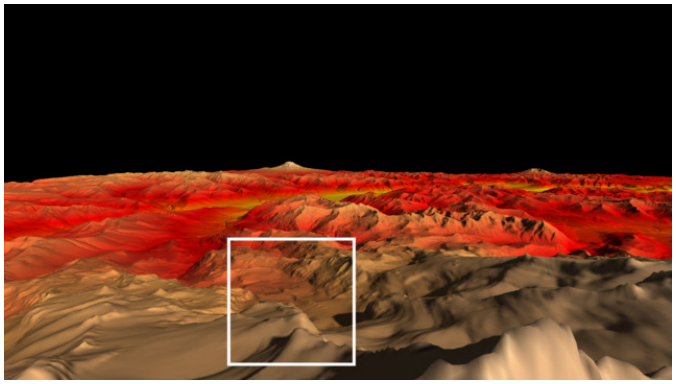
(e) PGM_{TWE}, 24.96db PSNR



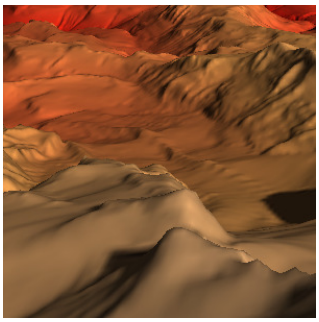
(f) PGM_{TWEB}, 30.24db PSNR

(g) *Frostbite*, 28.01db PSNR

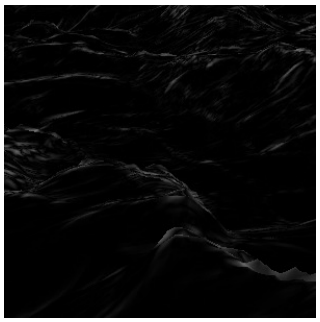
Figure 4.5: Differences of various render methods to the reference solution in frame 1104 of the camera animation. Image (a) shows the full camera frame (an I-frame in PGM_{TWEB}, so that there is no need to interpolate to arrive at a B-frame – this I-frame is the B-frame). The detail region that is analyzed is marked with a white rectangle and shown in isolation in image (b). Images (c) to (g) show the difference between each algorithm and the reference solution. The resulting PSNR values are given in the image caption.



(a) Frame 1126 of Animation, White Rectangle marks Area of Interest

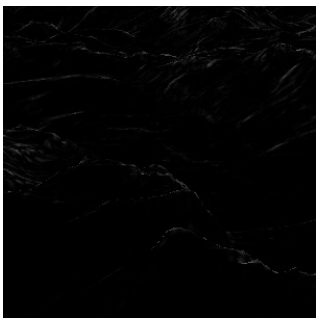


(b) Reference



(c) *PGM*, 26.25db PSNR

(d) PGM_{TE}, 32.60db PSNR



(e) PGM_{TWE}, 31.38db PSNR

(f) PGM_{TWEB}, 30.53db PSNR

(g) *Frostbite*, 34.32db PSNR

Figure 4.6: Differences of various render methods to the reference solution in frame 1126 of the camera animation. Image (a) shows the full camera frame. The detail region that is analyzed is marked with a white rectangle and shown in isolation in image (b). Images (c) to (g) show the difference between each algorithm and the reference solution. The resulting PSNR values are given in the image caption.

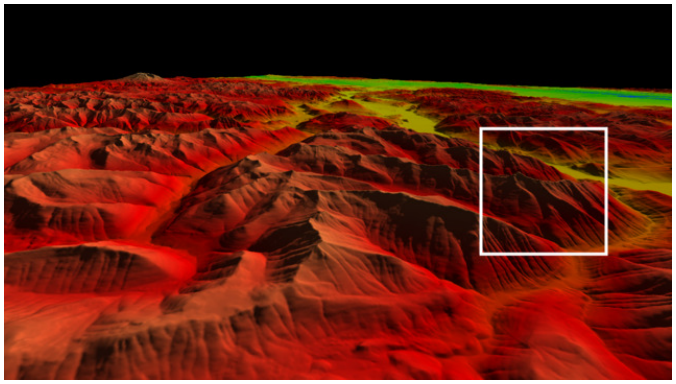
Figure 4.7 concludes the PSNR analysis with an example in which PGM_{TWEB} yields superior image quality, even when compared to *Frostbite*.

In Figures 4.8 to 4.11 we explore terrain-surface triangulations at ever-increasing distances from the sampling camera when rendering the landscape with basic PGM and several variations of our method, namely PGM_{TE}, PGM_{TWE}, and PGM_{TWEB}. Further, the figures also contain the surface triangulations as rendered by the reference solution and by *Frostbite*. For each method, we render a wireframe overlay on top of the terrain to reveal the underlying triangle mesh.

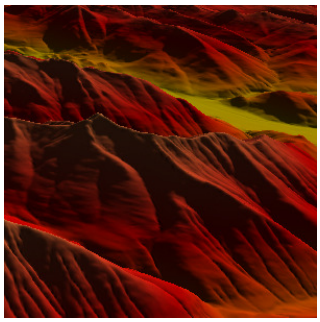
The sampling camera itself stays fixed at its initial position, while a meta view camera is used to inspect various regions of the resulting landscape mesh. This meta view camera does not influence terrain tessellation at all, i.e, the meta view camera merely serves illustrative purposes. Obviously, for the reference solution there is no notion of a sampling camera, and the terrain is rendered equally detailed everywhere. For *Frostbite*, the static sampling camera acts as a fixed position from which to build the quadtree (instead of updating this position to the current camera position in each frame), so that the triangle count of tiles decreases as the distance of a tile to the sampling-camera position increases.

Note that we do not include results for PGM_{TWEBU} in Figures 4.8 to 4.11, since each surface triangulation in PGM_{TWEBU} has the same sampling density as the tessellation created by PGM_{TWEB} – PGM_{TWEBU} simply combines slightly jittered terrain-surface approximations in screen space to finally arrive at a more faithful landscape reconstruction. If we were to take screenshots of the accumulated results of frames rendered with PGM_{TWEBU} while enabling the wireframe overlay, we would soon end up with wireframe lines covering the whole terrain surface, which for the intended illustration, would not make any sense at all.

