Rendering Geometric Complexity in Bandwidth-constrained Environments

by

Matteo Tanca

Dissertation

Presented to the

University of Dublin, Trinity College

in partial fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

(Interactive Entertainment Technology)

University of Dublin, Trinity College

August 2012

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Matteo Tanca

August 30, 2012

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this dissertation upon request.

Matteo Tanca

August 30, 2012

Acknowledgments

I would like to thank Dr. John Dingliana for his helpful advice on this project and throughout the whole year.
I also wish to thank my family, for their constant support over the years.

Matteo Tanca

University of Dublin, Trinity College
August 2012

Rendering Geometric Complexity in Bandwidth-constrained Environments

Matteo Tanca
University of Dublin, Trinity College, 2012

Supervisor: John Dingliana

Nowadays mobile and browser games are becoming increasingly popular and more and more graphically appealing, thanks to recent advancements in software and hardware technologies.
In March 2011 Khronos Group released the WebGL standard, which allows most major browsers to display complex 3D scenes natively, opening up new possibilities for browser-based game development. While this represents a huge advancement in terms of rendering capabilities, it does not overcome the issue of timely receiving the amount of data required to render a 3D scene. Although the network bandwidth available to the average user has been constantly increasing in the past few years, streaming complex assets still represents a challenging issue. Long download times can easily break up the flow of a game, making it less appealing, or even worse unplayable.
The dissertation analyses currently available approaches that could be used to over-

come this issue, focusing in particular on surface mapping, progressive level of detail and procedural content generation. After surveying the state of the art, it identifies and describes in more detail several techniques that are well suited for WebGL/OpenGL ES environments, in terms of hardware and software limitations. It then gives an overview of how the proposed solutions were implemented in a prototype framework, consisting of a client-side WebGL browser application, and a server implemented in JavaScript/Node.js. Lastly, it evaluates the results obtained with the implemented approaches, supporting the discussion with quantitative measurements where appropriate, and identifying current limitations and possible directions for future work.

Contents

List of Tables

List of Figures

Abbreviations

- **API** **A**pplication **P**rogramming **I**nterface
- **COLLADA** **COLLA**borative **D**esign **A**ctivity
- **CPU** **C**entral **P**rocessing **U**nit
- **FOV** **F**ield **O**f **V**iew
- **GJK** **G**ilbert **J**ohnson **K**eerthi algorithm
- **GLSL** **GL** **S**hading **L**anguage
- **GPU** **G**raphics **P**rocessing **U**nit
- **HSDPA** **H**igh **S**peed **D**ownlink **P**acket **A**ccess
- **JSON** **J**ava**S**cript **O**bject **N**otation
- **LOD** **L**evel **O**f **D**etail
- **MMI** **M**ulti-**M**esh **I**mpostor
- **PM** **P**rogressive **M**esh
- **POM** **P**arallax **O**cclusion **M**apping
- **PCG** **P**rocedural **C**ontent **G**eneration
- **PRNG** **P**seudo **R**andom **N**umber **G**enerator
- **QDMHB** **Q**uadtree **D**isplacement **M**apping with **H**eight **B**lending
- **X3D** e**X**stensible **3D** graphics

Chapter 1

Introduction

1.1 Context

Nowadays mobile and browser games are becoming increasingly popular and more and more graphically appealing, thanks to recent advancements in software and hardware technologies.
In March 2011 Khronos Group released the WebGL 1.0 API specification[16], which is now supported natively by all major browsers, except for Internet Explorer, which requires a third-party plugin[1] to enable its use.
This novel technology makes it possible to render complex 3D environments and models on compliant browsers, opening up new possibilities for browser-based game development. While this represents a huge advancement in terms of rendering capabilities, it does not overcome the issue of timely receiving the amount of data required to render a 3D scene. Although the network bandwidth available to the average user has been constantly increasing in the past few years, streaming complex assets still represents a challenging issue. Long download times can easily break up the flow of a game, making it less appealing, or even worse unplayable.
The dissertation aims to tackle this problem, as described in the next section, with a specific focus on browser-based game development.

[1] Internet Explorer WebGL plugin [6].

1.2 Objectives

Given a generic 3D client, which must download all necessary assets from a remote server, we aim to enable the client to render complex 3D scenes in real-time. Hence, the main goal of the dissertation is to develop a set of suitable techniques to reduce the amount of data required by the client at any given time, so that the necessary information can always be timely downloaded, without exceeding the available bandwidth limit (which would temporarily interrupt the rendering of the scene).
The research question is meaningful in all contexts where a 3D client, in order to render a scene, needs to receive the corresponding data over a bandwidth-limited network. In this sense, common examples are browser-based 3D games and applications (such as 3D viewers), as well as some recent multi-player, open-world games developed for mobile platforms.
Since complex browser-based 3D games are becoming commonplace, we decided to focus specifically on the WebGL platform, which is quickly emerging as a very common development choice for browser 3D applications. Nonetheless, due to the sufficient generality of the presented design and the strong similarities between the WebGL and OpenGL ES specifications, the implementation could be easily ported to any platform supporting OpenGL ES 2.0[14] (e.g., smartphones, tablets, ...).
In order to fulfil the research question, the following tasks were required:

1. Investigation of a number of research directions, in order to determine a set of promising techniques that could be used for our purpose. We mainly focused on the following directions:
   - *surface mapping*,
   - *progressive level of detail*,
   - *procedural content generation*.
2. Identification of which approaches in the state of the art are amenable to an efficient implementation into the WebGL context.
3. Based on the previous evaluation, design of a suitable solution and development of a prototype client-server framework implementing it.

1.3 Structure of the document

The rest of the document is organized as follows:

- **Chapter 2: Background** surveys current state-of-the-art approaches that could be applicable to solve the research question at hand. The chapter tries to assess which solutions are most suitable for the chosen implementation environment, in terms of performance and feasibility. Also, it gives an overview of WebGL frameworks available to date.

- **Chapter 3: Design** provides a high-level, detailed description of the techniques that were implemented in the prototype. First, the chapter describes two mainstream surface mapping approaches (parallax and relief mapping) which were included in the implementation. Then it delves into the mesh simplification algorithm, describing the approach employed to simplify 3D models in order to achieve progressive rendering. Lastly, it describes a pseudo-random procedural approach for terrain generation, implemented as a proof of concept of possible applications of PCG and its benefits.

- **Chapter 4: Implementation** describes how the techniques presented in the previous chapter were practically developed into a client-server framework, by using Node.js as a server-side technology, and any WebGL-enabled browser as a client.

- **Chapter 5: Conclusion** first presents and evaluates the results obtained with the implemented techniques. Then, it briefly sums up the content of the dissertation, providing some concluding remarks and possible development directions for future work and improvements.

Chapter 2

Background

WebGL is still a relatively new technology, so to date there are not many papers on relevant topics that specifically address this platform. However, in the past decade the problem of transmitting complex scenes over the network has been studied, for instance, in the field of virtual archaeology [44], with some recent results implemented in WebGL [59]. Of course, the objectives and the scope are slightly different from those considered in this dissertation.
More specifically, representing archaeological models requires high fidelity to the original to be preserved whenever possible, so research efforts are mostly focused on finding efficient ways of transmitting large 3D models in a progressive way.
In a browser game though, it is probably more important to provide a visually appealing experience to the user, rather than always preserving high accuracy of the models. Therefore it makes sense to consider a wider range of approaches:

- *Level Of Detail techniques*: as described in [44], Level Of Detail (LOD) solutions can be geometry-based or image-based. Within the first category, *progressive meshes* are probably the most used approach, while *impostors* are the most common in the latter.

- *Procedural content generation*: in order to render the game environment, a viable alternative to transmitting the corresponding geometry over the network is given by the use of procedural techniques, which allow a drastic reduction in the amount of data transmitted.

- *Surface mapping*: to increase the realism of rendered scenes, surface mapping techniques may also be applied (e.g., parallax mapping, relief mapping,...).

Before providing an overview of the state of the art, it is worth noting that the current WebGL specification is a subset of the OpenGL ES 2.0 standard, which targets mobile device hardware and is derived from the OpenGL 2.0 specification. Hence, WebGL lacks support for some of the more recent features offered by OpenGL 3.x/4.x standards[12, 13] (e.g., hardware instancing, geometry shaders and tessellation are not available). This fact makes some recent solutions less appealing in terms of performance, sometimes ruling them out completely.
In this chapter, we propose a review of the most promising algorithms and techniques in the previously mentioned research areas. Moreover, we briefly introduce currently available WebGL frameworks and assess their features.

2.1 Level Of Detail techniques

2.1.1 Geometry-based techniques

Geometry-based approaches employ a series of approximations, derived from an original model by progressively reducing the amount of geometric data, at the same time trying to preserve the visual quality of the output.
It is worth noting that, consistently with the topic of the dissertation, we are especially interested in techniques that allow both smooth *geomorphing*[1] and efficient transmission of the different LODs.
To date, mesh simplification algorithms can be categorized into three major classes, based on the underlying approach:

- *Edge contraction.*
- *Vertex clustering.*
- *Re-synthesis.*

[1]Geomorphing is the process of interpolating between models having different levels of detail, in order to avoid suddenly switching between different levels.

Edge contraction

In the past 15 years there has been great interest in this class of algorithms. Most of them differ only in the metric used to decide which edge should be contracted at each simplification step.
The majority employ different forms of quadric error metrics in order to keep track of the progressively simplified surfaces and evaluate the quality of the approximation.
As stated in [63], the major drawback of most edge-contraction based approaches lies in the fact that the input mesh must be manifold[2] in order to perform edge collapsing. Whenever there is no guarantee that the input is well-formed, a preliminary clean up of the mesh is required. This is typically not a major issue with game graphics assets, since it only adds an offline pre-processing step. However, it may cause a performance hit if the simplification needs to be performed in real-time. Algorithms based on edge contraction are also usually hard to parallelise, since the iterative approach used to collapse the edges is inherently sequential.
One of the earliest and most successful proposals in this class is the *progressive mesh* (PM) algorithm, as introduced by Hoppe in [31, 32]. A PM is a representation scheme for storing and transmitting arbitrary triangle meshes that allows smooth geomorphing of subsequent LOD approximations, progressive transmission of models and mesh compression. The simplification process can be applied to arbitrary meshes and preserves scalar appearance attributes (e.g., colour values, UV texture mapping and normals). It has been implemented in the DirectX API for several years now, so it can be considered an industrial-strength solution.
Given an input mesh M, its PM form consists of a coarser mesh M^0, plus a sequence of n detail records providing the information needed to iteratively refine M^0 into the input mesh $\mathrm{M} = M^n$. Basically, each record defines a *vertex split*, an elementary transformation which adds an additional vertex to the transformed mesh.
In other words, $\mathrm{PM}(M)$ defines a continuous mesh sequence M^0, M^1,..., M^n, from which LODs of increasing accuracy can be easily computed. Thus, progressive mesh forms can provide a continuous-resolution, lossless representation of a given input mesh.
In order to build the PM for an input mesh M, the algorithm applies to it a set of three

[2] A mesh is considered well-formed/*manifold* when it does not have any topological inconsistency, such as having three or more polygons sharing an edge, or two or more corners touching each other[18].

transformations (*edge collapse*, *edge split* and *edge swap*), although the *edge collapse* operation alone is already sufficient to effectively simplify the mesh.
Through a sequence of edge collapses the given mesh $M = M^n$ is transformed into the base mesh M^0. It is worth noting that, for $i = 1,\ldots, n$, the quality of the M^i intermediate approximation completely depends on the criterion the algorithm uses to select the vertices to be collapsed, and on the way it computes the attributes to be assigned to the unified vertex (e.g., position). Usually it is a trade-off between speed and output quality, chosen according to application requirements. Figure 2.1 shows a sample output of the progressive mesh simplification process.

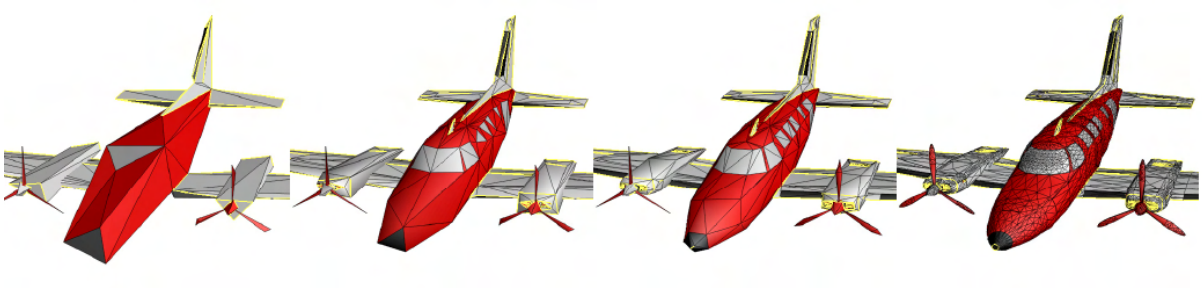

Figure 2.1: From left to right: M^0 (150 faces), M^{175} (500 faces), M^{425} (1000 faces), M^n (13525 faces)[32].

An alternative edge-contraction approach is proposed in [27], where a generalized simplification algorithm is described. It is based on iterative contraction of vertex pairs and uses quadric error metrics to assess the quality of the approximation. Unlike more traditional edge-contraction based approaches though, it is able to cope with non-manifold input meshes without the need for a clean-up preliminary phase. Also, in general it does not preserve mesh topology, because two disjoint components of a triangle mesh can potentially be joined by the algorithm. Not preserving the original topology is not necessarily a drawback, because two distinct shapes might well look like a single one if seen from a distance. However, it might not suit animated models: artifacts known as *webbing*, consisting of smears between originally unconnected parts of the model, may easily appear if topology is not preserved.
An important matter, regarding the use of progressive meshes into a networked application context (and more in general any LOD technique that requires a significant amount of data to be sent over the network), concerns the way LOD approximations

are transmitted to the recipient.

In [64], Zheng et al. propose a transmission model for progressive meshes, suitable for client-server multi-resolution rendering systems, where the server maintains the complete mesh structure, while the client only holds and renders the necessary portion of the model. To achieve interactivity despite network latency, a predictive parallel strategy is used. Client and server processes run in parallel, employing the rendering time to make up for network latency. A view-parameters prediction mechanism (depicted at high-level in Figure 2.2) is used to synchronize client and server, without imposing significant requirements in terms of extra memory. Even with long round-trip times, the system manages to overlap the network latencies for multiple frames, keeping visually acceptable rendering quality.

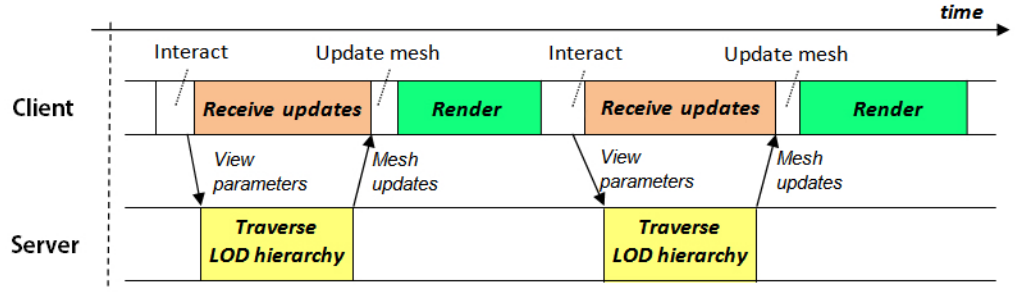

Figure 2.2: View-parameter based predictive mesh transmission system.

In [38], Li et al. describe a middleware solution for the transmission of triangle-based compressed 3D mesh representations (such as PMs) that targets progressive rendering systems, with specific focus on handling missing data over lossy networks. The middleware stands between the application layer and the transport layer and employs a combination of reliable and slower (TCP) and unreliable and faster (UDP) transmission channels, as a trade-off between delay (due to network latency) and distortion of the rendered output (due to missing data).

Although the system is probably too consuming in terms of resources to be integrated on the client side of a WebGL application, some of the key features could be integrated in an ad-hoc implementation. More specifically, the following seem desirable components for such a system:

- selection of the data to be transmitted based on client-side quality requirements,
- packet loss handling,
- monitoring of the distortion due to missing mesh data.

In [19] Cheng et al. describe an approach that specifically targets the transmission of progressive meshes. In order to allow scalability to a large amount of clients, they base the transmission on a receiver-driven protocol, where the sender is not responsible for deciding the transmission order of the data. As a first step, the server forwards the base mesh (coarsest approximation of the unsimplified model) to each client. Later, based on explicit client requests, it forwards the refinements, consisting of sequences of vertex-split[3] operations that allow the reconstruction of progressively more detailed approximations. The sending order is determined by the receiver, based on an algorithm (partly run on the GPU) that takes into consideration current visibility and expected visual contribution, in order to decide which refinements should be required first.
In [41] Maglo et al. present a view-dependent, event-based framework for the transmission of progressive meshes within a client-server web architecture. The X3D[17] XML format is extended with the information required to stream progressive LODs, as originally described in [25]. Although the proposed solution features good compression performance and produces good-quality intermediate levels of detail, it mainly focuses on the visualization of a limited number of models representing scientific data, and only takes into account a single attribute (colour) during vertex decimation. Thus, as presented in the paper, it does not allow the simplification and transmission of typical models used in games, which generally specify multiple attributes (such as normal and texture coordinates) for the vertices.

Vertex clustering

Rossignac et. al in [56] present one of the earliest mesh simplification algorithms based on vertex clustering. Vertices are clustered with respect to geometrical proximity, according to an uniformly divided surrounding grid. All vertices falling within a given cell are replaced by a unique vertex, which is chosen based on some importance metric,

[3]Recall from the past discussion about progressive meshes that the *vertex split* is the inverse of the *edge collapse* operation.

computed by averaging per-vertex weights, or by a higher order approximation (such as error quadrics).
After the clustering phase, the algorithm identifies and discards all degenerate[4] triangles. Figure 2.3 illustrates a high-level view of the algorithm. During the *Grading* step a weight table W is created and a weight is computed for each vertex. The *Triangulation* step occurs only if the input is a polygonal mesh, instead of a triangle mesh. The *Synthesis* step corresponds to the choice of the representative vertex for each cluster, while in the *Elimination* phase all degenerate triangles get discarded. Finally, the *Adjust normals* step calculates normal values for each simplified triangle.

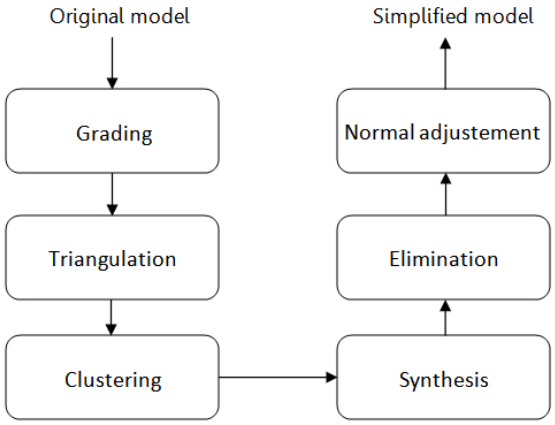

Figure 2.3: Vertex-clustering algorithm steps.

Low[39] describes a slightly modified version of the approach, where *floating cells* are employed for clustering, rather than using an equally subdivided surrounding grid. The cells are generated aggregating vertices in order of importance, as follows:

1. All vertices are sorted in non-increasing order with respect to their weight.
2. The highest-weight vertex is identified as the center of the new clustering cell; all vertices that are within the cell are replaced by a representative vertex (which is not put in the sorted list).
3. The previous step is repeated for the current highest-weight vertex.

[4]A triangle is called *degenerate* when its vertices are collinear. It is also referred to as a *zero-area triangle*[18].

The modified algorithm features better quality approximations for each LOD and preserves a larger number of important features, due to the fact that using the highest-weight vertex as the center of the floating cell greatly reduces the probability that two vertices having heavy weight get clustered together. Moreover, for each cell the maximum error, with respect to the highest-weight vertex, is always limited by half of the cell diagonal value.
Some major disadvantages of the illustrated vertex clustering techniques are:

- they do not provide a built-in mechanism to control the complexity (number of triangles) of the generated simplification;
- they do not cope well with preserving details from the input mesh;
- they are not flexible enough to allow different levels of simplification for different regions of the mesh, based on their complexity;
- they do not consider connectivity information, so they are not able to preserve input topology.

The vertex clustering approach was recently used by Willmott[63] as a starting point to develop a real-time mesh simplification algorithm, capable of handling ill-formed meshes without preliminary clean-up steps. The algorithm also features a finer-grain control over the simplification process, allowing to trade a loss of detail in flat areas for preserving more details in complex areas.

Re-synthesis

The approaches based on re-synthesis tackle mesh simplification by re-triangulating part (or all) of the mesh. As stated in [63], they typically accept only manifold inputs and are generally quite complex to implement. Also, they are significantly slower than edge contraction and vertex clustering solutions.
One of the main approaches is *vertex decimation*, as described in [58]. The algorithm applies local operations on the geometry and topology of the input mesh, iterating multiple times through all the vertices by executing the following three steps:

1. characterize the local vertex geometry and topology,

2. evaluate the decimation criteria,

3. triangulate the resulting hole.

At every iteration each vertex is a potential candidate for removal: the first step identifies the local topology and geometry of a given vertex, classifying it as:

- *simple*, where the vertex is surrounded by a cycle of triangles, with each edge using the vertex part of exactly two triangles;
- *complex*, if the vertex is surrounded as above, but an edge is not used by two triangles, or a triangle that is not in the cycle uses the vertex (in these cases the input mesh is not well formed);
- *boundary*, if the vertex is on the boundary of the input mesh;
- *interior edge*, if the vertex is simple *and* is used by two feature edges[5];
- *corner vertex*, if the vertex is simple and it is used by three or more feature edges.

