Distributed Texture-based Terrain Synthesis

Thesis by
Flora Ponjou Tasse

In Fulfillment of the Requirements
for the Degree of
Master of Science in Computer Science

Supervisors:
James Gain
Patrick Marais

University of Cape Town
Cape Town, South Africa

June 2011

I know the meaning of plagiarism and declare that all the work in the dissertation, save for that which is properly acknowledged, is my own.

Abstract

Terrain synthesis is an important field of Computer Graphics that deals with the generation of $3D$ landscape models for use in virtual environments. The field has evolved to a stage where large and even infinite landscapes can be generated in real-time. However, user control of the generation process is still minimal, as well as the creation of virtual landscapes that mimic real terrain. This thesis investigates the use of texture synthesis techniques on real landscapes to improve realism and the use of sketch-based interfaces to enable intuitive user control.

We present a patch-based terrain synthesis framework constrained by user-specified curvilinear features such as ridges and valleys. The framework copies patches of a Digital Elevation model from a real landscape onto the output terrain such that the output contains the curvilinear features specified by the user and exhibit the characteristics of the real landscape. A user specifies where terrain features appear in the generated terrain by providing a 2D sketch map or drawing $2\frac{1}{2}D$ sketched curves in the sketching interface. Features are automatically extracted from the user constraints and a real landscape, and a patch-based texture synthesis guided by these features produces a realistic terrain that fits the user constraints. A novel patch merging technique is proposed to remove boundary artifacts created by overlapping patches.

We show that terrains generated by our system are more realistic than current state-of-the-art terrain synthesis methods. We also present two GPU acceleration techniques and show that these parallel implementations accelerate the matching process by 6 to 30 times. We conclude that texture-based terrain synthesis provides an excellent solution to user control and realism challenges of virtual landscape generation.

Contents

Chapter 1

Introduction

One of the key challenges in Computer Graphics is the generation of believable landscapes. Researchers have explored noise-based algorithms for procedurally creating terrains and the use of physics-based techniques to simulate erosion effects. Recent methods have been proposed to apply texture synthesis algorithms to terrain generation. Texture synthesis takes a sample texture and attempts to create a new texture visually similar to the sample. Following a similar direction, this thesis presents a novel patch-based texture synthesis framework for generating realistic terrains from example landscapes. The system is controlled by a terrain sketching interface or user sketched maps that specify large features such as ridges and valleys as desired by the user. A novel patch merging technique is presented to remove visible boundary seams in the output landscape. The introduction provides a brief background and motivation for the project, followed by a description of its objectives and the structure of the thesis content.

1.1 Motivation

Landscapes form an integral part of many virtual environments. Virtual terrains are used in video games, films, advertisements, and many other software systems involving outdoor environments such as flight simulators. Believable terrains play an important role in ensuring immersion in a virtual environment. Some applications of landscapes in computer-generated words are illustrated in Figure 1.1.

When the landscape is only used as an aesthetic element (for example as the background) and is not involved in any direct interaction with the user, artists often draw a 2D terrain profile manually or use images of real landscapes. This practice arose in the early virtual environments and is still commonplace today (Half-Life 2, 2004). The terrain is simply added as a texture which reduces both the space requirements and the workload of the artist. However, it is often the case that the landscape must be navigable. In such cases, a 3D terrain model is needed.

The US Geological Survey (2011) provides freely downloadable models of real landscapes that can be added to 3D environments if needed. The use of real terrains has the advantage of making the environment more believable with little effort but limits the range of terrains that can be used. Furthermore, it is possible that no real landscape has the properties that the artist requires. Thus, artists often model the terrain manually using 3D modelling tools or painting a 2D image such that luminosity values represent height values. As the need for more believable and large

terrains increases, manually modelling terrains becomes more complex and tedious. Procedural terrain generation is a field of Computer Graphics that deals with algorithmically generating landscapes as an alternative to manual creation. Given a set of parameters, a procedural method generate landscapes of a user-defined possibly infinite size. Several commercial products for terrain generation use methods based on randomly displacing the height values of a flat terrain based on noise (Terragen, 2010). However, these methods cannot simulate the erosion effects present in real landscapes, such as drainage patterns.

(a) Image from the video game *Prince of Persia: The Forgotten Sands*, by Ubisoft

(b) Image from the movie *Lord of the Rings*, by New Line Cinema

(c) Image from the software *Microsoft Flight Simulator X*, by Microsoft

Figure 1.1: Examples of landscapes use in virtual environments

Often, the terrains generated by these methods need to undergo a physics-based simulation so that erosion effects can be incorporated. Many commercial packages are based on combinations of noise-based terrains and physics-based simulations (Pandromeda, 2010). These simulations have a high computational cost and demand that the user has a working knowledge of various erosion models. A new trend in procedural terrain generation is to create landscapes from information extracted from an example terrain. The current state-of-the art in this respect is Zhou et al. (2007)'s algorithm. This technique represents all terrains as 2D textures where each pixel value represents a height value. Given a user sketch map and a real landscape, the algorithm creates a new terrain by copying square blocks of pixels (*patches*) from the real terrain so that curvilinear features like ridges and valleys specified by the user in the sketch map appear in the output. This patch-based texture synthesis method produces a realistic terrain that has the same small-scale characteristics as the real terrain. Furthermore, the control lacking in most noise-based and physics-based terrain generation is implemented by allowing the user to specify the desired large scale features present.

Our proposed system is based on this technique. However, the use of 2D sketch maps gives limited control to the user, as they only specify the layout of the large scale features and do not constraint these features to have specific height values. In contrast, the Terrain Sketching Interface (Gain et al., 2009), enables users to fully control the generation of a new terrain by deforming a default terrain such that it fits $2\frac{1}{2}D$ silhouette and boundary curves drawn by a user. Multiresolution surface deformation and noise are used to deform and add detail to the terrain surface. Despite the full intuitive control users have over the generation, the terrains generated by this interface share the poor realism of noise-based terrains. *We propose to extend this sketch-based system by adding patch-based texture synthesis.* Combining the sketching interface and

the texture-based terrain generation will provide a system that gives wider control to the user and produces realistic terrains.

Most patch-based synthesis methods have $O(n^2)$ time complexity. They involve several iterations, where each iteration consists of comparing patches in order to select the best patch to place in the output. As the size of the example increases, so does the computation time of the synthesis. Programmable GPUs have evolved to be more powerful than CPUs for certain problems by an order of magnitude and new application interfaces have been developed to enable General Purpose GPU (GPGPU) computing. These have spawned new research focused on mapping applications with data parallelism onto graphics hardware. Texture synthesis methods, including patch-based texture synthesis, are a prime example of such applications. Thus, we accelerate the terrain generation computation time by implementing parallel work in the patch-based synthesis on the GPU.

1.2 Aims

The primary objective of this thesis is to develop a terrain synthesis system that both produces realistic landscapes and provides the user with intuitive control over the synthesis. This objective requires that we develop solutions that address the following requirements:

1. Control texture-based terrain synthesis with user sketched curves drawn in an existing terrain sketching interface.
2. Modify the state-of-the-art patch-based terrain synthesis to further improve the matching of user constraints and the realism of the output.
3. Implement a novel patch merging technique to remove boundary seams that appear in the output as the result of overlapping multiple patches.
4. Optimize the proposed framework with a parallelized implementation on graphics hardware.

The first objective gives control of the terrain generation to the user through an intuitive and yet powerful interactive sketching interface. Then we wish to improve the current state-of-art in patch-based terrain synthesis so that the output fits the constraints set by the user. Another goal of this project is to further enhance the realism of the generated landscapes by proposing a novel patch merging method that removes the boundary seams visible in the output. Previous merging techniques fail to eliminate boundary artifacts on 3D landscapes because they are targeted at 2D textures. Finally, we aim to speed-up the terrain synthesis by implementing a portion of the proposed algorithm on GPU.

1.3 Thesis structure

This thesis is structured as follows:

- Chapter 2 presents background on state-of-the-art techniques in landscape generation. Applications of texture synthesis to creating landscapes are also discussed. We provide

an introduction to exampled-based texture synthesis methods and various optimization schemes. An outline of current user controls for terrain generation is also presented.

- Chapter 3 first describes in more details the patch-based texture synthesis algorithm proposed by Zhou et al. (2007). The problems inherent to this method are outlined and potential solutions are proposed. The design of our proposed system on CPU is then presented. We specifically provide details on a novel patch merging technique and a brief visual assessment of this method in contrast to two other merging approaches.

- Chapter 4 provides details of the acceleration of the proposed framework on graphics hardware. Two separate GPU optimization techniques for patch-based texture synthesis are presented and a performance analysis of these techniques along with their CPU counterpart is performed.

- Chapter 5 presents a user study designed to compare the realism of the terrains generated by the proposed framework against real landscapes and terrains deformed to fit sketched curves drawn in a terrain sketching interface. The user study is also designed to investigate the realism of the proposed patch merging technique when compared to two other methods. An overview of the experiment design is presented, as well as data analysis and interpretation of the statistical results. This is followed by a visual assessment of the terrains synthesised by our system and a discussion of influential parameters.

- Chapter 6 concludes the thesis and proposes potential future work to further improve the diversity of the generated landscapes and acceleration the terrain synthesis computation time.

Chapter 2

Related Work

This thesis applies texture synthesis techniques to terrain generation. In this chapter, we first provide some background information on terrain generation strategies (Section 2.1), namely fractal-based generation, physics-based simulation, and texture-based synthesis. This is followed by a literature review of state-of-the art texture synthesis techniques (Section 2.2), such as pixel-based and patch-based schemes. Finally, we provide a more detailed discussion of the state-of-the-art technique in texture-based terrains synthesis.

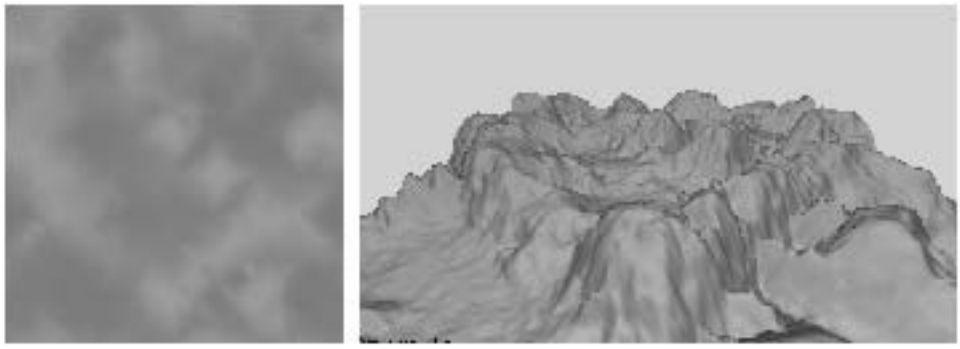
2.1 Terrain generation

The simulation of natural environments is a common application in computer games, movies, and training systems. As a vital element of any outdoor environment, terrain is extensively used in these applications. A terrain is a geometrically complex object, and manually creating one is a very long and tedious process, especially for large environments. Procedural terrain generation, also referred to as terrain synthesis is the process of algorithmically generating a landscape. For this purpose, landscapes are usually represented in digital form by digital elevation models (DEM). A DEM is a discrete quantitative model that represents the surface of a landscape.

2.1.1 Terrain representation

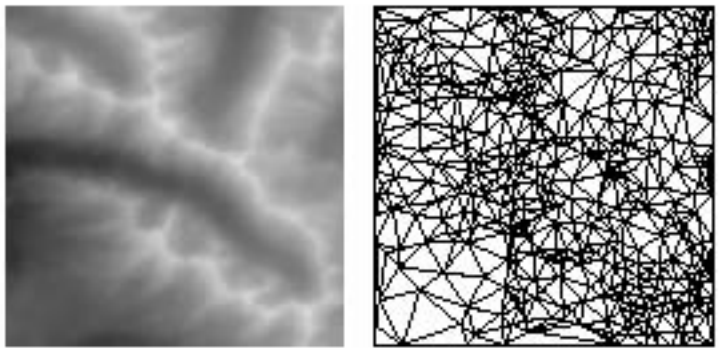
The most popular way of representing DEMs is a grid-based approach. The terrain is digitally represented by a 2D grid of elevation values sampled at regular intervals. A *heightmap* or *heightfield* is an image representation of this grid, with elevation values stored as pixel values. Figure 2.1 shows a heightfield and the terrain it represents. Heightmaps have several advantages. The storage capacity is reduced by storing a 2D array of sampled elevation values instead of all 3D points on the terrain surface. For this reason, a significant number of real-life terrains are preserved in a raster format. Moreover, the regular structure of the heightmap makes terrain manipulation and the optimisation of operations such as path finding and collision detection easier to achieve. Unfortunately, this format has finite resolution so that the terrain appears blocky at a close range. Also, heightmaps cannot represent three-dimensional structures like overhangs and caves, where a 2D point has multiple elevation values.

Terrain can also be represented with an arbitrary mesh of irregular 2D primitives such as polygons. A special case of such a non-uniform mesh is a Triangulated Irregular Network (TIN)



(a) Heightmap format **(b)** Corresponding 3D Terrain model

Figure 2.1: Heightmap representation. (a) A 2D texture representing a terrain. (b) 3D rendering of the corresponding terrain. Each luminosity value in the texture is equivalent to a height value in the terrain model.



(a) Heightfield format **(b)** TIN format

Figure 2.2: Triangulated irregular network representation.(a) A 2D texture representing a terrain. (b) A top-down view of the Triangulated Irregular Network representation of the same terrain. Smaller triangles are used along the ridge lines and larger polygons are used in smoother areas.

(Pajarola et al., 2002). A TIN models the terrain surface as a set of contiguous non-overlapping triangles of variable size, whose vertices are adaptively chosen (using an algorithm like Delaunay triangulation) to produce an accurate representation (Silva et al., 1995). An example of a landscape in the TIN format is shown in Figure 2.2. This format captures three-dimensional characteristics that the height map fails to represent and therefore support a wider range of terrains. Furthermore, they support variable level of detail. Detailed areas have more and smaller triangles whereas smooth regions have less and larger triangles. This significantly reduces the amount of data stored. However, TINs share the same limited resolution as heightmaps and are more complex to manipulate procedurally. This format is mostly suitable for manual terrain modelling and rendering.

Alternative implementations include voxel grids (Dorsey et al., 2006) and continuous functions (Pandromeda, 2010). A voxel grid is a 3D grid of voxels (the three-dimensional equivalent of pixels) that are either empty or filled. It supports truly 3D structures such as caves but uses a large amount of memory (during rendering) and storage space. Continuous functions on the

other hand reduce storage space and can be viewed at any resolution without the appearance of any artifact. Unfortunately, they are difficult to manipulate if one desires to modify the terrain and, depending on the complexity of the function, they can be very expensive to render.

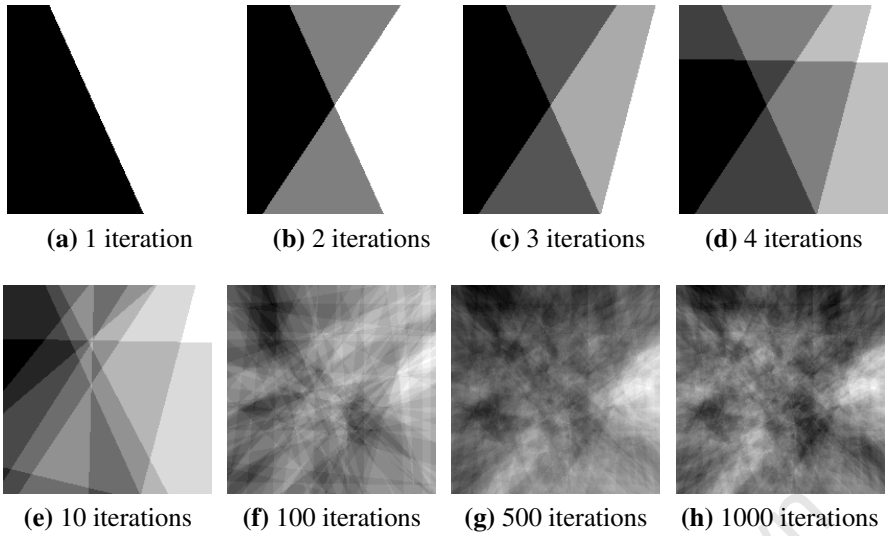
Most terrain generation and rendering systems in research and industry use the heightmap data structure because of its reduced storage capacity and simplicity (Terragen, 2010; Bryce, 2010; World Machine 2, 2010). Its image representation means that image processing techniques such as smoothing can easily be applied directly to the terrain. Furthermore, most real-life terrains are only available in heightmap format. The desire to have easy access to real landscape data and comparing the benefits of our framework over previous work led us to choose the heightfield representation as our terrain format. We now review current procedural techniques for generating terrain.

2.1.2 Fractal-based terrain generation

Fractal-based terrain generation was introduced by Mandelbrot in his book entitled "The fractal geometry of Nature" (Mandelbrot, 1982). In this book, he observes that natural shapes such as terrains, clouds, and trees share the property of *self-similarity*: they are statistically invariant under magnification. In other words, zoomed in sections of a shape are statistically similar to the shape itself. From these observations, Mandelbrot (1982) introduces Fractal Geometry, a mathematical model for shapes in nature (such as trees, clouds, and coastlines) that cannot be easily described with the traditional Euclidean Geometry. Self-similar shapes are described by fractional Brownian motion (fBm) also known as the "random walk process". fBm consists of steps in random directions, where the step-lengths are normally distributed with a mean of zero and a standard deviation determined by the desired fBm's roughness. Many others have extrapolated Mandelbrot's work with approximations of fBm able to produce fractal terrains (Voss, 1985; Miller, 1986; Lewis, 1987; Saupe, 2003).

Faulting methods are one of the earliest fractal terrain generation techniques (Mandelbrot, 1982; Voss, 1985). The Poisson faulting technique applies successive random displacements to the heightmap, creating a Brownian surface (Mandelbrot, 1982). This method is easily implemented by iteratively drawing a random line on an initially flat heightmap and displacing one side of the line by a random height value h. The range of the random value h decreases with each iteration to avoid abrupt height changes. Figure 2.3 illustrates this iterative process. Faulting techniques fix the resolution of the heightfield at the beginning of the generation process and therefore have no consideration for level of detail (LOD). LODs are important in terrain generation because terrain features appear at different scales. Features such as cracks occur at a finer level of detail while larger features such as mountains appear at a coarser level. LODs give the user control over the desired amount of detail, a feature not present in faulting techniques. Furthermore, they are extremely slow due to the large number of iterations required ($O(n^3)$ time complexity). Subdivision techniques are usually preferred over faulting, especially in computer games where low computation is an important factor.

Midpoint displacement methods are subdivision techniques that progressively increase the level of detail of the terrain by adding finer detail at each iteration (Fournier et al., 1982; Miller, 1986; Lewis, 1987). The subdivision interpolates between neighbouring points and displaces the new



(a) 1 iteration **(b)** 2 iterations **(c)** 3 iterations **(d)** 4 iterations

(e) 10 iterations **(f)** 100 iterations **(g)** 500 iterations **(h)** 1000 iterations

Figure 2.3: Fault formation. During each iteration, a line is randomly drawn on the terrain and one side of line is displaced by a small random number.

intermediate points by successively smaller random values. Midpoint displacement techniques differ mostly by the number of points interpolated during an iteration. The simplest implementation starts with a plane with random values assigned to each corner. The plane is subdivided into four smaller planes by interpolating between its four corner points and displacing the interpolated point (the midpoint) by a random value constrained by the desired roughness. This is repeated on the newly created planes until the desired level of detail is reached. The process is illustrated in Figure 2.4. Midpoint displacement methods such as the one described above are easy to implement and run in linear time. Furthermore, they support variable level of detail which can be useful in optimising the landscape rendering. However, the sharing of edges through subdivision steps often leave traces of the earlier stages of the subdivision (Saupe, 1988). This is manifested in 2D by creases that lie along straight lines. One way of dealing with this artifact is the use of the *successive random addition* method. In addition to the displacement of the midpoints, all points are displaced at each iteration by interpolating points of the previous iteration (Voss, 1985; Saupe, 1988). This extra work slows down the terrain generation, which explains the popular preference for Midpoint displacement. Both techniques produce very periodic terrains with repeating patterns, unlike landscapes in nature. Moreover, control is limited to the terrain roughness.

Spectral synthesis techniques in contrast offer better control over the nature of the landscape generated (for instance a hilly terrain). Spectral synthesis randomly selects the coefficients of a spectral representation of the terrain (Fourier or Wavelet transform) and then uses the inverse transform to generate the heightfield. Wavelet transform techniques are easier to implement and better at representing a structure such as a terrain at multiple levels of resolution (Bruun and Nilsen, 2001). Spectral synthesis purely interprets the fBm but every inverse transform has a $O(nlog(n))$ time complexity. In addition, the control over the nature of the terrain is only possible at a global level: local control is non-existent.

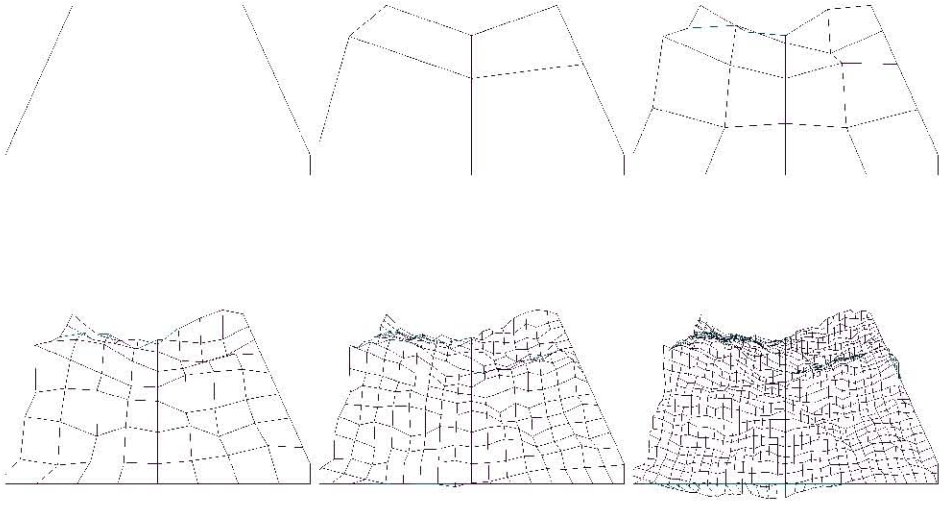


Figure 2.4: First six iterations of a midpoint displacement

Noise synthesis techniques address these issues and proceed by superposing scaled-down copies of a band-limited, stochastic noise function. The use of noise synthesis in terrain generation is very common in commercial terrain engines (Bryce, Mojoworld) and widely investigated in research (Miller, 1986; Saupe, 1988; Musgrave et al., 1989; Schneider et al., 2006). Terrains are created by summing up weighted bands of noise limited to a range of frequencies. The noise function commonly used in this context was introduced by Perlin (1985) and has been improved over the years (Perlin, 2002). However, Perlin noise is weakly band-limited: each Perlin band contains only frequencies of a power of 2. This creates a loss-of-detail and aliasing problems addressed by the Wavelet noise introduced by Cook and DeRose (2005). Wavelet noise is defined by a set of coefficients that are obtained by proceeding as follows:

- An image R of random noise is downsampled to create a half-size image $R\downarrow$.
- $R\downarrow$ is upsampled to an image $R\downarrow\uparrow$ the same size as R.
- A set of coefficients is obtained by subtracting $R\downarrow\uparrow$ from R. It is the band-limited part.

The downsampling and upsampling steps are performed with Wavelet analysis techniques (Stollnitz et al., 1995). Wavelet noise is almost perfectly band-limited and offers good detail with less aliasing. Fast and easy to implement, it is used in several terrain applications (Gain et al., 2009; Dachsbacher, 2006). We refer the reader to Lagae et al. (2010) for descriptions and comparisons of state of the art noise functions.

More information on fractal terrain generation methods can be found in Ebert et al. (2002). These methods are easy to comprehend and implement which is why they are used in many commercial products such as Terragen (2010). The main drawback common to the schemes covered above is the lack of user control or the use of non-intuitive parameter manipulation. Some researchers

address this problem by adding constraints to the landscape generation. User control models are discussed in Section (2.1.5). Furthermore, the fBm is statistically homogeneous and isotropic. At different positions the fBm looks similar. This is generally not the case with terrains in nature. Additionally, fractal terrains cannot capture erosion effects such as clear ridgelines and valleylines. Physical simulations of erosion are often performed on a fractal terrain to increase its realism.

2.1.3 Physics-based techniques

Physically-based techniques generate artificial terrains by simulating erosion effects. This approach dates back to Kelley et al. (1988) with the generation of stream network patterns to determine the topography of a terrain surface. Musgrave et al. (1989) present an erosion algorithm that simulates the erosion of a user-specified terrain, based on erosion laws from fluvial geomorphology.