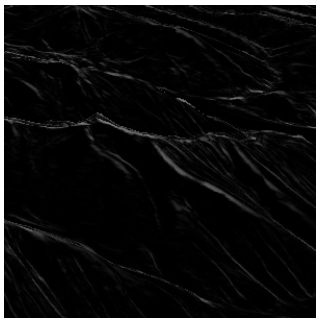
After this short introduction to the triangle-density comparison, let's actually take a look at the figures in more detail. Figure 4.8 shows a view of the terrain in which the meta view camera is equal to the main camera's position. Figure 4.8(b) represents the tessellation of the reference solution. Since the reference solution uses no LOD management, the wireframe lines in the distance are so dense on the screen, that the whole landscape appears in the wireframe's darker color. Figure 4.8(c) shows that *PGM* tessellates the peak immediately in front of the camera with a dense mesh in one direction, but at the same time the second, orthogonal dimension is sampled poorly. Figure 4.8(d) illustrates how our tailoring improvement brings the far-spaced samples, which are parallel to the sampling camera's view plane, closer together. In Figure 4.8(e) both tailoring and warping are applied to the persistent grid, thereby distributing grid vertices evenly on the ground plane toward the horizon. Since the grid cells are almost square, we can determine a mipmap level that matches both cell dimensions when fetching the displacement value from the heightmap. Note that importance-driven warping only considers the shape of projected grid cells on the ground plane, which means that triangles are still subject to arbitrary stretching after heightmap displacement. Figure 4.8(f) shows PGM_{TWEB}, which adds image-based bidirectional coherence to the equation. Note how the persistent grid is now actually twice as dense along both grid dimensions. Thanks to the distributed I-frame rendering, this is achieved at an unaltered triangle-per-frame count. The resulting quadrupled triangle count clearly boosts the faithfulness of the resulting terrain-surface approximation. Also remember, that the subscript "E" in PGM_{TE}, PGM_{TWE} and PGM_{TWEB} means that our local-edge-search



(a) Frame 1142 of Animation, White Rectangle marks Area of Interest

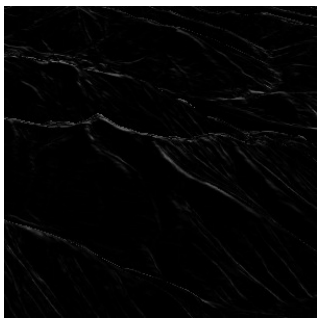


(b) Reference



(c) *PGM*, 22.69db PSNR

(d) PGM_{TE}, 27.97db PSNR



(e) PGM_{TWE}, 26.35db PSNR

(f) PGM_{TWEB}, 33.49db PSNR

(g) *Frostbite*, 29.33db PSNR

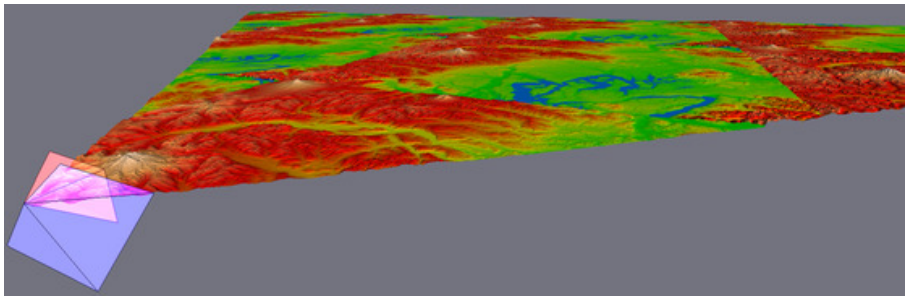
Figure 4.7: Differences of various render methods to the reference solution in frame 1495 of the camera animation. Image (a) shows the full camera frame (third interpolated B-frame in PGM_{TWEB}, i.e., only one "step" away from the future I-frame). The detail region that is analyzed is marked with a white rectangle and shown in isolation in image (b). Images (c) to (g) show the difference between each algorithm and the reference solution. The resulting PSNR values are given in the image caption.

improvement is active, no matter which of our other improvements are currently in use. Finally, Figure 4.8(g) reveals the triangles that are generated by *Frostbite*.

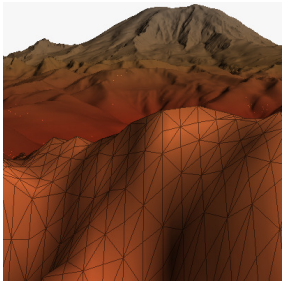
As mentioned above, in the remaining figures of this section, the sampling camera is still at the same position as in Figure 4.8, and we merely visit various parts of the static terrain-surface mesh that are further away from that position with a meta view camera. For the non-PGM-based render techniques, which do not use a sampling camera, this simply means that the detail levels of the terrain tiles are based on the position of the sampling camera and are not influenced at all by the position of the meta view camera either.

In Figure 4.9 we inspect a region of the landscape mesh that is just a little further away from the sampling camera's position than the area shown in the previous figure. As was illustrated in Figure 3.13, we can confirm that tailoring is most effective in the sampling camera's vicinity, i.e., that PGM_{TE} degenerates to *PGM* in the distance (see images (c) and (d) in Figure 4.9). Figure 4.9(e) shows that in more distant terrain sections warping has a bigger influence on heightmap-sampling quality than tailoring. Figure 4.9(f) illustrates how IBidirTC further boosts sampling density and thus the surface-approximation quality, coming close to the reference solution. Figure 4.9(g) reveals that *Frostbite* also does a good job at reconstructing the terrain. It is somewhat more accurate than PGM_{TWE}, but does not quite reach the sampling quality of PGM_{TWEB}.

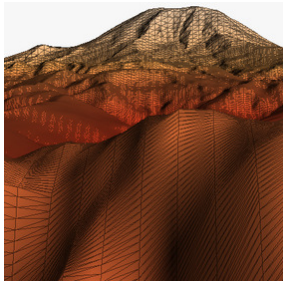
Figures 4.10 and 4.11 affirm the just predicted trend, namely that tailoring has a negligible influence on the improvement of the resulting image-quality with increasing distance, while warping and IBidirTC contribute a lot to the surface-reconstruction quality of the underlying digital elevation dataset. We can see that at this point both *PGM* and PGM_{TE} fail to reconstruct any terrain elevation. While in the sampling camera's vicinity *Frostbite* fell behind the image quality delivered by PGM_{TWEB}, it catches up in the distance. Eventually, toward the far end of the terrain mesh, the triangle tessellation created by *Frostbite* achieves a comparable triangle density as PGM_{TWEB}. In both figures we had to zoom out quite a bit to even see the elongated triangles from *PGM* and PGM_{TE}, which unfortunately leads to the reference solution becoming too densely covered by wireframe lines to see the underlying triangle structure created by the reference solution.