Complex vertices cannot be removed. All other types are potential candidates: if a vertex satisfies the decimation criteria[6], it is deleted together with all the triangles having it as a vertex. Then a local triangulation is applied to close the hole caused by the deletion.
The algorithm stops when the termination criterion (typically the reduction goal, in terms of decimated triangles, or a percentage of the initial amount) is met.
Figure 2.4 shows a simplification output produced by the vertex decimation approach.
Another solution based on re-synthesis is described in [33], where a bounded approximation approach is used, which guarantees that the output mesh approximates the input within a predefined tolerance: every vertex v in the input will lie within a user-specified ε distance in the simplified mesh. Faces are iteratively merged, and finally replaced with a *superface* (that is, a triangulation of the edges formed with other face clusters).

[5]If the dihedral angle between two adjacent triangles is greater than a specified feature angle, then a feature edge exists.

[6]Default decimation criteria are *vertex distance to plane* for simple vertices, and *vertex distance to edge* for boundary and corner vertices. If the distance is less than some threshold value, the vertex can be deleted.

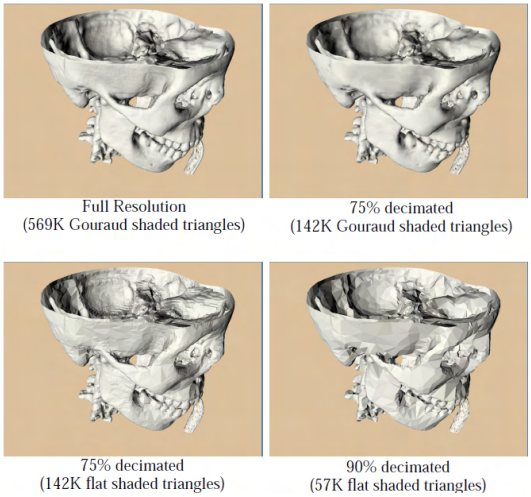

Figure 2.4: Vertex decimation simplification output[58].

2.1.2 Impostors

In their simplest form, *impostors* are 2D image textures mapped onto a rectangular card (also known as *billboard*)[18], in order to replace highly detailed geometry with a flat, quickly rendered image that gives the illusion of a complex object. Thus, the rendering effort is proportional to the number of pixels the impostor covers on the screen, instead of the number of vertices of the mimicked object. The impostor quadrilateral is opaque where the object is present, and transparent everywhere else.

Impostors are best suited to mimic static objects, especially if multiple instances of the same objects are present in the scene with similar viewing angles, allowing for re-instancing of the same impostor. Compared to using low-polygon approximations of complex objects, impostors usually provide better visual quality without high rendering costs. Another advantage is that the texture image can be processed with a low-pass filter to create an out-of-focus effect, in order to simulate depth of field.

In principle, an impostor provides an accurate representation only from a specific viewing direction, with the visual output quality rapidly degrading as the viewing angle is modified. To overcome such issue, several advanced variations of the original solution have been proposed.

In [54], Risser presents a technique called *true impostors*, where multiple depth layers representing non-height-field surface data are associated with billboards. As with the

original impostor technique, this approach performs a rotation of the impostor around its center, so that it always faces the camera. However, instead of using a static texture, it exploits the pixel shader programmability to perform ray-casting of the view vector through the billboard quad in texture-coordinate space[7], in order to intersect the 3D model and determine the colour at the intersection.
True impostors support self-shadowing, reflections and refractions. Figure 2.5 shows a sample scene where a large number of impostors is employed. In [53] *true impostors* are improved by allowing rendering of a texture volume onto the three visible faces of a box. Although more expensive than the original algorithm, this approach overcomes the usual restriction on viewing direction, achieving true volumetric rendering as well.

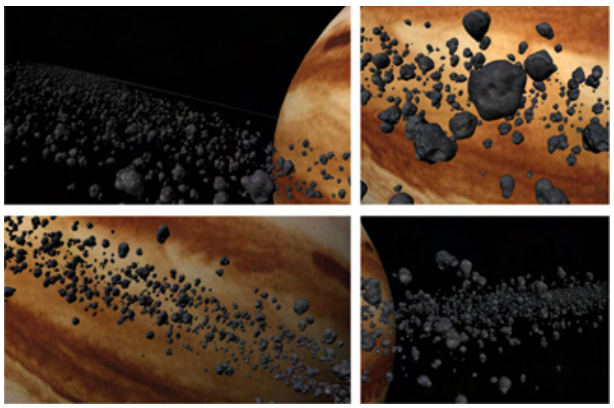

Figure 2.5: A model of Jupiter consisting of a quarter million impostors[53].

In [21], Decoret et al. propose a different variation on the basic technique, called *multi-mesh impostor* (MMI), where impostors are composed of multiple layers of textured meshes, which can be dynamically updated during visualization. The basic idea is to combine pre-generated and dynamically updated impostors into a single approach, allowing the use of cheaper pre-calculated representations where necessary, but also supporting a gradual transition to dynamic updates for better quality, when feasible within the given frame-time constraints. Moreover, this approach can bound de-occlusion errors to a user-specified amount, thus providing some sort of (coarse) control on output visual quality.

[7]Texture-coordinate space is defined by a frame with the center of the quad as the origin.

2.2 Procedural content generation

In this section, we discuss some techniques that allow us to procedurally generate game assets at runtime, thus reducing the amount of geometry data that needs to go over the network in order to display the scene on the browser.
For game content generation, a key requirement is the predictability of the generation process (e.g., in a multiplayer game we would need to make sure that the server and all clients share the same game environment), so we are interested in pseudo-random approaches that generate outputs based on a (common) seed. There is a wide scientific and technical literature on the topic, covering many different algorithms and types of content (e.g., urban and outdoor environments), some of which are designed to exploit GPU computing capabilities.
As we will detail in the following subsections, unfortunately not all recent approaches that run on the GPU are suitable for the WebGL platform. As mentioned before, several hardware capabilities, such as geometry shaders and tessellation, are not accessible via the API. Therefore, some recent procedural techniques, which for performance reasons heavily rely on these new features, cannot be at present considered viable options.

2.2.1 Outdoor environment generation

In [55] Rohleder and Netzel outline a procedural content generation approach performed on the GPU. The system combines the following components:

1. a real-time algorithm which employs fractal Brownian noise to generate and render infinite, deterministic heightmap-based terrains,

2. a thermal erosion algorithm (implemented as in [4]) to increase the realism of the generated heightmap,

3. a random tree distribution approach that takes into account information describing the previously generated terrain,

4. a semi-realistic sky model based on Rayleigh-Mie atmospheric scattering simulation.

The height map generation step relies on an underlying GPU implementation of Perlin noise[50], which is invoked whenever a new portion of the height map needs to be created. In order to do so, the terrain is split in a number of smaller nodes, with the camera initially placed at the center of all nodes. UV coordinates used for noise generation are determined based on the position of nodes in world space, and the camera direction.
To add more realism to the generated terrain, output is further processed by applying a thermal erosion pass, computed in the pixel shader in two steps. Then a normal map is generated, by applying a Sobel filter to the height map.
The following phase is tree placement, which consists of two separate steps. First a tree-density map is generated in the pixel shader, mimicking growth based on slope and height of the terrain (which are derived from normal and height maps). Then instancing is used to efficiently render trees based on the tree-density map, starting from those that are close to the camera, and imposing an upper limit to their number in order to avoid performance issues.
Lastly, the Rayleigh-Mie scattering simulation is applied, with an implementation that follows the outline provided in [49] and exploits *multiple render targets* (MRT) to achieve better performance[8].
The approach as a whole manages to produce large outdoor environments with good quality of the generated terrain, still achieving quite high frame rates on modern hardware.
In [34], Kamal et al. describe a solution for a parametrically-controlled generation of mountainous terrains. The approach focuses on generating pseudo-random terrain around distinguishing features (namely, positions with specified heights that the algorithm will try to recreate in the output), with the goal of mimicking existing geographical regions by feeding their rough description as an input to the algorithm. Although promising, it does not appear to be a suitable solution for the generation of game environments, since it is not easily applicable to more varied terrain types.

[8]Unfortunately, MRTs is another lacking feature in WebGL, so the Rayleigh-Mie approach cannot be implemented in the most effective way.

2.2.2 Procedural vegetation generation

A typical application of procedural techniques is the generation of semi-realistic vegetation in large outdoor environments. In [65] Zioma describes an approach which procedurally synthesizes believable motions of trees affected by a wind field. The technique takes advantage of GPU processing power by performing computations in the vertex shader. Instead of applying proper physical simulation methods, which are too time consuming to produce believable results in real-time, it models tree movements by means of a stochastic process.

Tree dynamics are synthesized by combining noise functions according to a set of predefined rules, while the wind field is defined as a bi-dimensional vector field, much similarly to the way fluids are defined in grid-based simulations.

For simulation purposes, each tree consists of a trunk and a number of branches composed by a single segment (that is, the branch flexibility is taken into account only as an optional rendering effect). The algorithm considers different simulation LODs, depending on the distance from the camera, because, as the viewer approaches the tree, the importance of branch movements becomes more significant than the trunk movement. Trunk animation is simulated by combining two noise functions (representing motions parallel and perpendicular to the wind direction), while branch motions are computed based on a set of rules, depending on the orientation of the branches with respect to the direction of the wind.

In order to introduce some additional variations in the movement of branches, a phase shift parameter, which is assigned to each branch in a pre-processing step, affects the function used to generate the motion. All simulation is performed in the vertex shader, by passing wind field state (as a 2D texture), bone hierarchy and tree parameters as inputs. The approach leverages hardware instancing[9] to achieve higher performance, managing to render visually believable (although not strictly physically correct) animations for large numbers of trees in real time.

[9]Note that HW instancing is not available in WebGL and an application-level implementation in Javascript is at least an order of magnitude slower. However, unofficial experimental extensions could be used to expose HW instancing capabilities[10].

2.2.3 Urban environment generation

In [29] a real-time solution for generating 'pseudo-infinite' cities is presented (which could likely be adapted to create urban game environments). Building generation parameters are created by a pseudo-random number generator, seeded on integers deriving from the building's position. All geometrical components of the city are generated as they appear in the viewing frustum of the camera, and only buildings and streets surrounding the camera position are generated and stored in memory, in order to limit memory requirements. The shape of a building is determined by its location. Since the amount of memory used by the algorithm remains more or less constant, the generated virtual city can be explored to a pseudo-infinite extent.

In [42], Marvie et al. introduce a novel rendering approach, called *procedural geometry mapping*, that combines rule-based grammars with surface mapping in order to render believable cityscapes at quasi-interactive framerates. The method performs real-time generation of procedural buildings, avoiding the need for actual geometry storage, and is based on a GPU, per-pixel lazy development of façade grammars. The approach drastically reduces memory requirements compared to more classical solutions, because the computations for objects spanning large areas are performed in image space, and as such do not depend on the actual size of the object, but only on the number of pixels it covers in the viewport.

The algorithm considers building footprints, and for each façade the bounding volume is projected towards the camera. For each pixel the split grammar is lazily developed in order to find potentially visible split rules and terminal shapes. To achieve better visual output, the approach employs normal and relief mapping to add further geometric details, and supports per-pixel self-shadowing for the façades.

Figure 2.6 shows a sample output of the *procedural geometry mapping* method. The algorithm applies four different LODs, depending on the distance of the façade from the viewer, performing smooth geomorphing between them: flat buildings (a), macroscopic geometry and shadowing (b), normal mapping (c) and relief mapping (d).

The major drawbacks of this solution are the low framerate achieved when applied to complex cityscapes (which at present makes it a less than ideal solution for real-time applications), and the distortion-due inconsistencies of the façades, arising when the approach is applied to highly-curved meshes.

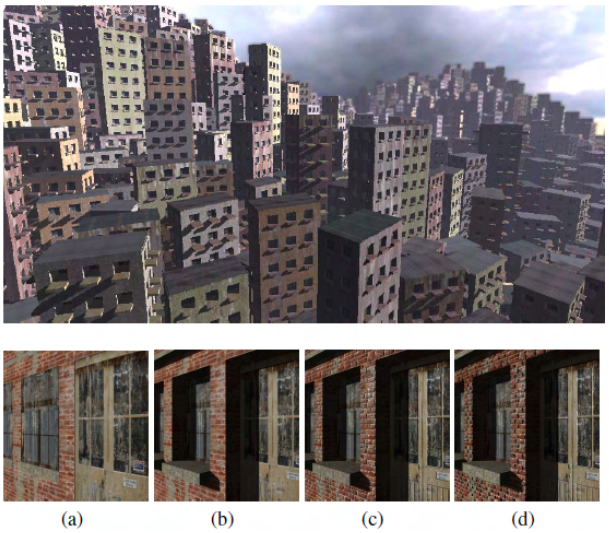

Figure 2.6: Cityscape with ≈10000 façades and the four different LODs employed[42].

Another content generation solution based on grammars is proposed by Magdics[40]. The framework is based on a context-free, parametric L-grammar, which is developed completely on the GPU in real-time. Many kinds of objects (e.g., buildings and vegetation) can be described by means of these formal grammars, and then procedurally generated at runtime. The approach also features an algorithm that effectively performs discrete collision detection against the scene, thus allowing proper interaction with the procedurally generated geometry.
As with the other procedural rendering techniques, only the potentially visible part of the scene is generated. Object instancing is performed completely on the GPU, and the solution achieves high framerates even when large numbers of objects are procedurally generated and tested for collision. Additionally, the paper shows how a grammar can be translated automatically to a shader program performing the procedural generation.

2.3 Surface mapping

In order to increase the realism of the rendered scenes, without requiring huge amounts of geometric data, surface mapping techniques may also be applied. Besides simple normal mapping, there are more advanced approaches that can be used to give the illusion of more realistic, highly detailed surfaces. The majority of the surface mapping

solutions detailed in the following are refinements of two major approaches: parallax mapping and relief mapping.
Relief texture mapping was presented for the first time in [47]. It is an extension to texture mapping, supporting the representation of 3D surface details and view motion parallax. The results are quite good for both static and moving viewpoints.
Parallax mapping was introduced by Kaneko et al. in [35], and represents the motion parallax effect on a single polygon surface by using per-pixel texture coordinate addressing. The result is a per-pixel level representation of view-dependent surface characteristics for each polygon.
The next subsections outline several advanced techniques, currently widely used in the game development industry, which improve on the approaches mentioned above.

2.3.1 Parallax occlusion mapping

As described in [23], *parallax occlusion mapping* (POM) consists of a high-precision ray-tracing intersection calculation, as illustrated in Figure 2.7.

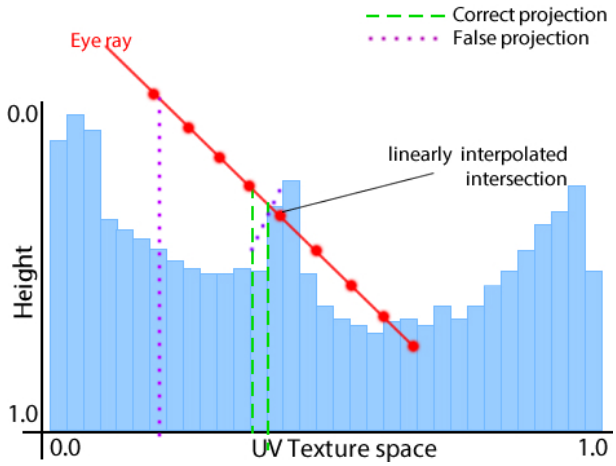

Figure 2.7: Parallax Occlusion Mapping.

The approach differs from the GPU implementation of relief mapping described in [52] in the way it determines the intersection point. After performing the linear search that discovers the points (p, r) and (k, l), defining the segment on which the actual intersection point lies, instead of using a binary search to find it, the ray is directly tested against the segment. Hence, the height profile is effectively approximated as a

piecewise linear curve.
Tatarchuk in [60] describes an advanced solution based on POM. The main features of the approach are:

- real-time per-pixel ray tracing performed on the GPU using an adaptive height field scheme, which reduces the presence of visual artifacts at oblique angles,
- estimation of light visibility for the displaced surface, which makes it possible to compute real-time soft shadows due to self-occlusions,
- an adaptive LOD control system[10], which enables the approach to control computational complexity at the fragment shader level, based on per-pixel information. The system also features smooth transitions between different LODs.

2.3.2 Per-pixel displacement mapping with distance functions

In [22], Donnelly describes a GPU implementation of the displacement mapping technique, which allows small-scale displacement on a per-pixel basis. As most recent techniques, this approach considers displacement mapping as a ray-tracing problem, and computes the texture coordinates corresponding to the point where the viewing ray intersects the displayed surface. In order to do that, a three-dimensional distance map is pre-computed, which provides distances between points in space and the displayed surface.
The main advantage, with respect to algorithms that sample the height map at uniformly spaced locations (as in [52]), is that using a pre-computed map eliminates the chance of "overshooting" the correct intersection point, which can cause visible aliasing or gaps in the rendered geometry.
The distance map of the surface is defined by means of a distance function $dist(p, S) = min\{d(p, q) : q \in S\}$, where S is the surface and p is a point in texture space, which returns the distance from p to the closest point on the surface S. In practical terms, the distance map for S is a 3D texture that stores, for each point p, the value of $dist(p, S)$. Conceptually, the algorithm follows the sphere-tracing approach presented in [30] for ray-tracing of implicit surfaces, but it applies that to the rendering of displacement maps.

[10]Note that this feature is only available on GPUs that support *texture LOD*.

The solution performs quite well even with surfaces that are generally problematic for surface mapping (e.g., displaced text): Figure 2.8 shows the output of the technique on such a case, with no visible artifacts. However, it still suffers from distortion when applied to high-curvature regions.

Figure 2.8: Per-pixel displacement mapping applied to text[22].

2.3.3 Relaxed cone step mapping

In [51], Policarpo and Oliveira improve on the GPU implementation of relief mapping presented in [52], by introducing a variation called *relaxed cone stepping*. The previous implementations of relief mapping performed ray-height-field intersection using a binary search, which refines the result produced by some linear search procedure. As already mentioned, the linear search phase is prone to aliasing, because it can "overshoot" some thin structures due to an overly large step size. To tackle this issue, the new approach employs a variation of pre-computed cone maps (described in [24]), called *quad-directional cone step maps*, which use four different radii for each fragment (one for each cardinal direction: north, west, east, south).
The approach combines together simple cone step mapping and binary search, by relaxing the restriction used to define the radii of the cones. The linear search used in previous relief mapping variations is replaced with a space-leaping approach, followed by the usual binary search. The algorithm is more efficient than simple cone step mapping because it allows wider cones (thus having faster convergence), and produces better-quality output, since it still uses the binary search to refine the result of the intersection test. Figure 2.9 illustrates a sample output of the technique.

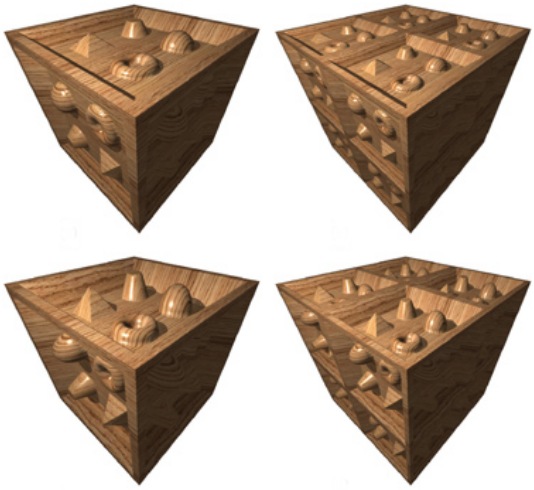

Figure 2.9: Relaxed cone stepping with different viewing ray directions and tiling factors[51].

Relaxed cone stepping has the same features of the previous versions of relief mapping, so it is able to correctly handle self-shadowing, silhouettes and intersections with non-height field surfaces. Moreover, having removed the linear search step for the more accurate cone stepping, it is less prone to artifacts, although maintaining comparable performance.

2.3.4 Quadtree displacement mapping with height blending

In [23], Drobot illustrates a novel technique that combines the strengths of the solutions outlined in the previous sections. The *quadtree displacement mapping with height blending* (QDMHB) approach specifically targets console hardware (which does not features most of the advanced capabilities available on latest GPUs[11]). The algorithm makes use of space-leaping techniques to avoid empty spaces, and exploits texture MIP levels to store the height quadtree, which can be prepared at runtime. Conceptually, it is a GPU implementation of the hierarchical ray-tracing algorithm for terrain rendering in [20], which uses heigh field pyramids, with bounding information stored in a mipmap chain.
Moreover, the approach was extended with a soft-shadows computation method that

[11]This fact should make the approach suitable for a WebGL implementation, according to the previous discussion about the capabilities of the API.

takes advantage of the quadtree structure and can also compute an ambient occlusion term. It also allows LODs in order to achieve better performance, and employs an effective surface blending method. The final result is a scalable, efficient and accurate displacement mapping of blended surfaces, which also includes ambient occlusion and soft-penumbra soft shadowing. Figure 2.10 illustrates a sample rendering output of this technique.

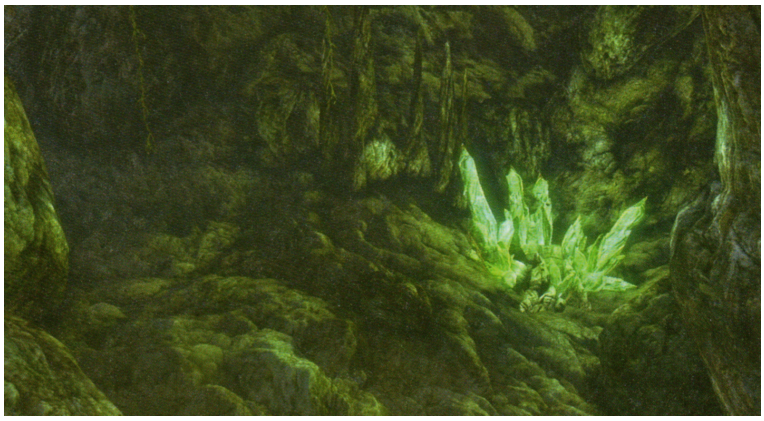

Figure 2.10: Quadtree displacement mapping with height blending[23].