A particularly easy to simulate erosion effect is thermal erosion. Thermal weathering iteratively distributes material from higher to lower points until the talus angle, the maximum angle of stability, is reached. The result is that material is removed from steep slopes and piled up at their feet. At each time step $t+1$, for each vertex v, the difference of the altitude a_t^v of vertex v and the altitude a_t^u of a neighbouring vertex u is compared with a user-defined talus angle T . If the difference is greater than T, a fixed portion c_t of the difference is moved from vertex v to the neighbour u.

$$a_{t+1}^u = \begin{cases} a_t^v - a_t^u > T: & a_t^u + c_t\,(a_t^v - a_t^u - T) \\ a_t^v - a_t^u \leq T: & a_t^u \end{cases}$$

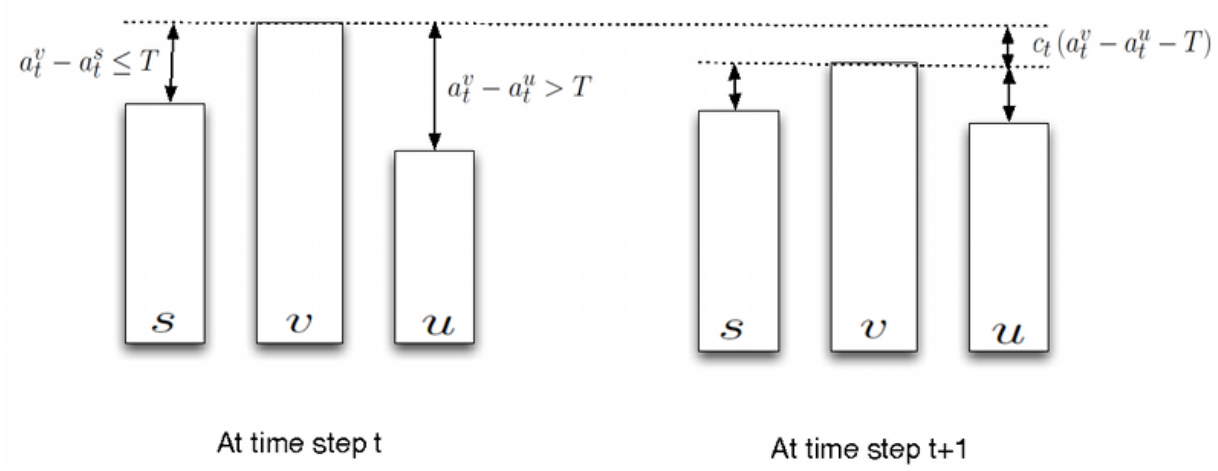
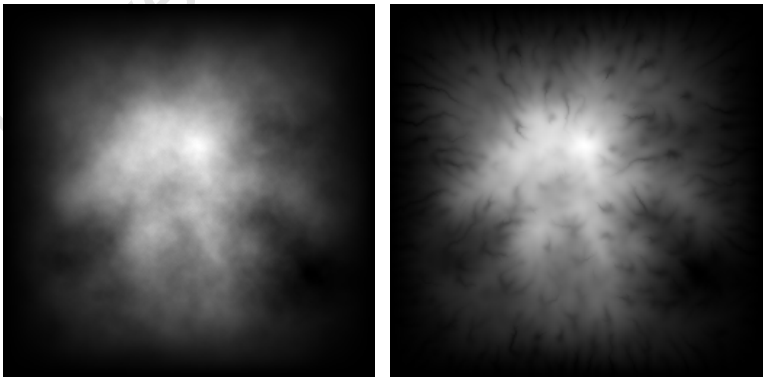


Figure 2.5: Thermal erosion. At time step t, the height difference between vertices v and u is larger than the talus angle T. Thus a portion of a_t^v is removed and added to a_{t+1}^u.

Figure 2.5 illustrates the erosion process. This is a simple technique that simulates the effects of thermal erosion. It is often used to improve the realism of fractal landscapes (Musgrave et al.,

1989; Olsen, 2004; Peytavie et al., 2008). Benes et al. (1997) extend this technique by proposing a hierarchical algorithm that distributes material to 8 neighbouring cells. The algorithm is then optimised by downscaling the heightmap before each erosion step and upscaling it afterwards. Thermal erosion is adequate for simulating particle deposition but cannot create drainage patterns, an effect of fluvial erosion.

Fluvial or hydraulic erosion is simulated by depositing water on the terrain vertices and allowing it to flow downhill, transferring sediment to lower neighbouring points. At each time step t, each vertex v is associated with an altitude a_t^v, a volume of water w_t^v and an amount of sediment s_t^v carried by the water. An excess of water and sediment is passed from the vertex v to every neighbouring vertex u. Figure 2.6 illustrates the effects of a simple fluvial erosion. This simple hydraulic model is used in several papers (Kelley et al., 1988; Musgrave et al., 1989) but more complex models exist (Chiba et al., 1998; Nagashima, 1998). Roudier et al. (1993) extend this model by attaching different materials to each grid point so that water at points with different materials behaves differently. This method simulates chemical dissolution but still uses a cellular automaton model similar to the hydraulic model above. It does not consider the inertia of water flow and so simulating large scale ridge and valleys is problematic. Chiba et al. (1998) propose a solution to this problem that is based on the vector field of the water flow. The vector field is obtained by placing water particles on each grid point of the heightfield and local gradient is used to approximate the velocity of each particle. When a water particle moves, an amount of sediment is attached to it and deposited according to the vector field. This method is able to produce hierarchical structures of ridges but is considerably more time-consuming than a simple hydraulic model. Nagashima (1998) combines thermal and fluvial erosion by using a river network pregenerated with a 2D fractal function. The river banks are then eroded to simulate fluvial erosion. More recent work by Neidhold et al. (2005) presents a physically correct simulation based on fluids dynamics and interactive methods that enable the input of global parameters such as rainfall or local water sources. These physics-based techniques give a more realistic look to otherwise very homogeneous terrains.



(a) **(b)**

Figure 2.6: Fractal terrain before and after fluvial erosion. After fluvial erosion, the terrain exhibit drainage patterns and is thus more realistic.

Physical simulation can be extremely slow when the number of vertices or time steps increases.

Anh et al. (2007) propose a fast erosion technique for generating high resolution terrain that takes advantage of GPU capabilities. Their algorithm is implemented on a 2D structure which stores height fields, water quantity, dissolved material and water velocities. The speed on GPU improves six to ten times over a sequential implementation. Other optimisations achieve realtime terrain generation at the expense of physical correctness (Olsen, 2004). Another drawback of physics-based methods is that they require a good understanding of the underlying physical laws. They are also difficult to control. Physics-based techniques limit user control to the setting of the initial parameters. A faster way of producing realistic-appearing terrains is the use of natural landscape data.

2.1.4 Texture-based techniques

Combining realism and easy user control is extremely difficult for fractal-based and physics-based terrain generation. An alternative to these techniques are texture-based or example-based methods that use texture synthesis techniques to generate a terrain by copying height information from a sample heightfield. By using a real-life heightfield as the sample, the realism of the resulting terrain is significantly improved. This is a very recent approach to terrain generation and it has been the object of little research (Chiang et al., 2005; Brosz et al., 2006; Dachsbacher, 2006; Zhou et al., 2007).

Dachsbacher (2006) adapts pixel-based texture synthesis (See Section 2.2.1) based on texture synthesis by non parametric sampling (Efros and Leung, 1999) to terrain generation. Unlike textures, differences in elevation and local derivatives are very noticeable in heightfields. The difference in height values can be noticed from the elevation image representation, but discrepancies between derivatives only become really noticeable during rendering. To determine which pixel from the sample terrain must be copied at a position $p = (x, y)$ of the output, Dachsbacher (2006) compares the distance between their neighbourhood with a function based on both the height values and derivatives. This technique suffers from the high computational cost of Efros and Leung (1999)'s method and the inability of pixel-based approaches to preserve edges. Dachsbacher (2006) argues that a patch-based approach is an attractive alternative as well as an implementation on hardware to speed-up the synthesis.

Brosz et al. (2006) present a terrain synthesis by-example that extracts small-scale characteristics from a target terrain that contains the desired small-scale features. These are then applied to a base terrain that has the desired large-scale characteristics. Patch-based texture synthesis is used to match areas in the base terrain with regions in the target terrain. This matching process ensures that information copied from one type of feature (for example a mountain peak) in the target is transferred to a similar type in the base terrain. The texture synthesis is based on Image quilting (Efros and Freeman, 2001) (see Section 2.2.2) and proceeds by first dividing the base terrain into a grid of overlapping square patches. Then, for each patch i in the base terrain, a similar patch j is chosen from the target and the extracted characteristics in block j are copied to block i. Unlike Image Quilting that copies blocks, this approach copies small-scale details and linearly blends them between patches. Therefore, the generated terrain does not have the boundary artifacts problem of patch-based texture synthesis techniques. However, while this technique increases the level of detail of a base terrain, it does not create a markedly new terrain. The realism of the

output is still strongly reliant on that of the base terrain.

Chiang et al. (2005) propose a patch-based approach that lets users construct geometric primitives to indicate the desired terrain profile. Then for each geometric or terrain primitive, a matching terrain unit is selected from a database of terrain patches characterised by features such as mountains and tablelands. These patches are manually segmented from real-life landscapes. The matching process maps a geometric primitive to a patch by computing cross-section, mountain, and terrain contour similarity. The selected terrain units are placed so that adjacent units partially overlap. The overlap areas are stitched using a cut approach. For each scanline of the overlap, a cutting pixel that minimises the elevation difference among pixels on the scanline is selected. Only pixels on one side of the cutting pixel are selected from the patch. Unfortunately, this patch stitching technique does not consider elevation differences in the vertical direction, which results in very noticeable boundary artifacts. Furthermore, the manual segmentation of the sample landscape increases the workload of the user.

Ong et al. (2005) implement patch-based terrain synthesis using genetic algorithms. The user provides a rough sketch of terrain region boundaries that indicates the approximate size, shape and position. Each region boundary is then broken down, with subdivision, into a sequence of points. A genetic algorithm (GA) is applied to the sequence to create more uneven and realistic boundaries. The user associates a specific terrain type with each enclosed region. Finally, a database of sample heightfields is used as the population of a genetic algorithm. The GA blends together elements of the provided database according to the modified boundary regions and user-specified terrain types. Because of the random nature of genetic algorithms, successive runs with the same input data will produce different results. One cannot intuitively produce the same terrain twice, which might impede the design process of the user. Furthermore, each sample heightfield in the database is hand-picked and represents only one terrain type. The amount of effort needed from the user is thus increased.

Zhou et al. (2007) propose a terrain synthesis from DEMs that generates compelling results (See Figure 2.8) by using a user-sketched feature map and a real-life landscape for patch-based texture synthesis. The algorithm is illustrated in Figure 2.7 and consists of three main steps:

- Terrain features are extracted from the example height field and the feature map. Terrain features are large-scale curvilinear features such as ridges and valleys. Feature extraction uses a technique borrowed from the Geomorphology literature named the Profile recognition and Polygon breaking Algorithm (PPA). This algorithm, proposed by Chang et al. (1998), produces a tree that represents the ridge lines or valley lines of a height field. A breadth first traversal of the tree of features is used to order patch placement.

- Patches are extracted from the example height field and placed in the output terrain to match the user sketch as closely as possible. The feature analysis scales down the search space for matching by reducing it to only those patches centred at a feature node. For each feature from the user sketch, a matching patch is found and placed in the output height field. The criteria used during this process are discussed in detail in Chapter 4.

- Once every feature is matched, the rest of the output field is filled by growing from the patches placed in the previous steps and placing patches with no strong features.

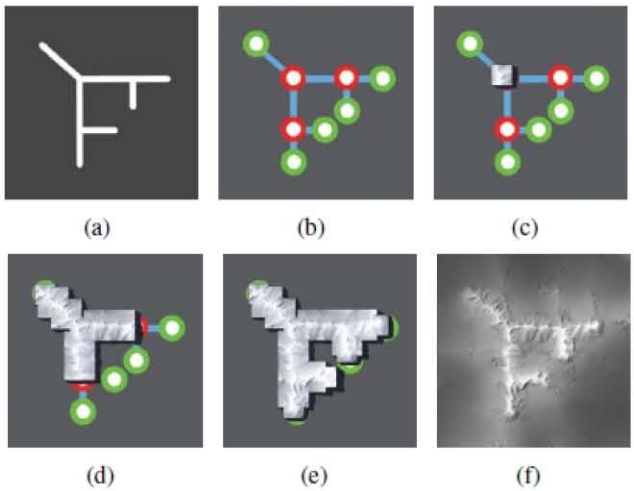


Figure 2.7: Illustration of patch placement order. (a) Sample sketch map. (b) Tree structure returned by PPA analysis. Branch point features and end point features are connected by curvilinear path features. (c) The root patch is placed first. (d) Breadth-first traversal guides the placement of additional patches. (e) Once tree traversal is complete, placement of non-feature patches begin. (f) Final result (Image and caption taken from Zhou et al. (2007))

Each newly placed patch overlaps with the already filled regions. Patch merging is applied to eliminate seams between the new patch and the synthesised regions (see Section 2.2.2 for more details on patch merging). This terrain generation method places less of a workload on the user because only two maps are provided. The user has control over the layout of the terrain, and the realism of the output is strongly related to the sample provided. However, feature extraction and patch-based texture synthesis are computationally expensive ($O(n^2)$). Generating a terrain takes approximately 5 to 6 minutes on an Intel P4 2.0Ghz processor with 2GB RAM. Moreover, the user can only specify the placement of the desired features and cannot specify desired height values for a ridge or valley. The framework proposed in this thesis addresses these issues.

While it is true that procedural methods are designed to reduce the work load of the artist, they often provide poor control over the precise content of the generated result. User control is thus as important as the technique itself. We now discuss current user control models.

2.1.5 User control

Procedural methods are designed to automate the terrain generation process, so that minimal input is required from the artist. However, artists desire some level of control over the appearance of the generated terrain. The mechanism by which this control is achieved is the subject of several user control models. The most commonly used model is *parameter manipulation*.

Parameter manipulation The terrain generation is controlled by the parameters of the algorithm used. For instance, a noise-based algorithm has parameters, such as the amplitudes and frequencies of the noise function, that can be manipulated to change the nature of the generated terrain. Most fractal methods have a roughness constant that specifies the irregularity of the terrain (Fournier et al., 1982). Physics-based method usually implement control by tuning

Figure 2.8: Terrain synthesis from DEMs (Image taken from Zhou et al. (2007))

a set of parameters such as the amount of sediment transferred during erosion (Musgrave et al., 1989). In the GA-based terrain synthesis approach proposed by Chiang et al. (2005), users specify parameters of geometric primitives representative of the desired terrain layout. Artists do not readily comprehend the parameters used in the above algorithms and simulating control by tweaking these parameters is a counter-intuitive approach. The artist does not know what effect a parameter change will cause or what value can be given to a certain parameter to achieve a desired result. An alternative to this are user control methods based on images.

Image-based control The manual design of terrain models is often achieved by painting a 2D height map with existing paint programs and using a system, such as Terragen (2010) to generate a 3D terrain model. Procedural methods also achieve control by using user-painted images. For instance, Schneider et al. (2006) use images to represent the fractal basis functions used to generate a terrain. Users can modify these functions by painting over the images which in turn changes the terrain appearance. Belhadj (2007) proposes a fractal-based technique, built on the Midpoint displacement method, that uses a set of elevation values to constraint the fractal generation. Zhou et al. (2007) employ a user-defined image containing a sketch of the terrain layout to constraint the texture synthesis of a real DEM. The use of a 2D map does not allow intuitive control over specific height values because it can be difficult for some users to convert the desired height values into luminosity values required by the map. A more detailed discussion of the limitations of Zhou et al. (2007)'s algorithm is presented in Section 2.3. People are accustomed to drawing sketches to represent ideas. Thus, sketching is a more intuitive alternative for controlling terrain generation.

User sketching Sketching does not require exact knowledge of what the terrain should be like and is well suited to users who do not want or need to carefully design a terrain. User-sketched strokes are used to roughly specify the desired landscape. The first terrain sketching facility was implemented as part of a sketching system named Harold (Cohen et al., 2000). In Harold, users design and edit hills and mountains by drawing 2D silhouette strokes. The end points of each stroke are used to form a projection plane on which the stroke is projected to form a silhouette curve. The resulting curve forms a shadow and points near the shadow are elevated according to their distance to the silhouette curve. Unfortunately, the depth of a mountain (extent of the mountain perpendicular to the screen) is constant and thus, the terrain looks unnatural from different viewpoints (Watanabe and Igarashi, 2004).

Watanabe and Igarashi (2004) use a similar approach but improve the boundaries of the feature using local extrema of the input stroke. Noise is added after the terrain is deformed, thus the output does not always match the heights specified by the stroke. Gain et al. (2009) present a more complex sketching interface to terrain generation that builds on Cohen et al. (2000). Users interactively control the placement and shape of landforms by drawing $2\frac{1}{2}$D silhouette, shadow, and boundary curves on the ground. These curves act as constraints for a multiresolution surface deformation and noise propagation. Wavelet noise (Cook and DeRose, 2005) is derived from Wavelet analysis of the silhouette stroke and incorporated during the surface deformation. Surface deformation and noise propagation occur at multiple scales, which produces better results than previous sketching techniques. This interface to terrain generation allows users to interactively and intuitively add features such as cliffs, steep slopes and indentations. However, wavelet noise does not model non-isotropic features such as erosion streams. Furthermore, under close inspection, modifications are visible if a real terrain is deformed with this sketching interface. Although this technique gives the user fine and easy control over the final output, the use of noise synthesis produces significantly less realistic terrains.

We have investigated techniques and user control models for terrain generation. We conclude that fractal terrains are unrealistic and physics-based methods are too expensive in both time and resources. Furthermore, neither can be easily controlled. Table 2.1 provides a comparison of these techniques based on their computational cost, the extent of user control and the realism of the synthesised landscape. The table shows that texture-based methods, particularly, patch-based methods, are realistic (see Figure 2.8) and provide better user control. However the computational cost is higher than that of fractal-based approaches . We believe that realism and user control outweigh performance except for real-time terrain generation applications. Additionally, a distributed texture-based algorithm can be used to lessen the computational cost. Therefore, our proposed framework uses a parallel patch-based texture synthesis method. We have also established in this section that user control is an important component of terrain generation and that sketching interfaces allow users to intuitively design terrain models. This project uses the terrain sketching interface proposed by Gain et al. (2009) as it provides an interactive environment that gives the user full control over the large scale terrain features. Our terrain generation framework is based on texture synthesis. The next section presents a range of texture synthesis methods and discusses their suitability for terrain generation.

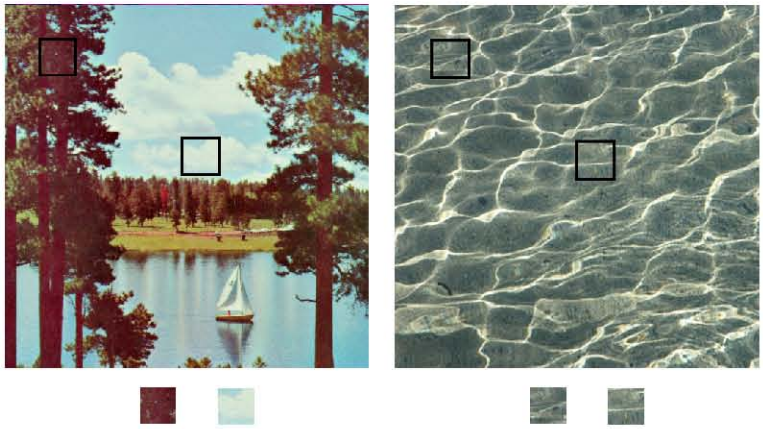
	Speed	User control	Realism	Main limitations
Noise-based	Isotropic terrain and absence of erosion effects			
MidPoint displacement (Fournier et al., 1982)	Less than a second	Low	Low	Crease artifacts
Successive random additions (Saupe, 1988)	Less than a second	Low	Low	Repeating patterns
Summing and-limited noise (Perlin, 2002; Cook and DeRose, 2005)	Less than a second	Low	Low	Homogeneous
Physics-based				
Thermal weathering (Kelley et al., 1988; Musgrave et al., 1989)	Several seconds	Low	Medium	Time-consuming and minimal control
Fluvial erosion (Musgrave et al., 1989; Chiba et al., 1998)	Few minutes	Low	High	Time-consuming and minimal control
Texture-based				
Pixel-based synthesis (Dachsbacher, 2006)	Several minutes	Medium	Medium	Smooths out features
Patch-based synthesis (Zhou et al., 2007)	Few minutes	Medium	High	Limited user control

Table 2.1: Comparison of terrain generation methods. Speed is an estimate of the time taken to generate a 1025×1025 landscape. User control is the level of control that the user has on the terrain generation. Realism measures the quality of the synthesised landscape against characteristics of real landscapes.

2.2 Texture synthesis

A texture is an image with stochastic or structured repeated patterns. Example-based texture synthesis, mostly referred to as texture synthesis, is a technique widely used in generating such textures. Given a sample texture I_{in}, it synthesises a new texture I_{out} that appears to have experienced the same stochastic process as the sample texture. This approach, contrary to hand-drawn pictures or photographic images, allows the automatic creation of a wide range of images, needed for texture mapping in Computer Graphics. Most texture synthesis schemes use the Markov Random Field (MRF) model, which models a texture as the product of a local and stationary stochastic process. An image is *local* if each pixel can be determined by a small set of neighbouring pixels, irrespective of the rest of the image. It is said to be *stationary* if given a small movable window of a proper size, the portion of the image observed by a user through the window always appears similar. Figure 2.9 illustrates the locality and stationary properties of an image. Based

on the MRF assumption that textures are local and stationary images, given the sample texture I_{in} , the output I_{out} is generated so that each output pixel has a neighbourhood similar to the neighbourhood of a pixel in I_{in}. We refer the user to an extensive survey on texture synthesis by Wei et al. (2009) for more insight into the techniques covered in this section.



(a) **(b)**

Figure 2.9: Local and stationary properties. (a) Different regions of the image are dissimilar and a value of a single pixel cannot always be determined by its neighbourhood. The image is neither local nor stationary. (b) Different regions of the image are similar and one can determine the value of each pixel from its neighbourhood. This is a texture, an image with local and stationary characteristics.

2.2.1 Pixel-based texture synthesis

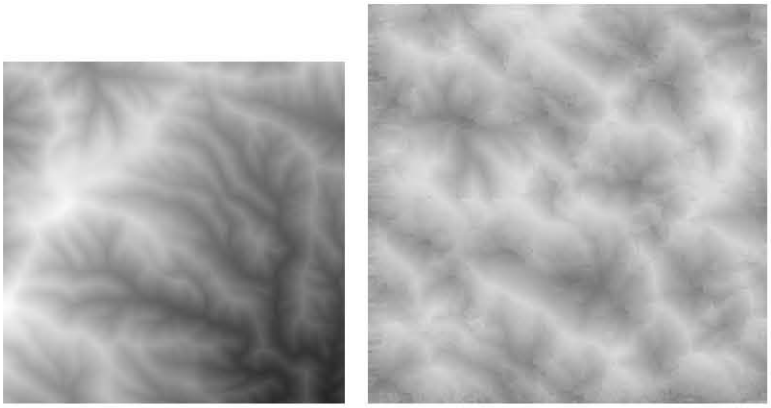
Pixel-based methods generate a new texture pixel by pixel, with each pixel value determined by its local neighbourhood. Efros and Leung (1999) inspired several per-pixel methods (Wei and Levoy, 2000) based on the algorithm presented in Algorithm 2.1. Efros and Leung (1999)'s algorithm first initializes the output texture I_{out} by copying a random pixel from I_{in}. Then it grows I_{out} by spreading out, pixel by pixel, from the seed. The value of a pixel $p \in I_{out}$ is obtained by looping through all the pixels $q \in I_{in}$ and comparing the neighbourhood N_p and N_q of p and q, respectively. The pixel q whose neighbourhood is closest to that of N_p is the best match for p and the value of q is assigned to p. The neighbourhood N_p of a pixel p is the square patch of user-defined size, centred at p. The size should be around the same size as the patterns in the texture. A sum of the squared differences (SSD) is used as a distance function to compare the similarity between neighbourhoods. Only pixels already synthesised in N_p are considered and the SSD is normalised by the number of known pixels. Thus, the neighbourhood size varies during the synthesis according to how far along the synthesis is. The variable change in the neighbourhood size and the use of an inside-out order of synthesis results in a non-uniform distribution of texture patterns. Wei and Levoy (2000) address these issues with a scanline order and a fixed neighbourhood size with an L-shape. The results of the synthesis do not differ significantly from Efros and Leung (1999) but the fixed neighbourhood size facilitates optimisation. However, both algorithms have a tendency to smooth out shapes and thus perform poorly on textures with variable pattern size and irregular shapes, such as textures of flowers and

Algorithm 2.1 Pixel-based texture synthesis. This algorithms takes as input a texture I_{in} and return an image I_{out} of user-defined size by synthesising I_{out} one pixel at a time using a nearest neighbourhood search.

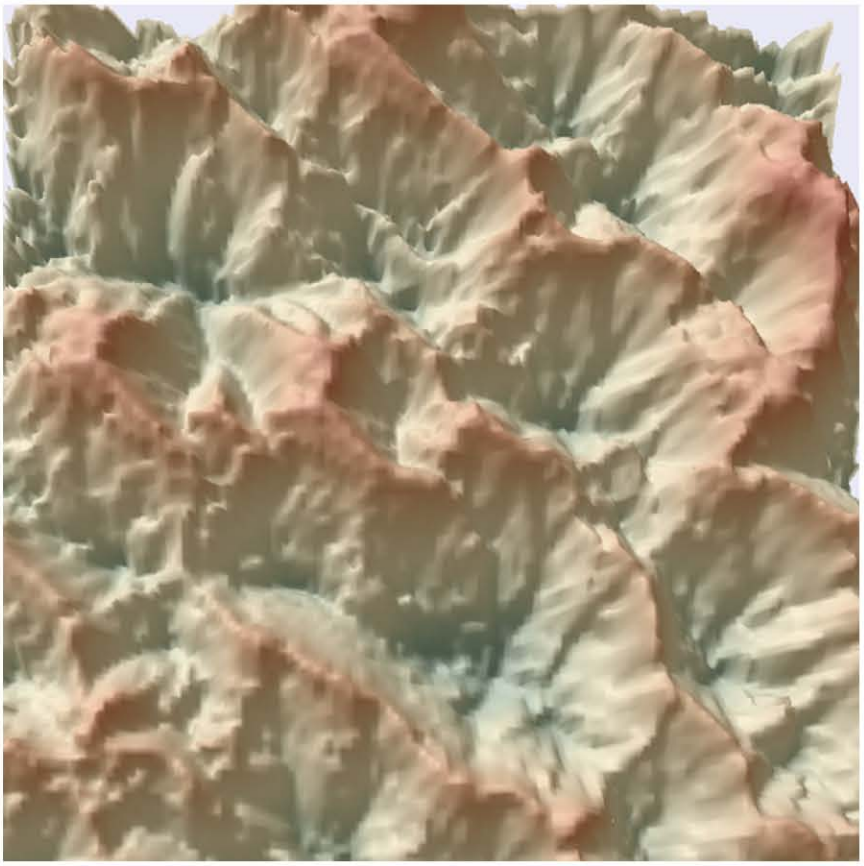
Initialize I_{out}
for all pixels p in I_{out} **do**
  $N_p \Leftarrow buildNeighbourhod(I_{out}, p)$
  $N_{best} \Leftarrow NULL$
  $q_{best} \Leftarrow NULL$
  for all pixels q in I_{in} **do**
    $N_q \Leftarrow buildNeighbourhod(I_{in}, q)$
    if $distance(N_p, N_q) < distance(N_p, N_{best})$ **then**
      $N_{best} \Leftarrow N_q$
      $q_{best} \Leftarrow q$
    end if
  end for
  $I_{out}(p) \Leftarrow I_{in}(q_{best})$
end for

leaves. Ashikhmin (2001) addresses this issue by presenting an algorithm that mainly targets natural textures. It encourages the verbatim copying of patterns by reducing the search space of the matching process to neighbouring pixels. The search space reduction allows interactive rates but produces poor results with arbitrary textures. Another avenue for better pixel-based synthesis performance is by reducing the neighbourhood size and using a multiresolution scheme.