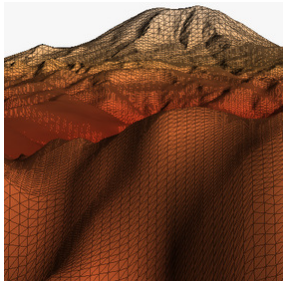
(a) Tessellation as seen from the Main Camera – showing Sampling Camera and Meta View Camera (coincides with Main Camera here)



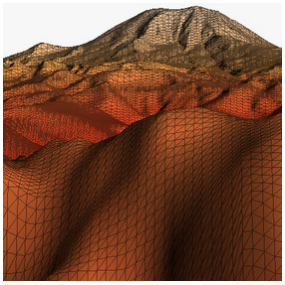
(b) Reference



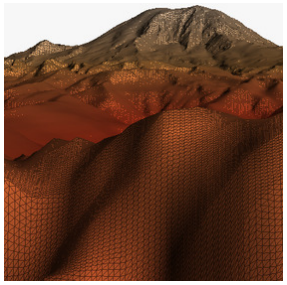
(c) *PGM*



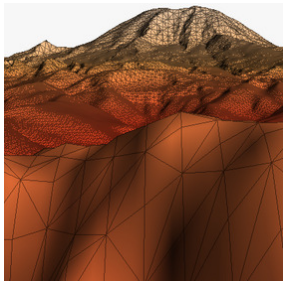
(d) PGM_{TE}



(e) PGM_{TWE}

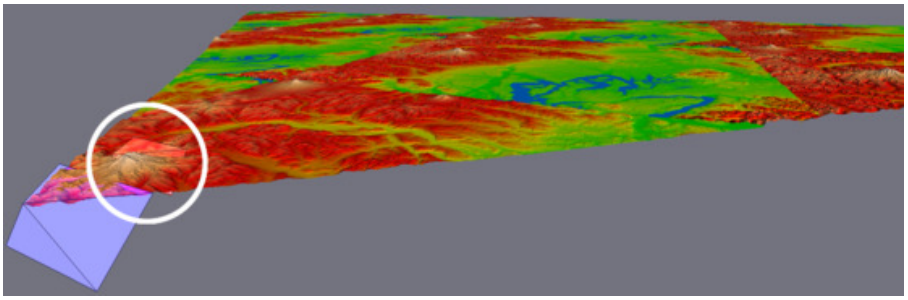


(f) PGM_{TWEB}

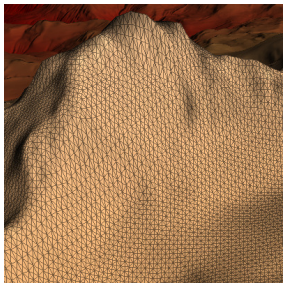


(g) *Frostbite*

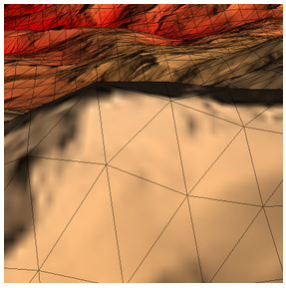
Figure 4.8: Illustrating terrain tessellation as seen from the main camera for various render methods through a wireframe mesh overlaid onto the surface. Image (a) shows where the sampling camera (blue view frustum) and the meta view camera (red view frustum) are located on the landscape. The meta view camera does not influence the detail level of the mesh. In this example, the meta view camera coincides with the main view camera. Images (b) through (g) show a cutout of the frame that is rendered on the screen from the meta view camera with various rendering algorithms. The respective captions reveal the employed algorithm.



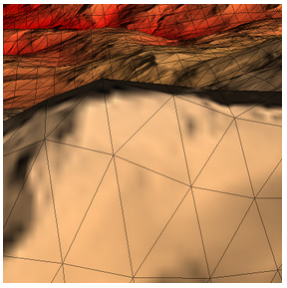
(a) Tessellation at Close Distance – showing Sampling Camera and Meta View Camera (marked with white circle)



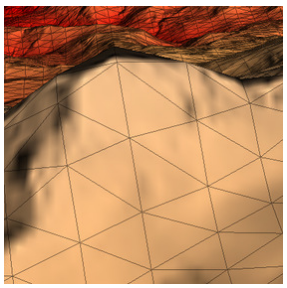
(b) Reference



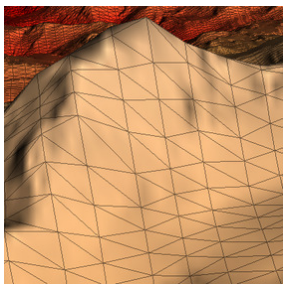
(c) *PGM*



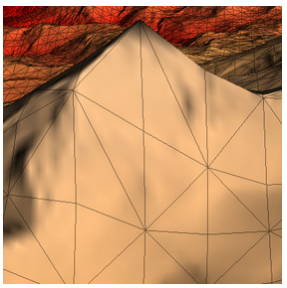
(d) PGM_{TE}



(e) PGM_{TWE}

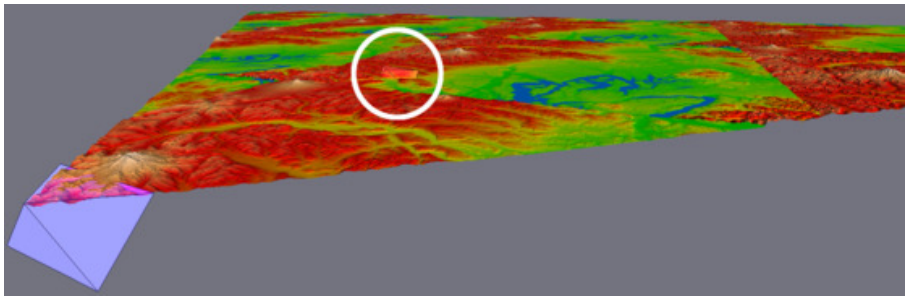


(f) PGM_{TWEB}

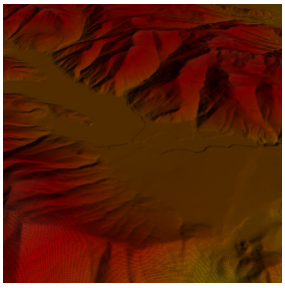


(g) *Frostbite*

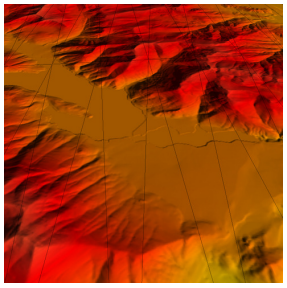
Figure 4.9: Illustrating terrain tessellation at a close distance to the sampling camera for various render methods through a wireframe mesh overlaid onto the surface as seen from a meta view camera. Image (a) shows where the sampling camera (blue view frustum) and the meta view camera (red view frustum) are located on the landscape. The meta view camera does not influence the detail level of the mesh. Images (b) through (g) show a cutout of the frame that is rendered on the screen from the meta view camera with various rendering algorithms. The respective captions reveal the employed algorithm.