According to [23], QDMBH tends to produce higher quality results than relaxed cone stepping and POM when applied to highly-detailed surfaces, with a comparable or lower amount of iterations.

2.4 WebGL frameworks

Although WebGL is still a novel technology, there have already been many attempts to wrap the exposed low-level API into higher-level 3D engines. While most frameworks are just prototypes, with very little documentation (if any), some of them offer a good set of functionalities and are quite well documented, so that they have reached a certain level of popularity among the web development community.
Based on [9], the following subsections introduce the most popular frameworks to date, highlighting their most prominent features.

2.4.1 CopperLicht

CopperLicht[2] is a commercial grade JavaScript 3D engine for creating games and applications that run into the Web browser, which exploits WebGL to render hardware accelerated 3D graphics. It is based on the well-known OpenGL engine Irrlicht, and is probably the most complete and advanced solution among available frameworks. Its major features are:

- scene graph support,
- 3D scene/world editor, known as CopperCube[12],
- skyboxes,
- billboards,
- automatic re-ordering of transparent objects at render time,
- optional 3D mesh binary compilation (which allows for reduced bandwidth requirements),
- integrated support for the most common 3D modelling formats,
- scenegraph-based 3D picking,
- optimised 3D maths library,
- collision detection and simple physics system,
- 3D skeletal animation support, with built-in importers for several formats (Milkshape ms3D, DirectX and B3D),
- cross-browser input management.

[12]Note that although CopperLicht is freely available, CopperCube is only available by purchasing a commercial license.

2.4.2 Three.js

Three.js[15] is an open-source, lightweight, cross-browser JavaScript library for creating and displaying animated 3D graphics on the browser. It offers both a SVG and a WebGL renderer, and includes the following features:

- scene graph support,
- camera system (both perspective and orthographic),
- animation support (both keyframe and morph-based),
- different light types (ambient, directional, point and spot lights),
- built-in shading models (flat, Lambertian, Gouraud, Phong),
- 3D model support for several popular formats (Blender, Wavefront OBJ, 3DS-Max),
- built-in maths library,
- ready-to-use post-processing effect shaders.

2.4.3 GLGE

GLGE[5] is a Javascript library intended to ease the use of WebGL, which masks the involved nature of the API and offers a higher-level interface to Web developers. Although the framework is still under heavy development, it already offers the following features:

- keyframe animation,
- COLLADA[1] format support,
- reflection and refraction effects,
- normal and parallax mapping,
- fog and environment mapping,
- integration with JigLibJS[7] rigid body physics library.

Chapter 3

Design

In this chapter, we provide a detailed description of the approaches that we propose to use for delivering geometric complexity to the client in real-time, without requiring the transmission of large amounts of data. In the following we focus on high-level design, deliberately leaving out the discussion of some implementation details and low-level issues, which will be the topic of Chapter 4.
The general architecture of the framework consists of a server and a set of N clients, as illustrated in Figure 3.1.

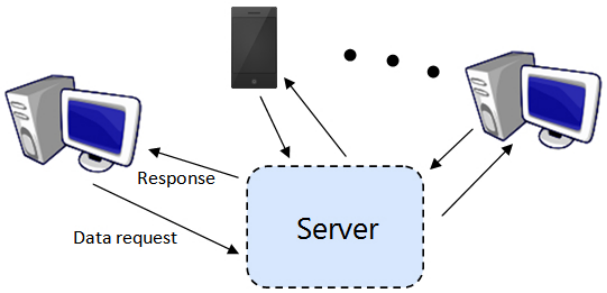

Figure 3.1: General framework architecture.

All clients render the exact same 3D scene, although possibly with different levels of detail and from different viewpoints, depending on local settings and camera position. The server is in charge of transmitting the required data to the clients on-demand, while the clients are supposed to timely request the data they need to render the scene.
The rest of the chapter is divided into three sections, corresponding to the three major contribution areas of the dissertation: surface mapping, progressive level of detail and

procedural content generation.

3.1 Surface mapping

As stated in the previous chapter, surface mapping techniques are used to give the illusion of geometric detail on surfaces that are actually described by a simple (usually flat) mesh. These approaches can increase the perceived amount of detail at the cost of some additional processing, and a small amount of extra data (typically a height map associated to the texture applied to the surface). Hence, we decided to include in the implementation two major techniques that have been successfully implemented in some commercial titles in the past few years: parallax mapping and relief mapping. Although, as stated in the background section, there exist some improvements on these techniques that produce slightly better visual outputs, these two serve well as a proof of concept of the surface mapping approach as a whole.

3.1.1 Parallax mapping

Parallax mapping was introduced by Kaneko et al. in [35], and approximates the motion parallax effect on a single polygon surface. The algorithm that we are going to describe in the following is detailed in [62], and is a slightly enhanced version of the original approach, which can be easily implemented in GLSL.
The parallax effect is the apparent motion that the areas of a non-flat surface exhibit with respect to one another, due to a change in the position of the viewer. The algorithm tries to approximate the effect by modifying the texture coordinates of each pixel, based on the corresponding surface height values.
Before delving into the algorithm itself, the concept of *tangent space* must be introduced. The tangent space consists of a coordinate system that is oriented relative to the surface: the three orthogonal axes defining it are tangent, bi-tangent (often referred to as bi-normal) and normal. While the normal is a vector perpendicular to the plane of the surface, tangent and bi-tangent lie on it. If the given surface is not flat, the tangent space will of course vary over the surface.
The concept of tangent space is fundamental to understand parallax and relief mapping, since they both require us to compute the intersection of the surface with the

vector departing from the viewer's position towards the point the viewer is looking at. In order to do that, the vector needs to be transformed into the same coordinate system of the height map associated to the surface, which is precisely the tangent space.
To be able to apply the parallax mapping algorithm to compute the displaced texture coordinates, for each pixel the following inputs are required:

- the original texture coordinates,
- the height value at the point,
- the eye vector in tangent space coordinates.

With simple texturing, when a texture representing an uneven geometry is mapped onto a flat polygon, the corresponding surface appears flattened. This is due to the fact that in general the point displayed (point A in Figure 3.2) does not correspond to the one at the intersection of the eye vector with the real surface (point B).

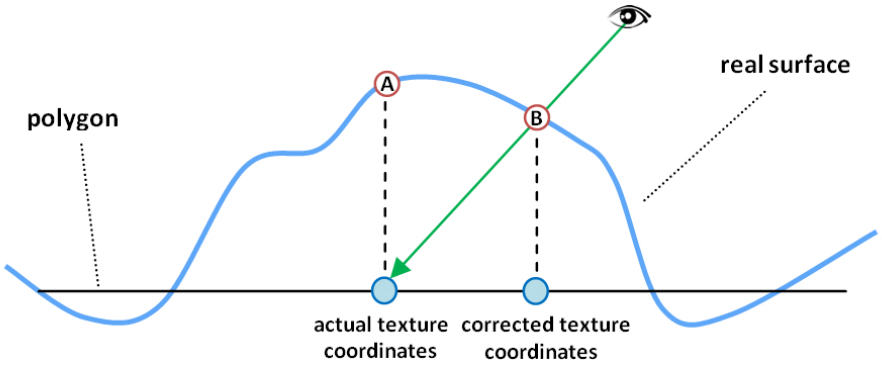

Figure 3.2: Simple texturing on an uneven surface.

The core idea at the base of parallax mapping is to correct the texture mapping, so that the coordinates correspond to point B instead of A. In general, the algorithm shifts lower areas towards the viewer, while higher areas are shifted away (thus effectively simulating the parallax effect). The amount of displacement applied to the incorrect UV coordinates is determined based on the height value at the point, as illustrated in Figure 3.3.
In order to represent height values, an extra map is associated to the regular texture, with a one-to-one correspondence between texels and height values, which are normalized in the range $[0.0, 1.0]$. Both texture coordinates and height values are in tangent

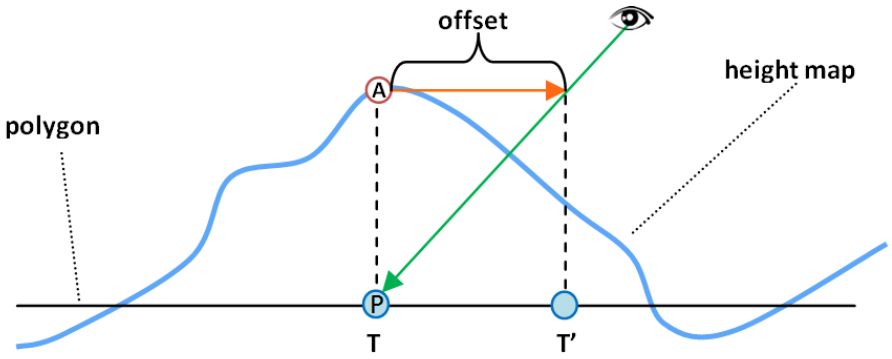

Figure 3.3: Parallax mapping: offset computation.

space, so as already mentioned the eye vector needs to be transformed into that space to correctly perform the necessary computations. The usual approach is to generate the eye vector on a per-vertex basis, by subtracting the position on the surface from the viewer position. The resulting vector is then put into tangent space, based on the rotation matrix constructed with the vertex tangent, bi-tangent and normal attributes. The transformed vector value is then linearly interpolated to be processed at the pixel-level, where it is normalized before starting to compute the texture displacement values.

The coordinate offset at a given point P is computed as follows:

1. the height value h corresponding to the original texture coordinates T is retrieved from the height map,

2. h is scaled by a factor s and subsequently biased by a term b (resulting in a new height $h_{new} = h \cdot s + b$),

3. the offset is obtained by tracing a vector parallel to the polygon, starting at the point situated directly above P on the surface, and directed towards the eye vector. The computed vector is added to the original coordinates to obtain the displaced coordinates $T' = T + (h_{new} \cdot V_{x,y}/V_z)$, where $V_{x,y}$ and V_z represent the bi-dimensional vector constructed from the x and y components of the normalized eye vector V, and its z component.

It is worth noting that the choice of the factor s and the bias term b affects the quality of the visual output and should be made considering the topology and physical properties

of the simulated surface. For surfaces featuring regular patterns (e.g., brick walls) a good value for the scaling factor is given by the average surface thickness, divided by the side of the area covered by the texture. A good value for b is typically $\frac{1}{2}s$, so that the remapped height field range does not exclude the 0.0 value, corresponding to the intersection of the real surface with the polygon used to model it. Finally, T' is used to retrieve the proper texel value for the pixel at hand.

The algorithm presented so far is the classic version of parallax mapping, which makes the assumption that the point at coordinates T' has the same height as the point at coordinates T. This is generally not true, especially for irregular surfaces. Typically, the steeper the viewing angle is, the smaller the computed offset gets, which usually makes the assumption nearly correct. However, at shallow angles the offset value tends to infinity, while the probability that T' and T index similar heights becomes in general very small. This results in artifacts and noticeable distortions in the visual output, due to totally inconsistent texel values.

A practical solution to the problem consists in conservatively limiting the offset value, so that it can never be greater than h_{new}, as depicted in Figure 3.4.

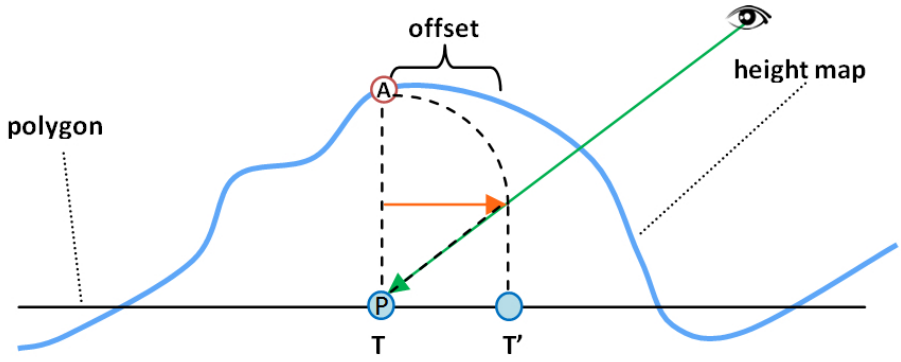

Figure 3.4: Parallax mapping: limited offset computation.

Thus, the revised equation for the offset computation becomes $T' = T + (h_{new} \cdot V_{x,y})$, which effectively limits the offset to being no greater than the height value at T. Considering that parallax mapping is itself an approximation, this solution is quite reasonable and reduces the parallax effect enough to preserve visual quality at shallow angles.

Although parallax mapping is quite an inexpensive technique, it still produces good

results. However, it has some strong limitations: besides the issue with shallow viewing angles, it is also unable to detect self-occlusion (that is, areas of the simulated surface that occlude other ones) and in general tends to produce better results when applied to height maps that have smooth gradients, while it can result in artifacts and distortions when used with steep gradients.

3.1.2 Relief mapping

Relief texture mapping was presented for the first time in [47] and is an extension to texture mapping, which supports the representation of 3D surface details and view motion parallax, by using image warping techniques. The results are quite good for both static and moving viewpoints. In the following we describe an enhanced, GPU-friendly version of the original algorithm, as presented in [52], which allows us to map relief textures onto arbitrary polygonal models.
Similarly to the previously illustrated parallax mapping, this technique uses a height map of values normalized in the range $[0, 1]$, expressed in tangent space, in order to compute a displacement of the original texture, based on the eye vector. However, due to the different approach, relief mapping can produce correct self-occlusions, interpenetrations and, with some additional processing, simple forms of shadows. Compared to parallax mapping, it also produces more consistent close-up results, usually generating less noticeable artifacts and texture distortions (if any), although at the same time producing a more pronounced relief effect.
The foundation of the algorithm is the ray-height-field intersection, which is computed on a per-pixel basis. Apart from the transformation into tangent space of the eye vector (which is required in order to perform the intersection with the height field), all other operations are performed per fragment, so that the per-vertex computations required to apply this technique are in fact the same as for parallax mapping.
At high-level, the algorithm consists of the following steps, which are performed for each fragment (note that the computation of the eye vector is performed per-vertex, as with parallax mapping):

1. compute EV (normalised, tangent-space-transformed eye vector associated to the current fragment),

2. compute texture coordinates of the point B, where EV intersects depth value

1.0, by using EV and A, the point corresponding to the texture coordinates of the fragment,

3. finally compute the intersection between EV and the height field surface by applying a binary search starting with points A, B,

4. use the texture coordinates of the intersection point to perform shading, that is: use it to index any map associated to the fragment (diffuse map, textures, ...).

The general idea is illustrated in Figure 3.5.

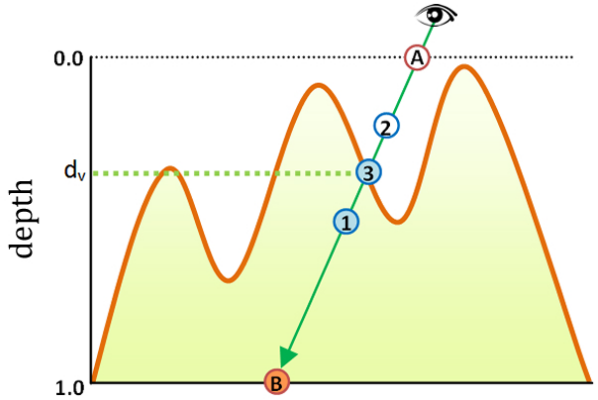

Figure 3.5: Relief mapping: ray intersection with a height field using binary search.

A depth of 0.0 and 1.0 are associated to point A and B, respectively. Every step, the current interval is split in half and the averaged depth and texture coordinates of the endpoints are assigned to the midpoint (which for the first step corresponds to point 1 in the illustration). The computed texture coordinates are used to access the depth map and determine whether the point is inside the height field surface (that is, the computed depth is lower than the stored depth) or not.
The search goes on, with one endpoint outside the surface and one inside it, until an iteration limit is reached. In [52], eight iterations are suggested as a satisfactory trade-off between the quality of the computed intersection and the processing cost required to determine it. In practice, that corresponds to subdividing the height field in 2^8 equal-size intervals.
As it is, the binary search may produce an incorrect intersection approximation. As shown in Figure 3.6, the issue arises if the ray intersects the height field surface in

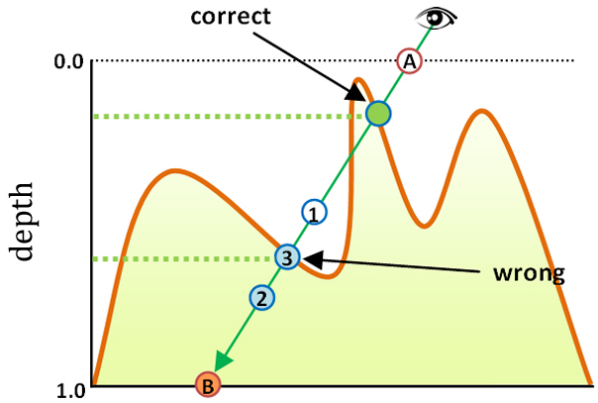

Figure 3.6: Relief mapping: overshooting issue with binary search.

more than one point. In this example, point 1 is the first midpoint calculated between A and B and, since it stands above the height field surface, the binary search would go deeper, although the ray has already pierced the surface before. As a result, it would completely overshoot the correct intersection, and finally return point 3 as the intersection point.

Fortunately, the problem can be efficiently overcome by coupling the binary search with a preliminary linear search, which starts from point A and samples the AB segment at regular intervals of configurable length δ, as illustrated in Figure 3.7. The search

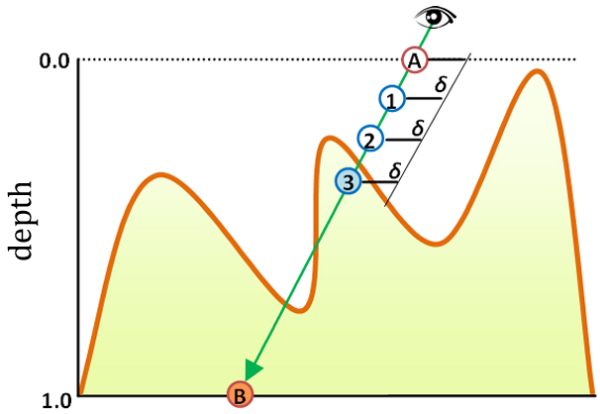

Figure 3.7: Relief mapping: linear search.

stops as soon as it finds the first point situated under the height surface. The value of δ should vary on a per-pixel basis, as a function of the angle between the transformed

eye vector EV and the interpolated per-pixel normal value. As the angle increases, δ decreases, thus effectively forcing a finer grain linear search in order to avoid missing possible intersections.
Once the first point under the height field surface is found, the binary search is performed using that point and the one previously sampled by the linear search (which by definition stood above the surface). Since the sampled interval is smaller, a lower number of binary-search steps is generally enough to achieve the same results as with binary search alone.

3.2 Progressive Level Of Detail

As mentioned in Chapter 2, progressive rendering and transmission of 3D models do not represent a novel research topic. While new ways of efficiently transmitting large 3D models are still actively researched for the purpose of visualizing remote scientific data, polygon simplification does not undergo a lot of research anymore.
The reason is that for most applications the increased GPU processing power now allows us to simply push large amounts of geometric data on the graphics processor, without noticeably affecting the rendering frame rate.
However, simplification still makes sense in some cases:

- most mobile devices still do not have fast dedicated graphics processors, so they cannot yet afford to draw high amounts of polygons at acceptable frame rates,
- browser-based 3D applications (such as WebGL and Flash games) can leverage HW acceleration to render the 3D scenes, but with respect to downloading the models on-demand they are limited, by available network bandwidth, in the amount of data they can feed to the GPU without interrupting the rendering flow. If waiting for possibly long times to pre-cache required data is not an option, this issue severely limits the complexity of the scene that could be rendered in real-time, especially if we consider average mobile connection bandwidths.

Therefore, we propose to reduce the amount of data that is sent over the network, by implementing an automatic simplification approach, run on the server side, which produces a sequence of Level Of Detail (LOD) approximations of decreasing complexity

and size. Such LODs can then be requested by clients on demand, based on their own requirements.

As described in Section 2.1, several different simplification approaches have been proposed in the past. Since we are focusing on simplifying and transmitting typical game models, the chosen approach should exhibit good results when applied to models having reasonably low polygon counts, and should take into account not only vertex positions, but also common vertex attributes (e.g., normals, UV mapping). It must also allow progressive transmission, so that the client can start rendering a model as soon as it gets its coarsest LOD.

For such reasons, we decided to implement a general polygon reduction approach known as *vertex contraction*. In this category, the two most known techniques are *progressive meshes*, originally presented by Hoppe in [31], and the Garland-Heckbert simplification algorithm, introduced in [26]. Both were originally developed to simplify models without vertex attributes, and later extended to cope with attribute simplification, under certain assumptions, in [57] and [27] respectively.

In the following we provide a high-level description of the solution implemented in the prototype, which is based on the basic idea of the Garland-Heckbert algorithm, although it uses an alternative metric to drive vertex contraction, and a different approach to cope with vertex attributes.

The original algorithm is founded on iterative vertex contraction and can be applied to manifold, as well as non-manifold meshes. It contracts couples of vertices (that is, it makes a single vertex out of the two) based on a cost metric. It is worth noting that the algorithm can potentially contract any couple of vertices (V_1, V_2) in the mesh, provided that they form a *valid pair*, where (V_1, V_2) is a valid pair if $|V_1 - V_2| \leq \tau$ for some threshold value $\tau \geq 0$, or there exists an edge with V_1 and V_2 as endpoints. Since disjoint components of the input mesh could be joined during the contraction process, the algorithm does not necessarily preserve mesh topology.

Although contracting vertices that are not connected by an edge is not necessarily a drawback, because distinct features of a mesh could well look like a single one from afar, it can potentially generate a so-called *webbing* effect, which consists in having features which would clearly be separated in the original model, but result somehow "glued" together in the simplified version. The effect is particularly noticeable for animated models.