Multiresolution pixel-based texture synthesis generates the output texture at multiple scales, in a coarse-to-fine fashion. A very popular data structure used for multiresolution schemes is the Gaussian stack. A Gaussian stack or pyramid is a multi-scale representation of an image that consists of successively low-pass Gaussian filtered, downsampled versions of the image. Wei and Levoy (2000) build a Gaussian stack G_{in} and G_{out} for I_{in} and I_{out}, respectively. At each level of resolution k, each pixel p in the image $G_{out}(k)$ is determined by a neighbourhood N_p that contains pixels from the current resolution k and lower resolutions. The algorithm iteratively adds details to the output at each level. It particularly enables the capture of large texture patterns from small neighbourhoods. The results are similar to the single resolution synthesis, and the multiresolution method is faster if patterns in the texture are large. Han et al. (2008) argue that the pixel-based algorithms covered so far rely on sample textures with a limited range of scales that consequently restrict the resolution of the output. Features larger than the sample or smaller than a pixel cannot be represented. In light of these limitations, the paper proposes a representation of the sample, named the *exemplar graph*, made of multiple user-defined scaled copies of the sample. A multiscale synthesis of this exemplar graph can create textures with large or infinite ranges of scale. Synthesising the exemplar graph has very high computational cost. Han et al. (2008) and many others (Wei and Levoy, 2000; Lefebvre and Hoppe, 2005) use different optimisation techniques to reduce the cost of pixel-based synthesis.



(a) Sample heightfield **(b)** Output heightfield



(c) 3D rendering of the output

Figure 2.10: Per-pixel texture-based terrain synthesis.

Optimisation To optimize the best neighbourhood search, which requires the most computation time in a pixel-based synthesis, schemes such as kd-tree(Zelinka and Garland, 2002), k-coherence (Han et al., 2008) or Approximate Nearest Neighbour (ANN) searching (Liang et al., 2001) are often used.

Wei and Levoy (2000) represent the neighbourhood N_p of a pixel p with a vector in a multidimensional space. Tree-structured vector quantization, a common data compression algorithm, then takes a set of neighbourhoods and generates a binary-tree-structured codebook. An approximation of the nearest neighbourhood is obtained by traversing the tree. This is faster than an exhaustive search through all the neighbourhoods of I_{in}, but the approximation reduces the quality of the final output.

Zelinka and Garland (2002) propose real-time texture synthesis by using a preliminary slow analysis phase. For each input pixel, a set of matching input pixels is stored during the analysis phase, which is later used in the synthesis. These techniques approximate the candidate set of good matches and may also provide less quality than an exhaustive search.

The rapid evolution of multiprocessors and the many-core multiprocessor GPU has spawned several parallel pixel-based algorithms (Lefebvre and Hoppe, 2005; Han et al., 2008; Garcia and Nielsen, 2009; Zhang et al., 2010). These methods are discussed in further details in Chapter 4. They provide better performance than the aforementioned optimisation techniques, with no loss of quality, and some are even capable of achieving real-time rates (Wei and Levoy, 2002; Lefebvre and Hoppe, 2005).

Although pixel-based methods allow a low-level control over individual pixels that is well suited for controlled synthesis (Hertzmann et al., 2001; Ashikhmin, 2001), they have several drawbacks. Because output pixels are determined in sequence, the output texture can easily lose global structure. Blurring often occurs, especially with structured textures. These drawbacks make pixel-based synthesis an unsuitable choice for terrain generation. Terrains often have well-defined features such as ridges and valleys. Smoothing them will produce highly unrealistic results. Figure 2.10 illustrates this with Wei and Levoy (2000)'s single resolution synthesis that generates a 150×150 height field from a 128×128 sample. Moreover, per-pixels methods are very computationally expensive. Pixels close to each other are likely to find matches that are close as well. Patch-based methods take advantage of this fact by synthesising a new texture a patch at a time instead of one pixel at a time.

2.2.2 Patch-based texture synthesis

Patch-based schemes are well-known for preserving the global structure and repeating patterns in the output texture (Nealen and Alexa, 2003; Wu and Yu, 2004; Lee and Yan, 2005). Instead of pixels, blocks of pixels called patches are copied from the source image to the output . Patch-based algorithms typically follow a pattern similar to Algorithm 2.2. The output texture I_{out} is filled in a scanline order, one patch at a time, minus an overlap region so that blocks of pixels partially overlap each other. To synthesise a patch $\Psi \subset I_{out}$, the algorithm loops over each patch $\Phi \subset I_{in}$ to determine a set of patches that satisfy the overlap constraints within a user-defined error tolerance ϵ. A random patch Φ is selected from the set as the best match for Ψ and Φ is copied into I_{out}, replacing the pixel values of Ψ. The overlap constraints are usually satisfied if the sum of the square differences on the overlap area is less than the error tolerance (Efros and Freeman, 2001; Kwatra et al., 2003). However, a simple comparison of pixel values may not be enough for a texture with large-scale features. Feature similarity is often added as a constraint to solve this limitation (Wu and Yu, 2004; Sinha et al., 2006). It is particularly useful in cases

where the large scale features in the output texture are controlled by the user (Zhou et al., 2007). A problem common to all patch-based methods is the issue of boundary artifacts: when two patches overlap, a seam is created along the boundary of the overlap. Patch merging deals with removing the seam and is tackled differently by the various patch-based algorithms.

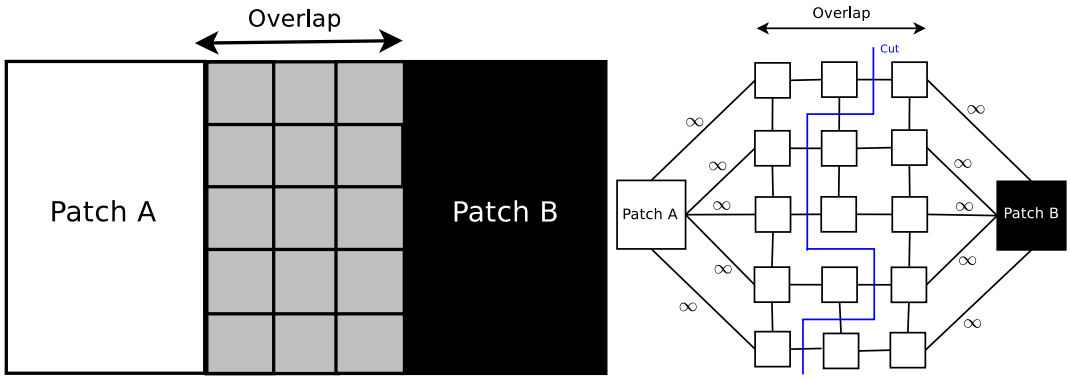
Algorithm 2.2 Patch-based texture synthesis. This algorithms takes as input a texture I_{in} and return an image I_{out} of user-defined size by synthesising I_{out} one patch at a time using a nearest neighbourhood search.

```
Initialize I_out
for all patches Ψ in I_out do
  ζ ⇐ ∅
  for all patches Φ in I_in do
    if distance(Ψ, Φ) < ε then
      ζ ⇐ ζ + Φ
    end if
  end for
  Φ_best ⇐ randompick(ζ)
  Ψ ⇐ Φ_best
end for
```

Efros and Freeman (2001)'s Image Quilting proposes the *minimum error boundary cut* method to make a cut through two overlapping patches that minimises the overlap error. If Ψ_1 and Ψ_2 overlap with regions of overlap Ψ_1^{ov} and Ψ_2^{ov}, respectively, then an error map $e = (\Psi_1^{ov} - \Psi_2^{ov})$ is constructed. Dynamic programming or Dijkstra's algorithm can be used to find the minimal cost path through the error map, from one end of the overlap to the other. Although the synthesised textures have better quality than with pixel-based algorithms, sudden colour changes along the patch boundaries still occur (Liang et al., 2001).Kwatra et al. (2003) extend the minimum error boundary cut to a *graph cut*.

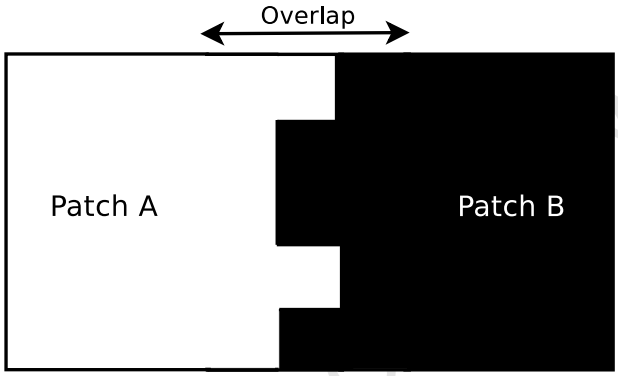
The graph cut algorithm determines the optimal seam in the overlap region. A graph representing the overlap area is constructed. Nodes are pixels p in the overlap area and adjacent nodes are connected by weighted segments. The weight of a segment is determined by comparing $\Psi_1^{ov}(p)$ and $\Psi_2^{ov}(p)$. Nodes on the boundary of the overlap are constrained to one of two patches by an infinitely weighted segment. The graph constructed is illustrated in Figure 2.11. The optimal cost cut of the graph can be determined by the *maxflow* or *minflow* (Sedgewick, 2001) graph algorithm. The cut effectively determines which pixels come from Ψ_1 or Ψ_2. Though the algorithm finds the optimal seam between overlapping patches, the seam may still be visible. Kwatra et al. (2003) alpha-blend a small area around the seam to hide it.

Alpha blending, also called feathering, assigns to each pixel p, a weighted sum of the pixel values $\Psi(p)$ corresponding to each overlapping patch Ψ. The weights vary according to the distance from p to the seam. This approach is very easy to implement and fast if it is not combined with another stitching method. It is adequate for situations where speed is preferred over quality (Liang et al., 2001) because feathering causes blurring artifacts around the overlap areas. Pyramid blending performs alpha-blending at different scales with different alpha masks (Adelson



(a) Patches A and B overlap

(b) A graphcut finds the optimal seam



(c) The cut determines which portion of patch B is discarded

Figure 2.11: Graph cut

et al., 1984). Coarse details are mixed over a wider region around the seam and finer details are mixed over a narrow area. This technique works well at reducing edge duplications that can occur in a simple alpha-blending. False edges appear when two very different areas are alpha-blended. This problem is less noticeable with pyramid-based feathering but does not completely disappear.

Pérez et al. (2003) introduce a set of new tools for seamless editing of images by modifying their gradient. One of the tools proposed is the seamless cloning of a portion of a source image in a target image. The Poisson Equation, a second-order differential equation, is solved over the overlap area, constrained by the 2nd order derivatives of the input texture and the boundary conditions defined by the target image. The pixel values in the overlap area are the unknowns and the solution of the Poisson equation thus provides the pixel values in the overlap area. As a result, a portion of the source image seamlessly appears in the target. Gradient editing has been widely used in seam removal (Levin et al., 2004; Zhou et al., 2007; Aliaga et al., 2008).

Levin et al. (2004) propose two techniques for seamless image stitching in the gradient domain. Their first approach minimises a cost function E which is a dissimilarity measure between the gradient of the stitched image and the gradients G_1 and G_2 of the two input images. E penalises derivatives in the stitched image that are neither in G_1 or G_2, as well as gradient values greater

than those in G_1 and G_2. This eliminates false stitching edges along the seam. Their second approach merges the gradients G_1 and G_2 of the input images with any stitching algorithm such as feathering or graphcut. Then a Poisson equation is solved to find an image with a gradient close to the stitched gradient field. Both methods remove the false edges in previous feathering techniques.

In their texture-based terrain synthesis, Zhou et al. (2007) use the graph cut algorithm to find the optimal seam in the overlap of terrain patches and then eliminate remaining visible seams by setting the gradient of pixels along the seam to zero. This is similar to the second technique proposed by Levin et al. (2004). A Poisson equation is solved using the modified gradient field as a guide. The result is a set of height values that smooths the transition from one patch to another. There is a limitation to the merging methods proposed by both Zhou et al. (2007) and Levin et al. (2004): although the result is artifact-free when viewed in 2D, the lighting in a 3D rendering environment highlights discontinuities in the gradient field. In Section 3.5, we propose a novel patch merging technique that addresses this issue.

Optimisation Patch matching is very expensive since, at each step, all patches from the input texture are examined to build a candidate set. Liang et al. (2001) present a patch-based sampling approach similar to image quilting. They achieve real-time speed by preprocessing the input texture. An optimised kd-tree, a quadtree pyramid, and principal component analysis (PCA) are used for feature vector dimension reduction, which speeds up the algorithm.

A patch map, introduced by Changzhen et al. (2007), is a set of feature-preserved irregular patches created during an analysis phase, before the synthesis. For every patch in the input texture, the patch map has a set of good matches obtained using graph cuts, feature deformation and matching. The analysis phase significantly speeds-up the synthesis phase, while preserving texture features.

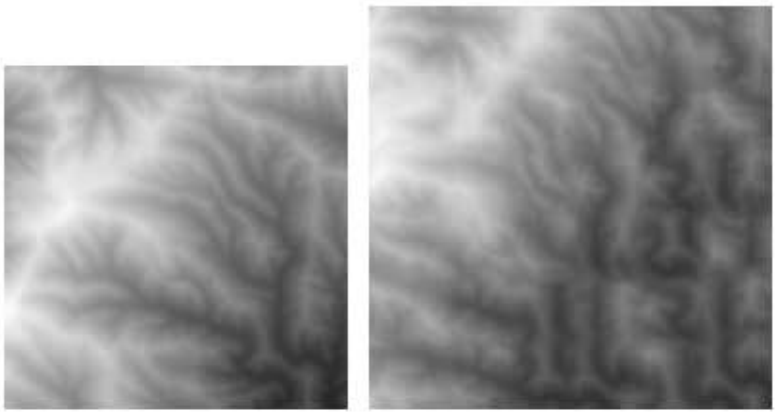
During a patch matching iteration, a patch from the output texture is matched against multiple patches from the input texture. This process is embarrassingly parallel and therefore can be easily optimised on multiprocessors and GPUs (Zou et al., 2010; Woodard, 2005; Sinha et al., 2006). Further details on patch-based parallel algorithms are provided in Chapter 4.

Patch-based algorithms are faster and better at preserving structures than pixel-based methods. However, the boundary artifact problem is a significant limitation and it is not trivial to both overcome this limitation and preserve the quality of the synthesised texture. Some prior work has focused on solving this issue by combining the two texture synthesis strategies.

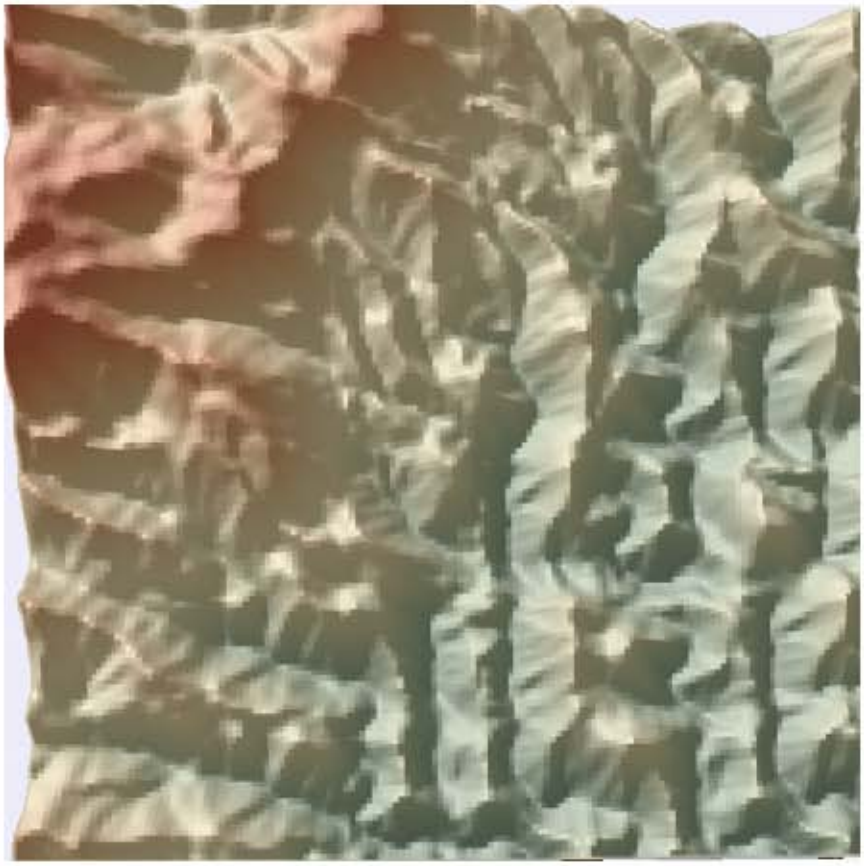
2.2.3 Hybrid methods

Nealen and Alexa (2003) propose a patch-based synthesis algorithm that uses a pixel-based re-synthesis on the overlap regions to remove seams. This technique benefits from the advantages of both pixel-based and patch-based synthesis. The overlap area repair produces fewer artifacts than a simple minimum cost cut (Efros and Freeman, 2001) or alpha blending (Liang et al., 2001). However, it inherits the high computational cost of pixel-based algorithms and it is not guaranteed that valid pixels can be found that will remove the boundary artifacts.

On the other hand, Kwatra et al. (2005) present a texture optimisation algorithm half-way be-



(a) Sample heightfield **(b)** Output heightfield



(c) 3D rendering of the output

Figure 2.12: Per-patch texture-based terrain synthesis

tween pixel-based and patch-based synthesis. The output texture is generated in pixel units but instead of computing their values one by one, all the pixel values are computed at once by merging the local neighbourhood similarity measures into one global quadratic cost function E and minimising E. The energy of a neighbourhood in I_{out} is the distance to its closest neighbourhood in I_{in} and the cost function E is the sum of all those energies. The global nature of the

cost function as well as the use of large neighbourhood sizes has patch-based characteristics. E is optimised by an iterative algorithm that progressively improves the texture after each iteration. Visually, this compares well with pixel-based (Efros and Leung, 1999; Wei and Levoy, 2000) and patch-based (Efros and Freeman, 2001; Kwatra et al., 2003) algorithms. However, the optimisation process can slip into a local minima and stay there, because distant neighbourhoods can only communicate through overlapping neighbourhoods (Kwatra et al., 2005). This limitation blurs some areas of the texture or results in misaligned texture patterns.

To determine the applicability of the surveyed texture synthesis methods to terrain synthesis, we prioritise low computational cost and quality. The authors of the aforementioned texture synthesis techniques evaluate success by visually comparing the generated textures against competing methods. They do not provide any quantitative assessment, and the visual inspection of their results only has small significance, because only a small subset of these is available. Given these considerations, Table 2.2 attempts to compare the various methods discussed, above according to computational cost and the quality of their output. Patch-based synthesis gives the best trade-off between performance and quality and thus a patch-based scheme is suitable for our terrain synthesis. However, heightmaps have different properties than textures. the Markov Random Field (MRF) model discussed at the beginning of this section models textures as local and stationary entities. Terrains do not have these properties. They are characterised by curvilinear features such as ridges and valleys. Figure 2.12 illustrates this fact with the patch-based synthesis of a real landscape. The sample has well-defined ridges and valleys but, the synthesised heightmap has some repeating patterns on its right. Although the result is still visually more representative of the sample than the pixel-based synthesis in Figure 2.10, it can be significantly improved by incorporating more constraints than just height difference on overlapping areas. Other criteria such as feature similarity and the level of detail are attractive alternatives.

2.3 Terrain synthesis from Digital Elevation Models

Zhou et al. (2007)'s algorithm is the state of the art method in using texture synthesis to generate more realistic landscapes. In this section, we briefly discuss the technique, its advantages and limitations.

Given a user-defined sketch map T_{tar} and a real heightmap T_{exm}, a patch-based texture synthesis is applied on the heightmap to generate a new terrain T_{out} that matches the features on the sketch map. We now present a more detailed overview of this algorithm.

Feature extraction
Users control the terrain synthesis by drawing a map representing the large scale features they desire. Features are automatically extracted from the sketch map and the real heightmap to reduce the work load of the user. This is done using the Profile and Polygon Algorithm (PPA) of Chang et al. (1998). To extract the ridge lines of a terrain, PPA proceeds as follows:

1. Profile recognition: All points that could be part of a ridge line are marked as candidates. To recognize if a point p is a candidate, PPA takes p as the center of a profile of user-defined length and if there is at least one point lower than p on both sides of the profile, p is marked as a candidate. The profile along the North-South, North East-South West,

East-West, and North West-South East directions are examined to check the candidacy of p. Figure 2.13(a) shows a set of points marked as candidates after profile recognition.

2. Target connection: Adjacent candidate points are connected by weighted segments or edges as shown in Figure 2.13(b). The weight of a segment is obtained by summing the height of the two points it connects. We say that a segment A is lower (or higher) than segment B, if A has a smaller (or larger) weight, respectively. If two segments cross each other, the lower segment is discarded.

3. Polygon breaking: The result of target connection is a set of connected segment groups, from which ridge lines must be extracted. This step simplifies the segment groups by repeatedly removing the segment with the lowest weight in each closed polygon until there are no more polygons (see Figure 2.13(c)). To achieve this, edges are sorted based on their weights. Then PPA starts with the lowest segment and checks if it is the side of a polygon. If it is, the segment is deleted, else the next lowest one is checked. This is repeated until all the segments are processed. This step produces a tree structure whose edges lie along the terrain ridges.

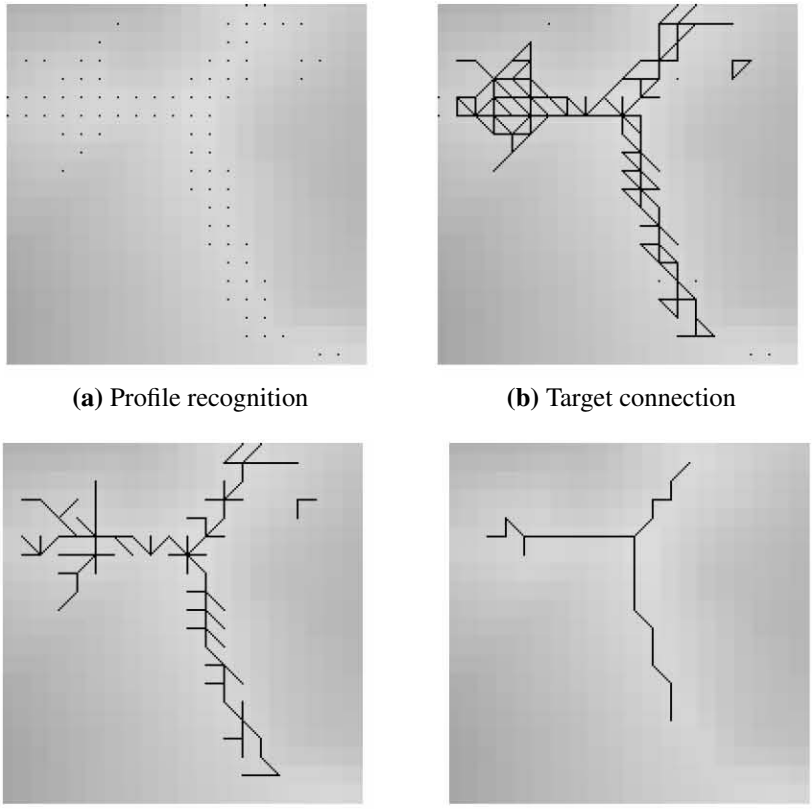
4. Branch reduction: Short branches are eliminated by repeatedly deleting their end points, a user-defined number of times. As shown in Figure 2.13(d), this process preserves the principal ridge lines.

5. Line smoothing: Finally, each vertex is moved to the average of its position and the positions of all its neighbours. The result is a tree structure that represents the terrain ridges.

Valley lines can easily be extracted by inverting the heightmap prior to the PPA algorithm. This technique attempts to extract the same features that a human operator would extract in a line-drawing process. However, Polygon Breaking is computationally expensive because for each segment s, the algorithm searches for a path that starts and ends at s. In the worst case scenario, the performance is $O(|E|^2)$, where $|E|$ is the number of segments. A large number of segments slows down the feature extraction considerably and reduces the performance of the terrain synthesis.

Patch-based texture synthesis Once feature extraction is performed on the sketch map and the sample terrain to obtain two separate tree of features, Zhou et al. (2007) perform *feature patch matching*. For each patch in the sketch map containing a feature, the best match in the sample terrain is copied and placed in the output. The feature patch matching differentiates between three types of features: branch points, end points, and path features. Figure 2.14 provides an example of each feature type. The order of the synthesis is guided by a breadth-first traversal of the sketch map's tree of features. To compare a target patch Ψ^p_{tar} centred at feature p to a candidate patch Ψ^q_{exm} centred at q having the same feature classification as p, the following criteria are used:

- If p and q are branch points, then their degree (or valence) must be equal and the angles of the outgoing paths must be similar. Furthermore, Ψ^q_{exm} is deformed to fit Ψ^p_{tar}. Sets of



(a) Profile recognition **(b)** Target connection

(c) Polygon Breaking **(d)** Branch reduction

Figure 2.13: Different steps of ridges extraction with the PPA algorithm

control points $\{p_i\}$ and $\{q_i\}$ are identified for p and q, respectively. The control points of a feature consist of the position of the feature itself and the points where outgoing paths intersect a circle inscribed in the patch. Figure 2.14 shows the control points for different types of features. Patch Ψ^q_{exm} is deformed by applying a continuous coordinate transformation that maps the set of points $\{q_i\}$ to $\{p_i\}$. The thin-plate spline (TPS) interpolation is used to obtain such a transformation with a minimal amount of distortion (Bookstein, 1989).

- If p and q are path features, the patch Ψ^q_{exm} is warped to fit Ψ^p_{tar} as with branch points.

The k candidate patches with the lowest deformation energy from the TPS are selected for further matching. The cost of selecting a candidate Ψ^q_{exm} is obtained by combining the following costs:

- c_d: *Deformation energy* from the TPS warping of Ψ^q_{exm} , if q is not an end point.
- c_g: *Graphcut score* to determine how well Ψ^q_{exm} matches the already filled pixels in the output T_{out}. Graph cut is an algorithm used to find the optimal seam in the overlapping region. The graph cut score indicates how noticeable the seam is, as will be discussed later in this section.

- c_f: *Feature dissimilarity cost* to measure the similarity of features along joining paths. For every path, height profiles orthogonal to the path are stored at uniformly sampled points on the path. When the path is joined to another path at a point P, the height profile at P is obtained by a linear interpolation of its two nearest neighbours. If the feature path p is joining another path in T_{tar} at point P and q is joining another path in T_{out} at point Q , the feature dissimilarity cost is the sum of squared differences of the height profiles at P and Q.