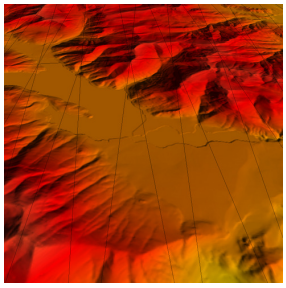
(a) Tessellation at Medium Distance – showing Sampling Camera and Meta View Camera (marked with white circle)



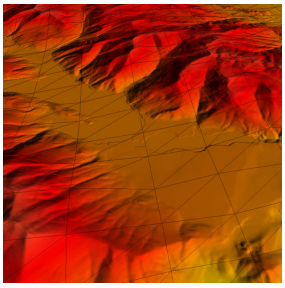
(b) Reference



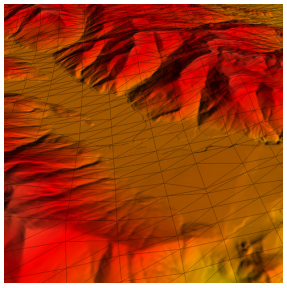
(c) *PGM*



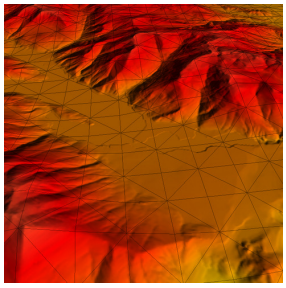
(d) PGM_{TE}



(e) PGM_{TWE}

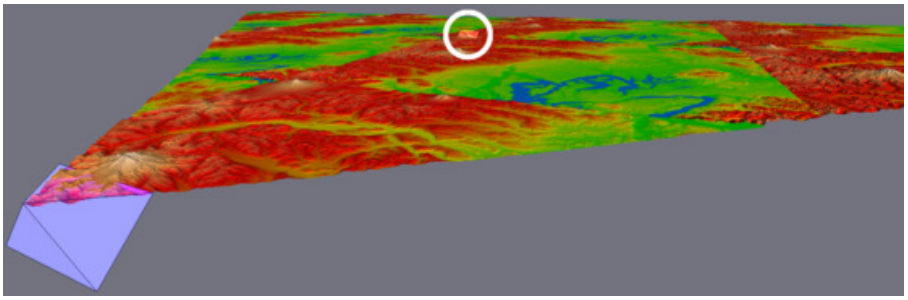


(f) PGM_{TWEB}

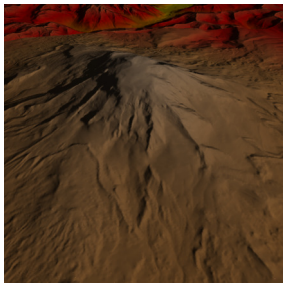


(g) *Frostbite*

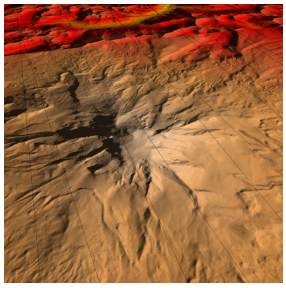
Figure 4.10: Illustrating terrain tessellation roughly in the middle of the line between the sampling camera and the far end of the visible terrain for various render methods through a wireframe mesh overlaid onto the surface as seen from a meta view camera. Image (a) shows where the sampling camera (blue view frustum) and the meta view camera (red view frustum) are located on the landscape. The meta view camera does not influence the detail level of the mesh. Images (b) through (g) show a cutout of the frame that is rendered on the screen from the meta view camera with various rendering algorithms. The respective captions reveal the employed algorithm.



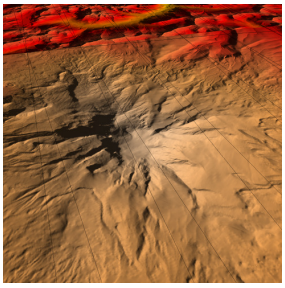
(a) Tessellation at Far Distance – showing Sampling Camera and Meta View Camera (marked with white circle)



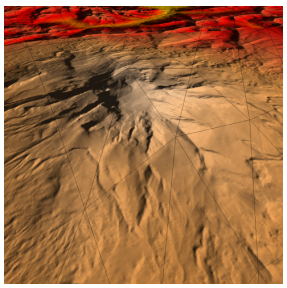
(b) Reference



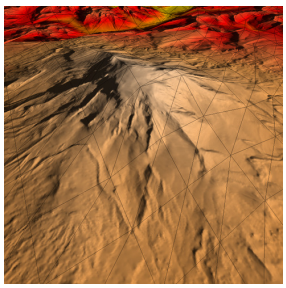
(c) *PGM*



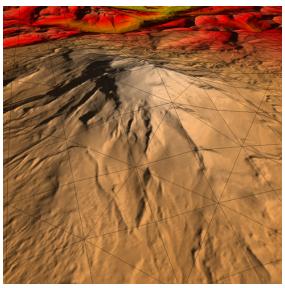
(d) PGM_{TE}



(e) PGM_{TWE}



(f) PGM_{TWEB}



(g) *Frostbite*

Figure 4.11: Illustrating terrain tessellation at a far-away distance from the sampling camera for various render methods through a wireframe mesh overlaid onto the surface as seen from a meta view camera. Image (a) shows where the sampling camera (blue view frustum) and the meta view camera (red view frustum) are located on the landscape. The meta view camera does not influence the detail level of the mesh. Images (b) through (g) show a cutout of the frame that is rendered on the screen from the meta view camera with various rendering algorithms. The respective captions reveal the employed algorithm.

CHAPTER 5

Conclusion and Future Work

In this thesis we presented a novel terrain-rendering algorithm that is based on and improves *Persistent Grid Mapping* (PGM) [32]. Our method preserves all desirable properties of PGM, which are, in no particular order:

- Approximates the terrain surface with a single watertight mesh with continuous LOD
- Issues only one draw call to render the visible part of the terrain
- Does not generate any degenerate triangles
- Automatic view-frustum culling
- Constant frame rate, since the same persistent-grid mesh is uploaded to the GPU only once and then used in every frame
- No need to transfer vertex data over the system bus at runtime, given that the GPU supports "vertex-texture fetch" or "render to vertex buffer".
- No additional preprocessing of terrain data necessary, hence enabling fast and easy modification of the terrain at runtime
- No additional terrain-geometry-related storage requirements other than the elevation data itself
- Intuitive performance fine-tuning possible by adapting the triangle count in the persistent grid
- Leaves the CPU almost idle
- Due to the similarity to Claes Johanson's real-time water rendering algorithm [30], PGM can serve as a unified framework to render both terrain and water surfaces

Further, we improve PGM with the following contributions:

- We tailor the persistent grid almost exactly to that part of the terrain which can possibly be seen through the main camera at hardly any computational cost. To achieve this, the persistent-grid cells are spatially compressed in order to make all of them fit into the determined area on the ground plane that has been found to contain terrain samples that potentially contribute to the rendered image, thereby wasting almost no grid triangles on invisible terrain areas, and reducing sampling intervals in the visible area.