3.2.1 Vertex placement

Given a valid couple of vertices (V_1, V_2), Garland and Heckbert in [26] describe several ways of computing the position where the vertex V_s, which replaces them, should be placed. These are:

- *subset placement*: one simple and fast solution consists in selecting V_1 or V_2, based on which one of the two has the best metric;
- *midpoint placement*: the contracted vertex position is determined as $V_s = (V_1 + V_2)$ / 2. This can be considered an alternative to subset placement, or used in conjunction with it as an additional option, to be used if the averaged position exhibits a better metric than the two endpoints;
- *optimal placement*: a more expensive choice is to compute a position that would produce the best possible cost metric for the substituting vertex. Depending on the function defining the cost metric, this problem may not admit a solution or, if the solution exists, it may not be unique. For the metric suggested by Garland and Heckbert, the problem is reduced to inverting the matrix defining the error at the vertex. If it is not invertible, the algorithm tries to find a point minimizing the metric along the segment (V_1, V_2), resorting to midpoint/endpoint selection if that attempt fails.

For the actual implementation, we decided to apply subset placement, because it is faster than the other options (since no additional costs are computed in order to choose placement) and it is also more amenable to progressive transmission (see Section 3.2.5 for details).

3.2.2 Mesh simplification algorithm

The approach we propose is a constrained version of the general Garland-Heckbert algorithm, in the sense that:

- the input mesh is assumed to be manifold,

- the τ value is set to 0, so that only *edge contractions* (also referred to as *edge collapses*) are allowed,
- the input mesh is assumed to be triangulated, in order to make the implementation of the algorithm a bit simpler[1].

The concept of edge collapse is depicted in Figure 3.8.

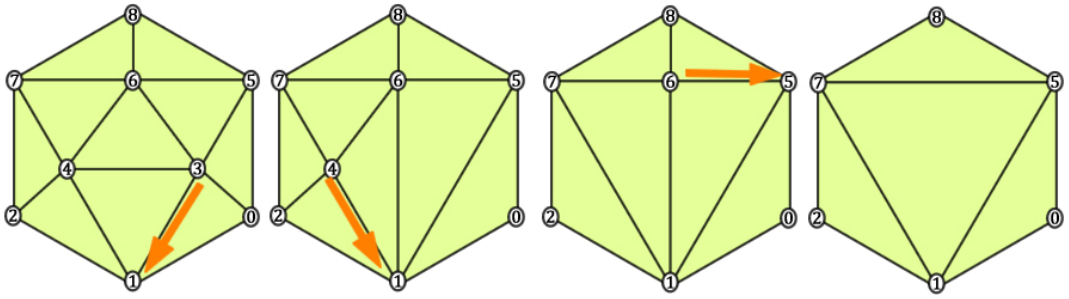

Figure 3.8: Mesh simplification: a potential sequence of edge collapses.

First, edge $(3, 1)$ is selected for contraction (that is, vertex 3 is thrown away, and substituted with vertex 1 as an endpoint in every edge it was part of). Then, edge $(4, 1)$ is collapsed, and finally edge $(6, 5)$ is. Typically, at each collapse one vertex, two faces, and three edges are removed.
From a high-level perspective, the algorithm processes a given mesh M_0 and performs a series of n edge contractions, which results in a sequence of meshes $M_0, M_1, \ldots, M_n$, where the generic M_i differs from the previous mesh (if any) only for a single edge contraction.
The simplification algorithm can be divided into the following steps:

1. Compute the cost metric for every edge in the mesh.
2. Select the minimum cost edge (u, v) and collapse it. The process consists of three distinct parts:

   2.1: remove any triangles that have both u and v as vertices (that is, remove any triangle on the edge (u, v)),

[1]Note that in practice this does not represent a serious limitation, because a non-compliant mesh can be easily triangulated with the help of any 3D modelling tool, or in a pre-processing step.

2.2: adjust mesh topology, by updating the remaining triangles that have u as a vertex to use v instead,

2.3: remove vertex u.

3. Recompute the cost metric for the edges that have been affected by the collapse.

4. Go back to step 2 until there are no more collapsible edges, or the desired level of simplification for the input mesh has been reached.

Figure 3.9 shows the output produced by applying the approach to a sample input mesh: the model on the left is the original unprocessed mesh, while the other two are its 50% simplification and 20% simplification, respectively.

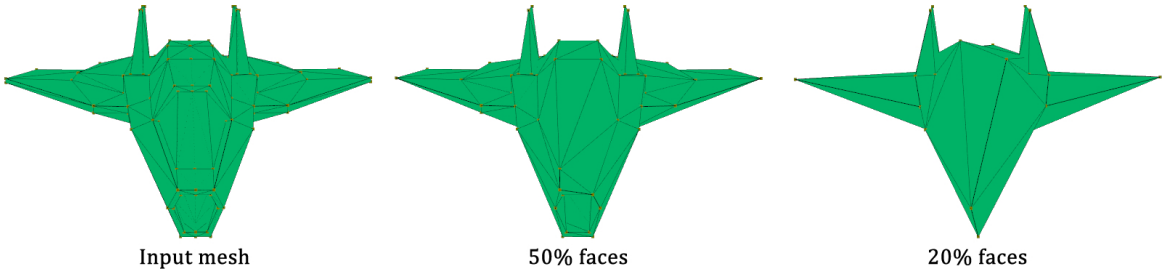

Figure 3.9: Mesh simplification: sample output.

3.2.3 Cost metric

So far we have described the algorithm, and mentioned that a cost metric is used to determine which edge should be collapsed, but we have not detailed the metric itself. Past research literature offers plenty of cost metrics that have been explored in association with edge contraction techniques, some of which produce very good results in terms of visual quality of the approximations and mesh topology preservation. However, most of them are computationally quite expensive and, most significantly, target the simplification of models with a high polygon count.
As previously stated, our goal is to achieve good quality simplification of relatively low -polygon models, so we opted for a cost metric proposed by Melax in [43], which we believe to be better suited for the task and comparatively faster to compute.

The metric tries to measure the visual change that each edge contraction would introduce in the mesh, by means of a comparison involving face normals and edge lengths. The heuristic tends to preserve detail in high-curvature surfaces, simplifying first nearly co-planar areas, which are likely to maintain similarity to the original, even when described with a reduced number of faces.

Triangle normals are computed with a commonplace approach, by taking the cross product of two edges of the triangle[2]. Therefore, for a triangle $< p1, p2, p3 >$, if we use the vector $U = p2 - p1$ and the vector $V = p3 - p1$, then the normal is computed as $N = U \times V$.

The following formula determines the cost of contracting a given edge (u, v).

$$C[(u,v)] = \|u - v\| \times \max_{f \in T_u} \{ \min_{h \in T_{(u,v)}} \{(1 - f.normal \cdot h.normal) \ / \ 2\}\} \tag{3.1}$$

Basically, the cost of collapsing an edge is defined as a curvature term, multiplied by the edge length, so that the cost of the collapse grows proportionally to the length of the edge. The curvature term is computed by comparing the scalar products of face normals, in order to determine the triangle adjacent to u that faces furthest away from the other triangles along edge (u, v).

It is worth noting that the proposed metric considers edge contractions to have a direction, which means that contracting edge (a, b) does not necessarily have the same cost as collapsing edge (b, a). The lower the cost for an edge, the smaller the expected visual change generated when contracting that specific edge.

The metric works reasonably well in most cases, usually computing costs that put off the simplification of corner vertices until no other candidates are available. This fact is generally positive, because it forces simplifications to preserve the overall original shape of the model and its symmetries.

However, in order to try to enforce shape and feature preservation even more, we introduced an additional check on top of the original metric, which can be easily disabled on a per-mesh basis if, for some specific inputs, the visual output quality happens to be better without it. The extra check is applied during the cost computation step: whenever a new potential minimum cost is computed, before updating the best contraction

[2]Note that, since the order of the vertices employed for the computation affects the direction of the normal, this has to be selected consistently all over the mesh.

candidate, the normal attributes of the edge endpoint vertices u and v are compared. The idea is again to use normals to measure the amount of visual change introduced by the collapse: more specifically, if the angle between u and v normal attributes is greater than 90 degrees, the edge is discarded as a possible candidate. This can be inexpensively checked by verifying whether or not the scalar product between the two vectors is less than zero: if that is the case, than the angle is greater than 90 degrees. With the additional check, the algorithm typically manages to preserve more distinctive features of the input model, at the cost of some extra computations, since the normal attributes of the involved vertices need to be computed when the check is performed (see Section 3.2.4 for details about a common way to compute such attributes). Figure 3.10 illustrates the point: with the additional check enabled, the overall shape of the ship is better preserved at the lower levels of detail.

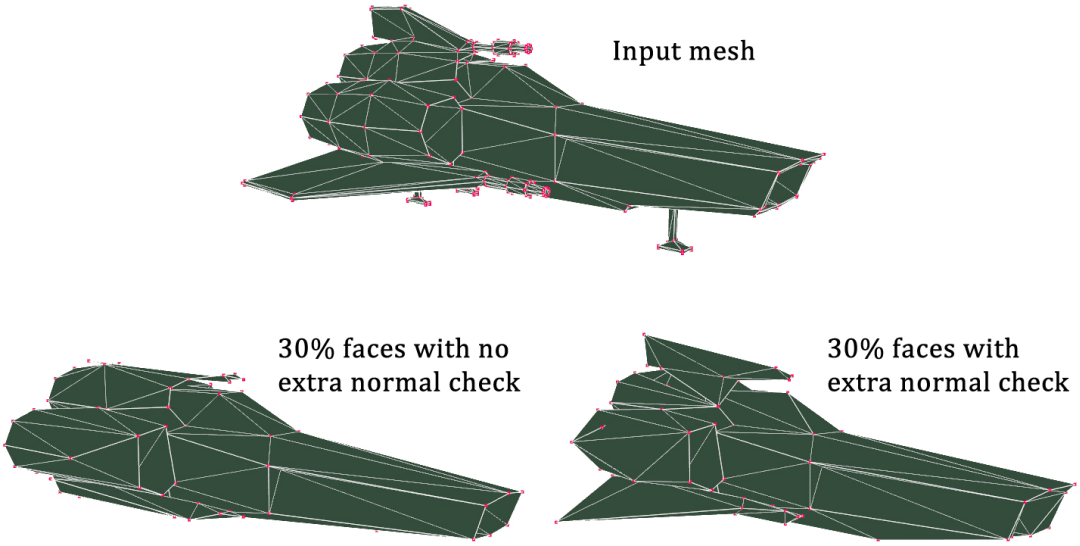

Figure 3.10: Mesh simplification: sample output with and without the extra check on normals.

3.2.4 Surface attributes

The simplification approach that was described in the previous sections does not take into account any vertex attribute, save for its position. Hence, its applicability is quite

restricted, because in general the 3D models used in a game specify a set of attributes for each vertex, typically consisting at least of normal vector and UV coordinates. Therefore, we decided to extend the approach to be able to cope with these two commonplace surface attributes.

With respect to vertex normal attributes, there are at least two possible strategies that the simplification approach can adopt to handle them. Assuming that the loaded model defines the normal attribute for every vertex in the mesh, a first possible attempt would be to simply retain the value associated to each vertex, without modifying it, through the entire simplification process (that is, each vertex that is not removed from the simplified mesh keeps its original normal value, unconcerned of any change to the mesh topology caused by the simplification process).

Unfortunately, this strategy is very likely to generate lighting artifacts, since vertex normal values will not in fact reflect the actual face normals in the simplified LODs.

To avoid, or at least mitigate this issue, an alternative approach consists in disregarding vertex normal attributes (if present) when the input mesh is loaded, and recalculating them during the simplification steps, using the updated triangle topology. Since all face normals are consistently updated during the simplification process in order to compute the cost metric, this strategy implies only a small additional cost: for each vertex, the normal is computed as the normalized sum of the normals of the triangles the vertex is part of, as expressed by the following equation.

$$V.normal = \| \sum_{f \in T_V} f.normal \| \tag{3.2}$$

As regards UV mapping, the problem is quite hard to solve in the general case. Hence, we focused on trying to achieve acceptable results, assuming that the mapping is used to index a single texture. Unlike the vertex normals, the UV mapping cannot be discarded at loading time, since the information would be lost and could not be consistently replaced. It is important to notice that in general the mesh could exhibit some degree of vertex position replication, because there may be couples of vertices V and U that are defined by the exact same position coordinates x, y, z, but differ for their UV mapping.

If the algorithm is run as it is, with the only difference that each vertex has an associated UV mapping attribute, the results are pretty disappointing. The algorithm works

as expected if the input does not contain any position replication (that is, the UV coordinates are continuous over the surface), but it produces very poor approximations otherwise. Figure 3.11 illustrates the results.

Figure 3.11: Mesh simplification: flawed UV-mapped output.

The illustration on the left represents the original mesh, whereas the image on the right is a 70% LOD approximation. It is immediately clear that the vertex replication is causing problems to the algorithm, which is generating "holes" in the mesh.
Therefore, we extended the algorithm to act slightly differently when applied to meshes with UV coordinates specified. Basically, each time an edge (u, v) is collapsed, so that vertex u is "moved" onto vertex v, the algorithm also checks which vertices share their position with u, and update their spatial coordinates with the position of v, leaving the UV mapping untouched. With this single modification, the approach becomes able to cope with texture coordinates in a much more effective way.
The idea, although conceptually quite simple, works reasonably well on meshes exhibiting a majority of high-curvature surfaces, as the one illustrated in Figure 3.12.
However, the simplification still generates some holes in models having a more geometrical shape, especially at the lower approximation levels, or when the input mesh has already a very low initial polygon count.
Overall, it is worth noting that lower level simplifications are supposed to be used when the lack of detail should be hardly visible, at least up to a certain extent. Thus, even though depending on the input mesh some holes may occur in the lower-detail approximations, it may still be possible to employ them, since there is a good chance that the imperfections will go unnoticed because of their relatively low screen coverage.
As the reader may have noticed, we have not mentioned at all the possible application

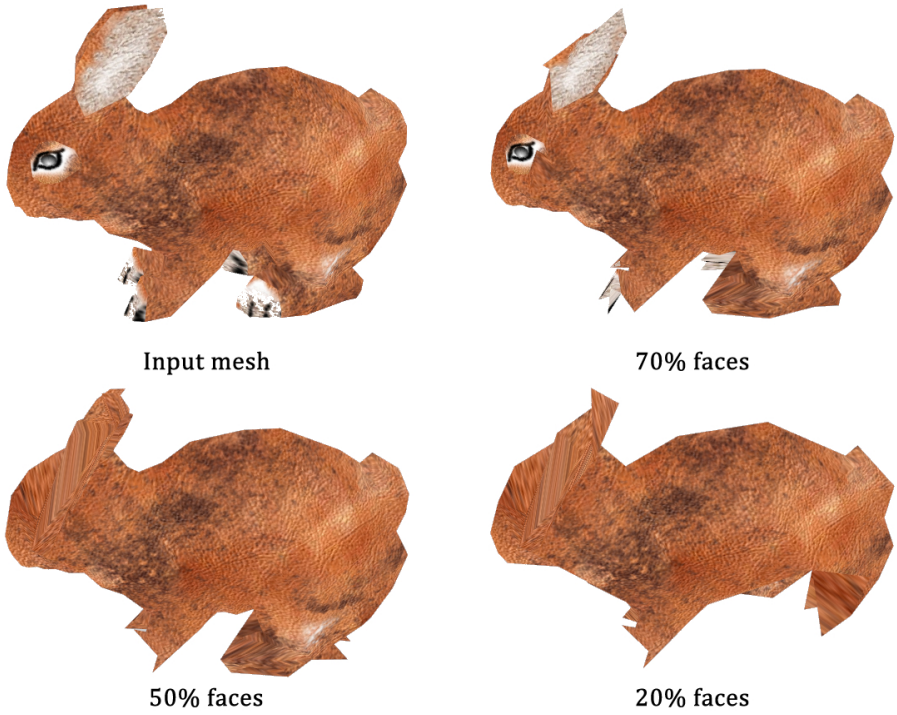

Figure 3.12: Mesh simplification: corrected UV-mapped output.

of the simplification algorithm to animated meshes. Since the problem is already quite difficult to solve when dealing with simple attributes, such as normals and UV coordinates, we decided to leave the simplification of animated meshes as a possible research direction for future work (see Chapter 5 for additional information).

3.2.5 LOD selection and transmission

So far the discussion of the mesh simplification approach has been centred around the algorithm itself and its possible extensions. However, generating the LODs for a given input mesh is only part of the problem at hand. The other main issues consist in LOD selection and its transmission from the server.

One problem that needs to be mentioned with respect to the rendering of LOD sequences is the phenomenon called *popping*, which is basically a noticeable, abrupt change in the appearance of the model, happening when different LODs are swapped. The greater is the change in shape between the swapped levels, the more noticeable becomes the effect. In order to mitigate the popping effect, *morphing* can be applied:

with this approach, the vertices of one LOD are mapped to the next one, in order to create a smooth transition over a short amount of time. In the case of the simplification algorithm, the information needed to perform the morphing may be extracted from the edge collapse sequence.
Although the selection of the level of detail could potentially be performed on the server, it makes more sense to let the client decide which LOD should get displayed, based on some kind of client-side metric, updated every time the 3D scene changes. Since the focus of the dissertation with respect to progressive rendering is on LOD generation rather than LOD selection, which is itself quite a broad and complex topic, the following discussion will only provide a brief overview of the matter.
Given a sequence of LOD approximations for an object, some of the most prominent factors that could drive the selection are:

- client rendering capabilities (that is, graphics hardware limiting the amount of geometry that can be rendered with acceptable frame rates),
- object distance from the view point,
- camera field of view[3] (FOV),
- screen-space coverage of the object.

The first one is somewhat orthogonal to the others: if the scene cannot be smoothly rendered, then the level of detail should be decreased until an acceptable frame rate is achieved, regardless of other considerations. Assuming the GPU can handle the load, the remaining options are part of typical metrics used to determine the minimum level of approximation required for each model.
The distance between the model and the viewpoint/camera can be quickly computed as the magnitude of a simple vector subtraction if we consider a fixed point of the object, such as its geometric centre, as a reference. Otherwise, assuming that the object is a convex polyhedron, the distance of its closest point to the viewpoint could be found by using one of several, relatively more expensive techniques, such as JGK.
The viewpoint-object distance is then compared to a sequence of thresholds to determine which level of detail should be selected (where the coarsest level is associated

[3]Field of view is defined as the extent of the observable game world that is displayed at a given moment.

with the farthest range, then the second-coarsest, and so on).
However, as described in [36], this is a simple but not optimal way to select the level of detail, because it does not take into account the field of view of the camera: if the FOV is very narrow (that is, the camera is zoomed), then the object could appear larger on screen even if it is distant from the camera, so that selecting a coarse LOD would be wrong.
Another issue associated to the use of basic distance thresholding is the fact that, if the model repeatedly crosses back and forth the threshold line within a short time span, the constant switch between levels of detail is likely to produce a noticeable flickering effect.
Instead of the simple distance from the camera, a better, related metric is the screen-space coverage of the object (that is, its projected size on the display), which takes into account both object-viewpoint distance and camera FOV.
More exactly, the *magnification factor* of the object is used, which is defined as the ratio between the screen size of the object and its physical size: the higher the value, the more detailed the model should become. Introducing screen size into the metric accounts for both object-viewpoint distance and camera field of view.
The magnification factor M is derived by transforming object position in view space, and then computing

$$M = scale_x \ / \ view_z \tag{3.3}$$

where $scale_x$ represents the scaling parameter used in the projection equation:

$$screen_x = (view_x \times scale_x) \ / \ view_z + center_x \tag{3.4}$$

The $view_z$ measurement, which accounts for camera distance, is expressed in world units, while $scale_x$ is measured in pixels, as it is related to camera FOV. Thus, M is defined in pixels per world unit. When M increases, more pixels are used per world unit, because the object becomes relatively larger on screen, thus a more detailed LOD should be used.
With respect to the flickering effect that happens when using a single threshold value for comparisons, a possible workaround comes from using hysteresis thresholding, which consists in thresholding against a range of values. Basically, as long as the compared value falls within the range defined by the lower and upper thresholds corresponding

to the current LOD, the selection does not vary. This approach mitigates the flickering issue, since toggling between levels of detail is less likely to happen.
As regards the transmission of the LODs to the client, in the perspective of sending as little data as possible over the network to reduce the time needed to fetch model approximations, it makes sense to build on the client side a simple caching system for the LODs. At the cost of the extra memory necessary to cache them, the client has the benefit of downloading each LOD only the first time it is needed, while every subsequent request will be served directly from the cache.
Moreover, it is worth noting that there are two possible ways to deliver the actual data from server to client:

- to transmit each LOD as a separate entity (that is, like it was a stand-alone 3D model),

- to send only the "difference" with respect to the previous, coarser LOD. This obviously implies that at least the very first transmission for a given model consists of a whole LOD (possibly the coarsest one). The idea is to use the edge collapse ordered sequence to perform a sort of progressive transmission. The information required to perform such a transmission can be extracted and saved at simplification time on the server, by storing the vertex to which each vertex is contracted, and sorting the vertices by the order in which they were collapsed. Thus, when a more detailed level is required, in order to allow the client to reconstruct the LOD from the one it has already got, only the extra vertices and the data required to reconfigure the mesh topology (that is, the indices defining the triangles) need to be sent.

3.3 Procedural Content Generation

Procedural content generation (PCG) refers to the process of generating (game) contents algorithmically. It has been used in game development for a long time, mainly with the following goals:

- *re-playability*: an element of (apparent) unpredictability is introduced into the game content, so that the user could potentially play again the same part of

the game without having the same experience (e.g., secondary features of a level could be procedurally generated in a platform game);

- *pseudo-infinite environment generation*: the environment is procedurally generated to allow the player to explore extremely large (possibly boundless) environments, which 3D artists could not generate by hand within reasonable amounts of time.