- $c_i's$: User-defined constraint(s) such as height constraints.

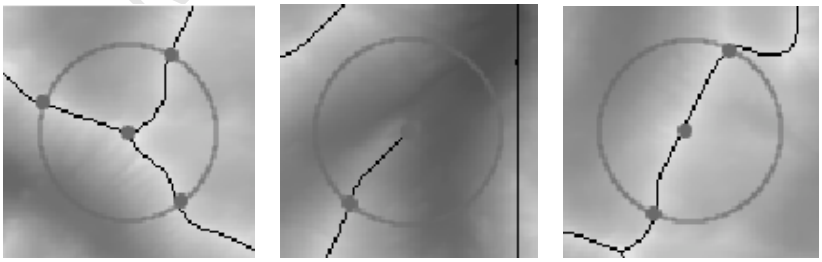
Thus, the cost of selecting a candidate patch Ψ^q_{exm} is a weighted sum of c_d, c_g, c_f and the $c_i's$. All the candidates patches are sorted according to their cost and the patch with the lowest cost is selected as the best patch and placed in the output T_{out}. This approach to feature patch matching is limited by several factors:

- The noise in both Ψ^p_{tar} and Ψ^q_{exm} is ignored. The addition of criteria that takes into account the similarity of noise variance in both patches will enhance the matching process.

- Zhou et al. (2007) argue that the TPS warping has degenerate cases, for instance in cases where the control points are collinear. A failed patch deformation can introduce artifacts in the output.

- Several patches are stopped from further matching because the cost of warping to fit the user sketch is too high. However, these patches could have been potential candidates if they were rotated.

- The feature dissimilarity cost only considers the height profile perpendicular to the path and not the height values along the path itself. Comparing the heights along the outgoing paths of the feature will provide a more constrained feature matching.



(a) Branch point **(b)** End point **(c)** Path feature

Figure 2.14: Examples of different types of features . The gray dots indicates the feature control points.

After the feature matching process, regions where the user did not specify any features are empty. Zhou et al. (2007) fill the rest of the output T_{out} during a *non-feature patch matching* phase. Patch positions are selected in coarsely-spaced increments in a descending order, from the already-filled areas. Patches are copied from the sample and pasted at these patch positions.

This selection is achieved by sorting candidate patches without strong features in an increasing order, based on the sum of the squared difference (SSD) of pixels in the overlap with the already placed patches. The first k candidates are chosen for further matching. The best candidate is selected according to a combination of the SSD score and the graphcut score, and placed in the output. However, this process the following issues:

- The order of the non feature patch matching does not take into account the edge information in the already synthesised regions. Newly placed patches without strong features can overwrite the features placed during feature patch matching. An order of synthesis that prioritizes patch positions with the most edge information would preserve the large scale features already placed.
- Checking each patch for an absence of strong features can easily be avoided by using a matching criteria that compares noise in patches from the sketch map and the sample.

Patch merging

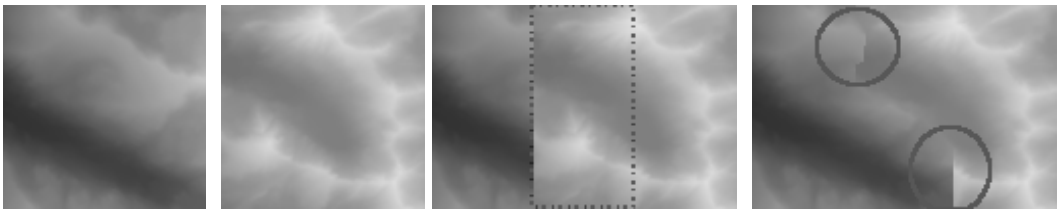
Every time a new patch is placed, it overlaps with previously filled pixels, creating boundary artifacts. Patch merging deals with merging patches seamlessly. Zhou et al. (2007) do this by combining two techniques: the graph cut algorithm (Kwatra et al., 2003) and Poisson seam removal (Pérez et al., 2003).

We revisit the graph cut algorithm discussed earlier in Subsection 2.2.2. The graph cut algorithm merges two patches Ψ_1 and Ψ_2 as follows:

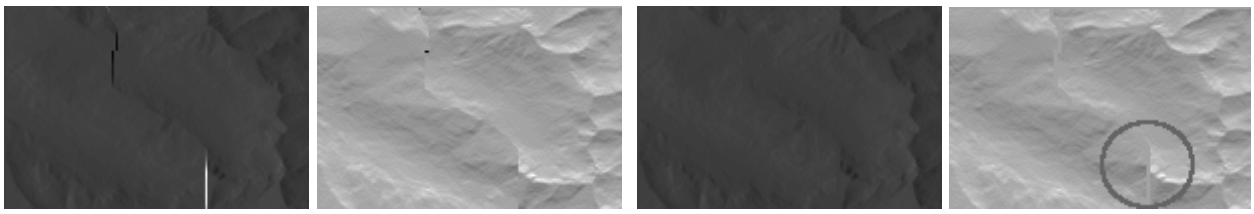
- A graph representing the overlap area ov is constructed: the nodes are pixels in the overlap region and adjacent pixels are connected by weighted edges. If p and q are adjacent pixels, then the weight of the edge connecting them is the sum of height differences in the two patches:
$$M(p, q, \Psi_1, \Psi_2) = |\Psi_1(p) - \Psi_2(p)| + |\Psi_1(q) - \Psi_2(q)|$$
Pixels on the overlap boundary are constrained to come from Ψ_1 or Ψ_2 by adding infinitely weighted edges to these patches.
- The optimal cut of the graph is obtained using a maxflow or mincut algorithm(Sedgewick, 2001). The cut determines the source Ψ_1 of pixels in the merged result Ψ. The score of the graphcut is
$$c_g = \sum_{<p,q> \, on \, the \, cut} M(p, q, \Psi_1, \Psi_2)$$

Figure 2.15(d) shows the result of a graph cut. Notice that the optimal seam is still visible. Poisson seam removal is used to remove the seam.

Poisson seam removal modifies the gradient field of the stitched output Ψ. The gradient field G_I of an image I is composed of two images G_I^x and G_I^y that indicate how the values I change in the horizontal and vertical directions. The gradient (g_i, g_j) of each pixel $(i, j) \in I$ is computed and $(G_I^x, G_I^y)(i, j) = (g_i, g_j)$. Figure 2.15(e) illustrates the horizontal and vertical components



(a) Ψ_1 **(b)** Ψ_2 **(c)** Before graph cut **(d)** After graph cut



(e) Gradient field **(f)** Modified gradient field

(g) Final result

Figure 2.15: Patch merging using Graph cut and Poisson seam removal. (d) The optimal seam computed by graph cut is still visible. (d) The horizontal and vertical components of the gradient field. (e) In both components of the gradient field, values along the seam are set to zero. Note the visible discontinuity in the modified gradient field. (g) A Poisson solver is used to find a set of height values that fits the modified gradient field.

of a gradient field. Zhou et al. (2007) modifies the gradient field of Ψ by setting the gradient of all the pixels along the cut to zero. Then the overlap area is reconstructed by solving a Poisson equation on ov constrained by the modified gradient field $G^{'}$ and the boundary of the overlap ∂ov. The Poisson equation is formulated as follows:

$$\nabla^2 \Psi^{'} = \nabla \cdot G^{'}, \qquad \Psi^{'} \mid_{\partial ov} = \Psi \mid_{\partial ov}$$

where $\Psi^{'}$ is the new set of elevations in the overlap area, ∇^2 is the laplacian operator and $\nabla \cdot G^{'}$ is the divergence field of $G^{'}$. Figure 2.15(f) illustrates the seam removal after solving the Poisson equation for a new set of height values. This patch merging technique has the following limitations:

- The weighting function M used during the graph construction in the graph cut does not consider the fact that discontinuities in low frequency areas stand out more than those in high frequency regions. The function can be modified so that seams passing through low frequency areas are more heavily penalized (Kwatra et al., 2003).

- Artificially setting some values of the gradient field to zero effectively creates second derivatives discontinuities that are visible in a 3D terrain fly-over as shown in Figure 2.16. This can be addressed by removing these discontinuities before solving the Poisson equation.

We have identified several points that can be improved in Zhou et al. (2007)'s algorithm to enhance the patch matching and reduce the number of artifacts in the generated terrain. This project addresses these issues.

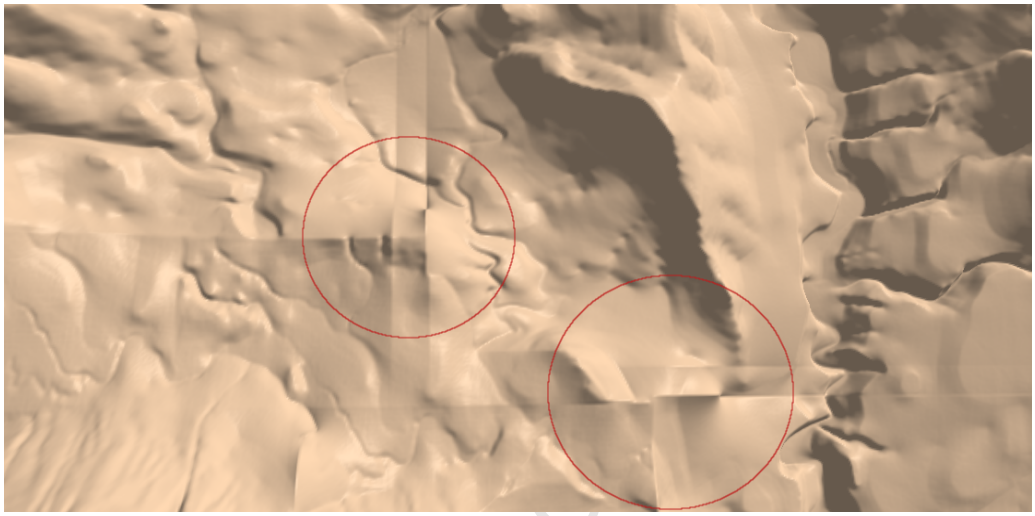


Figure 2.16: Boundary artifacts in a terrain generated with Zhou et al. (2007)'s algorithm

2.4 Summary

In this section, we surveyed techniques in terrain synthesis and texture synthesis. We discussed the issues with fractal-based and physics-based terrain generation and outlined a recent texture-based scheme (Zhou et al., 2007). This scheme generates new terrains by copying height values from existing terrains and offers better realism compared to fractal-based generation, better performance compared to physics-based schemes and a higher level of control.

Sketch-based environments that accept user input in the form of strokes offer a more intuitive user control alternative to parameter manipulation and image-based input systems. The artist can sketch an outline of the final terrain without the need to tweak parameters or paint over images. Gain et al. (2009)'s terrain sketching interface enables users to interactively create and edit landscapes by drawing $2\frac{1}{2}D$ curves. The terrain is deformed to perfectly fit the sketched curves using multiresolution surface deformation and noise propagation. This intuitive environment is however limited since the terrains generated are not that realistic, due to the underlying noise-based synthesis. We propose to replace the noise synthesis scheme with a texture-based terrain synthesis method that uses real landscape data to produce more realistic results.

Texture-based methods for terrain synthesis are derived from the texture synthesis literature which we discussed in Section 2.2. However, real landscapes do not generally have the local and stationary characteristics of textures. The original pixel-based and patch-based algorithms

rely on these characteristics, and applying them directly to landscapes will generate unrealistic terrains with repeating patterns. They need to be adapted to become a suitable option for terrain synthesis. Zhou et al. (2007) do this particularly well with a feature-guided patch-based texture synthesis algorithm. However, user control is specified by a user-defined sketch map which two-dimensional, and thus the artist can only specify the position and shape of the desired features with no control over their heights. We extend Zhou et al. (2007)'s work by changing from an image-based input system to the terrain sketching interface of Gain et al. (2009). Furthermore, we modify the time-consuming feature extraction PPA algorithm of Chang et al. (1998) to a more recent, robust, and faster technique based on minimum spanning trees (Bangay et al., 2010). Finally, we design a novel patch merging technique, more suitable for terrains, that overcomes the limitations of the patch merging used in Zhou et al. (2007). The next chapter discusses the design of our proposed framework.

	Cost	Quality	Limitations
Pixel-based			
Non-parametric sampling (Efros and Leung, 1999)	Several minutes	Medium	Slow and non-uniform pattern distribution
Multiresolution synthesis (Wei and Levoy, 2000)	Few minutes	Medium	Slow and does not preserve feature information
Synthesising natural textures (Ashikhmin, 2001)	Few seconds	Medium	Limited to textures with repeating patterns of variable size
Patch-based			
Graph cut-based (Kwatra et al., 2003)	Several seconds	High	Visible optimal seams
Feature-guided (Wu and Yu, 2004)	Several seconds	High	More constraints are needed to make it applicable to terrains
Hybrid			
Per-pixel resynthesis (Nealen and Alexa, 2003)	Few minutes	Medium	Artifacts in regions where no valid pixels were found
Texture optimization (Kwatra et al., 2005)	Few minutes	Medium	Blurring artifacts and misalignment of texture elements

Table 2.2: Comparison of texture synthesis techniques. Speed estimates the time taken to generate a 192×192 texture, given a 64×64 input. Quality measures the ability of the texture synthesis method to generate a texture that preserves the large features in the input, and seems to have through the same stochastic process as the input.

Chapter 3

Patch-based Synthesis Framework

This chapter presents the design of a framework that produces realistic terrains with large-scale features specified by user-sketched curves. The patch-based synthesis framework improves previous terrain generation algorithms (Zhou et al., 2007; Gain et al., 2009) by using $2\frac{1}{2}$D curves drawn by the user to generate a terrain. The terrain matches the large scale features and height constraints extracted from the user sketches, and yet contains the small scale characteristics of a real landscape. The terrain sketching interface (Gain et al., 2009) supports user interaction, and deforms a flat terrain to fit the user sketches. Then, a patch-based texture synthesis that takes the deformed heightmap and an example heightmap as input is executed. The output is a new heightmap built from patches in the example that matches the broad features extracted from the deformed heightfield. Finally, the new heightmap is deformed so that it perfectly satisfies the height constraints specified by the sketched curves. The quality of the generated terrain is strongly related to the realism of the example heightfield, and thus we use real landscapes downloaded from the US Geological Survey (2011). The patch-based texture synthesis is based on the algorithm of Zhou et al. (2007), with several modifications.

3.1 Enhanced framework for terrain synthesis

Zhou et al. (2007) produced impressive terrains with a patch-based texture synthesis technique. The user is, however, limited by the 2D nature of the sketch map. We propose using a terrain sketching interface (Gain et al., 2009) as the basis for our new hybrid scheme. Users will benefit from an interactive environment that allows them to intuitively control the output by drawing $2\frac{1}{2}$D constraint curves. An artist can also utilize other options of the interface, such as 3D multiresolution rendering and editing. In addition, the terrain generation of the sketching interface is upgraded from a noise-based scheme to a texture-based method that produces more realistic landscapes. Furthermore, we propose the following solutions to the texture synthesis limitations of Zhou et al. (2007):

- The performance issues of the PPA feature extraction are addressed by using minimum spanning trees in a fashion similar to Bangay et al. (2010)'s ridge and valley characterization;
- We improve the feature matching process of Zhou et al. (2007) by including a noise variance similarity cost and the sum of the squared difference on the overlapping region with

already placed patches;

- The thin-plate spline deformation is removed from the matching process. Instead, we mirror and rotate patches in the exemplar heightfield T_{exm} and consider them as new candidates. The set of potential candidates increases as well as the probability of finding good matches from the exemplar;
- The feature dissimilarity cost is modified to include the difference in heights along the outgoing branches of a feature. This criterion is not restricted to path features but also applies to end points and branch points;
- We change the filling order in non-feature matching to a best-first filling approach based on gradients, as used in Criminisi et al. (2004). Noise variance is also used as a criterion in this phase;
- Finally, we introduce a novel patch merging scheme more suitable for terrains. Instead of setting the gradient values along the optimal seam to zero, the discontinuities in the gradient field are removed with a scattered point interpolation method and a Poisson equation is solved for a new set of height values. This technique produces more realistic results than Zhou et al. (2007), especially when the results are viewed in 3D.

The rest of this chapter covers the design of our framework in more detail, from feature extraction to patch merging. An overview of our system is shown in Figure 3.1.

Variable	Meaning
T_{tar}	Target or deformed terrain
T_{exm}	Exemplar heightfield
T_{out}	Output heightfield
Ψ_X	An arbitrary patch in a terrain T_X
$\Psi_X^{(i,j)}$	A patch centred at (i, j) in a terrain T_X
Ψ_X^p	A patch centred at p in a terrain T_X
Ω	The set of empty pixels in the output
Γ_X	A tree of feature points in a terrain X
G_X	The gradient field of a patch or terrain X
$\{p_i\}$	The set of control points of a feature p
$ov_{A,B}$	The set of pixels in the overlap of A and B
∂X	The boundary of X

Table 3.1: List of symbols

3.2 Algorithm overview

This section describes our patch-based terrain synthesis. For a list of variables and their meaning, refer to Table 3.1. Initially, the terrain sketching interface presents a flat terrain upon which

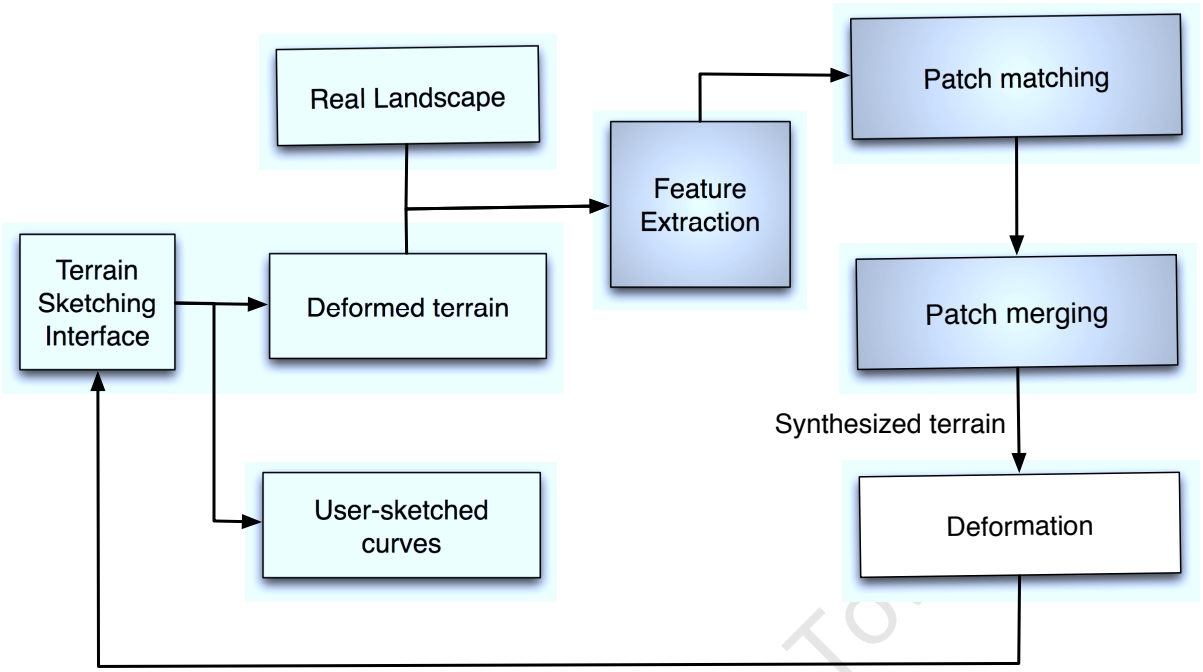


Figure 3.1: Overview of the patch-based terrain synthesis framework: gray components are the terrain synthesis steps described in this chapter. Feature extraction finds features in the target (or deformed) terrain and the exemplar. Patch matching finds patches in the exemplar that fit the target. Patch placing places a found patch in the output such that no seams are visible.

the user can draw curved strokes. Two curves are needed: one for the silhouette of the terrain feature and another for its boundary. Once the user validates these curves, the terrain is deformed to fit the constraints. An example heightmap, or *exemplar*, and the deformed terrain are then passed into a patch-based texture synthesis that will generate a new terrain that has the large scale features of the deformed terrain and yet exhibits the characteristics of the example heightfield. The new heightfield replaces the deformed terrain in the sketching interface. This new terrain no longer perfectly fits the sketched curves, and so it is deformed in a final phase to match the user height constraints. The user can edit the synthesised terrain by drawing another set of curves and the whole process restarts. No randomness is added to the texture synthesis, so that the same input data will always generate the same terrain. The interface, combined with the texture synthesis framework, provides the capability to navigate, examine, and edit the terrain as often as necessary.

To ensure that the terrain generated by the texture synthesis matches the target or deformed terrain, as closely as possible, several steps are needed. First, curvilinear features such as ridges and valleys are extracted from the target and the example heightmaps (Section 3.3). These features are used to constrain the feature matching (Subsection 3.4.1) so that the output, constructed from patches in the exemplar, has the same large scale features as the target. Non-feature matching then fills the empty areas in the output, where the user did not provide any information (Subsection 3.4.2). During both patch matching steps, the placement of a patch from the exemplar T_{exm} into the output heightfield T_{out} will be performed by patch merging (Section 3.5), to remove any

seams that occur when one patch overlaps previously placed patches. The pseudo-code in Algorithm3.1 presents these steps in a more concise format. Each function in the pseudocode will be further expanded in the next sections. The complete process, from user sketching to the final terrain is illustrated in Figure 3.2. The resulting terrain has the features of a real landscape and still satisfies the constraints sketched by the artist.

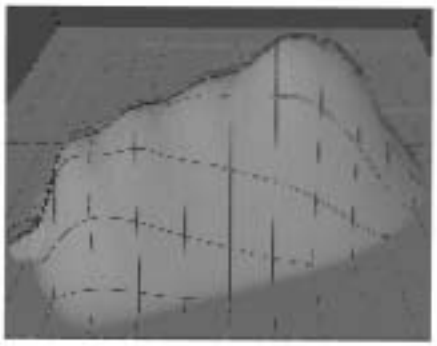
The rest of this chapter is dedicated to the design details of the different steps of the patch-based texture synthesis framework.

Algorithm 3.1 Terrain synthesis

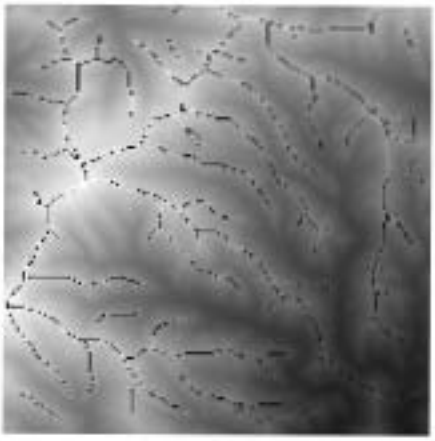
User sketches silhouette and boundary curves on the flat terrain T
T is deformed to fit the sketched curves
$T_{tar} \Leftarrow T$
$\Gamma_{tar} \Leftarrow FeatureExtraction(T_{tar})$
$\Gamma_{exm} \Leftarrow FeatureExtraction(T_{exm})$
Initialize T_{out}
$T_{out} \Leftarrow FeaturePatchMatching(\Gamma_{tar}, \Gamma_{exm}, T_{tar}, T_{exm}, T_{out})$
$T_{out} \Leftarrow NonFeaturePatchMatching(T_{tar}, T_{exm}, T_{out})$
$T \Leftarrow T_{out}$
T is deformed to fit the sketched curves

3.3 Feature extraction

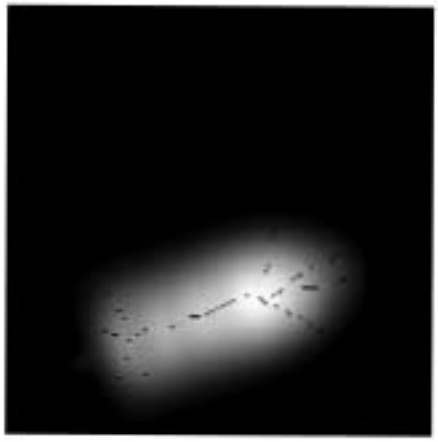
This step automatically extracts features such as ridges and valleys from the target and the exemplar. In Image Processing, it is common practice to use edge detection methods for feature extraction. These methods are based on locally-maximal derivatives, which result in spurious features (Zhou et al., 2007; Zhang et al., 2008). A method based on local extrema in the heightmap is needed to extract skeleton lines that characterize ridges or valleys. The Profile Recognition and Polygon Breaking Algorithm (PPA), by Chang and Sinha (2007), is designed for this purpose. The five basic steps of the PPA algorithm were described in Subsection 2.3 and consists of profile recognition, target connection, polygon breaking, branch reduction and line smoothing. Polygon breaking, which involves repeatedly deleting segments created during target connection until all the cycles are removed, has a significant computational cost. Although our framework can save features extracted from a heightmap to external files and load them back in the system when needed, the target terrain constantly changes according to the user's intentions. Thus, feature extraction of the target terrain is always executed at runtime. A slow process such as polygon breaking is not adequate for an interactive system. Furthermore, the performance drastically decreases as the size and the feature complexity of the terrain increases.



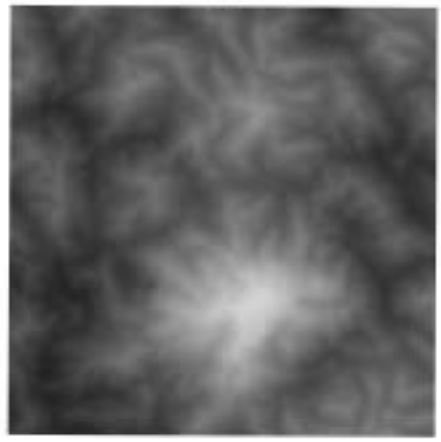
(a) A flat terrain is deformed to fit sketches.



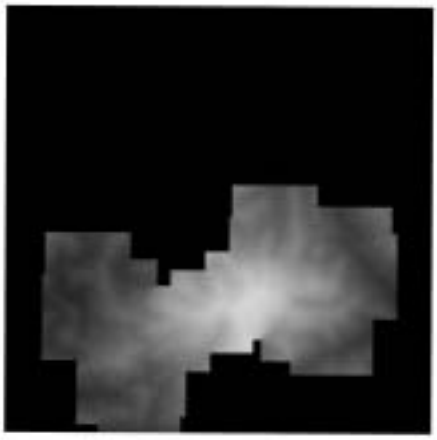
(b) Features are extracted from an example terrain.