- We present an entirely GPU-based algorithm to redistribute the vertices in the persistent grid in order to more evenly sample the ground plane. This redistribution makes better use of the available vertices in the grid, thereby drastically improving the terrain-surface approximation in the distance, where the basic PGM method tends to quickly fail to capture terrain features faithfully.

- We apply a simple local terrain-edge search for projected persistent-grid vertices in the camera's vicinity. For two adjacent vertices that share the same x-coordinate on the auxiliary camera's view plane before the grid projection of the grid onto the ground plane, i.e., vertices along the axis of anisotropy after the grid-projection step, we estimate their respective terrain slope. Based on the slope, we decide whether a terrain edge was potentially missed and if so, we move the vertex that is farther away onto said terrain edge.

- We present a novel B-frame interpolation algorithm for bidirectional temporal coherence, which is especially suited for static geometry. The B-frame reconstruction is based on point scattering, running efficiently on the GPU by employing the histogram-pyramid datastructure.

- When the camera position stays fixed for several frames, we approach an almost pixel-perfect terrain-surface solution by applying unidirectional temporal coherence to imperfect geometry. The challenge of this improvement was that normally temporal-coherence methods work on already correct geometry and only change the surface shading. We discuss the implications of accumulating rasterized samples that may be wrong and the circumstances under which the accumulation into a history buffer works nonetheless.

While PGM is undeniably elegant and simple, the algorithm is unfortunately far from perfect. The most distracting drawbacks of PGM are the vertex-swimming artifacts whenever the camera moves, and the insufficient sampling of distant terrain features. While our method does not fully solve these two issues, we showed that our proposed additions to PGM do indeed improve image quality significantly. Let us briefly recapitulate our contributions in the rest of this section.

We already said that the basic PGM method automatically performs view-frustum culling, but we must add that this automatic culling is often unnecessarily conservative. This leads to a considerable amount of persistent-grid vertices not contributing to the resulting image at all, since they ultimately fall off the screen. If at all, previous algorithms adjusted the projection matrix slightly before projecting the grid in order to lessen the area that the grid needs to cover for the current view. The adaptation of the projection matrix results in a persistent grid that is

still rectangular on the auxiliary camera's view plane, so that still a considerable amount of grid vertices fall off the screen. This simple approach was already described by Claes Johanson [30]. In contrast, we determine the minimal area on the ground plane that the grid needs to cover by intersecting the camera's view frustum with the terrain volume, and scale the grid such that all vertices lie within this area. Compared to the conservative tailoring, our approach typically leads to the grid having the shape of a general polygon on the auxiliary camera's view plane. Hence the proposed tailoring step leads to smaller sampling intervals on the ground plane after the persistent grid's projection, thereby making better use of the available mesh resolution. We outlined an efficient implementation on the GPU that is based on creating a redirection texture, which maps a vertex to its tailored position (see Section 3.3).

Another weakness of the basic PGM method that we addressed is that terrain features disappear quickly when projected grid vertices approach the horizon, due to the grid being stretched mainly along the axis of anisotropy. On the one hand, this stretching of the grid is a wanted side effect of the projective mapping of the persistent grid onto the ground plane, which is responsible for the automatic continuous LOD. On the other hand, the stretching is non-uniform, so that projected grid cell are stretched more on one axis than on the other. Since mipmap selection is governed by the longest cell edge in order to prevent aliasing artifacts, switching to coarser mipmap levels may happen "too early" due to the non-uniform cell stretching. Merely defining a static grid with more vertices along the y-coordinate or a tweaked vertex distribution does not solve the problem, since the amount of stretching varies across the grid as a function of the camera parameters, so that no single static grid-vertex distribution would perform well for all possible camera orientations and positions. Instead, we define an importance function based on the aspect ratio of projected grid cells. The importance-driven warping improvement then redistributes the grid vertices, ensuring that projected persistent-grid cells are mostly square. This way, the grid warping leads to a more appropriate mipmap-level selection across the whole projected grid and therefore helps preserve heightmap detail even towards the horizon. Note that we can efficiently combine the tailoring of the persistent grid with the grid-warping improvement by simply using the tailored grid-vertex positions when calculating the ratios of the projected grid-cells on the ground plane. An almost identical idea for importance-steered grid-vertex redistribution was proposed by Löffler et al. [34], the major difference being that they presented a CPU-based grid-vertex redistribution algorithm. Our warping algorithm is designed to run entirely on the GPU in an efficient manner, while not requiring any more advanced GPU features than those that were available in 2009, the year in which Löffler et al. published their warping algorithm.

To further increase image stability, we introduced local terrain-edge search in the camera's vicinity, a technique which, to our knowledge, has not been mentioned in any publication related to PGM or any other grid-projection method yet. With this improvement we try to detect missed terrain edges between two adjacent persistent-grid vertices that share the same x-coordinate in the persistent-grid mesh. Our heuristic to detect a missed terrain edge is to compare the terrain slope at two adjacent grid vertices along the axis of anisotropy, i.e. along the vector from the closer to the farther of the two projected adjacent grid vertices on the ground plane. This strategy may not detect missed terrain edges all of the time, but we found that the method represents a reasonable tradeoff between increased image quality versus low impact on performance. When

the slope at the closer vertex is positive, while the slope at the farther vertex is negative, we conclude that we just missed sampling a terrain edge between the two vertices, and move the farther vertex to that terrain edge. We perform a binary search that is guided by changes in the terrain slope to find the position of the missed terrain edge.

A major part of our work is devoted to exploiting temporal coherence between individual frames to further battle the remaining temporal aliasing, which manifests itself in vertex-swimming artifacts. We use *image-based bidirectional temporal coherence* (IBidirTC) by Yang et al. [65] to distribute the rendering of I-frames across four frames, which enables rendering a moderate per-frame triangle count that results in I-frames consisting of four times as many triangles. The B-frames between two I-frames are reconstructed from the I-frames by a greedy image-based search on the three-dimensional scene-flow field. Regarding image quality, IBidirTC offers two advantages. Firstly, the higher triangle count improves terrain-surface approximation quality in its own right, and secondly, the B-frame interpolation provides temporal smoothing. IBidirTC by itself does not solve all our open issues, though. For example, at depth discontinuities, B-frames that are reconstructed by the unaltered IBidirTC method may fail to find the corresponding fragment in the adjacent I-frames, which leads to degraded image quality. We addressed this weakness by introducing a new B-frame interpolation algorithm that scatters point sprites, which get their color and depth from I-frame texels that lie at or close to depth discontinuities. By interpolating the camera parameters of the two adjacent I-frames and assuming linear motion between the I-frames, we can perform exact reprojection from each adjacent I-frame into the current B-frame. The selective point-sprite scattering is performed efficiently on the GPU by employing the histogram-pyramid data structure.