Examples of mainstream applications of PCG are: *dungeon generation* (widely used in role-playing games, such as *Diablo*), *terrain generation* (commonplace in many real-time strategy titles, such as *Civilization*) and *rule-based vegetation generation*.
Although technically not restricted to that, game PCG is almost always associated to some kind of pseudo-random number generation, which is the source of the variability in the generated contents.
It is worth noting how a *seedable*, good quality pseudo-random number generator[4] (PRNG) were to be preferred to truly random generators, since it allows repeatability of the output. The term *seedable* denotes the possibility of feeding the generator with an input string (the *seed*) at creation time, which determines the sequence of pseudo-random outputs.
Repeatability is important for several reasons. First, it simplifies a lot the debug of procedural content generators. Moreover, in a multi-player environment, it allows to generate the exact same content on every client, which is typically a mandatory requirement for playability reason. Hence, from now on the discussion will assume the use of a seedable PRNG whenever the generation of pseudo-random numbers is required.
In the following section we describe an approach that allows real-time procedural terrain generation. The proposed solution, although limited to the generation of a specific type of game environment, demonstrates how PCG techniques can be applied to reduce the amount of data required to render the environment.

[4]Here, by *good quality pseudo-random generator* we refer to generators that do not exhibit recognizable patterns in their pseudo-random output.

3.3.1 Procedural terrain generation

Terrain generation is arguably the PCG application that has undergone the greatest amount of research to date: many different real-time solutions have been proposed, some of which are entirely performed on the GPU. Unfortunately, as already stated the WebGL API does not yet expose some of the most recent GPU capabilities, and this fact rules out some interesting techniques, such as [28].
However, in the following we describe an approach that can perform real-time terrain generation on a WebGL client, is parametrizable and allows output repeatability.

Fractal terrain generation

As described in [48], most terrain generation techniques, including the one implemented in the prototype, use some form of $1/f$ noise, which is also referred to as *pink* or *flicker noise*. This kind of noise allows us to generate surfaces that exhibit fractal behaviour, which resembles to a certain extent the morphology of natural terrain. Formally, it is characterised by the following formula:

$$S(f) = \frac{1}{f^a}, \tag{3.5}$$

where $S(f)$ is the power function of frequency, known as *spectral density*, and a is a value close to 1. There are many different methods to generate pink noise, of which the most commonly used are:

- *Spectral synthesis*: $1/f$ noise is simulated by accumulating a certain number of layers (called *octaves*) of noise, each one contributing on a single frequency. The octaves are created by producing evenly spaced pseudo-random numbers, corresponding to each octave's frequency, and filling in the remaining values by interpolation. The performance and quality of the output of spectral synthesis depend significantly on the underlying PRNG.

- *Midpoint displacement*: the value of each noise cell is computed in a single pass, by applying a *divide and conquer approach*, which recursively computes values that are half-way between already known cells, by averaging values from those cells, and then pseudo-randomly offsetting the result inside a range that depends

on the current level of recursion. Ideally, the offsets should have a Gaussian distribution inside the range. In practice though, uniformly distributed values are faster to compute and still provide acceptable results.

- *Simplex noise* (which is a revised version of classic Perlin noise, as described in [50]): this approach generates high-quality noise (that is, with no noticeable directional artifacts) and is quite fast, as it is amenable to a GPU implementation in the fragment shader. However, since it relies on look-up tables to generate pseudo-randomness, seeding the algorithm is not as simple as with a fairly standard PRNG, and would also require a larger amount of data to compose a seed.

Based on the considerations above, we decided to use midpoint displacement to generate the fractal terrain. More specifically, we implemented it by using the diamond-square algorithm, which represents a good trade-off between generation speed and output quality.

Diamond-square algorithm

The diamond-square algorithm, as described in [48], takes a bi-dimensional grid as input, which, for the sake of simplifying the implementation, should be square, with dimension $(2^N + 1) \times (2^N + 1)$ (the plus one is required to have the midpoints placed exactly on the grid intersections). The only height values that need to be initialized in the input are those of the four corners, which can potentially be zero as well.
Figure 3.13 graphically illustrates the basic idea of the algorithm, for a small 5×5 2D grid.

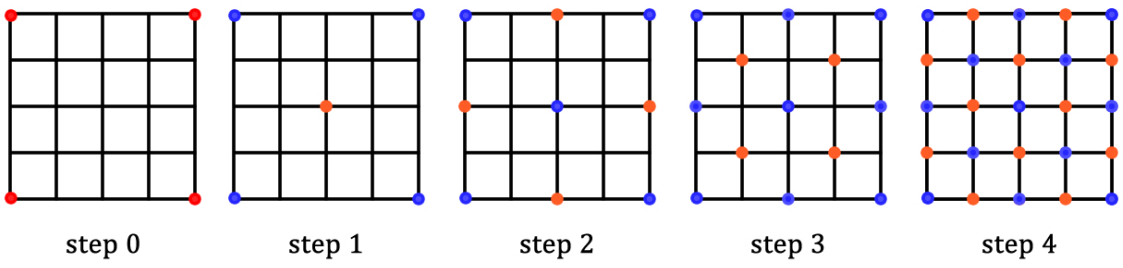

Figure 3.13: Diamond-square algorithm iterative subdivision.

In the preliminary step, the four corner values (highlighted in red) are initialized.

Then at step 1 (the *diamond* step) the midpoint value (highlighted in red) is computed, by averaging the four corners (highlighted in blue), and offsetting the result by some pseudo-random value in the range $[0, rangeScale]$.
During step 2 (the *square* step) the remaining midpoints (highlighted in red) are computed, averaging the corresponding four corner values and offsetting the average by a pseudo-random value in $[0, rangeScale]$, as done in the previous step. Note that the grid is assumed to wrap around, in order to enable the selection of corner points for the midpoints that are located on the borders.
Step 3 and 4 illustrate the next iterations, which fill in all remaining uninitialized points.
The *rangeScale* parameter is quite important, because it controls the amount of variation in height that the terrain is allowed to have at a given point. In order to get more consistent results, it is generally varied dynamically during the algorithm execution, by halving its value at the start of each grid subdivision.
The number of subdivisions performed by the algorithm is obviously equal to $\log_2(M-1)$, where M is the edge length of the grid. Conceptually, the higher the number of iterations (and therefore the larger the grid size), the finer the grain of the surface tessellation. Figure 3.14 illustrates the increasing complexity at different subdivision stages: after the first pass, only nine points have been computed and the corresponding wireframe surface is extremely coarse, but after only five passes, the surface is already composed of a high number of triangles.

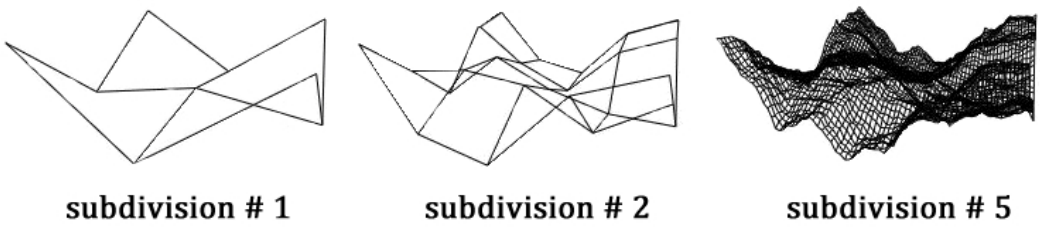

Figure 3.14: Diamond-square algorithm: tessellation granularity.

A good trade-off value for the number of subdivisions is 8, which corresponds to a dimension of 257×257 for the grid. An example of terrain surface generated with the suggested size is shown in Figure 3.15.
With that value, the surface is complex enough to effectively simulate a terrain. More-

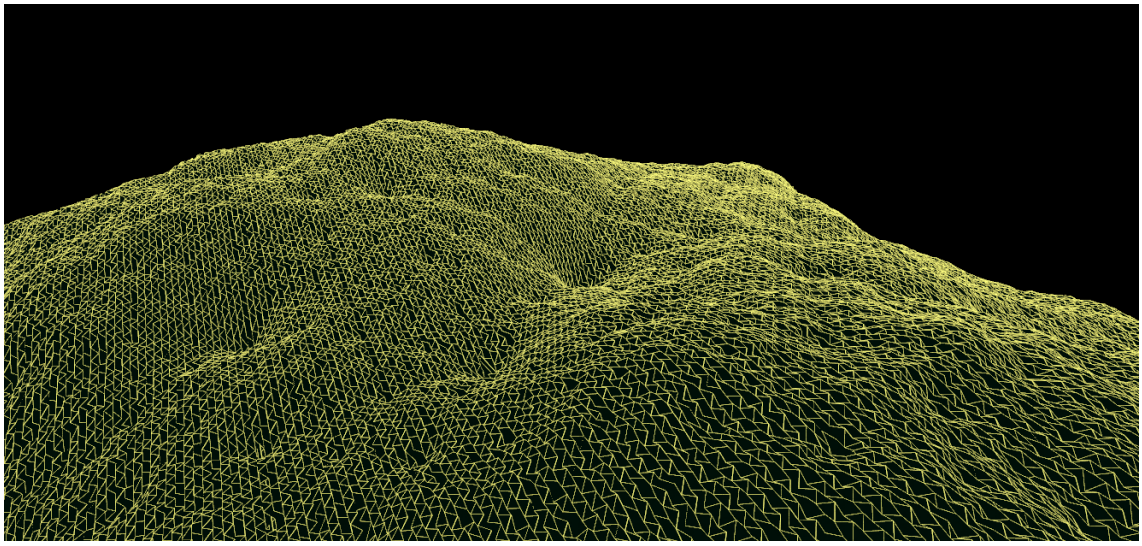

Figure 3.15: Procedural terrain generation: wireframe example output.

over, a larger dimension, although producing a finer-grain tessellation, would not allow to perform the process in real-time, thus making the approach impractical for generating terrain patches on the fly.
It is worth noting that in order to use the diamond-square algorithm within a patch-based terrain system, patch borders require an extra post-processing step, since in general there will be visible discontinuities in the height values at the edges of adjacent patches. However, the problem can be effectively mitigated, by considering a sort of "transition area" around the border of the most recently added patch. In order to smooth the transition, the height values in this area would be recomputed by interpolating their current values with the values from the corresponding, equal-size area around the border of the adjacent, pre-existing terrain patch.

Procedural height-based texturing

So far we have described how to procedurally create a terrain patch, but we have not discussed how it should be rendered. In fact, this choice is crucial in determining a believable terrain.
The simplest solution is to map a single, high-resolution seamless texture over the terrain patches. Unfortunately, this does not look right at all, because the texture pattern is easily recognised by the human eye. Also, it does not take into account in

any way the terrain height at the given point, so the result tends to appear flat.
Therefore, we decided to use a different approach that, although computationally more expensive than simple texturing, produces quite believable results. The approach is run entirely on the GPU and combines several low-resolution textures on a per-pixel basis, in order to determine the final colour of a given point.
The fundamental idea is to have a set of height zones, each one associated with a texture, which contribute to the colour of the fragment in a measure that depends on the height value at the point. The version implemented in the prototype uses four zones and associated textures, which are depicted in Figure 3.16.

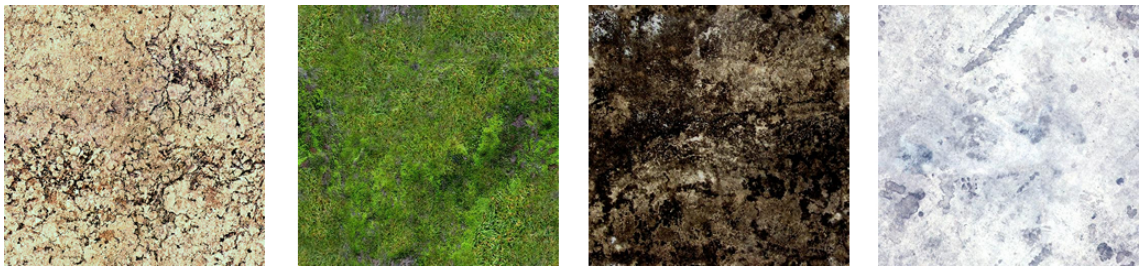

Figure 3.16: Procedural height-based texturing: texture reference set.

However, it might be possible to achieve even better results by increasing the number of height zones, thus applying a finer-grain procedural texturing. At a high level, the routine is described by the following steps, which are applied for each fragment, and perform an additive blending between the textures:

1. Receive the per-vertex, interpolated texture coordinates u, v and height value h.
2. Compute the height range for each zone, which is defined as the difference between the maximum and minimum altitudes associated to the zone.
3. Calculate the weight for each zone as:

$$zone.weight = \max(0, (zone.range - |(h - zone.max)|) \ / \ zone.range), \quad (3.6)$$

4. Compute the final colour as:

$$fragmentColour = \sum_{z \in heightZones} z.weight \cdot z.texture[u, v] \quad (3.7)$$

A significant aspect in this approach is the choice of which tiling factor tf should to be applied to the textures. If the scale is too small, the blend will show visible patterns. On the other extreme, if the scale is too large, the terrain will have a coarse-grained, *"pixelated"* look, due to the insufficient resolution of the textures. We empirically found that a good value for tf is

$$tf = 10 \times terrainScale, \tag{3.8}$$

where $terrainScale$ is the scale factor applied to the rendered terrain grid. Figure 3.17 illustrates the textured version of the previously shown terrain sample.

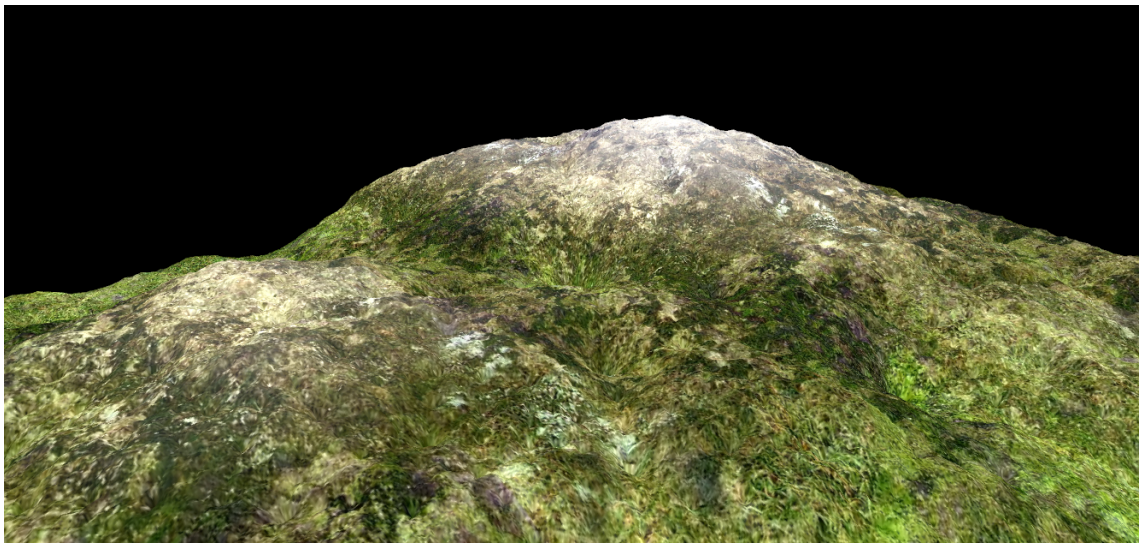

Figure 3.17: Procedural height-based texturing.

Feature injection

Many kinds of games can potentially benefit from using procedurally generated terrains. However, most often than not, because of mandatory game design requirements, the game environment needs to satisfy some constraints, such as the presence of specific features at predetermined positions.
Hence, a nice feature for a game-oriented procedural terrain generation system would be the capability of injecting specific features into the creation process, so that they are reproduced in the output.

We slightly modified the original generation algorithm to take into account feature injection. The fundamental idea is to bias the generation process by introducing into the grid a certain amount of pre-generated height values, and let the algorithm fill the rest. The pre-initialized height values are simply provided as a list of 3D points (whose components are the value itself, and the corresponding map coordinates), which is scanned when the grid is generated.
We investigated two possible ways of handling the pre-computed heights in the algorithm:

1. To compute a new value even for pre-computed heights: this approach basically runs the diamond-square algorithm in an additive fashion, so that the value computed for the midpoint is added to its current value.

2. To leave pre-computed values as they are.

With both approaches, the effectiveness of feature injection is significantly dependant on the complexity of the behaviour we would like to introduce on the surface. For instance, given the fact that the algorithm as it is tends to simulate non-eroded terrains, it is quite hard, if possible at all, to inject steep and sharp features, such as single high mountain peaks.
Also, the number and placement of the pre-initialized height values is crucial in driving the injection process.
With respect to both the options specified above, we noticed that quite interesting features can be reproduced, without the need to specify a large amount of height values in the corresponding areas. However, if new values are generated for pre-computed height points, the output tends to be less consistent over different PRNG seeds, because of the larger variability introduced in the area.
Figure 3.18 illustrates a quite successful attempt to inject a mountain valley feature in the middle of a terrain patch.
The pre-computed height list contains $\frac{1}{16}$ of the height values in the area covered by the valley, and the non-additive method was used to handle them. The screenshots differ for the underlying PRNG seed, but the feature is consistently reproduced in all of them.

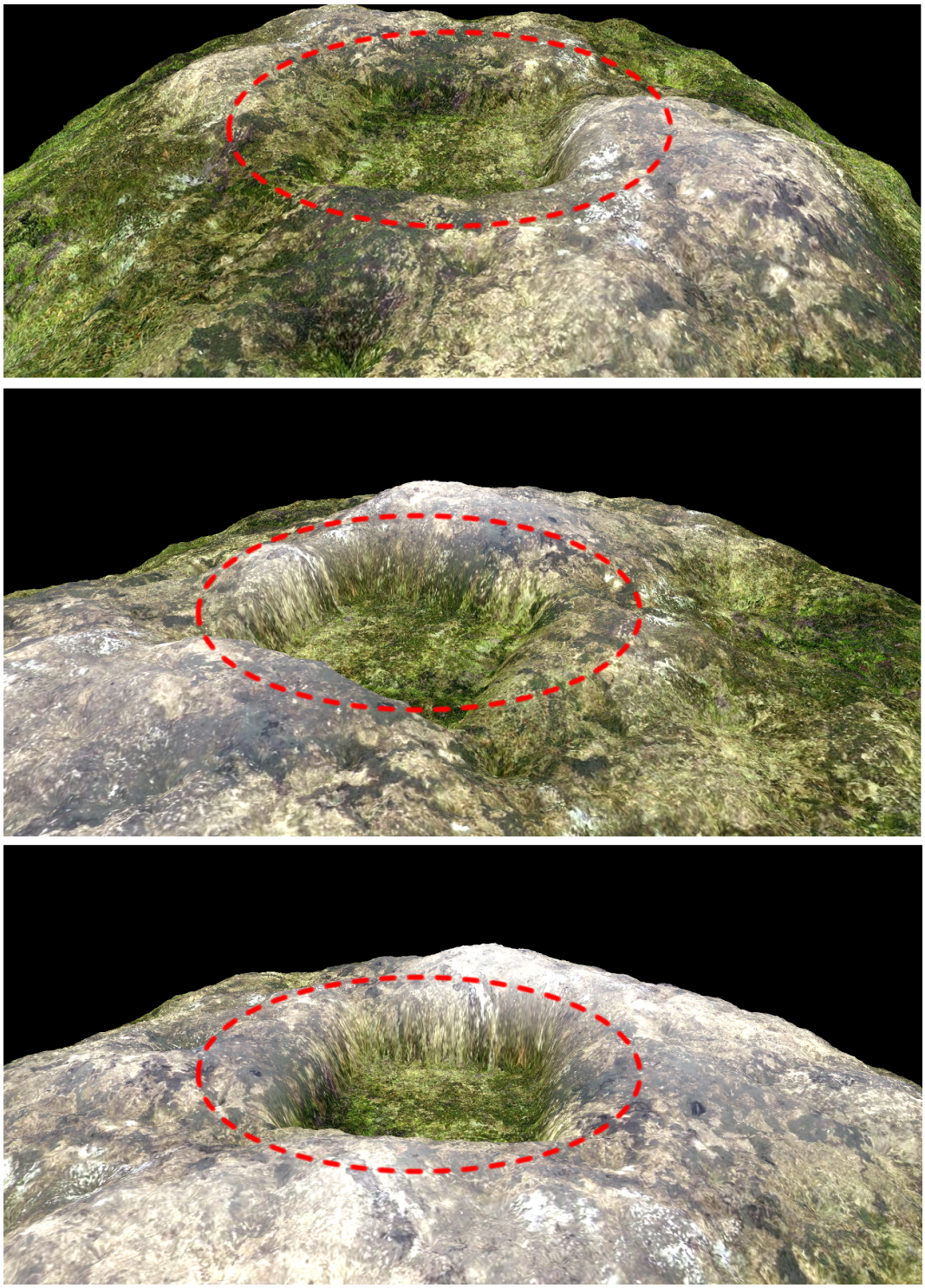

Figure 3.18: Procedural terrain generation: feature injection sample output.

Chapter 4

Implementation

In this chapter, we complete the description of the proposed solution, illustrating the remaining relevant implementation details. Before that, we briefly discuss some general implementation choices, and introduce all the external libraries that were used.

4.1 Implementation choices

Implementation language

In the implemented prototype framework, all application and server code was entirely written in JavaScript. On the client side, this was the only option to access the WebGL API.
On the other hand, many other programming languages (e.g., PHP, Java, ...) could have been used for developing the server. However, we decided to use the increasingly popular combination of JavaScript and Node.js[1][11], mainly for the following reasons.

- *Code re-usability.* If the framework were used as a base to develop an actual multi-player browser game, it is highly likely that some part of the game logic would require the same code to be run both on the clients and the server alike. With that in mind, although the benefit was minimal for the actual prototype, using a single programming language for both application and server code was the ideal choice.

[1]Node.js is a relatively new development technology, which enables the use of the JavaScript programming language for coding server-side applications.

- *Scalability.* Node.js exhibits a very low memory footprint per open connection, so it represents a sensible choice for developing data-intensive applications, which could potentially need hundreds or thousands of open connections at the same time. Moreover, Node.js has a completely modular architecture (so that new functionalities can be easily plugged in), and is capable of offloading computationally intensive routines onto background threads.