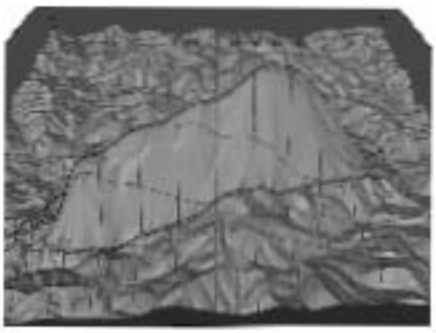
(c) Features are extracted from the deformed terrain.



(e) The rest of the output is filled with patches with no strong features.



(d) Patches that match features of the deformed terrain are copied from the exemplar.



(f) The synthesized terrain is deformed in the sketching interface, so that it fits the sketched curves.

Figure 3.2: Patch-based synthesis framework

Bangay et al. (2010) propose a modification of the PPA algorithm that connects all the terrain points into a graph using a height-based or curvature-based weighting and computes the *minimum spanning tree* (MST) of the graph. The minimum spanning tree of a graph is a subset of edges that connect all the nodes and minimize the total weight of the edges. It is commonly created using greedy MST algorithms such as Prim (1957) and Kruskal (1956) algorithms. Similarly to Chang et al. (1998); Chang and Sinha (2007), the feature extraction ends with branch reduction and line smoothing of the tree. Results show that while the original PPA's cost is polynomial, the new version runs in quasilinear time with respect to the number of edges in the graph. However, all the terrain points are characterized, even points on flat regions. Even though a point is in a valley, it will become part of a ridge line and vice-versa. Because we are mainly concerned with performance and the extraction of large-scale terrain features, we simply replace the polygon breaking in the PPA algorithm, with a *minimum spanning forest* algorithm, preserving the profile recognition and target connection steps.

The graph constructed by connecting candidates points needs not be strongly connected. The minimum spanning forest is the union of minimum spanning trees of the connected components of the graph. We choose to use Kruskal's algorithm, which extracts minimum spanning trees as well as forests, and has a $O(|E|log|E|)$ time complexity, where $|E|$ is the number of edges in the graph. Kruskal algorithm performs several orders of magnitude better than Polygon breaking.

Algorithm 3.2 FeatureExtraction(T)

$E \Leftarrow \emptyset$
for all adjacent points p and q in T **do**
  $E \Leftarrow E + edge(p, q, (T(p) + T(q))/2.0)$
end for
if two edges cross **then**
  Delete the less significant edge
end if
for all edges e E **do**
  Remove parallel edges on either of e if there are less significant than e
end for
if ridge lines are extracted **then**
  Negate the weights of all the edges in E
end if
Extract the minimum spanning tree Γ of E
for all nodes p in Γ **do**
  if p does not have lower (upper for valley lines) height values on both sides **then**
    Remove p from Γ
  end if
end for
Remove short branches
Smooth the branches in Γ
Remove small isolated trees
return Γ

3.4 Patch matching

Once features have been extracted, patch matching selects square patches from a list of candidate patches in T_{exm} that match the user constraints. The default patch size is set to 80×80 but the user may modify it at runtime according to the detail desired in the output and the resolution of the exemplar. We increase the flexibility and variability of the output by allowing mirrored and rotated patches to be selected. Mirrored versions of each patch about the x-axis and the y-axis are added to the candidates. We choose to rotate each patch 8 times by $\pi/4$, which is enough to increase the quality of patch matching. Patch matching is executed in two steps: searching for patches to place in T_{out} that match the tree of features Γ_{tar} extracted from T_{tar} (Subsection 3.4.1) and filling the remaining empty areas in T_{out} with patches that contain a minimal amount of detail (Subsection 3.4.2).

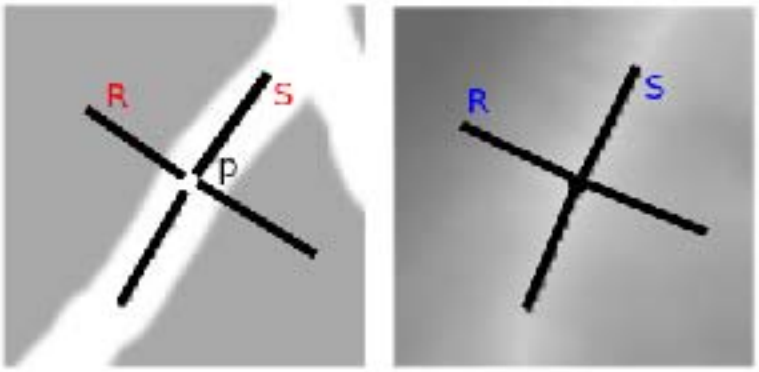
3.4.1 Feature patch matching

Feature patch matching selects patches from the exemplar that match the features extracted from T_{tar}. This process starts with a preprocessing step that compiles a list of potential candidates C_{exm}. For each feature $q \in \Gamma_{exm}$, the patch Ψ^q_{exm} centred at q and the control points $\{q_i\}$ of q are added to the list, as well as their mirrored versions. $\{q_i\}$ consists of the position of q and points where its outgoing paths intersect a circle inscribed in Ψ^q_{exm} . Then, Ψ^q_{exm} and $\{q_i\}$ are incrementally rotated by $\pi/4$ about q and the rotated versions are added to C_{exm}, to increase the probability that good matches will be selected.

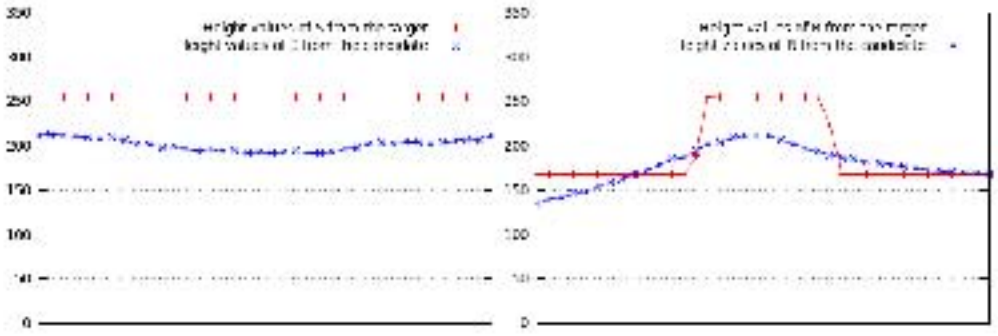
As with Zhou et al. (2007), end points and branch points of Γ_{tar} are treated as nodes of a tree, connected with chains of path features. A breadth-first traversal of this tree is used to guide the order of patch placement. The tree traversal travels along the chain of path features, in steps of one-half of the patch size, to ensure that consecutive patch placements are within a certain distance of each other and patches do not completely overlap. A root for the graph is picked, preferably the node with the highest degree, and the graph is traversed until every node has been visited.

For each feature $p \in \Gamma_{tar}$ visited in the breadth-first traversal, its control points $\{p_i\}$ are computed and a cost is assigned to each candidate in C_{exm} to indicate how similar it is to $P = (\{p_i\}, \Psi^p_{tar}, \Psi^p_{out})$. Candidates having a different feature classification to p are given a very high cost plus the overlap area cost c_o described below. The cost of a candidate $Q = (\{q_i\}, \Psi_{exm}) \in C_{exm}$ with the same classification as p is computed by combining several cost functions:

Feature dissimilarity c_f This cost function determines feature similarity between Ψ^p_{tar} and Ψ_{exm} by comparing the height profiles of p with the L_2 norm (sum of the squared differences). The height profile of p consists of height values along the outgoing segments of p. Figure 3.3(a)(b) shows a path feature p with the set of outgoing segments denoted by S. The graph in Figure 3.3(c) illustrates the difference in height values along S. When p is a path feature, height profiles of the segment orthogonal to the path are also compared by adding their L_2 norm to the feature dissimilarity cost. Candidates with a low feature dissimilarity cost are more likely to be good matches for p. This differs from the feature dissimilarity in Zhou et al. (2007) which only compares height profiles perpendicular to a path. Here, height values along the features are



(a) Target patch Ψ^p_{tar} **(b)** Candidate patch Ψ_{exm}



(c) Height profiles

Figure 3.3: Feature dissimilarity between the target patch Ψ^p_{tar} and the candidate patch Ψ_{exm}.

also taken into account.

Angle differences c_a The angles of the outgoing paths of the candidate and the feature p are compared using the normalized sum of squared differences. The angle differences indicate how similar the structure of p and that of the candidate are.

Noise variance c_n The noise variances of the candidate Ψ_{exm} and Ψ^p_{tar} are computed at multiple levels of resolution and their SSD is added to the candidate cost. The noise variance of an arbitrary patch $\Psi\ |_l$ at level $l > 0$, where $l = 0$ is the coarsest resolution, is the variance of the Gaussian noise computed by consecutively downsampling and upsampling $\Psi\ |_l$ to obtain the lower resolution $\Psi\ |_{l-1}$ and subtracting $\Psi\ |_{l-1}$ from $\Psi\ |_l$. This criterion compares the level of detail in both candidate and target patches at multiple levels of resolutions. A lower noise variance difference increases the chances that a patch with similar characteristics in terms of bumpiness at different resolutions will be selected.

Overlap area c_o Placing Ψ_{exm} in the output T_{out} may overlap with previously placed patches. The difference in height values in the overlapping region should be minimized to reduce the severity of the seam created by the overlap. This difference is easily computed with the L_2 norm of already synthesised pixels in Ψ^p_{out} and their corresponding values in Ψ_{exm}.

The total cost c of selecting a specific candidate $Q = (\{q_i\}, \Psi_{exm}) \in C_{exm}$ as a match for

the target $P = (\{p_i\}, \Psi^p_{tar}, \Psi^p_{out})$ is computed by combining the above matching costs:

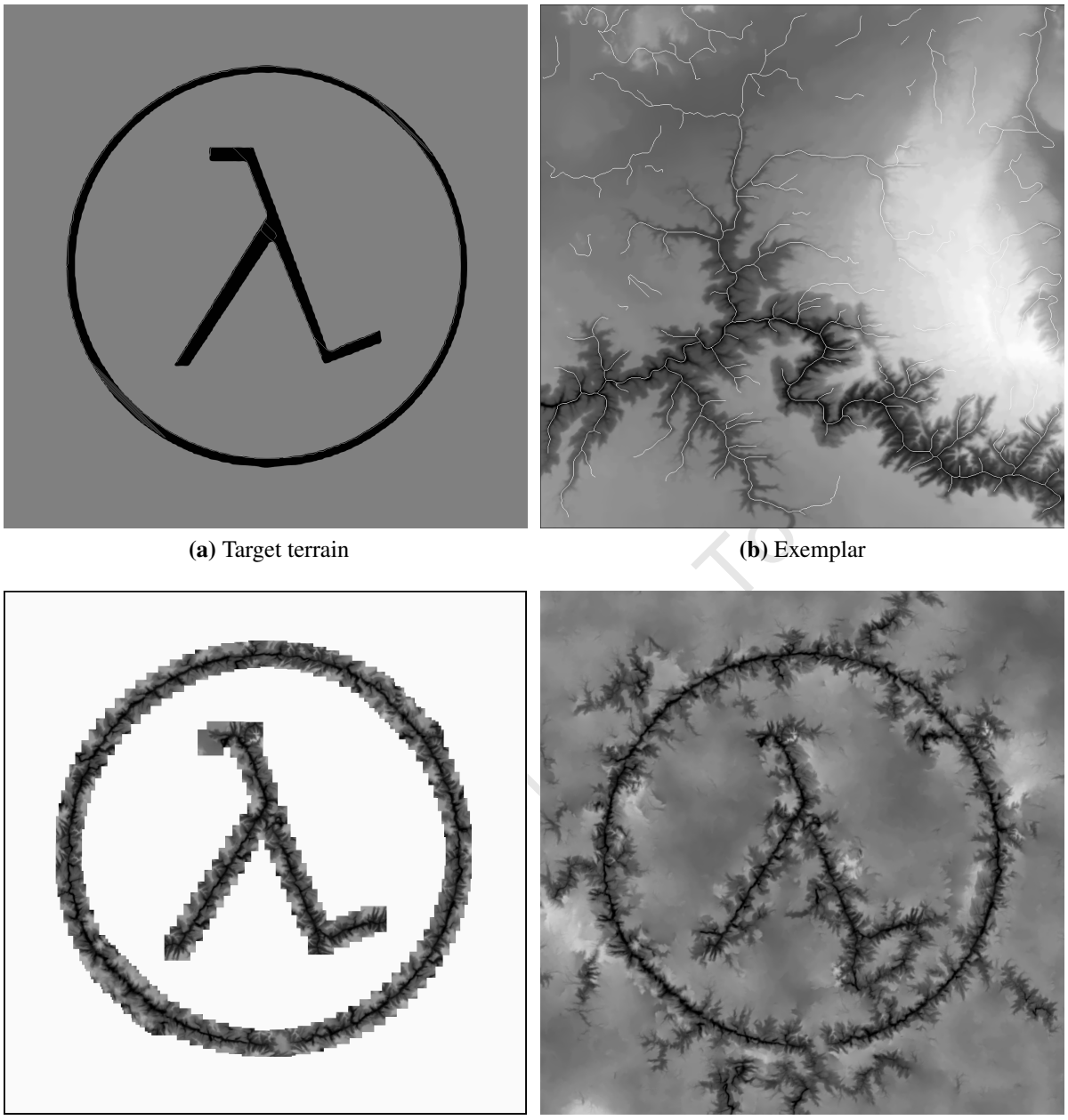
$$c(P,Q) = \alpha_f(\{p_i\}, \Psi^p_{tar}, \Psi_{exm}) + \alpha_a c_a\left(\{p_i\}, \{q_i\}\right) + \alpha_n c_n(\Psi^p_{tar}, \Psi_{exm}) + \alpha_o c_o(\Psi^p_{out}, \Psi_{exm})$$

where $\alpha' s$ determines the influence of a particular criterion. In most of our test cases , we use the following values: $\alpha_f = 5$, $\alpha_a = 2$, $\alpha_n = 0.001$ and $\alpha_o = 1$. Feature dissimilarity is the dominant criterion as it leads the system to select the candidate that closely matches the height constraints of the target features. Candidates are not deformed to fit the target and thus, the angle difference is the next dominant criterion in ensuring the feature structure in generated terrain fits with the target. The overlap area cost has a lower weight as patch merging will remove seams in the overlapping region, even if the seam is severe. The noise variance cost influenced the matching process the least. Not only does c_n have a significantly higher value than other costs, it is an overall measure of the difference in level of detail. A high weight may result in the selection of patches that do not match the dominant features in the target.

The list of candidates C_{exm} is sorted in increasing order according to their cost. If there is no patch of the same type as Ψ^p_{out}, c_o is the only criteria used for sorting. A set of k candidates with the lowest costs is selected and from that smaller set, the candidate with the lowest graph cut cost is chosen as the best match for p. Whereas the pixels difference on the overlapping region determines how severe a seam may be, the graph cut cost indicates the severity of the optimal seam. A low cost means that the patch will be less affected during patch merging. Once the best match from the feature p is found, it is placed in the output heightfield T_{out} at position p. Figure 3.4(c) shows the result of feature patch matching.

Algorithm 3.3 FeaturePatchMatching(Γ_{tar}, Γ_{exm},T_{tar},T_{exm}, T_{out})

$C_{exm} \Leftarrow BuildCandidateList(\Gamma_{exm}, T_{exm})$
for all feature p in Γ_{tar} **do**
  $\zeta \Leftarrow \emptyset$
  for all patches Ψ^q_{exm} in C_{exm} **do**
    $\zeta \Leftarrow \zeta + (\Psi^q_{exm}, cost(p, q, \Psi^p_{out}, \Psi^p_{tar}, \Psi^q_{exm}))$
  end for
  Sort ζ by cost
  Select from the first k candidates in ζ the patch Ψ_{exm} with the lowest graph cut score
  Place Ψ_{exm} at position p
end for



(a) Target terrain

(b) Exemplar

(c) Feature patch matching

(d) Non-feature patch matching

Figure 3.4: Patch-based texture synthesis. (a)-(b) Valley lines are extracted from the target and the exemplar. (c) Patches are selected from the exemplar and placed in the output. (d) The rest of the output is completed. (The white segments represent valley lines)

3.4.2 Non-feature patch matching

Some areas of the target terrain may not have any features and hence the corresponding regions in the output will be empty after feature patch matching. To compensate for this, non-feature patch matching synthesises the empty areas. The unknown region Ω of the output terrain T_{out} is

filled by iteratively copying patches with a minimal amount of detail from the exemplar T_{exm} that match the already synthesised pixels in T_{out}. Criminisi et al. (2004) show that in exemplar-based filling, the quality of the output is highly affected by the order of the filling process and propose a filling algorithm that prioritizes patches along structures. We use a similar filling order during the non-feature patch matching to ensure terrain features are preserved and correctly propagated.

Patch-based filling order

This process selects the position of the next patch placement. A best-first filling approach is used, that depends on priority values associated with each point on the boundary of Ω, $\partial\Omega$. Let Ψ^p_{out} be the patch centred at the point $p \in \partial\Omega$, as illustrated in Figure 3.5. Its priority value is influenced by a confidence term and a data term. The confidence term $C(p)$ indicates the known pixel portion around p. During initialization, $C(p) = 0$ for all $p \in \Omega$ and $C(p) = 1$ for all $p \in T_{out} \setminus \Omega$. At any point in the non feature matching process, the confidence of a point $p \in \partial\Omega$ is:

$$C(p) = \frac{\sum_{q \in \Psi^p_{out} \cap (T_{out} \setminus \Omega)} C(p)}{\sum_{q \in \Psi^p_{out}} 1}$$

The data term $D(p)$ measures the strength of the linear structure (*isophote*) orthogonal to $\partial\Omega$ (Criminisi et al., 2004). This term encourages the propagation of linear structures and is computed with the following formula:

$$D(p) = \frac{\left|\nabla T^{\perp}_{out}(p) \bullet \mathbf{n_p}\right|}{\gamma}$$

where $\nabla T^{\perp}_{out}(p)$ is the vector perpendicular to the gradient value of p, $\mathbf{n_p}$ is the normal to $\partial\Omega$ at p, $(\bullet)$ is the dot product operator and γ is the normalization factor ($\gamma = 255$ for 8-bit gray images). Figure 3.5 illustrates the vectors $\nabla T^{\perp}_{out}(p)$ and $\mathbf{n_p}$at the point p.

The filling priority of p proposed by Criminisi et al. (2004) is$\Upsilon(p) = C(p)D(p)$. However, multiplying the confidence and data terms discards pixels with a data term equal to zero even if their confidence term is very large. Instead, as in Nie et al. (2006), we use a filling priority that is the addition of both terms:

$$\Upsilon(p) = C(p) + D(p)$$

At each iteration, the pixel $p \in \partial\Omega$ with the highest priority value is selected. The best match Φ_{exm} for the patch Ψ^p_{out} is chosen from a pregenerated list of candidates and placed in the output. Then Ω is updated and $C(q) = C(p)$ for all $q \in \Psi^p_{out} \cap \Omega$. The algorithm stops when $\Omega = \varnothing$.

Matching criteria

In non-feature patch matching, no feature is associated with the patch Ψ^p_{out}. Thus, the list of candidates C_{exm} only contain patches Ψ_{exm} from the exemplar and their transformations. As with feature patch matching, patches from the exemplar are incrementally rotated by $\pi/4$ and the rotated and mirrored versions are added to C_{exm}.

The criteria used to find the best match also differ from those used in feature patch matching. The cost associated with choosing a candidate patch Ψ_{exm} is the combination of only two of the criterion used in feature patch matching: the *noise variance difference* c_n and the *normalized sum of squared differences on the overlap area* c_o. The noise variance difference encourages the selection of patches with minimal bumpiness and c_o ensures that Ψ_{exm} matches the previously synthesised pixels in Ψ^p_{out}. The total cost c for selecting Ψ_{exm} is computed from c_n and c_o as follows:

$$c(\Psi^p_{out}, \Psi_{exm}) = \alpha_n c_n(\Psi^p_{out}, \Psi^p_{tar}) + \alpha_o c_o(\Psi^p_{out}, \Psi_{exm})$$

with $\alpha_n = 0.0001$ and $\alpha_o = 10$.

The candidates are ranked in decreasing order according to their cost, and the first k candidates are selected. Then, the first k candidates are ranked in decreasing order according to the graphcut merging score as before. The first candidate is selected as the best match and placed at position p. Figure 3.4(d) shows a final output terrain after non-feature patch matching is performed.

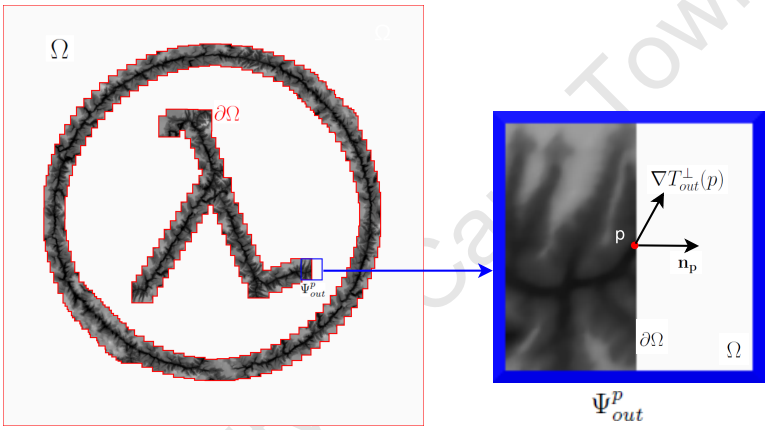


Figure 3.5: The unknown region Ω and a patch Ψ^p_{out} centred at $p \in \partial\Omega$ ($\partial\Omega$ is painted in red)

3.5 Patch merging

Once the system finds a match Ψ_{exm} for the target Ψ^p_{out}, Ψ_{exm} must be placed in the output field T_{out} at a position p, replacing Ψ^p_{out}. If Ψ_{exm} is simply pasted in the output and overlaps with already-placed patches, a seam will appear if it has different pixel values in the overlapping region. Patch merging is used to seamlessly place Ψ_{exm} in the output.

Let Ψ_1 be the old patch and Ψ_2 a new patch that overlaps with Ψ_1 over a region ov. We merge Ψ_2 with Ψ_1 into a patch Ψ such that the seam across their region ov is invisible. This is achieved by combining three different techniques: Graphcut (Kwatra et al., 2003), Shepard Interpolation (Shepard, 1968) and a Poisson equation solver (Pérez et al., 2003).

3.5.1 Graphcut

Our patch merging algorithm starts by performing a graphcut, which determines the optimal seam between Ψ_1 and Ψ_2. Figure 3.6 illustrates a graph that represents an overlap area ov. During the graph construction, we used a weighting function M that penalizes seams going through low

Algorithm 3.4 NonFeaturePatchMatching(T_{tar}, T_{exm}, T_{out})

$C_{exm} \Leftarrow BuildCandidateList(T_{exm})$
while Ω is not empty **do**
  $p \Leftarrow NULL$
  $\Upsilon \Leftarrow NULL$
  for all pixels q in $\partial\Omega$ **do**
    $\Upsilon_q \Leftarrow getPriority(q)$
    if $\Upsilon_q > \Upsilon$ **then**
      $\Upsilon \Leftarrow \Upsilon_q$
      $p \Leftarrow q$
    end if
  end for
  $\zeta \Leftarrow \emptyset$
  for all patches Ψ_{exm} in C_{exm} **do**
    $\zeta \Leftarrow \zeta + (\Psi_{exm}, cost(\Psi^p_{out}, \Psi^p_{tar}, \Psi_{exm}))$
  end for
  Sort ζ by cost
  Select from the first k candidates in ζ the patch Ψ_{exm} with the lowest graph cut score
  Place Ψ_{exm} at position p
  Update Ω
end while
return T_{out}

frequency regions. If p and q are two adjacent pixels in the overlap region Ω, then the weight of the edge connecting them is

$$M(p, q, \Psi_1, \Psi_2) = \frac{|\Psi_1(p) - \Psi_2(p)| + |\Psi_1(q) - \Psi_2(q)|}{\left|G^d_{\Psi_1}(p)\right| + \left|G^d_{\Psi_1}(q)\right| + |G_{\Psi_2}(p)| + |G_{\Psi_2}(q)|}$$

where d is the direction of the gradient determined by the direction of the edge between p and q. $G^d_{\Psi_1}$ and $G^d_{\Psi_2}$ are the gradients of Ψ_1 and Ψ_2 along d computed with the forward difference. Finding the minimum cut of the graph is a well-known graph problem solved by min-cut or max-flow (Sedgewick, 2001). Boykov and Kolmogorov (2004) provide a library that solves the max-flow problem and we make use of it. Graphcut finds the optimal seam ζ in the overlap ov and in the process, partitions Ψ into two portions: the *sink* A consisting of pixels with values coming from Ψ_1 and the *source* B containing pixels Ψ_2. Figure 3.7(d) shows the result of a graph cut algorithm. The optimal seam is still visible and further processing is needed to eliminate this seam.

3.5.2 Shepard Interpolation

The optimal but still visible seam can be removed by deforming the source B to match the sink A along the cut. Let x_i, $i = 0, 1, \ldots, N$ be the set of pixels along the seam. The following equation constrains the deformation of B:

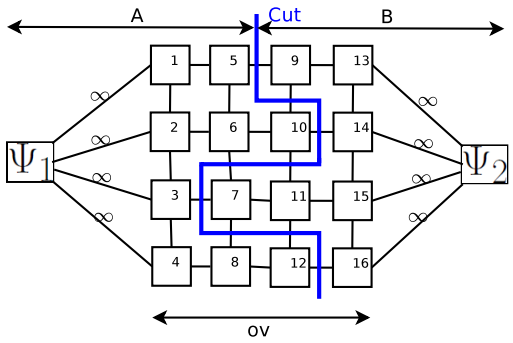
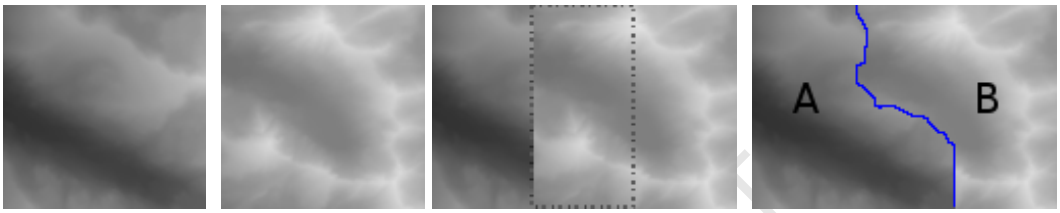


Figure 3.6: Graph construction in the graph cut algorithm. Adjacent pixels are connected and the optimal cut of the graph partitions the overlap area in two.