Unfortunately, IBidirTC also introduces some issues. Apart from the increased memory requirements and the performance impact, the distributed I-frame rendering causes an additional user-input delay, since we can't change the camera or anything in the scene until a new I-frame is started, which, in our method, happens every fourth frame. The same delay needs to be enforced when performing terrain modifications at runtime.

In our final improvement, we explored combining PGM with unidirectional temporal coherence. We showed how to deal with imperfect geometry, which is a new challenge, since typically, temporal-coherence methods merely deal with surface shading, and assume that geometry was rendered correctly. We saw that even if each rasterized frame contains terrain-surface approximation errors, we can accumulate the results when visibility does not change, i.e., when the camera position does not change. When updating the history buffer, we employ the simple heuristic of sticking to the closest fragment. Even though this does not always prefer the correct value, it nevertheless leads to less noticeable artifacts. Other methods that build on unidirectional temporal coherence typically employ a more elaborate confidence function to decide whether to update the history buffer with a newly calculated fragment or to skip this calculation and simply reuse the already stored fragment from the history buffer. Just like most other unidirectional-temporal-coherence methods, we randomly force refreshing of small blocks of fragments from time to time to prevent over-usage of values from the history buffer, which would introduce unwanted blurring.

By slightly jittering the auxiliary camera in a random manner each frame, we virtually increase the projected grid's mesh resolution, thus refining image quality gradually. With this

previously unexplored PGM improvement, we tackle another inadequacy of basic PGM, namely that the algorithm cannot guarantee a maximum screen-space error. While technically speaking, our method still does not guarantee a maximum screen space error, our improvement approximates a near pixel-perfect solution – typically within one to two seconds. We backed this claim by comparing the images generated by our method to images calculated by POV-Ray. Note that Löffler et al. [34] enhanced PGM in such a way that a maximum screen-space error can be guaranteed, however additional data related to the elevation map needs to be prepared in a preprocessing step, and that data is used at runtime to perform ray casting at those fragments that violate the maximum screen-space error tolerance. The varying number of fragments for which raymarching needs to be performed paired with indeterminate raymarch durations per fragment lead to a less predictable frame rate. In contrast to the method from Löffler et al., our method has a constant frame rate, though.

Finally, we demonstrated that our algorithm generally achieves even better image quality than the Frostbite terrain-rendering algorithm (or, for that matter, any other similar quadtree-based terrain-rendering algorithm that draws tile meshes at various resolutions), at nearly equal triangle-per-frame counts, albeit at a lower frame rate. The lower frame rate is due to bandwidth limitations that arise from several screen-sized render targets that are needed for the image-based bidirectional-temporal-coherence improvement.

In Chapter 5.1, we sketch a modification of the bidirectional-temporal-coherence part of our algorithm which gets rid of the backward- and forward flow fields and the greedy image-based search algorithm for B-frame reconstruction. Our idea is based on image-space raymarching and it sounds promising in that not only storage requirements can be reduced (no need for storing, updating and accessing velocity fields anymore), but performance should benefit as well due to the reduced bandwidth requirements. It would be interesting to compare the performance of the just sketched modification of our terrain-rendering algorithm to the Frostbite terrain renderer.

When disabling the image-based bidirectional temporal coherence improvement, our terrain-rendering method approximates the digital elevation dataset less accurately than the Frostbite landscape rendering system, but does so at a slightly faster frame rate.

As just hinted, a notable feature of our technique is that the individual improvements can be employed independently of each other, so that each improvement can be seen as a modular block, which enables tweaking image quality and performance. This may be interesting for systems with less computational power, since skipping some of the improvements will obviously increase the overall performance of the algorithm.

While we hardly talked about rendering large datasets, and did not implement out-of-core rendering in our prototype, our method does not rule out texture streaming. In fact, the exact same approach to handle vast landscapes that was presented in the original publication of PGM [32] can also be used in our method. In short, this approach involves a relatively small terrain-data working set that fits into video ram and represents the currently visible part of the landscape at several detail levels. As the camera moves, the working set is incrementally updated from the large dataset, which may reside in main memory and/or on disk, or is synthesized on the fly in real time.

Apart from describing our new method in detail, we hope that we also managed to give an overview of the history of terrain rendering, as well as other popular terrain-rendering algo-

rithms. We also discussed related topics like calculating the screen-space error of terrain-surface approximations, and how to utilize GPUs efficiently when rendering regular grids.

5.1 Future Work

In the process of developing our new method, we discovered several ideas worth exploring, which we want to sketch briefly here.

One of these ideas, which we touched upon in Chapter3.2.1, is to build the grid from equilateral triangles, yielding diamond-shaped grid cells. We would then be able to compare the image quality that results from rendering the terrain using a regular grid made from right triangles to the images generated by projecting and displacing a persistent grid employing a diamond tessellation.

Another approach that we would like to explore further is using hardware tessellation on the silhouettes of the terrain. We could stick to our tailoring and grid-warping steps, and then adapt the local-edge-search improvement. Instead of estimating the slope of the heightmap in world space, we would look for silhouettes in eye space by inspecting eye-space normals. Triangles at the detected silhouettes would then be subdivided in order to better approximate the actual terrain shape with the persistent grid. With hardware tessellation at hand, we could even enhance our grid-warping step and account for triangle stretching due to steep slopes, instead of only looking at the ratios of projected grid cells on the ground plane, which currently ignores the height displacement altogether.

Talking of our grid-warping step, we think that it should be possible to derive an analytic warping equation to get rid of the multipass evaluation when redistributing grid-vertices.

Finally, we would like to explore yet another way of interpolating B-frames in IBidirTC when dealing with static geometry only. IBidirTC reconstructs B-frames with an image-space greedy search algorithm, that employs forward and backward three-dimensional flow fields, which are part of each I-frame. We showed that we can augment interpolated B-frames by scattering point sprites from adjacent I-frames without using these flow fields at all. Instead, we established an intermediate camera frame for each B-frame, which enables us to reproject any I-frame sample into the B-frame. Reconstructing B-frames by point scattering alone is not doable, though, since we can't guarantee that each and every B-frame pixel is hit by a reprojected point sprite. Furthermore, we would have to render at least as many point sprites as there are pixels on the screen, which would lead to degraded performance. A similar technique that is potentially more promising than our point scattering is to perform ray marching for each B-frame pixel. We would start by setting up a ray through each B-frame pixel and transform that ray into each of the two adjacent I-frames. In each I-frame we would perform an eye-space raymarch in order to intersect the ray with the I-frame's depth buffer. Finally, the color and depth of the I-frame intersection point would be assigned to the B-frame pixel that we started with. If an intersection is found in both the past and future I-frame, we could apply the same logic to combine them as was used for the image-space search results, which was outlined in Section 2.4.1. The big advantage of the just sketched B-frame reconstruction technique is that we wouldn't need to create and store the forward and backward optical flow fields, thereby reducing memory and bandwidth requirements. While the intersection of the transformed ray with the I-frame's depth

buffer would involve several dependent texture fetches, for reasonably small camera movements, the transformed ray, which is clipped to the I-frame's near and far plane, would project to a line consisting of only a few pixels in the respective I-frame, so that only those few pixels would need to be processed. And let's not forget that after all, IBidirTC's greedy B-frame interpolation algorithm performs several dependent texture fetches as well. We therefore expect this new B-frame interpolation algorithm to outperform both the original image-based search algorithm as well as our proposed point-scattering augmentation, while yielding high-quality results – at least for static scenes.