- *Fast prototyping.* JavaScript represents a good trade-off between code execution speed and *"development-friendliness"*, with effective debugging tools available for both client-side and server-side code.

External libraries

Besides the already mentioned WebGL and Node.js libraries, in order to avoid writing large amounts of *boilerplate* code, we also made use of several other external libraries:

- *sylvester*: matrix/vector manipulation library, also available as a Node.js module;

- *node-static*: Node.js module providing basic server functionalities for static content (such as non-dynamic HTML pages, shader programs, images, ...);

- *zlib*: Node.js module implementing gzip compression/decompression functionalities;

- *glMatrix*: client-side matrix/vector library, featuring specific functions to ease matrix manipulation within WebGL applications;

- *custom-seed*: a replacement for the standard JavaScript pseudo-random number generator, which enables custom seeding of the *Math.random()* function[3].

Transmission formats and compression

In order to transmit the required data structures to the clients, one or more data-interchange formats have to be agreed between client and server. Complex data is sent with both the progressive LOD and the procedural terrain generation approaches.
Since the first needs to transfer much more data than the latter, we primarily investigated which format would best suit the transmission of mesh data structures, described

by arrays of vertex positions, texture coordinates, normal attributes, and indices. We narrowed the range of options to two possible alternatives: using the *Extensible 3D Graphics* format (X3D)[17], or a custom format based on the *JavaScript Object Notation* (JSON)[8].
JSON is a language-independent, lightweight data-interchange format, structured as simple text, and conceptually based on two basic structures:

- *collections of name/value pairs* (called *objects* in JSON), which are enclosed in curly braces, where each name is followed by a colon and its associated value, and name/value pairs are separated by commas;

- *ordered lists of values* (JSON *arrays*), which are enclosed in square brackets, with values separated by commas.

Values can be *null*, double quoted strings, numbers, boolean values, objects or arrays. Hence, nesting of arrays and objects is allowed, so that marshalling and un-marshalling of complex data structures to and from JSON strings is usually an easy process.
X3D is a file format used to represent and communicate 3D objects using XML. It is the evolution of the Virtual Reality Modeling Language (VRML), and allows us to describe complex 3D scenes for use in engineering and scientific visualization, CAD and multimedia applications. Being a XML-based language, the major strengths of X3D are its easy extendibility and the possibility of syntactically checking X3D documents, by feeding them (along with the XML Schema defining the X3D language) to a generic XML validator tool.
Unfortunately, these features come with the following costs:

1. Overhead is introduced in the transmitted data, since all mandatory X3D-related information needs to be specified in addition to the actual data, in order to get compliant, valid X3D files.

2. A X3D parser is required on the client side. Since writing one from scratch would be a very demanding task, the best option is to integrate an existing one into the application. For WebGL applications, a sensible choice is represented by the X3DOM parser, since it is HTML5-compatible. However, this would considerably increase the amount of application code that needs to be downloaded in form of JavaScript files, even though only a small set of its features would be employed.

Since we were not interested in producing LOD representations that could be used in environments other than the prototype framework, we considered implementing a X3D-based transmission system to be overkill with respect to the prototype simple requirements. Hence, we decided to transfer all data structures, for both progressive LOD and PCG approaches, with simple JSON objects and arrays (see the corresponding sections for further details).
It is worth spending a few words about the use of data compression. There are several good reasons to apply compression to transferred data:

1. JSON-formatted structures are basically strings, so they are amenable to generic compression (e.g., gzip) with good compression rates. In fact, almost all data transfers (HTML pages, JavaScript files, ...), except for already compressed textures, can benefit from compression in terms of reduced transfer times.

2. Gzip compression of outbound data flows can be integrated in the Node.js server application by using the aforementioned *zlib* module, at the cost of some extra-processing time before the data is actually sent.

3. All browsers are capable of accepting gzip compressed content and efficiently decompress it upon reception. This is performed automatically, provided that the corresponding header is specified when the request is sent to the server. Since the inflation routine is quite fast, it is generally more convenient to receive compressed content, rather than downloading its uncompressed version.

Hence, the reference implementation by default applies compression to all transfers of uncompressed data (that is, any data type, save for already compressed textures), as long as the balance between the reduced download time and the extra-processing time is favourable. In practice, this means that all client requests specify compressed content as accepted. On the server side, provided that the received request accepts compressed content, all outbound data is compressed, except for the procedural terrain generation initial parameters, which are so small that adding the compression headers would actually increase the total amount of sent data.

4.2 Surface mapping

A high-level description of all the involved algorithms was provided in Section 3.1, for both parallax and relief mapping. However, it is worth extending that discussion with some relevant implementation details.
First, we did not describe how the height map is represented in the actual implementation. Although the height values could be embedded in the texture map, by using the alpha channel[2], we decided to use a separate grayscale image associated to the texture, with the convention that the brighter the point, the greater its height. An example of a texture and associated height map is shown in Figure 4.1.

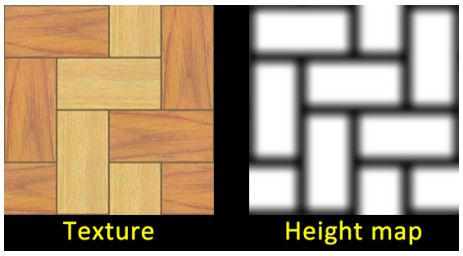

Figure 4.1: Parallax/Relief Mapping: texture and height-map.

Tangent space computation

A key aspect of parallax and relief mapping implementations is the transformation of the eye vector in tangent space. In Chapter 3 we assumed that each vertex had normal, tangent and bi-tangent attributes, which were used to build the required transformation matrix in the vertex shader. However, very often 3D models only specify the normal attribute. In such cases, we need a way to compute the other two attributes. The approach used in the implementation is described in [37], and works for arbitrary triangle meshes.
We first consider how to compute the attributes for a single triangle T, defined by vertices $\mathbf{P_0}, \mathbf{P_1}$ and $\mathbf{P_2}$, with associated texture coordinates (u_0, v_0), (u_1, v_1) and (u_2, v_2). First, the following quantities are computed:

- $\mathbf{Q_1} = \mathbf{P_1} - \mathbf{P_0}$ and $\mathbf{Q_2} = \mathbf{P_2} - \mathbf{P_0}$,

[2]This approach assumes that the original texture does not contain transparency information.

- $(s_1, t_1) = (u_1, v_1) - (u_0, v_0)$ and $(s_2, t_2) = (u_2, v_2) - (u_0, v_0)$.

Then the following equations need to be solved for $\mathbf{T}$ and $\mathbf{B}$:

- $\mathbf{Q_1} = s_1\mathbf{T} + t_1\mathbf{B}$
- $\mathbf{Q_2} = s_2\mathbf{T} + t_2\mathbf{B}$

The equations form a system with six unknowns and six equations (three for the x, y, z components of $\mathbf{T}$, and the same for $\mathbf{B}$). The system can be written in matrix form as:

$$\begin{pmatrix} (\mathbf{Q_1})_x & (\mathbf{Q_1})_y & (\mathbf{Q_1})_z \\ (\mathbf{Q_2})_x & (\mathbf{Q_2})_y & (\mathbf{Q_2})_z \end{pmatrix} = \begin{pmatrix} s_1 & t_1 \\ s_2 & t_2 \end{pmatrix} \begin{pmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{pmatrix}$$

and solved by multiplying both sides by the inverse of the (s, t) matrix:

$$\begin{pmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{pmatrix} = \frac{1}{s_1t_2 - s_2t_1} \begin{pmatrix} t_2 & -t_1 \\ -s_2 & s_1 \end{pmatrix} \begin{pmatrix} (\mathbf{Q_1})_x & (\mathbf{Q_1})_y & (\mathbf{Q_1})_z \\ (\mathbf{Q_2})_x & (\mathbf{Q_2})_y & (\mathbf{Q_2})_z \end{pmatrix}$$

The solution provides us with the non-normalised tangent and bi-tangent vectors $\mathbf{T}$ and $\mathbf{B}$. The per-vertex tangent and bi-tangent attributes are respectively computed as the normalised sum of tangents and bi-tangents of the faces the vertex is part of[3].
It is important to note that the resulting tangent, bi-tangent and normal attributes are not necessarily perfectly orthogonal to one another. Assuming they are close to that, Gram-Schmidt orthogonalization can be used to rectify this issue, without introducing relevant distortions. Thus, non-normalised tangent and bi-tangent attributes $\mathbf{T}'$ and $\mathbf{B}'$ can be computed, for a vertex having normal attribute $\mathbf{N}$, with the following formulas:

- $\mathbf{T}' = \mathbf{T} - (\mathbf{N} \cdot \mathbf{T})\mathbf{N}$,
- $\mathbf{B}' = \mathbf{B} - (\mathbf{N} \cdot \mathbf{B})\mathbf{N} - (\mathbf{T}' \cdot \mathbf{B})\mathbf{T}'/\mathbf{T}^2$.

Once all the necessary vertex attributes are available, parallax and relief mapping can be computed with the GLSL shaders described in the following.

[3]The approach is exactly the same one described in Section 3.2.4 for vertex normals.

Eye vector transformation

The proposed implementation calculates the eye vector transformation in the vertex shader, but this step could be alternatively performed in the fragment shader, with slightly different results, by passing the eye vector and the normal, tangent and bi-tangent attributes to it, as *varying* variables.
Table 4.1 illustrates the vertex shader code used for both parallax and relief mapping.

```
varying vec2 vTextureCoord;
varying vec3 vEyeDir;

attribute vec3 aVertexTangent;
attribute vec3 aVertexBitangent;
attribute vec3 aVertexNormal;
attribute vec3 aVertexPosition;
attribute vec2 aTextureCoordinates;

uniform vec3 uEye;

uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;

void main (void) {

    gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);

    vTextureCoord = aTextureCoordinates;

    // Compute the direction of (Eye Position - vertex position)
    vec3 eyeDir = vec3(uEye - aVertexPosition);
    normalize(eyeDir);

    // Put the Eye Direction into tangent space
    vEyeDir.x = dot(aVertexTangent, eyeDir);
    vEyeDir.y = dot(aVertexBitangent, eyeDir);
    vEyeDir.z = dot(aVertexNormal , eyeDir);
}
```

Table 4.1: Parallax/Relief Mapping: vertex shader.

Note how the transformation matrix is not built explicitly, since the x, y, z components of the eye vector, transformed in tangent space, can be computed with three separate scalar products.

Parallax and relief mapping: per-fragment computations

As already stated, the fragment shader is where the parallax and relief mapping actually differ. The corresponding code for the parallax shader is shown in Table 4.2.

```
precision highp float;  // Set precision specifier

varying vec2 vTextureCoord;
varying vec3 vEyeDir;

// scale and bias depend on the physical
// properties of the simulated surface
uniform float uScale;
uniform float uBias;

uniform sampler2D uTextureSampler; // Texture
uniform sampler2D uHMSampler; // Height-map

void main (void) {

    // Normalize eye vector
    vec3 eyeDirN = normalize(vEyeDir);

    // Sample height-map value and compute the offset
    vec4 offset = texture2D(uHMSampler, vTextureCoord);
    offset = offset * uScale + uBias;

    // Calculate the limited offset and the resulting UV coordinates
    vec2 texCoords = offset * eyeDirN.xy + vTextureCoord;
    gl_FragColor = texture2D(uTextureSampler, texCoords);
}
```

Table 4.2: Parallax Mapping: fragment shader.

The shader computes the limited version of parallax mapping, as described in Section 3.1.1. Texture and height map are passed to the program as *uniform* input 2D samplers. Scale and bias factors are also provided by the application, so that they can be adjusted with respect to the surface being simulated. The code in the main function does not need any specific comment, as it implements exactly the steps detailed in the high-level description of the approach.

The fragment shader code implementing the relief mapping technique is shown in Table 4.3.

The structure of the shader is somewhat similar to the parallax mapping shader. The ray intersection computation is performed in the *computeIntersection* function, shown in Table 4.4.

```
precision highp float;  // Set precision specifier

varying vec2 vTextureCoord;
varying vec3 vEyeDir;

uniform sampler2D uTextureSampler; // Texture
uniform sampler2D uHMSampler; // Height-map

// This function performs the linear and binary
// searches to find the intersection
vec2 computeIntersection (sampler2D hMap, vec2 uv, vec3 eyeVector);

void main (void) {

    // Normalize eye vector
    vec3 eyeDirN = normalize(vEyeDir);

    vec2 texCoords = computeIntersection(uHMSampler, vTextureCoord,
                                         eyeDirN);
    gl_FragColor =texture2D(uTextureSampler, texCoords);

}
```

Table 4.3: Relief Mapping: fragment shader.

The function performs a linear search first, sampling the search interval with step size equal to $1.0/linearSteps$, in order to find the interval containing the correct intersection point. Then, it computes an approximation of the intersection point by performing 8 steps of bisection on the interval. For the sake of clarity, it is worth noting that height values are used, rather than depth values (as illustrated in Section 3.1.2), so that the same height map can be used for both parallax and relief mapping techniques.

4.3 Progressive Level Of Detail

In order to complete the discussion about progressive level of detail, it is worth examining the actual implementation of mesh simplification, LOD transmission and LOD selection in somewhat more detail.

Mesh simplification

As stated in Chapter 3, the mesh simplification routine is run entirely on the server side. At startup time, the server computes the sequence of discrete levels of detail for each model. The simplification is performed using several ad-hoc structures:

```
vec2 computeIntersection (sampler2D hMap, vec2 uv, vec3 eyeVector) {
    const int linearSteps = 10; // Linear search step width = 1/linearSteps
    const int binarySteps = 8; // Number of binary search steps

    float height = 1.0;
    float step = 1.0/linearSteps;
    int currStep;
    vec4 tHeight = texture2D(hMap, uv);
    vec2 delta = vec2(-eyeVector.x, -eyeVector.y) / eyeVector.z;

    // Linear search
    for (int i = 1; i <= linearSteps; i++) {
         if (tHeight.x <= height) {
             height -= step;
             currStep = i;
             tHeight = texture2D(hMap, uv - delta * height);
         }
    }

    // currStep - 1 is above the surface,
    // so use that and currStep for the following bisection
    currStep--;
    height += step;

    // binary search
    for (int i = 0; i < binarySteps; i++) {
         step *= 0.5;
         height -= step;
         tHeight = texture2D(hMap, uv - delta * height);

         if (tHeight.x >= height) {
              height += step;
          }
    }

    return uv - delta * height;
}
```

Table 4.4: Relief Mapping: per-fragment intersection.

- *Mesh*: container class, which encapsulates the lists of *Vertex* and *Triangle* objects defining the model. It also stores a reference to the current best candidate for contraction.

- *Triangle*: an object containing references to the three *Vertex* instances which define the triangle, and its own normal vector.

- *Vertex*: class storing all information about a vertex and its attributes. Each instance contains the position, UV mapping and normal attributes of the corresponding vertex. Moreover, each *Vertex* object P stores information relevant to

the simplification process:

- the *collapse candidate* vertex Q, which is the other endpoint of the edge (P, Q), where (P, Q) is the best possible candidate for contraction within the set of edges departing from P;
- the cost of collapsing (P, Q) (based on the metric described in Section 3.2.3), which is used at the *Mesh*-object level to determine the overall best contraction option;
- references to all its neighbouring vertices and the triangles it is part of, in order to efficiently apply changes caused by collapses.

It is worth spending a few words to clarify how the input structures for the mesh simplification algorithm are built. Current implementation accepts input 3D models in Wavefront Object Format (OBJ): a custom parser is in charge of creating the *Mesh* object, based on the OBJ representation[4]. The parser is capable of importing any triangulated mesh, with or without normal and/or UV mapping vertex attributes.
Note that the resulting *Mesh* object has vertex normals re-computed, as described in Section 3.2.4, and replicated vertices in case of UV attribute discontinuities over the surface of the model.
Since progressive LODs are only generated once (at startup time), optimising the execution speed of the algorithm was not a primary goal, and current implementation simplifies the models sequentially, one by one. However, it is worth noting that Node.js provides a module (*threads-a-gogo*), which enables applications to perform efficient parallel data processing on multi-core processors. The mechanism could be easily plugged into current implementation, by running the simplification of different models on separate threads, and collecting the computed discrete LOD sequence through a callback function, which would be fired by the spawn thread upon completion.

Transmission

As anticipated in Section 4.1, meshes are sent to the client using JSON objects, the format of which is described in Table 4.5.

[4] 3D models that are not in OBJ format can be easily exported in such format with most 3D modelling tools (e.g., *Blender*).

```
{
    vertexPositions: [...],
    vertexNormals: [...],
    vertexTextureCoords: [...],
    indices: [...]
}
```

Table 4.5: Progressive LOD: mesh transmission format.

The object properties have the following meaning:

- *vertexPositions*: an array of floats, which represent the x, y, z position components of the vertices defining the mesh, so that the array will have the form $[(V_0)_x, (V_0)_y, (V_0)_z, (V_1)_x, (V_1)_y, (V_1)_z, \ldots]$.
- *vertexNormals*: an array of floats, representing the x, y, z components of the normal attributes associated to the vertices. The normals are defined in the same order used for vertex positions. Therefore, assuming that N_i is the normal attribute of vertex V_i, the array will be defined as $[(N_0)_x, (N_0)_y, (N_0)_z, (N_1)_x, (N_1)_y, (N_1)_z, \ldots]$.
- *vertexTextureCoords*: an array of floats in the range $[0, 1]$, which define the u, v coordinates at each vertex. As with normals, the values are specified in the same order as vertex positions.
- *indices*: an array of integers, representing the vertex indices, with respect to the order used in the *vertexPositions* array, that define the triangles of the mesh.

Once the client receives the JSON object, it parses each property and fills the GPU buffers with their content, setting the correct item size for each one of them, depending on the type of data (e.g., 3 for the position buffer, since each elements consists of x, y and z components). Note that this operation is extremely fast, since it is performed using JavaScript typed arrays[5], instead of regular arrays.
As mentioned before, the greatest benefit of using JSON objects to deliver the mesh description is that only the required information is transmitted, with no additional

[5]JavaScript typed arrays allow for interoperability with native binary data. They were introduced shortly after the first WebGL draft was published, in order to provide a faster way to feed data to the GPU.

overhead. Moreover, the object format could be easily extended with new properties if needed (e.g., to send additional attributes, such as tangent, bi-tangent, blend weights, ...).

By default, the server keeps all simplified models in memory, into a hash-table associating each model name with the corresponding array of discrete LODs, so that they can be quickly served to the requesting clients. However, due to possible memory constraints, this may become prohibitive when a large number of models is simplified. Hence, the server can optionally save the intermediate levels of detail to file (in JSON format), so that they can be served as static content, which allows the server to save memory, at the cost of an increased delivery time.

Each time a client needs to receive a specific LOD, it sends an asynchronous *XMLHttpRequest* to the server, specifying the name of the model and the desired level of detail as parameters, where the index 0 corresponds to the 100% LOD (that is, the original mesh M_0) and $1, 2, \ldots, n$ to increasingly coarser approximations $M_1, M_2, \ldots, M_n$. Once the corresponding data is received, a callback function takes care of filling the proper GPU buffers, as previously explained.

The framework also provides a mechanism to receive only incremental updates: the client sends an upgrade request, specifying model name and currently available LOD M_j, and the server replies with the list of changes that need to be performed to build the more detailed LOD M_{j-1}. In practice, the list consists of a sequence of vertices (and corresponding normal and UV attributes) that need to be substituted into the arrays corresponding to M_j, or added to them. Also, the server sends the list of indices that defines the triangles of M_{j-1}, which replaces the previous index array.

Once the necessary changes are applied, the client can fill the relative GPU buffers with the updated information. In order for this mechanism to work efficiently, the client has to store the arrays corresponding to the currently available LOD, instead of throwing them away once the GPU buffers are filled. Moreover, the server has to sort the vertices during simplification, so that they are consistently ordered, and the client does not have to re-order the corresponding arrays when switching from a coarser LOD to a finer one. This is easily implemented by sorting the vertices in the order in which they were collapsed.

LOD selection

As described in Section 3.2.5, LOD selection is performed on the client on a per-model basis, based on its *magnification factor*. The metric is recalculated at each update cycle, in order to consistently refresh the selected LODs.
At present, the prototype does not perform any *morphing* between subsequent levels of detail, so *popping* effects may be noticeable if the LODs are very different from each other in terms of mesh topology. A simple form of morphing, based on linear interpolation of the vertex attributes, could probably be added without too much effort. However, linear interpolation works well when the values vary continuously. Thus, discontinuities in the texture coordinate mapping would still produce *popping* effects, if the interpolated value crossed a texture seam.
Current client implementation interprets the selected LOD as the minimum level of detail required for properly displaying the model at the given time. In other words, if currently available LOD is more detailed, the client just renders that instead of the selected one. An alternative approach would possibly cache all retrieved LODs, in order to enable the client to display lower levels of detail, without re-downloading them.
However, the solution would require more memory on the client-side, and it would potentially benefit only configurations having low-end GPUs. Hence, we decided not to implement this alternative into the WebGL client, although it could be a smarter choice for a version of the framework targeting mobile development.

4.4 Procedural terrain generation

As previously mentioned, in order to procedurally generate the terrain, the client needs to fetch a set of initial configuration parameters from the server. In Section 4.1, we anticipated that this information is transferred using JSON as an interchange format. The expected object structure is shown in Table 4.6.
The parameters have the following meaning:

- *patchSize*: a single integer[6], which determines the size of the 2D grid used for terrain generation;

[6]Note that the client expects an integer of the form $2^n + 1$, for $n \in Z^+$.

```
{
    patchSize: ...,
    scaleFactor: ...,
    seed: ''...''
}
```

Table 4.6: Procedural terrain generation: initial configuration format.