(a) Patch Ψ_1 **(b)** Patch Ψ_2 **(c)** **(d)**

Figure 3.7: Graph cut algorithm steps. (c) Ψ_1 and Ψ_2 overlap over a region ov (enclosed by the dotted lines). (d) Ψ_1 and Ψ_2 are merged along the optimal seam.

$$B(x_i) = A(x_i),\ i = 0, 1, \ldots, N$$

The constraint removes the difference in pixel values along the seam by ensuring that B and A have the same values along the cut.

We draw upon the deformation technique based on point features proposed by Milliron et al. (2002) to compute $B^{'}$, the deformation of B. Let $x \in B$, then the height value at x is displaced by an amount $\triangle(x)$ computed by summing displacements $A(x_i) - B(x_i)$ at points x_i on the seam, scaled by a distance-based normalised weights $\hat{w}_i(x)$. In other words, the deformation of B at a point x is

$$B'(x) = B(x) + \triangle(x)$$

$$\triangle(x) = \left(\sum_{i=0}^{N} \hat{w}_i(x) \right) (A(x_i) - B(x_i))$$

using normalised weights:

$$\hat{w}_i(x) = \frac{w_i(x)}{\sum_{j=0}^{N} w_j(x)}$$

where $w_i(x)$ is 1 at x_i and falls off radially to a distance $d_\emptyset$. We choose the weighting function w_i to be a simple Inverse Distance weighting function defined by Shepard (1968):

$$w_i(x) = \begin{cases} \left(\frac{d_\phi - d(x,x_i)}{d_\phi d(x,x_i)}\right)^\alpha & if \quad d(x,x_i) < d_\phi \\ 0 & otherwise \end{cases}$$

where $d(x, x_i)$ denotes the distance between x and x_i, d_ϕ determines the area of influence, and α specifies the smoothness of the deformation. Figure 3.8 illustrates the normalized weighting function $\hat{w}_i(x)$ with different α values.

This type of deformation is often referred to as Shepard Interpolation. $B^{'}$ now has the same values as A on the seam and hence, replacing B with $B^{'}$ removes the discontinuity (see Figure 3.9).

Shepard Interpolation does not take into account gradient values and so the gradient field of the merged terrain may have discontinuities. Although a top down view of the output will show no artifacts, a fly-over immediately reveals that the terrain is not smooth (See Figure 3.11(c)). Instead of removing the optimal seam in Ψ, we remove the seam in the gradient field G_Ψ. G_Ψ is partitioned into two: the gradient field G_A of A and the gradient field G_B of portion B. Similarly to the above interpolation that deforms B to fit A along the seam, G_B is deformed to fit G_A along the optimal cut so that

$$G_B(x_i) = G_A(x_i),\ i = 0, 1, \ldots, N$$

where x_i, $i = 0, 1, \ldots, N$ are of pixels along the seam. We use the same interpolation as before. For each $x \in B$,

$$G'_B(x) = G_B(x) + \triangle(x)$$

$$\triangle(x) = \left(\sum_{i=0}^{N} \hat{w}_i(x)\right)(G_A(x_i) - G_B(x_i))$$

G_A and G_B now have the same values along the seam and thus, G_Ψ is discontinuity free. An example of the Shepard interpolation of a gradient field is presented in Figure 3.10(a)(b). Patch merging ends with a new set of elevations Ψ' reconstructed from the modified gradient G_Ψ by solving a Poisson equation.

3.5.3 Poisson equation solver

To find the new patch $\Psi^{'}$, the following Poisson equation with Dirichlet boundary conditions must be solved:

$$\nabla^2\Psi^{'} = \nabla \cdot G_\Psi, \qquad \Psi^{'} \mid_{\partial\Omega} = \Psi \mid_{\partial\Omega} \tag{3.1}$$

where

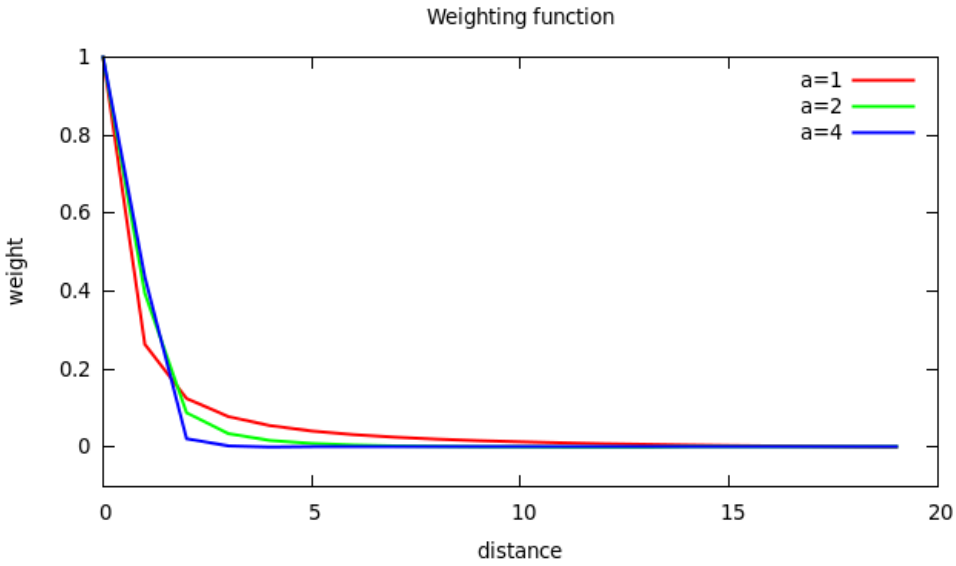
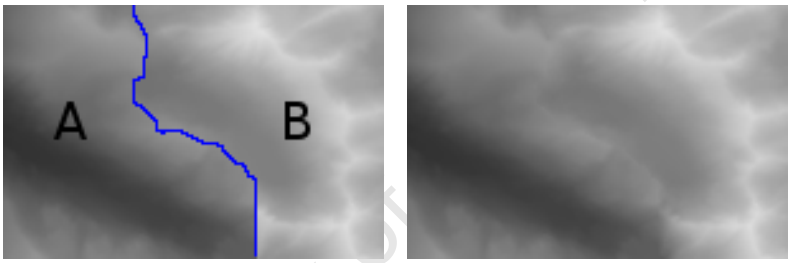


Figure 3.8: Weighting function $\hat{w}_i(x)$, $d_\emptyset = 20$.



(a) Graphcut

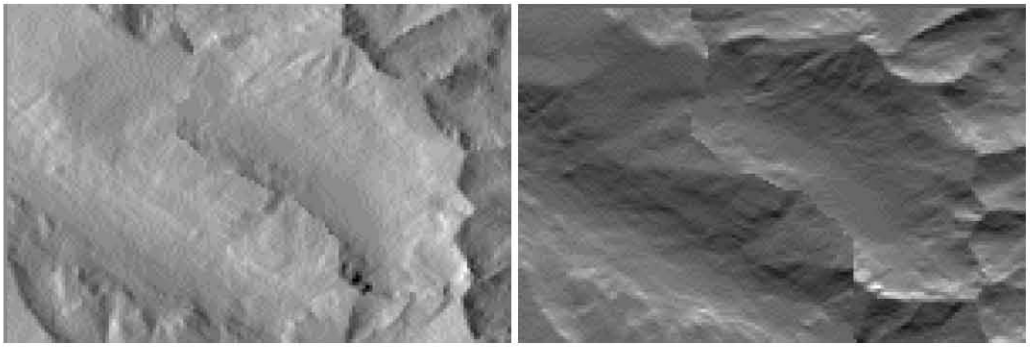
(b) B is deformed to fit A along the seam

Figure 3.9: Shepard interpolation

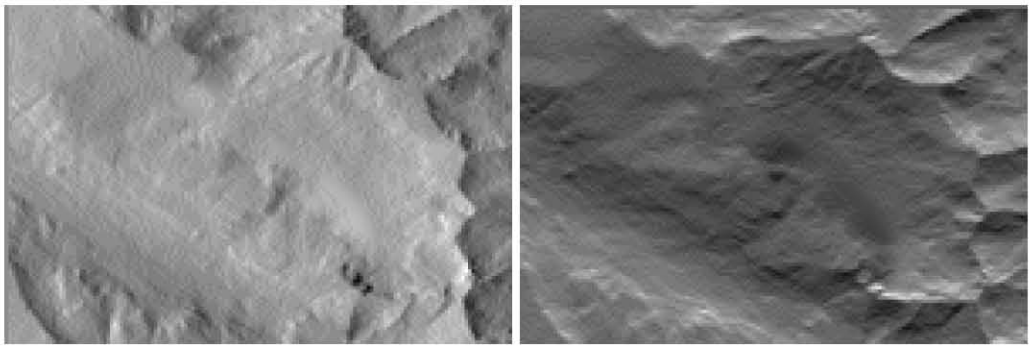
$\nabla^2\Psi' = \frac{\partial^2\Psi'}{\partial x^2} + \frac{\partial^2\Psi'}{\partial y^2}$ is the Laplacian of Ψ', $\nabla \cdot G_\Psi = \frac{\partial G^x_\Psi}{\partial x} + \frac{\partial G^y_\Psi}{\partial y}$ is the divergence of $G_\Psi = (G^x_\Psi, G^y_\Psi)$ and Ω is the whole patch area. The reconstruction is not restricted to the overlap area because pixels outside of that area may have been deformed during Shepard Interpolation.

Both the Laplacian and divergence are linear operators and thus, we use finite differences to build a large system of linear equation that approximates Equation 3.1 (George, 1970). The iterative Conjugate Gradient method (Shewchuk, 1994) is used to solve the linear system for the unknown values of Ψ'. The new set of height values Ψ' is pasted in the output heightfield and the result is a smoothly merged terrain as shown in Figure 3.10(c).

Figure 3.11 illustrates an example of our method compared with the patch merging techniques discussed in this chapter. An experiment designed to evaluate the success of this technique based on user opinions is presented and analysed in Chapter 5.



(a) Gradient field of Ψ



(b) Modified gradient field

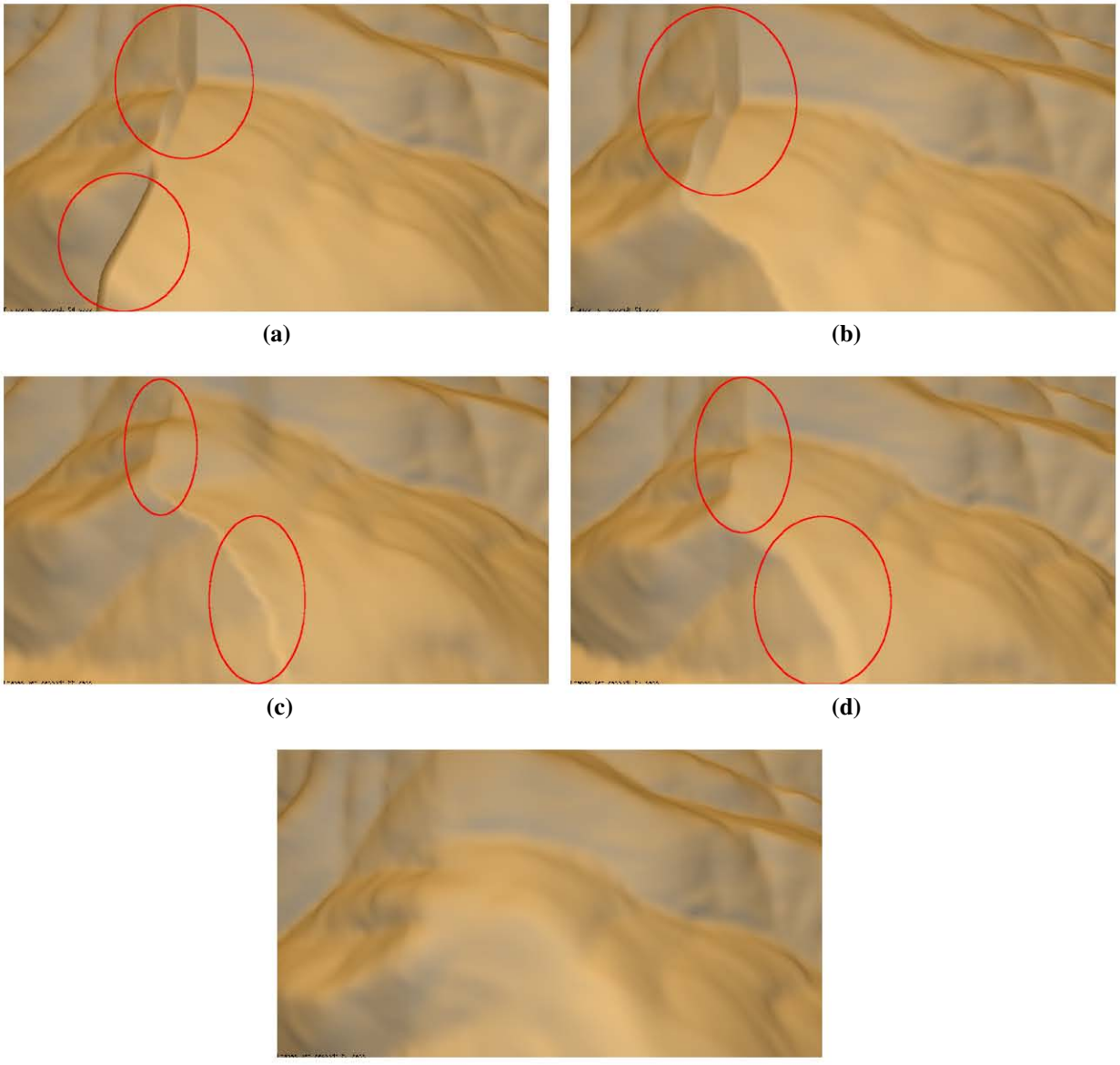
(c) Final result

Figure 3.10: Our patch merging method. (d) The horizontal and vertical components of the gradient field after the graph cut. (e) Each component of the gradient field is deformed using Shepard Interpolation. Discontinuities at the seam are removed (g) A Poisson solver is used to find a set of height values that fits the modified gradient field.

3.6 Summary

We have presented a patch-based synthesis framework that takes a terrain deformed by user-sketched curves and produces a new terrain with the same broad features as the deformed height-field and characteristics similar to a real terrain. This synthesized terrain is once again deformed in the terrain sketching interface so that it fits the original user sketches. The result is a terrain whose features are controlled by the user and that has the characteristics of a real land-

scape. The disadvantage of this patch-based synthesis is its relatively high computation cost. The 2000×2000 output terrain in Figure 3.4 was synthesized from a 1025×1025 terrain in 7 minutes. Users can be easily put off by waiting minutes to see results and it prevents interactive design cycles. To overcome this issue, a parallel solution is designed and implemented on NVIDIA's Compute Unified Device Architecture (CUDA). The next chapter describes the techniques used in the parallel implementation of the patch-based synthesis scheme presented in this chapter.



(a) **(b)** **(c)** **(d)** **(e)**

Figure 3.11: Comparison of patch merging techniques. (a) No patch merging (b) Graphcut algorithm. (c) Shepard Interpolation. (d) Graphcut+Poisson seam removal proposed by Zhou et al. (2007). (e) Our method: Graphcut+Shepard Interpolation of the gradient+Poisson equation

Chapter 4

GPU Acceleration

Texture synthesis algorithms based on neighbourhood matching are generally very slow. Pixel-based and patch-based synthesis both repeatedly find the k nearest neighbours of a patch or group of pixels. The general form of the k-neighbourhood search can be formalized as follows:

Let $R = \{r_0, r_1, r_{2,} \ldots, r_m\}$ be a set of m reference points with values in the multidimensional space $\mathbb{R}^d$ and $P = \{p_0, p_1, p_{2,} \ldots, p_n\}$ a set of n query points in the same multi-dimensional space. Given a specific distance, the k-neighbourhood problem consists of finding the k nearest neighbours of a query point $p_i \in P$ in the reference set R. In texture synthesis, a query point p_i is a block of pixels in the output texture and a reference point r_j is a block of pixels in the exemplar. The dimension d is the block size. Usually, the L_2 norm computes distances between multi-dimensional points. The naïve approach to solving this search problem, also known as the brute force method, selects the nearest neighbour of the query point p_i by following these steps:

- Compute all the distances between p_i and r_j $\forall j \in [0, \ldots, m]$
- Sort the m reference points according to the computed distances
- Select the k reference points with the smallest distances

Our framework uses the brute force approach with a more complex metric that incorporates noise variance, angle differences, feature dissimilarity and height differences in overlapping regions. This approach has a high time complexity: $O(nmd)$ for computing distances and $O(nmlog(m))$ for sorting. Several attempts to optimise this process have been implemented using tree-structured vector quantization (Wei and Levoy, 2000), kd-trees (Zelinka and Garland, 2002) or Approximate Nearest Neighbours (Liang et al., 2001). However, they rely on reducing the search space which can decrease the quality of the texture synthesis output. We refer the reader to Clarkson (2006) for an extensive survey of nearest-neighbour methods.

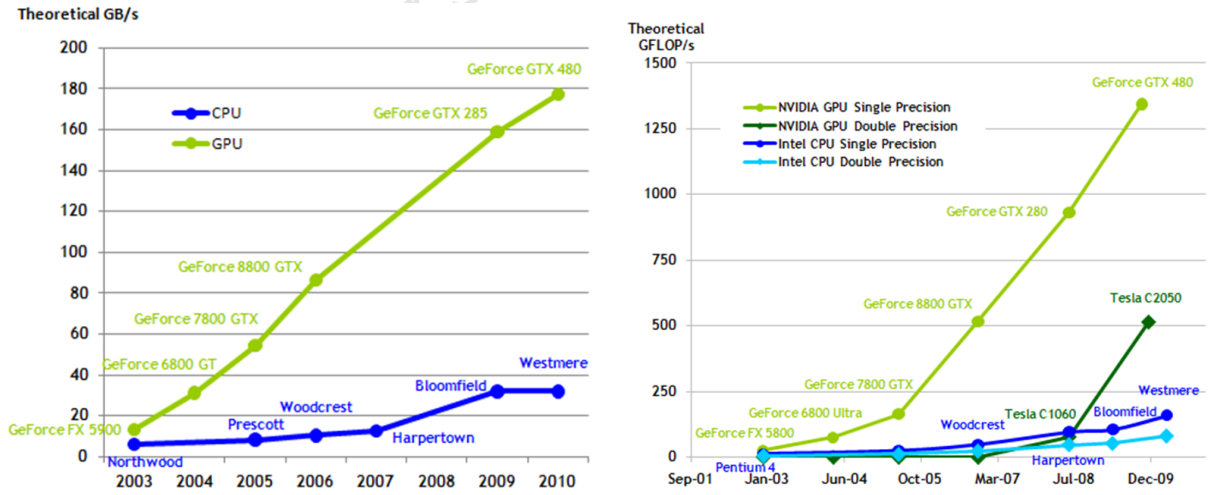
The recent progress of Graphics Processing Unit (GPU) from a static graphics rendering pipeline into a powerful multithreaded many-core processor has led to the rise of a new range of parallel algorithms. GPUs focus on highly parallel data processing at the expense of data caching and flow control, a capability well suited to the brute force of the k nearest neighbourhood problem (Garcia and Nielsen, 2009). In this chapter, we present a brief background on GPUs and specifically the NVIDIA Compute Unified Device Architecture (CUDA). We also discuss previous

texture synthesis algorithms on graphics hardware. Finally, we propose a GPU implementation of the patch-based texture synthesis framework presented in Chapter 3.

4.1 Graphics Processing Units and NVIDIA CUDA

According to Moores' law, the number of transistors that can be put inexpensively on a single processor doubles every two years. However, scaling the transistor density and clock rates increases power consumption, which means that Moore's law is hitting a power wall. Leading chip manufacturers such as Intel and AMD have addressed this problem with *multi-core processors*. By increasing the number of cores per chips, peak performance is exponentially scaled while clock frequency roughly stays the same. Although multi-core chips do not have power issues related to clock frequency, power density is still a problem as transistor density scales. Moreover, the increase in the number of transistors does not directly translate into a similar change in computing power as more transistors in modern CPUs are dedicated to accelerating memory access through caching. Allocating more transistors to lower memory latency successfully improves the communication capabilities of the CPU but mostly optimises single thread programs. It is not suited for applications such as 3D rendering that consist of small computations on each element of the input and for which computing power is far more important than caching.

For the last two decades, the gaming industry has driven the creation and evolution of GPUs, after realising that computer games needed more computer power than the CPU could offer. The Graphics Processing Unit (GPU), commonly called *graphics hardware*, is a computing unit primarily designed to handle 3D graphics rendering in applications such as computer games and 3D visualisation tools. The first GPU was introduced by NVIDIA in 1999 with its GeForce 256. The GPU follows a different trend than the CPU by devoting more transistors to data processing and less to caching. Thus, the processing speed and memory bandwidth of current GPUs far outpaces that of CPUs, as illustrated in Figure 4.1.



(a) Processing speed **(b)** Memory bandwidth

Figure 4.1: CPU vs GPU performance timeline [Images from (NVIDIA Corporation, 2010b)].

4.1.1 GPU Architecture

The GPU has significantly improved over the years, from a static graphics pipeline to a massively parallel programmable processor with special-purpose functions. A graphics pipeline transforms a representation of a 3D scene into a 2D raster image by going through the following steps:

- Vertex processing: The GPU takes as input a list of geometric primitives such as lines and triangles, formed from individual vertices. Each vertex is converted from its 3D position to a 2D screen position. Also, the colour at the vertex is determined from the vertex properties (such as material) and the light properties. A scene generally contains many vertices that can be processed independently and thus works well on GPU.
- Primitive assembly: The transformed vertices are arranged into triangles.
- Rasterization: Each triangle is decomposed into primitives called *fragments* that correspond to a screen-space pixel location.
- Fragment processing: Information such as colour, texture, and depth is used to shade each fragment. Again, this processes a large number of fragments independently, which is well suited to the GPU.
- Composition: The fragments are assembled to form a 2D image.

In the earliest GPUs, vertex and fragment processors were fixed and the programmer could only control the rendering process through the configuration of parameters such as the position and colour of a vertex or light. The GPU evolved from a fixed-function to a programmable pipeline by enabling the control of the vertex or fragment operations though vertex and fragment programs called *shaders*. This has followed a different direction from the CPU with a focus on large parallel computation instead of reducing memory latency. For example, the CPU would process vertices or fragments for 3D rendering in a sequential order, whereas the GPU processes many elements within a stage simultaneously. GPUs follow a single program multiple-data programming model, where the same operation is applied on many elements in parallel.

The rapidly evolving programmable capabilities of the GPU spawned a new research community focused on mapping general-purpose applications onto GPU hardware.

4.1.2 General Purpose GPU computing

Because the GPU dedicates fewer transistors to caching, memory latency is the major bottleneck for applications executed on graphics hardware. GPUs makes extensive use of multi-threading to hide high memory latency and thus, they are well suited for data-parallel applications with a high *arithmetic intensity*. The arithmetic intensity of a program is the ratio of floating point operations to memory operations. Applications suited to GPU implementation have large computational requirements, high data parallelism and a high arithmetic intensity. Examples of such applications are general-purpose programs such as linear algebra solvers, fluid simulations, and N-body simulations.

Originally, programming on graphics hardware was only possible though graphics APIs. Programmers had to specify input and output in terms of vertices and textures and write a fragment shader that operated on each fragment or pixel in parallel. Shading languages such as the DirectX High Level Shading Language (HSLS), Nvidia CG, and the OpenGL Shading language (GLSL) were used to write shaders for general purpose GPU programming. The main drawback of these languages is the fact that they are inherently designed for Graphics and thus, computation must be expressed in terms of structures such as vertices, textures, fragments and graphics operations such as blending, filtering, and rendering. The use of graphics terminology made programming on graphics hardware inaccessible to typical programmers.

The need for an API with a higher level of abstraction was addressed by the BrookGPU project (Buck et al., 2004). The project extends the C language with a stream programming model that represents data with streams and computation with kernels. A *stream* is an ordered set of elements. A *kernel* is a function that processes each element in a stream in parallel, and writes the output into one or more streams. Input elements are processed independently from each other. Streams are mapped to textures and kernels to fragment shaders, while reading and writing data is performed through framebuffer readbacks and texture updates. The parallel program is then executed by rendering a rectangle covering the pixels in the output texture (stream).

Following the same direction, several companies have developed programmable scalable architectures and associated APIs that bypass the use of graphics terminology or API functions. Examples of such systems are low-level hardware abstraction layers such as CTM from ATI and high level interfaces such as CUDA from Nvidia and DirectCompute from Microsoft. Other GPU technologies such as OpenCL allow the execution of general purpose programs, independent of the hardware vendor. These new technologies have boosted the number of applications mapped to the graphics hardware. Several of these applications are implemented on NVIDIA CUDA because its programming environment provides extensions to the C language which enables the management of both CPU and GPU computing, and eases interactions with the hardware for programmers familiar with C syntax.

4.1.3 CUDA Programming model

Originally, programmers could only use the GPU capabilities through graphics APIs. General-purpose algorithms were expressed with graphics primitives such pixels and textures, and output was obtained from the GPU by rendering to a texture and reading it back. In 2006, NVIDIA CUDA was released to facilitate the implementation of general purpose programs on graphics hardware. CUDA comes with a more general parallel programming model and an application programming interface (API), named CUDA C, that allows C/C++ developers to include CPU and GPU operations in the same program (NVIDIA Corporation, 2010b). Other programming languages and APIs such as CUDA Fortran, OpenCl, Direct Compute are also supported. CUDA C provides the ability to easily incorporate parallel code in a sequential program with C extensions.

The NVIDIA G8 series, released in 2006, was the first series of graphics cards designed to support CUDA and all CUDA compliant GPUs are based on its architecture. This consists of an array of multiple units called *streaming multiprocessors* (SM). Each SM is an aggregation of

Single Instruction Multiple Data (SIMD) processor cores and has access to special functions units, a multi-threaded instruction set, shared memory, local registers and a constant cache. Processors within a SM execute the same instruction at the same time on possibly different data. The *compute capability* (CC) of a GPU indicates the features supported by a CUDA device. The earliest CUDA compliant hardware, such as the GeForce 8800 GTX, has a compute capability of 1.0, while the latest devices like the Quadro 2000, has a 2.1 compute capability.