Bibliography

[1] ARB_timer_query. http://developer.download.nvidia.com/opengl/specs/GL_ARB_timer_query.txt.

[2] Evans & Sutherland. http://en.wikipedia.org/wiki/Evans_%26_Sutherland.

[3] Geometry shader stage. https://msdn.microsoft.com/en-us/library/windows/hardware/ff566757

[4] NVIDIA GPU Programming Guide. http://developer.download.nvidia.com/GPU_Programming_Guide/GPU_Programming_Guide_G80.pdf.

[5] NVIDIA GPU Programming Guide. http://developer.download.nvidia.com/GPU_Programming_Guide/GPU_Programming_Guide.pdf.

[6] Peak signal-to-noise ratio. https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio.

[7] Peak Signal-to-Noise Ratio as an Image Quality Metric. http://www.ni.com/white-paper/13306/en/.

[8] Post transform cache. https://www.opengl.org/wiki/Post_Transform_Cache. Accessed: January 2016.

[9] QueryPerformanceCounter function. https://msdn.microsoft.com/en-us/library/windows/desktop/ms644904(v=vs.85).aspx.

[10] Tessellation Evaluation Shader. https://www.khronos.org/opengl/wiki/Tessellation_Evaluation_Shader.

[11] Vertex shader inputs. https://www.khronos.org/opengl/wiki/Built-in_Variable_(GLSL)#Vertex_shader_inputs.

[12] Andersson, Johan. Terrain Rendering in Frostbite using Procedural Shader Splatting. http://developer.amd.com/wordpress/media/2013/02/Chapter5-Andersson-Terrain_Rendering_in_Frostbite.pdf. Accessed: January 2016.

[13] Arul Asirvatham and Hugues Hoppe. Terrain rendering using GPU-based geometry clipmaps. In Matt Pharr and Randima Fernando, editors, *GPU Gems 2*, chapter 2, pages 27–45. Addison Wesley, March 2005.

[14] Iain Cantlay. DirectX 11 Terrain Tessellation. `https://developer.nvidia.com/sites/default/files/akamai/gamedev/files/sdk/11/TerrainTessellation_WhitePaper.pdf`, 2011.

[15] Albert Cervin. Adaptive Hardware-accelerated Terrain Tessellation. Master's thesis, Linköpings universitet, 2012.

[16] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. BDAM – Batched Dynamic Adaptive Meshes for High Performance Terrain Visualization. *Computer Graphics Forum*, 22(3):505–514, September 2003. Proc. Eurographics 2003 – Second Best Paper Award.

[17] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Planet–sized batched dynamic adaptive meshes (p-bdam). In *Proceedings IEEE Visualization*, pages 147–155, Conference held in Seattle, WA, USA, October 2003. IEEE Computer Society Press.

[18] Malte Clasen and Hans-Christian Hege. Terrain rendering using spherical clipmaps. In Beatriz Sousa Santos, Thomas Ertl, and Kenneth I. Joy, editors, *EuroVis*, pages 91–98. Eurographics Association, 2006.

[19] Philippe Decaudin and Fabrice Neyret. Volumetric billboards. *Computer Graphics Forum*, 28(8):2079–2089, 2009.

[20] Christian Dick, Jens Krüger, and Rüdiger Westermann. GPU Ray-Casting for Scalable Terrain Rendering. In *Proceedings of Eurographics 2009 - Areas Papers*, pages 43–50, 2009.

[21] Christian Dick, Jens Krüger, and Rüdiger Westermann. GPU-Aware Hybrid Terrain Rendering. In *Proceedings of IADIS Computer Graphics, Visualization, Computer Vision and Image Processing 2010*, pages 3–10, 2010.

[22] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Mille, Charles Aldrich, and Mark B. Mineev-weinstein. Roaming terrain: Real-time optimally adapting meshes. In *IEEE Visualization*, pages 81–88, 1997.

[23] Christopher Dyken, Gernot Ziegler, Christian Theobalt, and Hans-Peter Seidel. High-speed Marching Cubes using HistoPyramids. *Computer Graphics Forum*, 27(8):2028–2039, September 2008.

[24] Fabian Giesen. A trip through the Graphics Pipeline 2011, part 6. `https://fgiesen.wordpress.com/2011/07/06/a-trip-through-the-graphics-pipeline-2011-part-6/`, July 2011.

[25] Sven Forstmann and Jun Ohya. Visualizing large procedural volumetric terrains using nested clip-boxes. Departmental bulletin paper, Waseda University, October 2011.

[26] Anton Frühstück. Gpu based clipmaps. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, April 2008.

[27] Mark Harris and Ian Buck. Gpu flow-control idioms. In Matt Pharr and Randima Fernando, editors, *GPU Gems 2*, chapter 34, pages 547–555. Addison Wesley, March 2005.

[28] Damien Hinsinger, Fabrice Neyret, and Marie-Paule Cani. Interactive animation of ocean waves. In *ACM-SIGGRAPH/EG Symposium on Computer Animation (SCA)*, july 2002.

[29] Hugues Hoppe. Optimization of mesh locality for transparent vertex caching. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 269–276, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[30] Claes Johanson. Real-time water rendering: Introducing the projected grid concept. Master's thesis, Lund University, March 2004.

[31] Yotam Livny, Zvi Kogan, and Jihad El-Sana. Seamless patches for gpu-based terrain rendering. *Vis. Comput.*, 25(3):197–208, February 2009.

[32] Yotam Livny, Neta Sokolovsky, Tal Grinshpoun, and Jihad El-Sana. Persistent Grid Mapping: A GPU-based Framework for Interactive Terrain Rendering. Technical report, Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel, October 2006. http://www.cs.bgu.ac.il/ grinshpo/publications.html.

[33] Yotam Livny, Neta Sokolovsky, Tal Grinshpoun, and Jihad El-Sana. A GPU persistent grid mapping for terrain rendering. *The Visual Computer*, 24(2):139–153, 2008.