- *scaleFactor*: a float value, which determines the scale of the pseudo-random variation applied in the computation of midpoint values;
- *seed*: the common seed used on every client to initialize the custom PRNG, implemented by the *custom-seed* library. The value is computed at startup time by concatenating a predefined amount of bytes from a local entropy source. For instance, this could be implemented on Unix/Linux systems by reading the output of the *dev/urandom* process. Fortunately, Node.js provides an ad-hoc function for generating random bytes, which takes care of finding a proper source of entropy, based on the underlying operating system, and returns the requested number of bytes.

In order to apply feature injection, the set of predefined heights is sent with a different JSON object, consisting of two properties:

- *patchCoordinates*: an array of two integer values, which specify the horizontal and vertical offsets of the patch with respect to the one at the origin of the map (that is, the first patch generated);
- *precomputedHeights*: conceptually this would be an array of objects, each having *X*, *Y* and *height* properties. *X* and *Y* would specify the point on the grid, while *height* would provide the pre-computed height value to be set. However, to limit the amount of extra strings inserted in the JSON object, *precomputedHeights* consists of an array of simple numbers, which is split in groups of three on the client side. The first two are assumed to be the x and y coordinates of the point on the 2D grid, while the third is the pre-computed height.

If feature injection is enabled, whenever a client has to generate a new patch of terrain, it sends a request to the server containing the corresponding patch coordinates.

The server replies with the proper JSON object, containing a (possibly empty) set of predefined height values, which are used on the client to bias the terrain generation process for the corresponding patch.

Procedural texturing

As already mentioned in Chapter 3, the procedural terrain generation consists of the actual surface generation, which is performed in application code, and its procedural texturing, which is run entirely on the GPU. In order to render the terrain, the 2D grid of height values is converted to a triangle-strip mesh, having a number of vertices equal to the number of points defining the grid.
The position and texture coordinates of each vertex are determined as follows:

- the x and z position components are computed from the horizontal and vertical coordinates of the corresponding point on the grid, multiplied by a scale factor, which determines the extent of the area occupied by the patch in world space;
- the y component is equal to the height value of the surface at the corresponding point;
- the UV mapping is procedurally generated to consistently map the texture(s) on the surface.

The vertex shader code for rendering the terrain is quite trivial, as it just passes the texture coordinates and the height value to the fragment shader. Tables 4.7 and 4.8 illustrate the fragment shader. The code implements exactly the same steps described at a high level in Section 3.3.1.
As a collateral benefit of applying procedural texturing to the surface, the terrain becomes amenable to a simple form of animation, where the height values of the points on the grid potentially vary over time. Although current prototype does not implement this feature, it could be easily added by:

1. Adding a *targetHeight* attribute to each vertex in the mesh.
2. Morphing the height value passed from the vertex shader to the fragment shader, over a predefined amount of time, from the original y position of the vertex towards the $targetHeight$ value.

```
precision highp float;  // Set precision specifier

struct ZoneLevel {
     float min;
     float max;
};

varying vec2 vTextureCoord;
varying vec3 vHeight;

// min-max height ranges for each zone (sand, grass, ...)
uniform ZoneLevel uSandLevel;
uniform ZoneLevel uGrassLevel;
uniform ZoneLevel uHillLevel;
uniform ZoneLevel uSnowLevel;

uniform sampler2D uSandSampler; // Sand texture
uniform sampler2D uGrassSampler; // Grass texture
uniform sampler2D uHillSampler; // Hill texture
uniform sampler2D uSnowSampler; // Snow texture

// This function implements the weighted blend between the
// four low-resolution textures
vec4 blendTerrain();

void main (void) {

    gl_FragColor = blendTerrain();

}
```

Table 4.7: Procedural terrain texturing: fragment shader.

Since the final colour for each pixel is determined procedurally at render time, the variation in height would be reflected in a change in the texturing output, which would progressively adapt to the different altitude at the point.

```
vec4 blendTerrain () {
    // base color
    vec4 terrainColor = vec4(0.0, 0.0, 0.0, 1.0);

    float height = vHeight;
    float regionMin = 0.0;
    float regionMax = 0.0;
    float regionRange = 0.0;
    float regionWeight = 0.0;

    // sand
    regionMin = uSandLevel.min;
    regionMax = uSandLevel.max;
    regionRange = regionMax - regionMin;
    regionWeight = (regionRange - abs(height - regionMax)) /
                   regionRange;
    regionWeight = max(0, regionWeight);
    terrainColor += regionWeight * texture2D(uSandSampler,
                   vec2(vTextureCoord.s, vTextureCoord.t));

    // grass
    regionMin = uGrassLevel.min;
    regionMax = uGrassLevel.max;
    regionRange = regionMax - regionMin;
    regionWeight = (regionRange - abs(height - regionMax)) /
                   regionRange;
    regionWeight = max(0, regionWeight);
    terrainColor += regionWeight * texture2D(uGrassSampler,
                   vec2(vTextureCoord.s, vTextureCoord.t));

    // analogous code for other height ranges and textures
    //...

    return terrainColor;
}
```

Table 4.8: Procedural terrain texturing: per-fragment blend.

Chapter 5

Results and Conclusion

In this chapter we present an evaluation of the results obtained with the previously detailed techniques, supporting them with quantitative measurements where appropriate, and identifying their current limitations. We then point out several possible directions for future work and improvement, finally providing some concluding remarks concerning the proposed solution as a whole.

5.1 Results evaluation

All the measurements presented in this section were performed on an average laptop, with the following configuration:

- *CPU*: Intel core i5 - 2.26 Ghz,
- *GPU*: ATI Radeon 5650 HD Mobility,
- *Browser*: Mozilla Firefox, ver. 14.0.1.

For some types of data that the clients receive from the server, it makes sense to assess expected download times, as they directly affect the amount of time required to actually start rendering a scene. Therefore, a reference download speed has to be set for the measurements.
The theoretical download bandwidth limit for current HSDPA hardware is 14.4Mbit/s.

However, most mobile Internet providers do not have enough resources to offer high download speeds, so in practice the limit is much lower. Based on recent surveys ([46] for mobile broadband providers and [45] for fixed-line providers), we decided to consider, for benchmark purposes, a lower threshold of 2Mbit/s (average HSDPA download speed in the UK) and a higher one of 15.91Mbit/s (average download speed for European consumer broadband services).

5.1.1 Progressive Level Of Detail

Whenever the mesh simplification algorithm is able to produce acceptable-quality simplifications of the input model, progressive LOD can be an effective method to reduce the amount of data that is required to begin rendering the mesh.
To evaluate the performance of the approach, the models illustrated in Figure 5.1 were used as a reference. As previously stated, the proposed solution focuses on models with a low polygon count. Hence, all of them were chosen following this primary criterion. However, since we want to show both the benefits and the limitations of the approach, we included samples that exhibit different visual-quality levels in their approximations, in order to try to identify the reasons behind the different results.

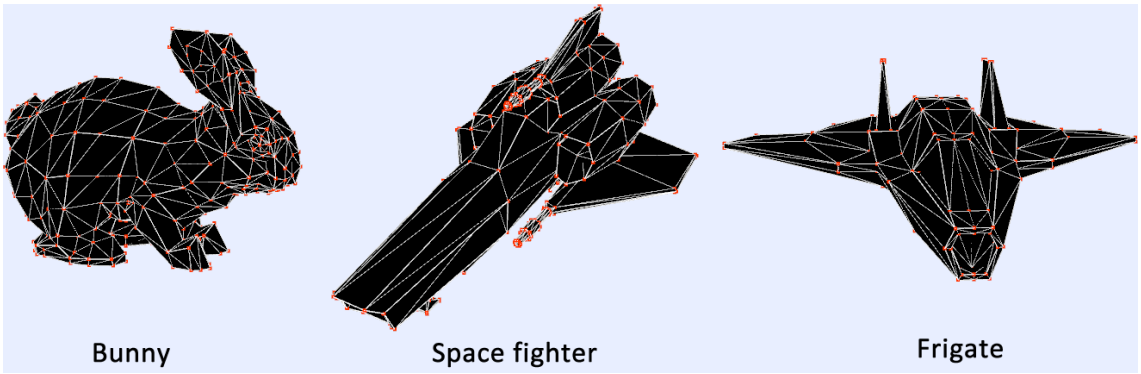

Figure 5.1: A set of sample models used to evaluate the results of progressive LOD.

The reference set contains the following three models:

1. *"Bunny"*, consisting of a mesh with a high degree of curvature and a considerable number of shared vertices, as evident from the ratio between vertex and triangle counts, indicated in Table 5.1.

2. *"Space fighter"*, which has a more geometric shape, with a certain number of thin features and a lower percentage of shared vertices.

3. *"Frigate"*, having a fairly simple geometric shape, about half of the shared vertices of the *"Bunny"* model (in percentage), and an extremely low triangle count.

Figure 5.2 shows the texture-mapped versions of the reference models.

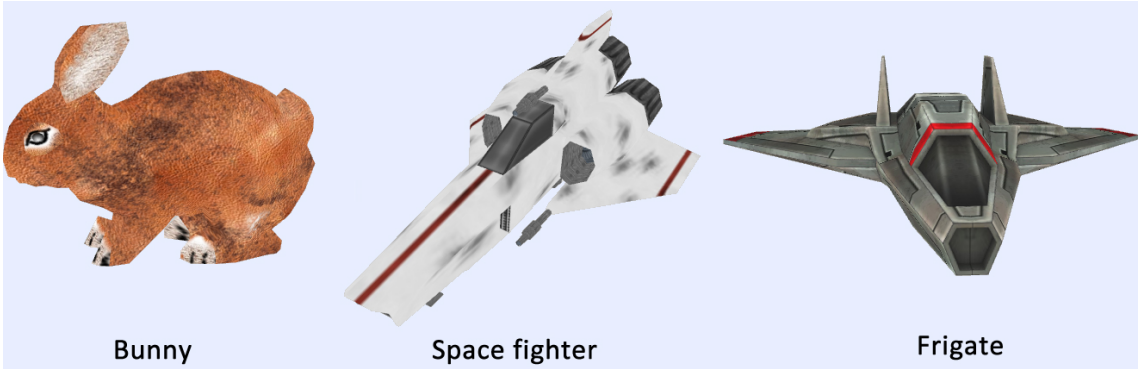

Figure 5.2: Textured reference models.

Table 5.1 provides a measurement of the potential benefits of using progressive level of detail, compared to simply downloading the original mesh. As expected, with respect to the input model, the approximations have sizes that are proportional to the level of simplification. Hence, depending on the quality of the lower LODs (in terms of texture distortion and presence of "holes" in the mesh), the client is able to start rendering all the models that do not require close-up details, in only a small fraction of the time necessary for the fully detailed scene.
It is worth noting that these figures regard the uncompressed JSON representation of the meshes. As stated in Chapter 4, in practice sent data can be compressed on the server side and decompressed on the fly on the client side, so that the download time can be further reduced (on average to $\frac{1}{4}$, compared to the uncompressed LOD data).
We now evaluate current limitations of the mesh simplification algorithm, by analysing its output for the reference models.
We first consider an initial sequence of approximations generated without taking into account UV mapping, which causes vertex replication wherever the texture coordinates attribute is not continuous on the surface of the mesh. Figures 5.3, 5.4, 5.5 and 5.6

Model / LOD	Vertex Count	Triangle Count	Size (KB)	Expected DL Time (s) [2 Mbit/s]	Expected DL Time (s) [15.9 Mbit/s]
Bunny / 100%	549	902	92,3	0,3605	0,0453
Bunny / 70%	385	685	67	0,2617	0,0329
Bunny / 50%	275	516	48	0,1875	0,0236
Bunny / 30%	165	300	28,5	0,1113	0,014
Bunny / 10%	55	86	9,3	0,0363	0,0046
Frigate / 100%	338	298	44,8	0,175	0,022
Frigate / 70%	238	211	32,3	0,1262	0,0159
Frigate / 50%	170	133	26,2	0,1023	0,0129
Frigate / 30%	102	76	15,8	0,0617	0,0078
Frigate / 10%	34	25	5	0,0195	0,0025
Space fighter/ 100%	1239	950	180,7	0,7059	0,0888
Space fighter/ 70%	868	748	137,5	0,5371	0,0675
Space fighter/ 50%	620	553	98,6	0,3852	0,0484
Space fighter/ 30%	372	367	59,2	0,2313	0,0291
Space fighter/ 10%	124	150	19,9	0,0777	0,0098

Table 5.1: Comparison between different levels of detail.

illustrate how the reference models look like, once simplified at the $70\%, 50\%, 30\%$ and 20% level, respectively.

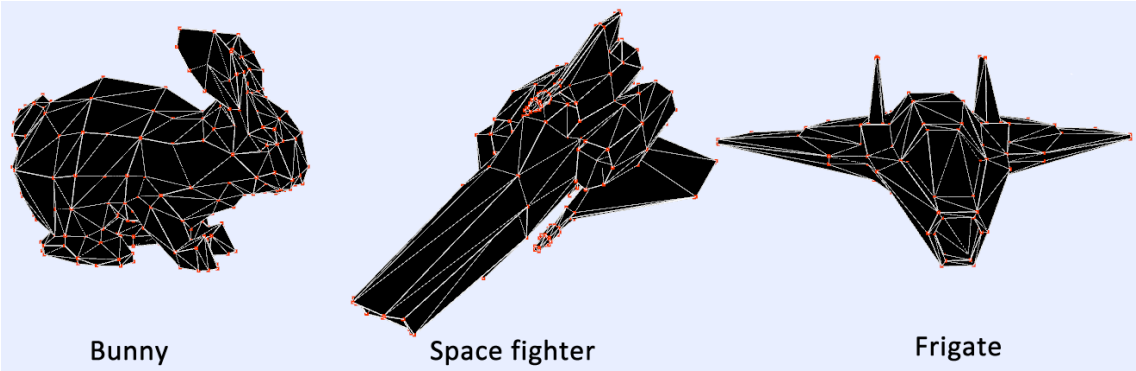

Figure 5.3: Sample models: 70% LOD.

When not considering UV coordinates, the algorithm produces consistent results on all the reference models. It preserves model shapes reasonably well, even at the lower levels of detail, and tends to maintain symmetries, which helps improve the overall

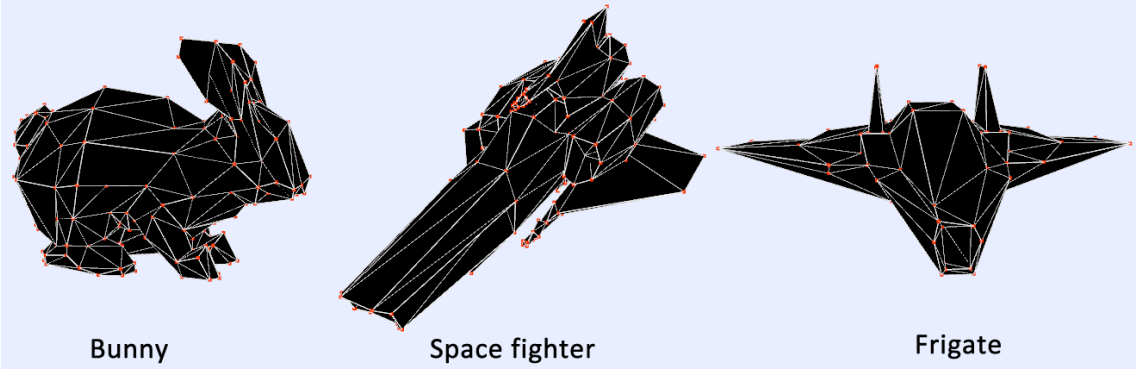

Figure 5.4: Sample models: 50% LOD.

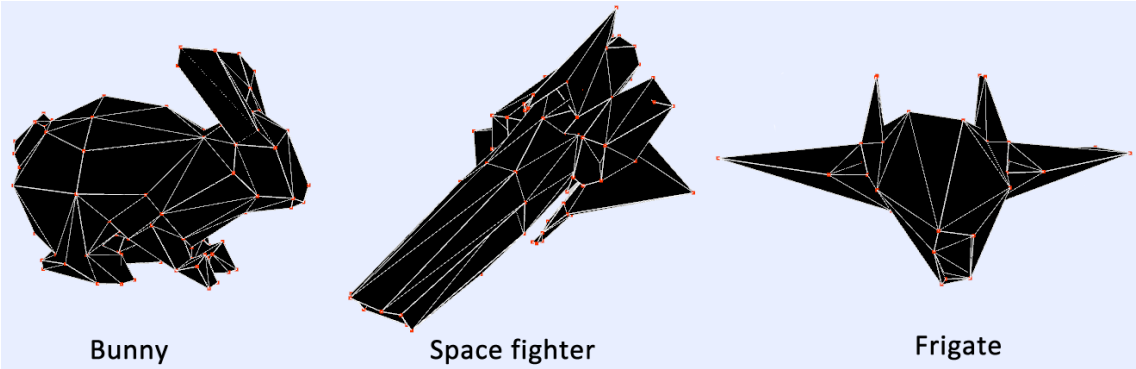

Figure 5.5: Sample models: 30% LOD.

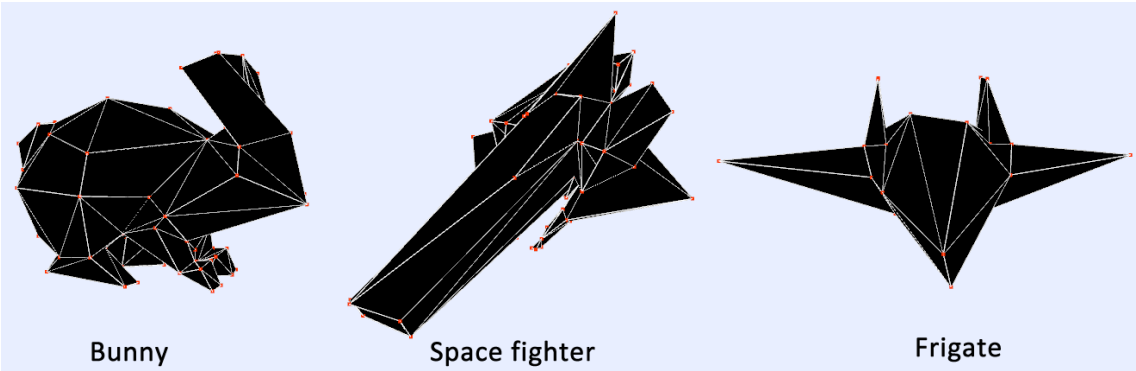

Figure 5.6: Sample models: 20% LOD.

quality of the approximations.
Up to a certain extent, it also manages to keep thin features (as is shown by the *"Space fighter"* sample): even if they are not preserved exactly, this is not a major drawback, since from afar the precise shape of these features would not be clearly distinguishable.
As mentioned in Section 3.2.4, if UV mapping is taken into account, the mesh simplification algorithm does not exhibit the same consistency in the results. We purposely inserted the *"Frigate"* sample in the reference set, because it is a good representation of a category of models that the algorithm is not capable of handling properly.
Figures 5.7, 5.8 and 5.9 show the textured sample models at the $70\%, 50\%$ and 30% LODs, respectively.

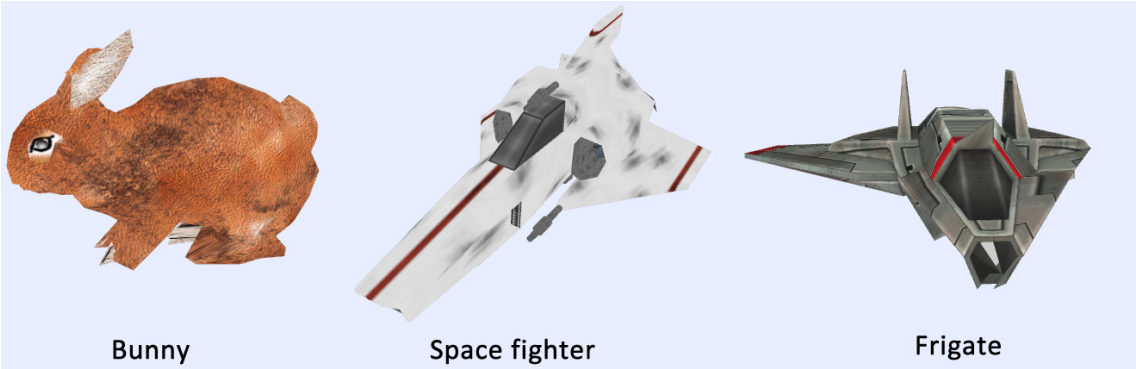

Figure 5.7: Textured sample models: 70% LOD.

By inspection of the output, it is quite evident that the quality of the simplifications is acceptable for the *"Bunny"* model, although at the 30% LOD, even if the mesh has no "holes", the texture starts warping a little.
With respect to the *"Space fighter"* sample, the results are not as good, but face preservation still degrades nicely: some "holes" are introduced in the simplified meshes as the approximation process becomes more aggressive, but the original shape is overall preserved until the 30% level, where the alteration becomes evident, along with a prominent texture warp.
Unfortunately, the levels of detail for the *"Frigate"* mesh become noticeably flawed from the 70% LOD onwards, to the extent that the coarsest approximations are not usable at all, as evident from the other outputs.
The existence of a relation, between the quality of the simplification and the percentage

Figure 5.8: Textured sample models: 50% LOD.

of replicated vertices, becomes evident when considering the different properties of the models. In other words, if texture coordinates are continuous over the surface (such as on a textured plane), then the simplification is expected to produce very good-quality LODs. However, when discontinuities are introduced in the UV mapping, thus causing vertex replication, the simplification quality starts degrading. If the number of replicated vertices becomes quite significant compared to the vertex count (as with the third sample mesh), our approach is not able to produce usable approximations at the coarsest levels.