In CUDA terminology, the CPU is referred to as the *host* and the GPU as the *device*. Code meant to be executed in parallel is encapsulated in functions called kernels. Every kernel call from the host creates and executes a user-configured number of concurrent threads on the device. The CUDA programming model mainly relies on three key features to provides a high-level hardware abstraction: a thread hierarchy, a memory hierarchy and barrier synchronisation.

Thread hierarchy CUDA abstracts the number of available cores on the GPU from the programmer with the use of a thread hierarchy. Threads are batched together into groups called *blocks*. Ideally, a parallel program is divided into sub-problems each solved by a block and within a block, a sub-problem is further divided in small pieces executed by threads. Blocks are automatically scheduled to run on a variable number of multiprocessors, either concurrently or sequentially, such that the kernel scales the data parallelism to the number of available cores at run time. Thus, the thread hierarchy allows threads within the same block to cooperate or synchronise and enables automatic scalability.

Blocks are one-dimensional, two-dimensional or three-dimensional and threads are identified with a one-dimensional, two-dimensional or three-dimensional thread index. Blocks are organised into a one-dimensional or two-dimensional grid. This provides a natural way of assigning threads to elements in a data set such as a vector, a matrix or a volume.

A grid may have up to $2^{16} = 65356$ blocks per dimension and a block consists of up to $2^{10} = 1024$ threads on high-end GPUs such as the NVIDIA Fermi architecture with a compute capability 2.x, resulting in a maximum number of $2^{42} = 2^{16\times2} \times 2^{10}$ threads. The CUDA thread hierarchy is illustrated in Figure 4.2. Each block resides on a multiprocessor, and it is possible for multiple blocks to share one SM. In which case, the SM resources such as shared memory and registers are split between the blocks.

For every kernel invocation, blocks are divided into SIMD groups of contiguous threads called *warps*. The number of threads per warp or *warp size* is a hardware-based constant and is equal to 32 on current GPUs. Warps are scheduled on the processors of a SM and the same instruction is issued to all active threads within a warp simultaneously. If the 32 threads diverge at some point because of a conditional branch, each path taken is serially executed and threads that are not on that path are paused. Once all paths are completed, all the 32 threads converge back on the same execution path (NVIDIA Corporation, 2010b). Data-dependent branching can thus substantially decrease the performance of a parallel program and must be avoided whenever possible unless it is guaranteed that all threads in a warp will take a particular branch.

A kernel launch does not block the host and thus, the host and the device can run concurrently. This is referred to as *asynchronous execution* . The host and device are synchronised explicitly with specific synchronisation calls or implicitly when data is transferred from or to the device

memory. Device memory cannot be accessed by the host when a kernel is active and so when the CPU attempts a memory transfer from or to the device, it is blocked until the kernel execution is completed. Asynchronous execution is typically used when part of the sequential code can be executed independently from the kernel output.

Although a kernel launches a number of threads that are supposed to run in parallel, in practice, only some threads are executed simultaneously. The number of threads that are actively processing at any given time is limited by the number of multiprocessors and the number of cores per SM. Furthermore, the number of warps residing on a multiprocessor depends on the block size and grid size, the memory resources of a SM, and the resource requirements of the kernel, such as the amount of registers required. Warps may also have to wait for a memory read or write, because of the high latency of memory transactions on GPU. Efficiently switching between warps hides memory access latency and maximises GPU utilisation. Maximum occupancy is desired for more efficient latency hiding. Occupancy is the ratio of warps residing on the multiprocessor to the maximum number of warps that can reside on the multiprocessor. The device has various memory spaces that differ according to their scope, access latency, and location on the device. The suitable use of memory is a key performance-determining factor in CUDA programs.

Memory hierarchy Unlike the CPU, the GPU focuses more on concurrent data processing than memory caching or control flow. Thus, maximising memory throughput plays a very important role.

The host and the device have their own separate memory, and kernels can only compute from device memory, which implies that data must be copied to device memory for processing and the output copied back to host memory. The device peak bandwidth is much higher (141 GBps on Nvidia GeForce GTX 280) than the peak bandwidth between host and device memory (8 GBps on the Gen2 PCIe bus). Thus, data transfer between host memory and device memory should be minimised even if that means running functions on the device that do not provide any increase in performance compared with running them on the host. Also, CUDA programs gain in performance when the overhead associated with each transfer is reduced by batching multiple small transfers into a larger one. Higher bandwidth between the host and the device is achieved with *page-locked* (also known as *pinned*) host memory. Page-locked memory is guaranteed to be on the RAM as opposed to the regular pageable memory. On GPUs with compute capability 2.x, pinned memory can be mapped directly on the device, which eliminates the need for data transfer between the host and the device. Furthermore, asynchronous transfers immediately return control to the host once the transfer invocation is issued. Page-locked memory is a scarce resource and excessively using it reduces the overall performance of the operating system.

Memory throughput is mostly dependent on the device memory space used and access patterns. Each level of the thread hierarchy has a corresponding memory space in the memory hierarchy, as shown in Figure 4.4. Each thread has access to a *register* and *local memory* during its execution. *Shared memory* is commonly only available to threads within the same block. Finally, *global*, *constant* and *texture memory* are accessible by all threads from all blocks and the host thread. Table 4.1 provides details on the locations, scopes, access permissions and lifetimes of each memory space. To minimise latency, the use of on chip memory such as register and shared memory is preferable to off-chip memory. Unfortunately, they are not accessible from the host

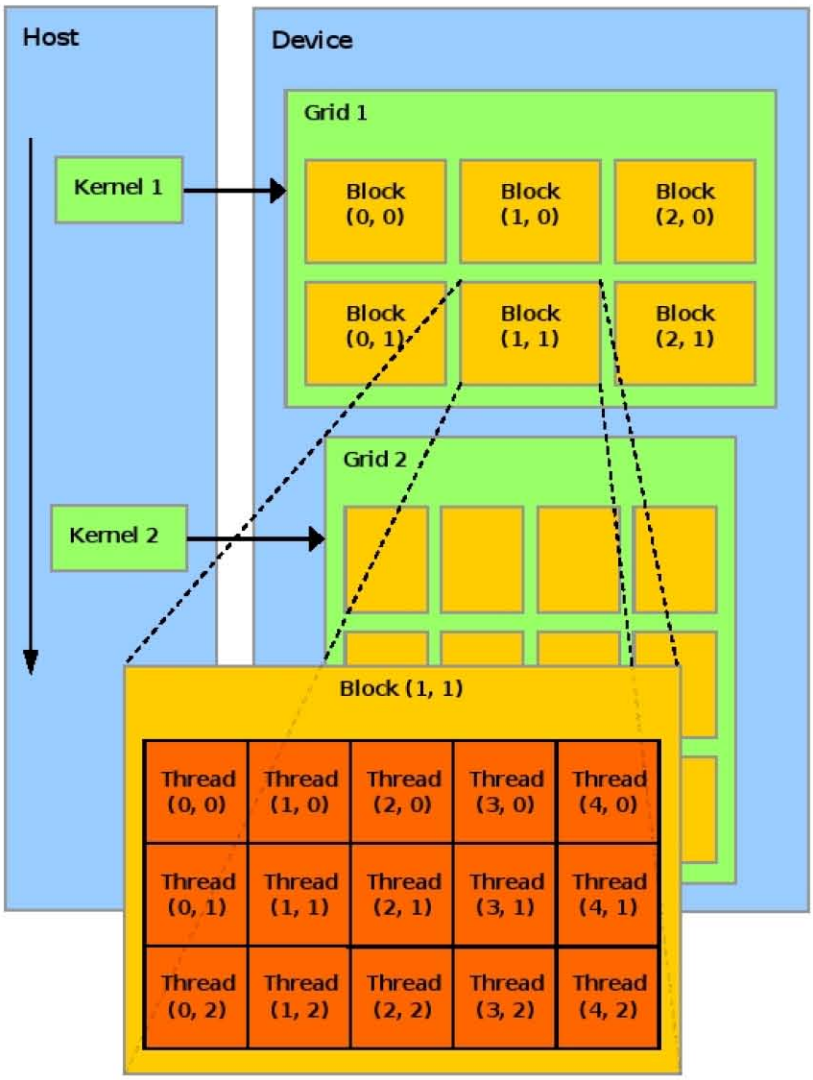


Figure 4.2: Thread hierarchy. A kernel launch creates a grid of blocks and each block is composed of an array of threads. A specific thread is addressable by its position within a block and the block index in the grid. [Image from (NVIDIA Corporation, 2010b)]

and their small size may present some limitations. On the other hand, global and texture memory have a global scope and are the largest. But along with local memory, they have the greatest access latency. Every memory space has different characteristics tailored for special purposes. Choosing the right memory space determines the performance gains.

A number of 32-bit registers are allocated to each multiprocessor and shared between cores. A multiprocessor has up to 48K registers on the latest Tesla GPU series. Accessing register memory has zero latency, making it the fastest memory on the device. However, the latency may increase due to read-after-write dependencies or bank conflicts. Memory is divided into equally-sized memory modules called *banks* that are accessible simultaneously. Memory read or write to addresses in distinct banks are serviced simultaneously. A bank conflict occurs when two or

Memory	Location: on/off chip	Cached	Access	Scope	Lifetime
Register	On	N/A	R/W	one thread	Thread
Local	Off	No	R/W	one thread	Thread
Shared	On	N/A	R/W	all threads in a block	Block
Global	Off	No	R/W	all threads + host	Host allocation
Constant	Off	Yes	R	all threads + host	Host allocation
Texture	Off	Yes	R	all threads + host	Host allocation

Table 4.1: Device memory spaces

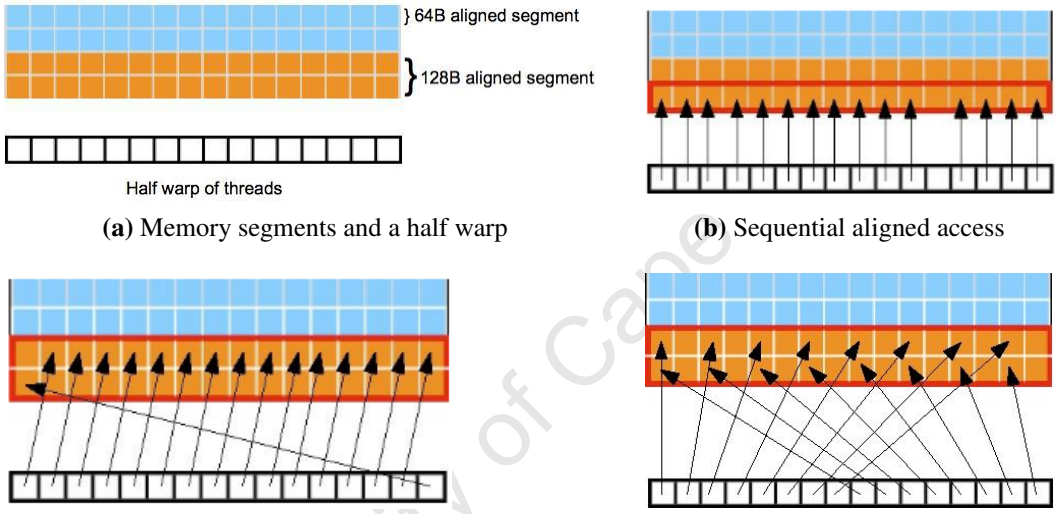
more addresses of a memory request fall in the same bank. In this case, the memory request is split in as many separate conflict-free requests as possible, decreasing memory throughput. A read-after-write dependency requires that a thread wait 24 clock cycles after writing to a register before the register can be accessed again. The latency of read-after-write dependencies is hidden if at least times $24 \times n$ threads reside on a multiprocessor, where n is the the number of cores per SM. Instructions are scheduled as optimally as possible to minimise bank conflicts. They are best avoided by ensuring that the number of threads per blocks is a multiple of 64.

Local memory is off-chip memory local to a single thread with a size that varies from 16 to 48 kilobytes depending on the GPU generation. The CUDA C compiler automatically determines which variables are held in local memory, mostly variables that are too large to fit in registers.

Shared memory is fast on-chip memory shared between threads in the same block but not accessible outside the block. On high-end GPUs such as Fermi, shared memory is up to 48 kilobytes per multiprocessor. Similarly to register memory, shared memory is split into distinct 32-bit banks accessible in parallel. For devices of compute capability 1.x, the number of banks is 16 and each shared memory request is divided into two requests: one for the first half of the warp and another for the other half. A bank conflict cannot occur if a half warp of threads access only one memory address. A *half warp* is consists of 16 contiguous threads. For devices of compute capability 2.x, the number of banks is 32 and shared memory requests are not partitioned. Thus, conflicts can occur between threads in the first half and threads in the second half of the warp. As with register memory, access latency can be increased by read-after-write dependencies and hidden if the block size is a multiple of 64. Shared memory enables the cooperation between threads in a block and is useful in avoiding off-chip memory access.

Global memory is the device equivalent of the RAM and it is situated far from the GPU cores. The high memory latency can be hidden by coalesced access to global memory. When certain access patterns are used, memory access by threads of a half warp (for devices of compute capability 1.x) or of a warp (for devices of compute capability 2.x) is coalesced into as little as one transaction. Global memory consists of rows of 64-byte aligned segments (16 floats) that are accessed in transactions of 32, 64 or 128 bytes. For instance, 16 aligned floats will be accessed in a single 64-byte transaction. When a warp is issued a memory access instruction, all memory requests of the active threads are coalesced into one or more transactions dependent on the size of the data accessed and the access pattern. Thus, if the 16 threads of a half warp request aligned floats, the requests will be serviced by one 64-byte transaction. Figure 4.3 illustrates various access patterns. The simplest access pattern is sequential and aligned, achieved when the k-th

thread within a warp or half a warp accesses the k-th word in a segment. As shown in 4.3(b), not all the threads need to access an element. 4.3(c) illustrates another access pattern in which unaligned sequential addresses fit within a single 128- byte segment. On a device with compute capability 1.2 or higher, a 128-byte transaction is performance, wasting about 50% of bandwidth. If the half warp request memory locations that are sequential but cross the 128-byte boundary, the request is divided into a 64-byte transaction and a 32-byte transaction. The strided access pattern illustrated in 4.3(d) occurs frequently with multidimensional data. In the example, the stride is 2 and all memory addresses are coalesced into a 128-byte transaction. However, half of the elements in the transaction are not used and thus 50% of memory bandwidth is wasted. The bandwidth waste increases as the stride increases until the 16 threads in the half warp are serviced by 16 serialised 32-byte transactions, resulting in poor performance.



(a) Memory segments and a half warp

(b) Sequential aligned access

(c) Misaligned sequential access

(d) Strided access with a stride of 2

Figure 4.3: Coalesced access. (a) Global memory is partitioned into 64-byte and 128-byte aligned segments. The GPU tries to fit coalesced memory request into as few transactions as possible [Images from (NVIDIA Corporation, 2010a)]

Constant memory is a read-only off-chip memory cached in the constant cache with a total size of 64 kilobytes. A read from constant memory has a performance identical to global memory on cache misses. When all threads in one half of a warp access the same address in the constant cache, the speed is equivalent to reading from the register. Otherwise, memory requests to different memory addresses by threads in a half warp are serialised.

Texture memory is read-only off-chip memory cached in the texture cache of the GPU. Texture fetches costs one global memory read on a cache miss. Otherwise, the texture fetch costs one read from the texture cache and has register like performance. Memory is accessed with texture fetches that offer a lower latency than non-coalesced access to global memory.

These memory spaces have different advantages in size and access latency. Selecting the right memory access patterns is essential to best exploit the GPU device (NVIDIA Corporation, 2010b).

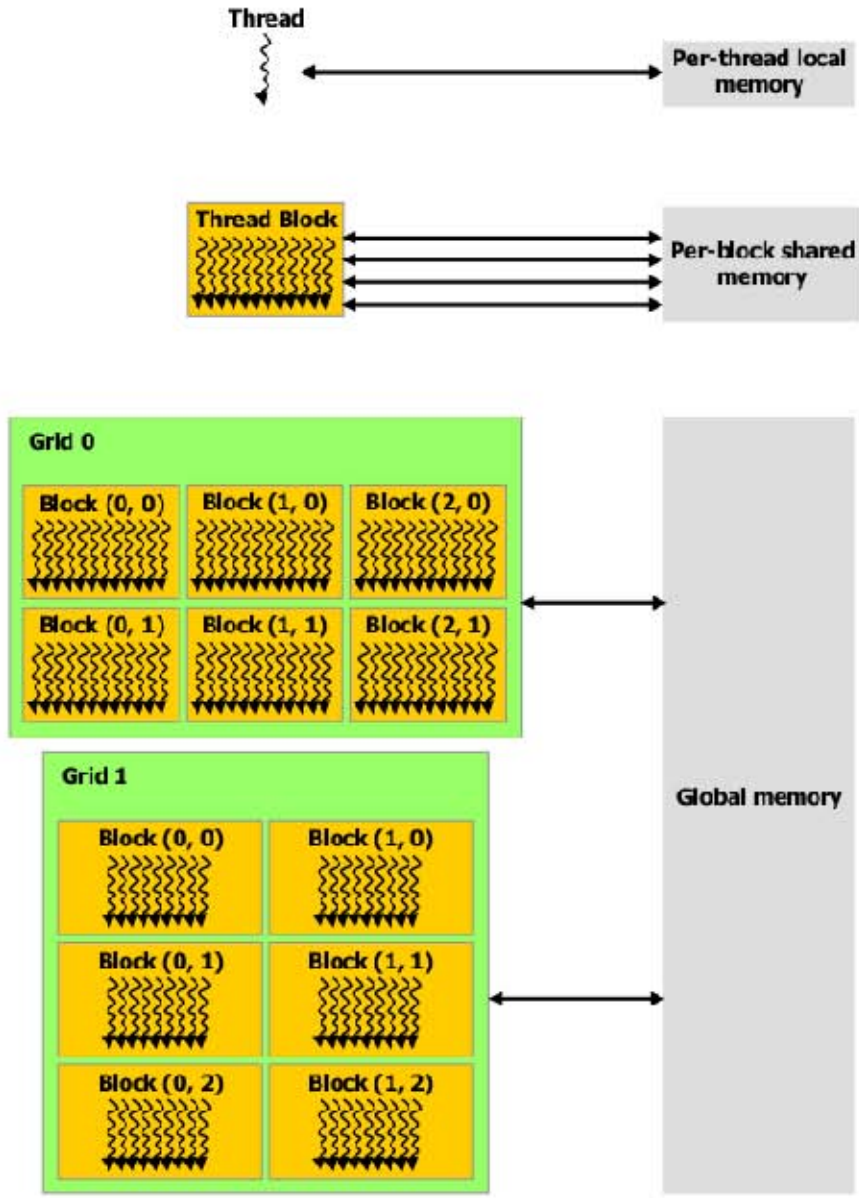


Figure 4.4: Memory hierarchy. Each level in the thread hierarchy represents the scope of a memory space in the memory hierarchy. Local memory is restricted to individual threads, shared memory is only accessible at a block level and global memory is accessible from one or more grids. [Image from (NVIDIA Corporation, 2010b)]

Barrier synchronization When barrier synchronisation is implemented in a kernel, any thread within a block stops at a barrier point and only resumes its execution when all the threads within the same block reach the barrier. This tool, along with shared memory, enables cooperation between threads in a block. It is commonly used when data is copied from global memory to shared memory in a block and all block threads must complete the data transfer before shared memory can be used. Barrier synchronisation is limited to the block level and thus, unlike multithreading on a CPU, it is impossible to synchronise all threads. This allows blocks to

run independently from each other and ensures that the GPU is always busy.

The high computational power and memory bandwidth of the GPU can only be fully exploited if a parallel algorithm follows a few key optimisations strategies:

- Maximise parallel execution: Optimising the ratio of parallel computation to sequential computation provides tremendous performance benefits because most of the work is executed on GPU. When possible, asynchronous calls should be used to enable concurrent execution between the host and the device.
- Maximise memory throughput: Data transfers between the host and the device should be minimised because they have a lower memory bandwidth than transfers within the device. The memory access pattern must determine the most suitable device memory space for data storage. Global memory transactions have the highest memory latency, which is significantly reduced when coalesced. Shared memory must be used if there are few bank conflicts, and its small size is not a limitation.
- Maximise instruction throughput: Instructions with low computational requirements should be avoided, as large throughput hides memory latency. Branch divergence within a half warp must also be avoided if possible to ensure that all the half warp threads perform the same instruction concurrently.

4.2 Previous texture synthesis on GPU

Texture synthesis on the GPU is not as widely studied as applications such as physics simulation. Designing a GPU texture synthesis algorithm is challenging because the texture is generated by evaluating each pixel or patch, in sequential order, dependent on already synthesised regions.

Multiresolution pixel-based schemes, popular for generating textures of better quality than single resolution approaches, are even harder to parallelise since, at each resolution level, the output pixels are evaluated one after another. The value of an output pixel is obtained by comparing its spatial neighbourhood against all neighbourhoods from the exemplar and selecting the input with the closest matching neighbourhood. The spatial neighbourhood depends on the most recent synthesised results, which introduces cyclic neighbourhood dependencies (Wei and Levoy, 2002). Because of this, the synthesis depends on the order in which pixels are processed and generating the output texture at multiple level of resolution in parallel is impossible.

Wei and Levoy (2002) propose an order-independent approach that removes cyclic dependencies by using a multiscale pyramid representation of the output. The coarsest level is a small noise image created by randomly assigning pixel values. A pyramid is also created for the input images and the output pixels are processed independently from each other from the coarser to the finest level of resolution. At each level, multiple neighbour-matching passes are applied on each output pixel for correction independently of the order in which the pixels are processed.

Lefebvre and Hoppe (2005) propose a parallel algorithm based on Wei and Levoy (2002)'s order-independent scheme that executes pixel correction passes on GPU. Windows of an infinite, deterministic and aperiodic texture are generated by using a Gaussian output pyramid consisting of

jittered pixel values from the exemplar and correction passes implemented as a sequence of pixel shading passes on GPU. The pyramid construction, jittering, and passes are all executed on the GPU device so that CPU utilisation is nearly zero.

Zou et al. (2010) propose a graph cut-based algorithm built on the work of Kwatra et al. (2003), that repeatedly places an instance of the exemplar with a certain displacement, in the output texture. Specifically, at each iteration, a portion P_t of the exemplar starting at an offset t is selected for placement so that the difference in pixel values in the overlapping region of the output and P_t is minimised and structural information is preserved. A GPU implementation of the graph cut method (Vineet and Narayanan, 2008) is used to determine the optimal portion of the patch to placed in the output. The synthesis is further accelerated by a preprocessing step on GPU that minimises the search space by selecting relative offsets with minimal matching errors. On a NVIDIA 8800GTX with 128 cores, this approach is 12 times faster than its sequential implementation. However, it is limited by the fact that the exemplar is repeatedly placed in the output, causing repeating patterns with no variation.

A simpler approach to texture synthesis on the GPU treats the neighbourhood matching process as a k-neighbour search problem and implements the brute force approach to this problem on GPU. The computations of neighbourhood distances (matching costs) are independent of each other and thus are fully parallelized in several algorithms (Garcia and Nielsen, 2009; Zhang et al., 2010; Xiao et al., 2010). Garcia and Nielsen (2009) present a CUDA-based brute force approach that stores the multi-dimensional query points in global memory for coalesced reading and the set of reference points or candidates in texture memory. Finding the k reference points with the k smallest distances from a given query point is performed using an insertion sort on GPU. Using this brute force approach in Efros and Leung (1999)'s pixel-based synthesis of a 128^2 image from a 64^2 exemplar yields a 48X speed-up compared to using Approximate Nearest Neighbours (Indyk and Motwani, 1998). Zhang et al. (2010) also parallelize the computation of neighbourhood distances in the Wei and Levoy (2000) pixel-based algorithm and accelerate the algorithm by an average of two orders of magnitude. This technique takes advantage of the data parallelism of matching, is easy to implement and applicable to patch matching in patch-based synthesis.

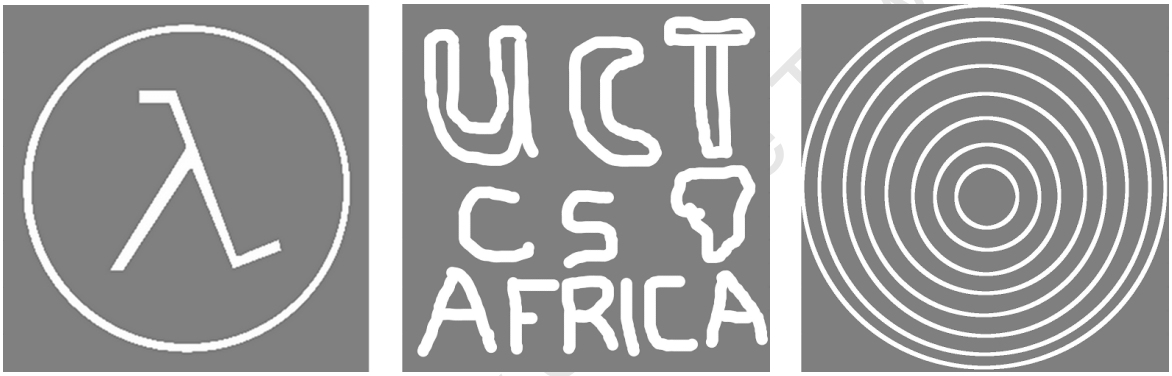
We conclude that the GPU is a very valuable tool for accelerating texture synthesis algorithms and its focus on computer-intensive parallel data processing is well-suited for the patch matching in our proposed framework. Garcia and Nielsen (2009) and Zhang et al. (2010) adopts a simple approach to the k nearest neighbour problem that consists of computing costs or distances on GPU for every query point. A similar concept is used to accelerate feature and non-feature patch matching in our patch-based terrain synthesis.

4.3 GPU Implementation

Before attempting to parallelize our framework, we ran a performance analysis of the patch-based synthesis to identify the bottlenecks. We investigated CPU texture synthesis performance of three different target terrains against exemplar terrains with sizes ranging from 513×513 to 2049×2049. The three target terrains, illustrated in 4.5 have different feature density and therefore the number of patches placed during feature matching varies between 25%, 50% and

70% of patches placed during terrain synthesis. Six example terrains are used to observe the change in performance according to the exemplar size. These exemplars are scaled versions of one terrain, and the number of candidate patches associated with each is shown in Table 4.2. These target and example terrains are used to investigate the performance of our sequential and parallel solutions.