[34] Falko Löffler, Stefan Rybacki, and Heidrun Schumann. Error-bounded gpu-supported terrain visualisation. In Min Chen and Vaclav Skala, editors, *WSCG proceedings 2009*, WSCG 2009 Communication Papers, pages 47–54, Plzen, Czech Republic, 2009. ISBN 978-80-86943-94-7.

[35] Frank Losasso and Hugues Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 769–776, New York, NY, USA, 2004. ACM.

[36] David Luebke, Benjamin Watson, Jonathan D. Cohen, Martin Reddy, and Amitabh Varshney. *Level of Detail for 3D Graphics*. Elsevier Science Inc., New York, NY, USA, 2002.

[37] Joseph D. Mahsman. Projective Grid Mapping for Planetary Terrain. Master's thesis, University of Nevada, Reno, December 2010.

[38] Jalal Eddine El Mansouri. Rendering 'Rainbow Six | Siege'. http://www.gdcvault.com/play/1022990/Rendering-Rainbow-Six-Siege, 2016. GDC.

[39] Oliver Mattausch, Daniel Scherzer, and Michael Wimmer. High-quality screen-space ambient occlusion using temporal coherence. *Computer Graphics Forum*, 29(8):2492–2503, December 2010.

[40] Morgan McGuire and Peter Sibley. A heightfield on an isometric grid. *SIGGRAPH 2004 Sketch*, Aug 2004.

[41] D. Nehab, P. V. Sander, J. D. Lawrence, N. Tatarchuk, and J. R. Isidoro. Accelerating real-time shading with reverse reprojection caching. In *ACM SIGGRAPH/Eurographics Symposium on Graphics Hardware*, pages 25–35, August 2007.

[42] Ray L. Page. Brief History of Flight Simulation. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.132.5428&rep=rep1&type=pdf.

[43] Renato Pajarola and Enrico Gobbetti. Survey of semi-regular multiresolution models for interactive terrain rendering. *The Visual Computer*, 23(8):583–605, 2007.

[44] Alex A. Pomeranz. Roam using surface triangle clusters (rustic). Master's thesis, Center for Image Processing and Integrated Computing, University of California, Davis, 2000.

[45] Paul Rosen. Rectilinear texture warping for fast adaptive shadow mapping. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, pages 151–158, New York, NY, USA, 2012. ACM.

[46] D. Scherzer, L. Yang, O. Mattausch, D. Nehab, P. V. Sander, M. Wimmer, and E. Eisemann. A survey on temporal coherence methods in real-time rendering. In *Eurographics State of the Art Reports*, May 2011.

[47] D. Scherzer, L. Yang, O. Mattausch, D. Nehab, P. V. Sander, M. Wimmer, and E. Eisemann. Temporal coherence methods in real-time rendering. *Computer Graphics Forum*, 31(8):2378–2408, 2012.

[48] Daniel Scherzer, Stefan Jeschke, and Michael Wimmer. Pixel-correct shadow maps with temporal reprojection and shadow test confidence. In Jan Kautz and Sumanta Pattanaik, editors, *Rendering Techniques 2007 (Proceedings Eurographics Symposium on Rendering)*, pages 45–50. Eurographics, Eurographics Association, June 2007.

[49] Jens Schneider, Tobias Boldte, and Ruediger Westermann. Real-time editing, synthesis, and rendering of infinite landscapes on GPUs. In *Vision, Modeling and Visualization 2006*, 2006.

[50] Jens Schneider and Rüdiger Westermann. Gpu-friendly high-quality terrain rendering. *Journal of WSCG*, 14(1-3):49–56, 2006.

[51] Adrian Secord, Wolfgang Heidrich, and Lisa Streit. Fast primitive distribution for illustration. In *Proceedings of the 13th Eurographics workshop on Rendering*, EGRW '02, pages 215–226, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[52] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification*. http://www.opengl.org/registry/doc/glspec43.core.20130214.pdf, February 2013.

[53] Andrew Selle, Ronald Fedkiw, Byungmoon Kim, Yingjie Liu, and Jarek Rossignac. An unconditionally stable maccormack method. *J. Sci. Comput.*, 35(2-3):350–371, June 2008.

[54] Dave Shreiner, Graham Sellers, John M. Kessenich, and Bill M. Licea-Kane. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. Addison-Wesley Professional, 8th edition, 2013.

[55] Ron Sivan. Surface modeling using quadtrees. Technical Report CS-TR-3609, Univ. of Maryland, College Park, Ctr. for Automation Research, Feb 1996.

[56] Filip Strugar. Continuous distance-dependent level of detail for rendering heightmaps. *Journal of Graphics, GPU, and Game Tools*, 14(4):57–74, 2009.

[57] László Szécsi and Khashayar Arman. Procedural ocean effects. In Wolfgang Engel, editor, *ShaderX6*, pages 13–39. Course Technology, 2008.

[58] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. The clipmap: A virtual mipmap. In *SIGGRAPH*, pages 151–158, 1998.

[59] Thatcher Ulrich. Chunked LOD: Rendering Massive Terrains using Chunked Level of Detail Control, 2004. Course at SIGGRAPH 02, http://www.tulrich.com/geekstuff/chunklod.html (01-07-2004).

[60] USGS and The University of Washington. Puget Sound Model. `http://www.cc.gatech.edu/projects/large_models/ps.html`.

[61] Michal Valient. Taking Killzone Shadow Fall Image Quality into the Next Generation. `https://www.guerrilla-games.com/read/taking-killzone-shadow-fall-image-quality-into-the-next-generation-1`, March 2014. GDC.

[62] Harald Vistnes. GPU Terrain Rendering. In *Game Programming Gems 6*, pages 461–471. Charles River Media, 2006.

[63] Z. Wartell, W. Ribarsky, and L. Hodges. Efficient ray intersection for visualization and navigation of global terrain using spherical height-augmented quadtrees. In *Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization*, May 1999.

[64] Widmark, Mattias. Terrain in Battlefield 3: A modern, complete and scalable system. `http://www.frostbite.com/wp-content/uploads/2013/05/GDC12_Terrain_in_Battlefield3.pdf`. Accessed: January 2016.

[65] L. Yang, Y.-C. Tse, P. V. Sander, J. D. Lawrence, D. Nehab, H. Hoppe, and C. L. Wilkins. Image-based bidirectional scene reprojection. *ACM Transactions on Graphics (Proceedings of the ACM SIGGRAPH Asia 2011)*, 30(6):150, 2011.

[66] Egor Yusov. Real-Time Deformable Terrain Rendering with DirectX 11. In Wolfgang Engel, editor, *GPU Pro 3*, pages 13–39. A K Peters, 2012.

[67] Gernot Ziegler, Art Tevs, Christian Theobalt, and Hans-Peter Seidel. Gpu point list generation through histogram pyramids. Research Report MPI-I-2006-4-002, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, June 2006.