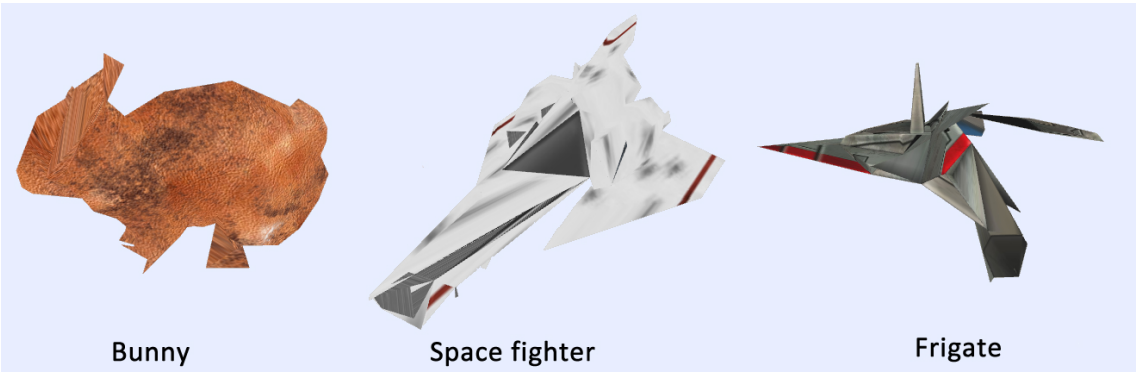

Figure 5.9: Textured sample models: 30% LOD.

Typically, every model consisting of a non-planar mesh has a certain amount of discontinuities in the texture mapping attribute, corresponding to the areas where the texture map was "cut" to better adapt to the mesh topology. The proposed approach is able to tolerate relatively sparse discontinuities, but degenerates if the number of

discontinuities (hence, the amount of replicated vertices) weighs too much on the total vertex count, such as in the *"Frigate"* sample.
It is worth comparing this result with several approaches, described in the literature, which attempted to extend edge contraction to handle vertex attributes.
In [57], Sander et al. try to extend Hoppe's PM approach, in order to create a common texture parametrization for the entire progressive mesh sequence of a given input model, by minimising texture stretch and deviation over all the meshes. However, this requires several pre and post-processing steps, which involve partitioning the mesh into charts, and then creating an ad-hoc texture atlas to sample texture maps. Moreover, in the examples presented in the article, only colour and normal maps are considered, with no results shown for texture maps.
As regards the original Garland-Heckbert approach, its extension, introduced in [27], modifies the general metric to take into account vertex attributes. As described in the paper, the approach produces the best results when attributes vary continuously over the surface of the mesh. If that is not the case, vertices need to be replicated along the seams, as with our solution. The Authors suggest that forcing boundary constraints to maintain the seam might produce acceptable results, although they do not show any example simplification of textured, non-planar meshes with discontinuities. Moreover, they explicitly assert that, if constraints proliferate beyond a certain extent, the suggested solution would not work well. In other words, if texture discontinuities are not sparse enough, the quality of the simplifications is bound to decrease.
Although based on a different simplification idea (*vertex clustering* instead of edge contraction), it is worth mentioning the approach described in [63]. In fact, to date that appears to be the only technique that has been extensively used in production. Most importantly, it seems to be able to produce good results with arbitrary meshes, coping well with thin features, multiple attributes with discontinuities, and animation information, such as blend weights and indices.

5.1.2 Procedural terrain generation

The performance of the procedural terrain generation technique was assessed with respect to two factors:

- execution time,

- amount of transferred data, compared to using a set of meshes describing the same surface.

We measured the execution time of the algorithm for a single patch of terrain, and compared the results obtained with different grid sizes, to determine which ones are suitable for real-time terrain generation.
In order to have statistically meaningful measurements, the execution times were computed as an average over 1000 different generations. The results are shown in Table 5.2.

Terrain Patch Size	Triangle Count	Generation Time * (ms) [min]	Generation Time * (ms) [average]	Generation Time * (ms) [max]
129 X 129	32893	9	9,592	44
257 X 257	131325	36	38,508	79
513 X 513	524797	142	149,65	182
1025 X 1025	2100221	564	591,671	698
				*: data acquired over 1000 generations

Table 5.2: Terrain generation performance assessment.

The algorithm is quite fast for grids of small and medium size: 129×129 and 257×257 patches can certainly be generated in real-time. With a grid of size 1025×1025, the algorithm becomes too slow for real-time generation, while the intermediate size (513×513) is still amenable to real-time use, provided that the generation is performed by a background process, so as to avoid blocking frame rendering[1].
However, from our tests, a 257×257 size seems to be the best trade-off between speed and visual quality of the output, because it quickly produces a surface that is enough tessellated to appear quite realistic. Also, that size fits well with using low-resolution textures in the procedural texturing phase.
With the intent of estimating the benefits of procedural terrain generation, in terms of reduced download times, we also measured the average size of the meshes corresponding to the surface generated by the algorithm. Table 5.3 illustrates the results, as well as expected download times for the four different patch sizes. Note that those figures could be reduced, by compressing the transmitted JSON representation of the meshes (as stated before, the weight would be decreased to slightly more than $\frac{1}{4}$ of the original).

[1]JavaScript code can be run in the background by using HTML 5 *Web worker*[61] objects, supported on all major browser, save for Internet Explorer.

Terrain Patch Size	Equivalent Model Size * (KB)	Expected DL Time (s) [2 Mbit/s]	Expected DL Time (s) [15.9 Mbit/s]
129 X 129	1038	4,0547	0,5098
257 X 257	4330	16,9141	2,1267
513 X 513	18173	70,9883	8,9258
1025 X 1025	75932	296,6094	37,2947
			*: uncompressed JSON format

Table 5.3: Procedurally generated terrain patches: equivalent model sizes.

However, even if compression was used, in order to render non-procedurally generated terrain the client would constantly need to download new patches from the server, which would likely result in missing terrain parts, if the required data is not timely received.
Our approach, if feature injection is not used, only needs to initially fetch the set of configuration parameters. Once this very small packet ($\sim$ 80 bytes) is received, the algorithm can generate an arbitrary number of patches, with no need for further data. When using feature injection, the comparison still remains extremely favourable, since the generation algorithm only needs to receive the pre-initialized points, and only for those terrain patches that actually have injected features in them.
On average, specifying height values for 10 points of a grid requires 300 bytes of data from the server. Of course, to further reduce the impact of feature injection with respect to download times, the lists of pre-initialized points can be compressed. As an example, injecting the mountain valley shown in Section 3.3.1 requires $\sim$ 38 KB of data (less than 10 KB if compressed).
Regarding the limitations of current implementation, they can be synthesized by the following points.

1. The solution is not able to generate caves and arches. Unfortunately, this is a limitation of the algorithm itself. These behaviours could still be reproduced, by using a sort of degenerate feature injection, where all points in the area occupied by the feature are specified. That way, an arch or a cave could be simply "put in place". However, the size of the required data would increase to that of the mesh describing the entire surface of the feature, so this is a feasible workaround only if these cases happen rarely.

2. The simulated terrain is not affected by erosion. However, a post-processing pass could be easily added to simulate that, at the cost of extra computations.

3. The diamond-square algorithm can occasionally introduce artifacts in the generated surface, which show up as localized spikes or creases (as in Figure 5.10). This issue could be addressed by analysing the generated surface and smoothing affected areas, if any. However, this phenomenon only happens very rarely, and when it does it is often hardly noticeable, once procedural texturing has been applied.

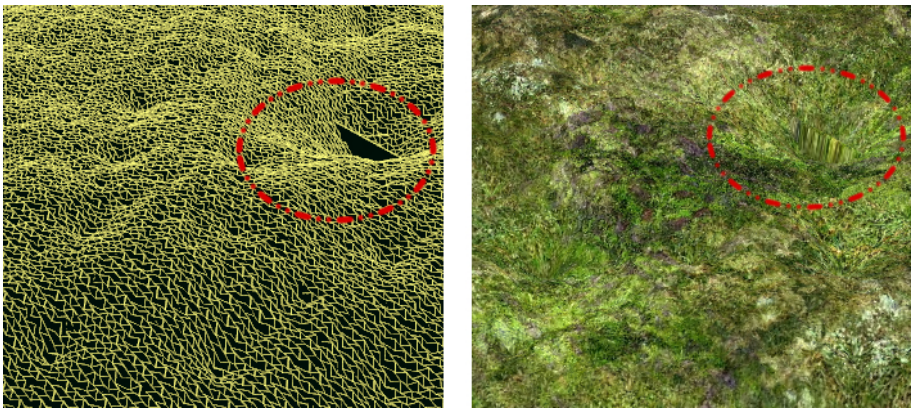

Figure 5.10: Diamond-square algorithm artifacts.

5.1.3 Surface mapping

As regards parallax and relief mapping, it is important to point out that the main reason for implementing them was to prove that mainstream surface mapping approaches are amenable to an efficient, GPU-based implementation in WebGL/OpenGL ES.
With respect to download times, a quantitative evaluation of these techniques would require a comparison between the size of the extra data (that is, the height map) downloaded, and the size of a hypothetical model describing the simulated surface. However, estimating the geometric complexity of such a mesh is not trivial, which makes it hard to setup a direct measurement. Since finding an alternative way to compare the two quantities falls beyond the scope of the dissertation, we decided not to provide quantitative results for parallax and relief mapping.

For reference, we provide a visual comparison of the different results achieved by applying the two approaches to the same inputs. Figure 5.11 shows a close-up view of the same surface, rendered with simple texturing, parallax and relief mapping, respectively.

Figure 5.11: Simple texturing, parallax and relief mapping close up view.

It is worth noting how the offset produced by parallax mapping is limited (as described in Section 3.1.1), while in the corresponding scene rendered with relief mapping the protrusion effect is more pronounced.

5.2 Future work

Concerning the progressive LOD implementation, as previously described the mesh simplification algorithm has some limitations. Hence, the algorithm and the related cost metric could be improved by:

- taking into account a wider range of vertex attributes, possibly considering their value in the cost metric;
- properly handling non-sparse attribute discontinuities, which currently represent the major obstacle to a broad use of the simplification approach;
- enforcing strict preservation of thin features;
- extending the algorithm to the simplification of animated meshes.

With respect to the procedural terrain generation approach used in the framework, an interesting improvement would be the addition of an erosion factor to the current

algorithm. Moreover, as already mentioned, when the next major update to the WebGL API specification will be released, it is very likely that recent GPU features (namely, HW instancing and geometry shaders) will become available, so that more complex, GPU-based terrain generation algorithms (such as [28]) will not be precluded any more.
As regards extending the framework with additional PCG techniques, there are many pre-existing approaches that are amenable to a direct implementation on the WebGL platform, or could be easily adapted. For instance, PCG could be used for:

- *vegetation generation*, based on some type of rule-based system, such as the one described in [40];
- *urban environments*, for instance generated with approaches using a custom-seeded PRNG, as the one presented in [29].

A harder research direction, but well worth the effort, would be the combination of two or more procedural techniques into a single solution (e.g., extending current terrain generation system with procedurally generated rural environments and vegetation).
Although the two surface mapping techniques implemented in the framework are arguably the most used in commercial game development, there are some newer approaches that might be worth exploring. More specifically, *parallax occlusion mapping* ([60]) and *quadtree displacement mapping* ([23]) seem promising candidates to investigate, in terms of execution speed and visual quality of the simulated surfaces.

5.3 Concluding remarks

The dissertation explored several different research directions, with the intent of identifying a set of approaches that allows a 3D client to render complex scenes with limited amounts of data.
The main motivation behind the research was to enable WebGL/OpenGL ES clients to render 3D scenes in real-time, based on data streamed from a remote server. Hence, we focused on techniques which effectively reduce the amount of data that the client must receive to begin rendering the scene.
The proposed approaches were then integrated in a prototype framework, ideally designed to be used as a barebones multi-player game architecture, the implementation

of which consists of a JavaScript/Node.js server, and a WebGL browser application as a client.
Although obviously susceptible to improvements and extensions, the prototype clearly shows how surface mapping, progressive level of detail and PCG techniques could be successfully applied for developing 3D browser games and applications, so that download times are drastically reduced to enable rendering in real-time, but the complexity of the scenes is preserved.
As a final remark, it is worth noting that the WebGL client implementation also works on mobile devices, if the corresponding version of Firefox is used[2]. Therefore, it is technically possible to run the same WebGL application both on desktop computers and on mobile devices. However, since mobile CPUs and GPUs are usually significantly less powerful than their PC counterparts, it would make sense to investigate proper client-side parametrizations, so that the techniques implemented in the framework could be dynamically adjusted to be used, with smooth frame rates, even on mobile hardware.

[2]The implementation was tested on Firefox Mobile Beta (ver. 15.0), on a Samsung Nexus S.

Bibliography

[1] COLLADA - Collaborative Design Activity. URI: http://collada.org.

[2] CopperLicht. URI: http://www.ambiera.com/copperlicht.

[3] D. Bau. A JS seedable random number generator. URI: http://davidbau.com/archives/2010/01/30/random_seeds_coded_hints_and_quintillions.html.

[4] Davide Coppola. GPU Terrains. URI: http://www.m3xbox.com/index.php?page=p_gputerrains.

[5] GLGE. URI: http://www.glge.org.

[6] Internet Explorer WebGL plugin. URI: http://www.iewebgl.com.

[7] JigLibJS - Javascript Rigid Body Physics Library. URI: http://www.jiglibjs.org.

[8] JSON - JavaScript Object Notation. URI: http://www.json.org.

[9] Khronos Group Wiki: WebGL Frameworks. URI: http://www.khronos.org/webgl/wiki/User_Contributions#Frameworks.

[10] Neonux unofficial HW instacing extension for Firefox. URI: http://neonux.com/webgl/instancing.html.

[11] Node.js. URI: http://www.nodejs.org.

[12] OpenGL 3.3 specification. URI: http://www.opengl.org/registry/doc/glspec33.core.20100311.pdf.

[13] OpenGL 4.2 specification. URI: http://www.opengl.org/registry/doc/glspec42.core.20120427.pdf.

[14] OpenGL ES 2.25 specification. URI: http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf.

[15] Three.js - JavaScript 3D library. URI: http://mrdoob.github.com/three.js.

[16] WebGL 1.0 specification. URI: http://www.khronos.org/registry/webgl/specs/latest/.

[17] X3D - Extensible 3D Graphics file format. URI: http://www.web3d.org/x3d.

[18] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[19] Wei Cheng and Wei Tsang Ooi. Receiver-driven view-dependent streaming of progressive mesh. In *Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '08, pages 9–14, New York, NY, USA, 2008. ACM.

[20] Daniel Cohen-Or and Amit Shaked. Photo-realistic imaging of digital terrains. *Comput. Graph. Forum*, 12(3):363–373, 1993.

[21] Xavier Décoret, François X. Sillion, Gernot Schaufler, and Julie Dorsey. Multi-layered impostors for accelerated rendering. *Computer Graphics Forum*, 18(3):61–73, 1999.

[22] W. Donnelly. Per-pixel displacement mapping with distance functions. *GPU Gems*, 2, 2005.

[23] Michal Drobot. Quadtree Displacement Mapping with Height Blending. *GPU Pro*, pages 117–148, 2010.

[24] Jonathan Dummer. Cone step mapping: An iterative ray-heightfield intersection algorithm. 2006. URI: http://www.lonesock.net/files/ConeStepMapping.pdf.

[25] Efi Fogel, Daniel Cohen-Or, Revital Ironi, and Tali Zvi. A web architecture for progressive delivery of 3D content. In *Web3D*, pages 35–41, 2001.

[26] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, pages 209–216, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[27] Michael Garland and Paul S. Heckbert. Simplifying surfaces with color and texture using quadric error metrics. In *Proceedings of the conference on Visualization '98*, VIS '98, pages 263–269, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[28] Ryan Geiss. Generating Complex Procedural Terrains Using the GPU. *GPU Gems*, 3, 2007.

[29] Stefan Greuter, Jeremy Parker, Nigel Stewart, and Geoff Leach. Real-time procedural generation of 'pseudo infinite' cities. In *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, GRAPHITE '03, New York, NY, USA, 2003. ACM.

[30] John C. Hart. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12:527–545, 1996.

[31] Hugues Hoppe. Progressive meshes. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 99–108, New York, NY, USA, 1996. ACM Press/Addison-Wesley Publishing Co.

[32] Hugues Hoppe. View-dependent refinement of progressive meshes. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 189–198, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[33] Alan D. Kalvin and Russell H. Taylor. Superfaces: Polygonal mesh simplification with bounded error. *IEEE Comput. Graph. Appl.*, 16:64–77, May 1996.

[34] K. Raiyan Kamal and Yusuf Sarwar Uddin. Parametrically controlled terrain generation. In *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, GRAPHITE '07, pages 17–23, New York, NY, USA, 2007. ACM.

[35] Tomomichi Kaneko, Toshiyuki Takahei, Masahiko Inami, Naoki Kawakami, Yasuyuki Yanagida, Taro Maeda, and Susumu Tachi. Detailed shape representation with parallax mapping. In *In Proceedings of the ICAT 2001*, pages 205–208, 2001.

[36] Yossarian King. Never Let 'Em See You Pop — Issues in Geometric Level of Detail Selection. In Mark DeLoura, editor, *Game Programming Gems*, pages 432–438. Charles River Media, 2000.

[37] Eric Lengyel. *Mathematics for 3D Game Programming and Computer Graphics, Second Edition*. Charles River Media, Inc., Rockland, MA, USA, 2003.

[38] H. Li, M. Li, and B. Prabhakaran. Middleware for streaming 3D progressive meshes over lossy networks. *ACM Trans. Multimedia Comput. Commun. Appl.*, 2(4):282–317, November 2006.

[39] Kok-Lim Low and Tiow-Seng Tan. Model simplification using vertex-clustering. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, I3D '97, pages 75–ff., New York, NY, USA, 1997. ACM.

[40] Milán Magdics. Real-time generation of L-system scene models for rendering and interaction. In *Proceedings of the 2009 Spring Conference on Computer Graphics*, SCCG '09, pages 67–74, New York, NY, USA, 2009. ACM.

[41] Adrien Maglo, Guillaume Lavoué, Celine Hudelot, Ho Lee, Christophe Mouton, and Florent Dupont. Remote scientific visualization of progressive 3D meshes with X3D. In ACM, editor, *International Conference on 3D Web Technology (Web3D)*, July 2010.

[42] Jean-Eudes Marvie, Pascal Gautron, Patrice Hirtzlin, and Gael Sourimant. Render-time procedural per-pixel geometry generation. In *Proceedings of Graphics Interface 2011*, GI '11, pages 167–174, School of Computer Science, University of

Waterloo, Waterloo, Ontario, Canada, 2011. Canadian Human-Computer Communications Society.

[43] Stan Melax. A Simple, Fast and Effective Polygon Reduction Algorithm. *Game Developer*, 5:44–49, 1998.

[44] F.J. Melero, P. Cano, and J.C. Torres. Progressive transmission of large archaeological models. In *In Beyond the artifact: Computer Applications in Archaeology. Bar International Series*. Ed. Archaeopress, 2004.

[45] Netindex.com. European Household Download Index Survey, August 2012.

[46] Ofcom. Measuring Mobile Broadband in the UK, December 2010.

[47] Manuel M. Oliveira, Gary Bishop, and David K. McAllister. Relief texture mapping. In *SIGGRAPH*, pages 359–368, 2000.

[48] Jacob Olsen. Realtime procedural terrain generation - Realtime synthesis of eroded fractal terrain for use in computer games. Technical report, Department of Mathematics and Computer Science [IMADA], University of South Denmark, October 2004.

[49] Sean O'Neill. Accurate atmospheric scattering. *GPU Gems*, 2, 2006.

[50] Ken Perlin. Improving noise. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 681–682, New York, NY, USA, 2002. ACM.

[51] F. Policarpo and M. Oliveira. Relaxed Cone Stepping for Relief Mapping. *GPU Gems*, 3, 2007.

[52] Fábio Policarpo, Manuel M. Oliveira, and João L. D. Comba. Real-time relief mapping on arbitrary polygonal surfaces. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, I3D '05, pages 155–162, New York, NY, USA, 2005. ACM.

[53] Eric Risser. Rendering 3D volumes using per-pixel displacement mapping. In *Proceedings of the 2007 ACM SIGGRAPH symposium on Video games*, Sandbox '07, pages 81–87, New York, NY, USA, 2007. ACM.

[54] Eric Risser. True impostors. *GPU Gems*, 3, 2007.

[55] Pawel Rohleder and Aleksander Netzel. Procedural Content Generation on the GPU. *GPU Pro*, 2:29–37, 2011.

[56] Jarek Rossignac and Paul Borrel. Multi-resolution 3D approximations for rendering complex scenes. In *Modeling in Computer Graphics: Methods and Applications*, pages 455–465, 1993.

[57] Pedro V. Sander, John Snyder, Steven J. Gortler, and Hugues Hoppe. Texture mapping progressive meshes. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 409–416, New York, NY, USA, 2001. ACM.

[58] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '92, pages 65–70, New York, NY, USA, 1992. ACM.

[59] Christopher Schwartz, Roland Ruiters, Michael Weinmann, and Reinhard Klein. WebGL-based Streaming and Presentation Framework for Bidirectional Texture Functions. In Franco Niccolucci, Matteo Dellepiane, Sebastián Peña Serna, Holly E. Rushmeier, and Luc J. Van Gool, editors, *VAST*, pages 113–120. Eurographics Association, 2011.

[60] Natalya Tatarchuk. Dynamic parallax occlusion mapping with approximate soft shadows. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, I3D '06, pages 63–69, New York, NY, USA, 2006. ACM.

[61] W3C. Web Workers - W3C Candidate Recommendation. URI: `http://www.w3.org/TR/workers/`, May 2012.

[62] Terry Welsh. Parallax mapping with offset limiting: A per-pixel appproximation of uneven surfaces. 2004.

[63] Andrew Willmott. Rapid simplification of multi-attribute meshes. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 151–158, New York, NY, USA, 2011. ACM.

[64] Zhi Zheng, Tony K. Y. Chan, and Edmond C. Prakash. Rendering of large 3D models for online entertainment. In *Proceedings of the 2006 international conference on Game research and development*, CyberGames '06, pages 163–170, Murdoch University, Australia, Australia, 2006. Murdoch University.

[65] Renaldas Zioma. GPU-Generated Procedural Wind Animations for Trees. *GPU Gems*, 3, 2007.