The performance analysis of the CPU patch-based terrain synthesis using a 2049×2049 exemplar is summarized in Figure 4.7 The performance results clearly shows that patch matching accounts for most of the computation time. Patch matching finds the k most similar candidate patches to a target patch and then select from these candidates, the patch that minimises the graph cut cost associated with placing it in the output. We determine the k most similar candidate patches by computing the costs of each candidate patch, sorting the candidates set according to their costs and choosing the first k candidates. The cost of each candidate is updated independently. This is a highly parallel problem with no dependencies that can easily be implemented on a GPU.

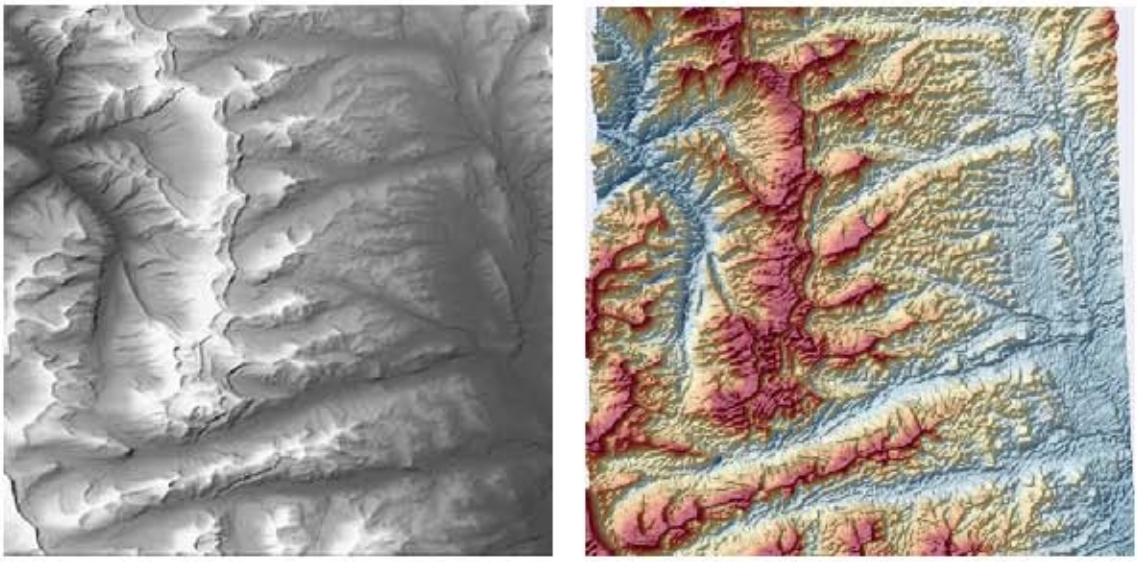


(a) **(b)** **(c)**

Figure 4.5: Target terrains. (a) 25% of the synthesised patches are placed by feature matching. (b) 50% of patches placed in the output are processed by feature matching. (c) Most patches, 75%, are placed during the feature matching process.

Exemplar size	Number of features or candidates for feature patch matching	Number of candidates for non-feature patch matching	Percentage of feature vs non-feature candidates
513×513	2980	1210	$71\% - 29\%$
1025×1025	10550	5290	$66\% - 34\%$
1537×1537	18730	12960	$59\% - 41\%$
2049×2049	26060	24010	$52\% - 48\%$

Table 4.2: Number of candidate patches per exemplar. Each exemplar is a scaled copy of the Flathead mountain range in Figure 4.6. They are used to investigate the synthesis performance as the exemplar size varies (see Figures 4.7, 4.11, 4.10, 4.12, 4.13, and 4.14).



(a) Heightmap **(b)** 3D rendering

Figure 4.6: Flathead Mountain range (MT, USA)

4.3.1 Patch matching

Patch matching is the bottleneck of the framework, accounting for more than 85% of the computation time. There are two possible ways of accelerating this process. The first approach consists of simultaneously finding multiple best matches for different locations and placing them in the output. Selection of the best matching candidates takes place independently and may result in the placement of patches that differ significantly from each other in height values. At some point, patches placed by independent threads must be joined and the merge may not yield realistic results if the patches differ significantly in the overlapping region.

The second approach, used by most patch-based texture synthesis schemes on GPUs, places patches sequentially. However, the selection of the best match is accelerated by computing the costs associated with candidate patches in parallel. The candidate with the lowest score is then obtained by a parallel reduction of the list of candidates. In cases where the k candidates with the lowest score are needed, a parallel sort of the list is performed. We use this approach to increase the performance of our framework. The *parallel insertion sort* function provided by the Thrust library is used for sorting the list of candidates.

Similarly to Garcia and Nielsen (2009), we accelerate the process by computing candidate matching costs on GPU. This is achieved this by allocating each thread to calculate the matching cost of a candidate, so that the number of threads is equal to the number of candidates. Each thread accesses the output, the target patch, the corresponding candidate patch, and other characteristics such as control points and noise variances. Thus, memory latency is a likely bottleneck that should be reduced as much as possible to increase the performance of a GPU implementation.

Shared memory is often used to access data with minimum latency. Data is copied once from off-chip memory to shared memory, and the threads can access shared memory as often as needed

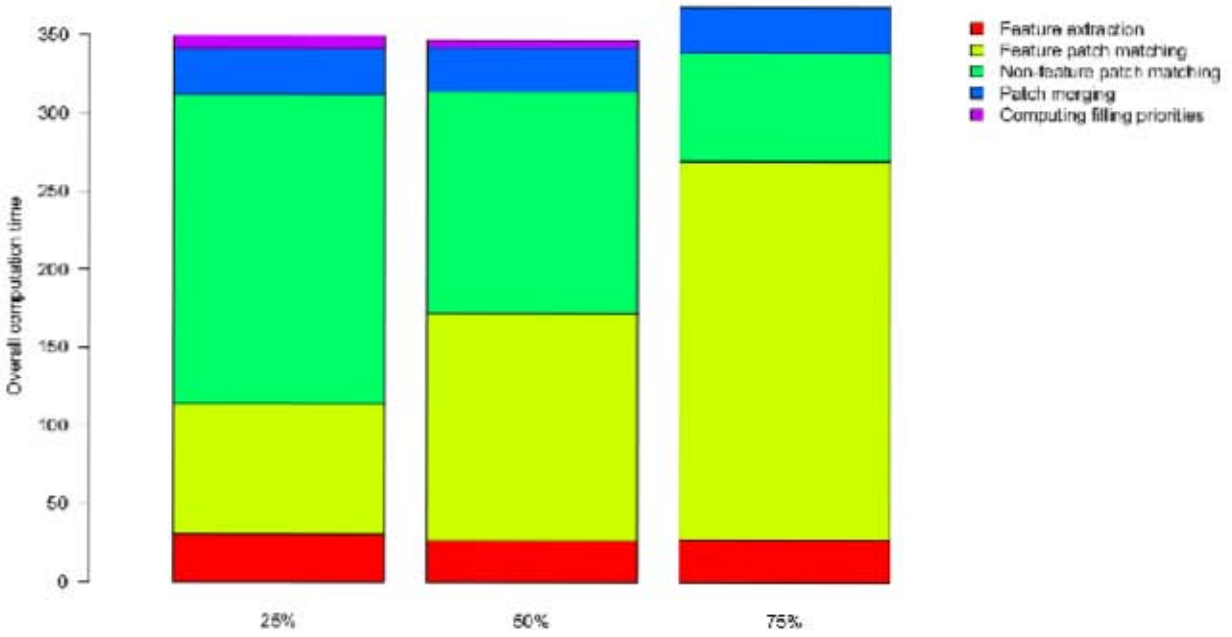


Figure 4.7: Distribution of work on the CPU for situations where 25%, 50 % and 75% of patches are placed by feature patch matching. The overall computation time is measured in seconds. Patch matching is the bottleneck of terrain synthesis, accounting for more than 80% of the CPU time.

without the high latency that off-chip memory transactions will incur. Specifically, the exemplar can be divided according to the number of blocks and parts of the image copied into the shared memories of these blocks. As a result, the shared memory of a block will contain candidate patches needed by the block threads. However, the largest shared memory size is 48 KB for devices of compute capability 2.x and above. Thus, shared memory can only contain $49,152$ bytes equivalent to $12,288$ floats and therefore, cannot contain square patches of size greater than 110×110. For devices of compute capability 1.x, the maximum patch size that the shared memory of a block (16KB) can accommodate is 64×64. Hence, the use of shared memory is not a viable option as it places severe restrictions on the patch size. Means of reducing memory latency, as discussed in the Section 4.1.3 are coalesced access to global memory and the use of caches. Constant memory size is 64KB on all CUDA-enabled GPUs, which is too small for exemplars of size greater than 128×128. So our choices are reduced to global memory and texture memory. In this section, we present two different GPU designs based on the use of these memories and we discuss their advantages and shortcomings. These optimisation techniques target the feature patch matching phase, as it is similar to non-feature patch matching.

Coalesced global memory Best performance is achieved by global memory transactions when threads in a half warp or a warp (depending on the compute capability) access memory in a sequentially aligned pattern. When computing the matching costs for a target feature p, all threads access the same address locations in the target patch (centred at p) , control points and noise variances of p. However, they access different candidate patches that may be rotated or mirrored, in a fashion that is not sequential nor aligned. To encourage coalescing behaviour, the

k-th thread in a half warp must access the k-th element in a linear array. To achieve this, all possible candidate patches are copied into a linear array and their pixel values are arranged in a such a way that consecutive threads access consecutive array indices. The first pixel of each patch is copied into the linear array, then the second element of each patch and so on. Specifically, if $C = \{C_1, \ldots, C_m\}$ is a set of m possible candidates and each C_r has pixels values in a scan line order $\{c_{r1}, c_{r2}, \ldots, c_{rd}\}$ where $d = s \times s$ is the patch size and c_{rk} is the pixel value at position k in the patch C_r, then a large linear array A of size md is created and stores the pixel values $\{c_{11}, c_{21}, \ldots, c_{m1}, c_{12}, c_{22}, \ldots, c_{m2}, , \ldots, c_{md}\}$. To avoid copying a large array from host to the device, this transfer of candidate patches to A is performed on the device. Also, instead of computing the pixel values of each candidate in a separate step, candidates store their top left position in the exemplar as well as information such as rotation angle and mirror axis. This information is used to compute the pixel values of each candidate in the kernel. This process is formalised in more detail in Algorithm 4.1 and illustrated in Figure 4.8

Algorithm 4.1 Transfer candidates patches to device memory

function $buildCandidateSet(A, T_{exm}, C, m, s)$
$t \Leftarrow blockId \times blockDim + threadId$
$x \Leftarrow$ `column offset of` $C[t]$
$y \Leftarrow$ `row offset of` $C[t]$
$rot \Leftarrow$ `rotation angle of` $C[t]$
$mir \Leftarrow$ `mirror axis of` $C[t]$
for $j = 0$ `to` s **do**
  for $i = 0$ `to` s **do**
    $k \Leftarrow i + j * s$
    $A[t + k * m] \Leftarrow T_{exm}[transform(x + i, y + j, rot, mir)]$
  end for
end for

During patch matching, if threads $\{t_1, \ldots.., t_m\}$ access a pixel at position (i, j) in their corresponding candidate patch, they are accessing values in consecutive locations $\{A_{1k}, \ldots, A_{mk}\}$ where $k = i + j * s$. Thus, all threads access sequentially aligned memory addresses and so do threads in a half warp or warp. By using the design described above, a memory request by threads in a half warp for a pixel value in their candidate patch results in 16 4-byte memory requests coalesced into a single 64-byte transaction.

Control points and noise variances of candidate patches are copied into linear arrays on the device. Unlike 2D candidate patches, the set of control points and the list of noise variances for a specific candidate feature q is linear and small (a feature rarely has more than 3 branches and a default value of 3 is used for the number of levels of resolution). The control points and noise variances for all candidates are concatenated into 1D arrays A_{con} of control points and A_{var} of noise variances. Threads in a half warp or a warp access these values in a strided pattern where the stride is the number of levels of resolutions or the number of control points per candidate. The strided pattern also encourages coalescing behaviour although the performance benefits are

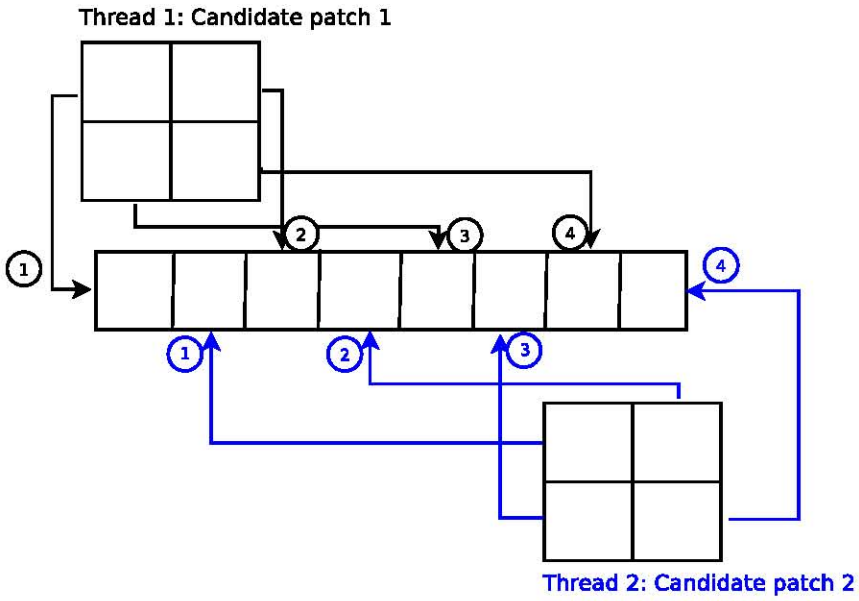


Figure 4.8: Candidates patches are placed a linear arrays such that threads can access their corresponding candidate in a sequentially aligned pattern. The encircled digits represent a patch pixel value and the arrows indicates where that pixel value is placed in the linear array.

not as significant as those gained by a sequential aligned pattern. This GPU implementation of the patch matching process is very similar to the CPU version presented in Chapter 3, except for coalesced memory access and the fact that the loop across the candidate patches is removed by assigning each candidate to a thread. Using a large array that contains all candidate patches not only enables a sequentially aligned access pattern but also reduces the amount of computation. It removes the needs for rotating and mirroring patches from the exemplar in the process of finding the best match every time a new patch must be placed in the output.

Thus far, we have used memoization of candidate patches in the sequential version (building a set of candidates) and in the GPU optimisation discussed here. It demands large memory resources for even small exemplars. Indeed, every square patch in the exemplar yields 9 rotated and mirrored patches that must all be placed in memory for the duration of the texture synthesis. A small increase in the exemplar size produces a significant increase in the number of candidate patches and space needed. For these reasons, large exemplars (such as 2500×2500 real landscapes) cannot be used as they will quickly use up all the available memory. Furthermore, memoization decreases the instruction throughput of the GPU implementation by reducing the amount of computation. Thus, we propose another GPU optimisation method based on texture memory, that computes candidate patches on the fly and does not limit exemplars to small sizes.

Texture memory Instead of coalesced memory access, caching is used to reduce high memory latency. The exemplar is bound as a texture and accessed through a texture cache. Accessed patches have a 2D spatial locality and pixel values are often reused, which makes texture memory highly suitable for texture synthesis. Furthermore, CUDA textures have capabilities such as interpolating between pixel values that are useful for operations such as rotation. In addition, texture memory supports very large exemplars without running out of memory. During patch

matching, the pixel value of a candidate patch is obtained by computing the corresponding pixel in the exemplar, transforming it accordingly (rotation or mirroring), and computing the interpolated pixel value of the transformed pixel coordinates. The transformation and interpolation is repeated each time a pixel value from a candidate patch is requested. Such operations would have significantly decreased the performance of a sequential implementation but the GPU has special function units designed to handle these computations. The use of texture memory does not demand any rearrangement of data and so reduces the implementation complexity of the GPU acceleration. Control points and noise variances are still concatenated in linear arrays and accessed as described in the previous subsection. The performance benefits gained by using this design arise from the 2D spatial locality provided by the texture cache and the fact that once cached, pixel values in the exemplar can be accessed with a low memory latency similar to that of register memory. A comparison of this scheme against the design based on coalesced access memory is performed in the next subsection.

4.3.2 Patch merging

Patch merging contributes less than 10% of the computation time but this percentage becomes significant as the target and the patch size grow larger. As discussed in Chapter 4, the graph cut algorithm, Shepard Interpolation and a Poisson equation solver are used during merging. We optimised the Shepard Interpolation and Poisson image solver on GPU to achieve a better overall performance.

Shepard Interpolation Shepard Interpolation deforms the gradient field to remove the discontinuities along the optimal seam created by the graph cut. The sequential version presented in Chapter 4 deforms each pixel value by displacing it by an amount that depends on the discontinuities at the seam and the distance from the pixel to the seam. The parallel Shepard Interpolation copies an array of pixel positions and the discontinuities at the seam from the host to the device. The displacement values are computed in parallel for each pixel position, and an array of displacements is copied back to the host. The host then uses the computed values to deform the gradient field. The speed obtained from the parallel solution is 2X faster than the sequential interpolation. We attribute the small speed-up to the fact that when patch sizes are small, Shepard Interpolation does not have the high computational requirements necessary to fully utilise the GPUs computational power.

Poisson image solver The Poisson equation consists of two main steps: building a large system of linear equations to approximate the Poisson equation and solving the system. Finding a numerical solution to the system of equations is the most time consuming step. We use the conjugate gradient method, an iterative method based on multiple matrix multiplications, scaling and additions. Let x be a vector of unknown values in the equation$Ax = b$. Initially, the values of x are set to an approximate solution or, in this case, to 0. A is a $d \times d$ matrix and b is a $d \times 1$ vector, where d is the patch size and the number of unknowns. Algorithm 4.2 presents the Conjugate Gradient algorithm used to solve the equation $Ax = b$ for x on the CPU.

The conjugate gradient method on GPU is well-known and proceeds as follows: A, x and b are copied from the host onto the device and all matrix and vector operations are executed on GPU.

But the GPU implementation only delivers performance benefits if the number of unknowns (the size of x) is large enough to hide the overhead of calling the kernels. Indeed, a kernel is called for every matrix and vector operation and the overhead associated with each invocation makes the whole GPU solver slower than its CPU equivalent if x is not large enough. In our framework, the unknowns are all the pixels in a patch and hence the size of unknowns is the patch size. Figure 4.9 shows how the performance of the parallel and sequential solver responds to variations in patch size. Notice how the performance of the GPU solver surpasses that of the CPU version for patches of size larger than 175×175. However, the typical patch size is 80×80. To accommodate all patch sizes, the GPU implementation of the framework was modified to use the GPU conjugate gradient method for patches larger than 175×175.

Algorithm 4.2 Conjugate gradient

function $ConjugateGradient(A, x, b, niter, tol)$
$Ax \Leftarrow A * x$
$b \Leftarrow b - Ax$
$r1 \Leftarrow b * b$
$k \Leftarrow 1$
$p \Leftarrow b$
while $r1 > tol * tol$ `and` $k <= niter$ **do**
  if $k > 1$ **then**
    $p \Leftarrow (r1/r0) * p$
  end if
  $Ax \Leftarrow A * p$
  $a \Leftarrow r1/(p * Ax)$
  $x \Leftarrow x + a * p$
  $b \Leftarrow b - a * Ax$
  $r0 \Leftarrow r1$
  $r1 \Leftarrow b * b$
  $k \Leftarrow k + 1$
end while

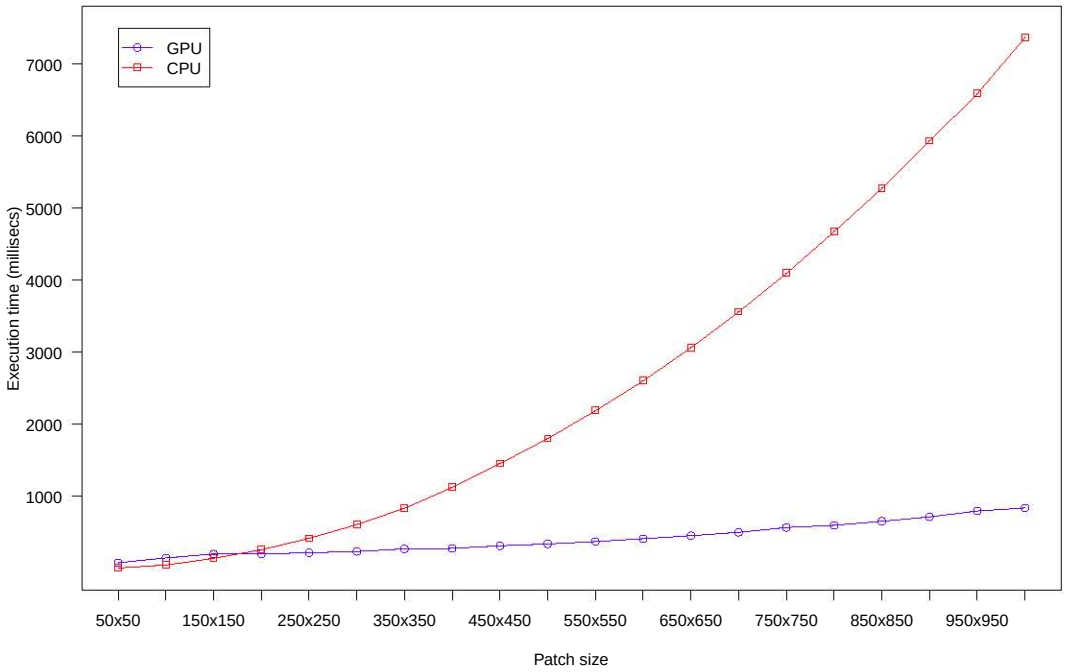


Figure 4.9: CPU vs GPU Poisson solver performance

4.4 GPU benchmarking

We consider five implementations of the patch-based terrain synthesis: a CPU implementation with memoization (CPU1), a GPU solution with memoization and non-coalesced memory access (GPU1 NC), a GPU version with memoization and coalesced memory access (GPU1 C) and a GPU solution that uses texture memory with no memoization of candidate pixel values (GPU2). A fixed target terrain of size 1024×1024 is used in all the performance analysis, while the exemplar size and patch size vary. The number of candidates and execution times of the feature patch matching and the non-feature patch matching are recorded. Refer to Table 6.2 in Appendix A for a record of these timings. The computer used for these experiments is an Intel Core 2 Quad 2.33GHz with 3GB of DDR3 memory. The GPU device used is an NVIDIA GTX 280 with compute capability 1.3, 1GB of DDR3 memory and 30 multiprocessors ($30 \times 8 = 240$ cores) interfaced with a PCI Express 2.0x16 port.

According to Figure 4.11 and 4.10, GPU2 patch matching is 30 times faster than the equivalent sequential version CPU2. In both implementations, patches are extracted from the exemplar during the computation of matching costs and therefore, the same transformations are repeated every time a new patch must be placed in the output. The high computational requirements of this approach makes it suitable for a GPU implementation. However, each rotation transformation incurs a cubic interpolation that interpolates the height value of a transformed pixel value from 16 neighbours. The GPU2 solution is therefore limited by the extra computation. The use of memoization removes the necessity to interpolate values. Instead of a transformation and cubic

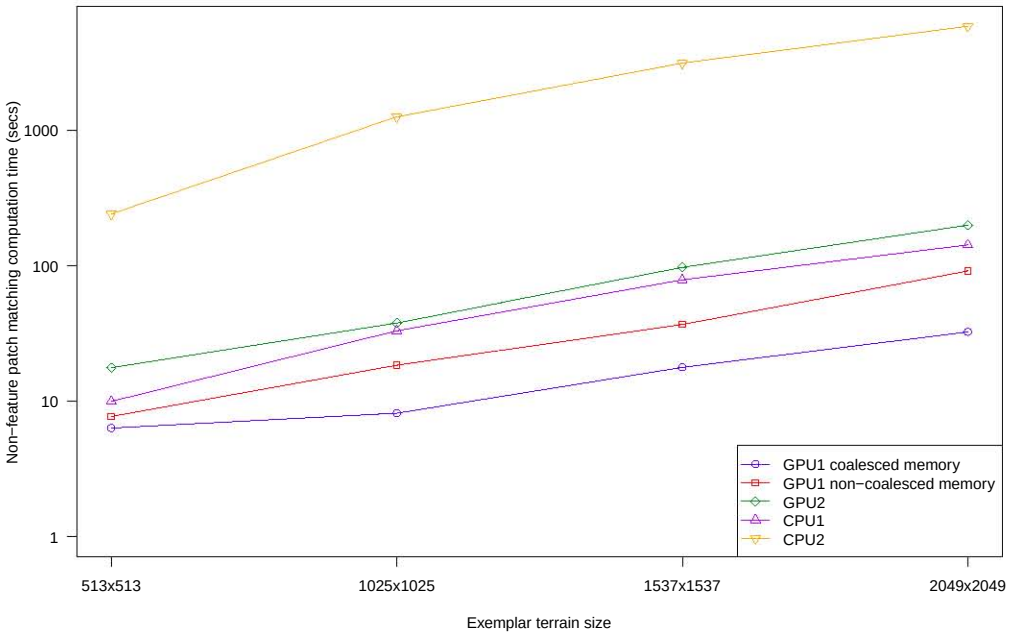


Figure 4.10: Performance analysis of non-feature patch matching. Execution times were obtained by adding up the time taken for matching 288 80×80 patches containing target features.

interpolation, getting the value of a pixel costs one read from global memory. GPU1 C is 6 times faster than CPU1 and GPU2, and 2 times faster than GPU1 NC.

Figure 4.12, 4.13 and 4.14 shows the overall performance of the framework implementations based on CPU2, CPU1, GPU1 NC with non-coalesced memory access, GPU C based on coalesced access and GPU2 patch matching with texture memory. The GPU C version has the highest performance, with a 2X speed-up compared to CPU1 and GPU2. As the exemplar gets larger, more time is spent on feature extraction, which accounts for the slow speed-up compared to patch matching. Although GPU1 has the lowest computational cost, it is severely limited by the size of the exemplar itself. We limit the exemplar size in the performance analysis results presented so far mainly because 2049×2049 is the maximum size that implementations using memoization could handle for a patch size of 80×80. On the other hand, GPU2 is as slow as the sequential solution using memoization but supports large exemplar sizes without running out of memory. We use it to generate the terrains presented in the next section whenever the exemplar size is too big for the use of memoization.

4.5 Discussion

This chapter presented two GPU implementations based on different device memory spaces, namely coalesced global memory and texture memory. The GPU solution based on coalesced global memory (GPU1 C) will run out of memory before the similar CPU solution (CPU1) due to the fact that the host memory (RAM) is generally larger than device memory. However it is

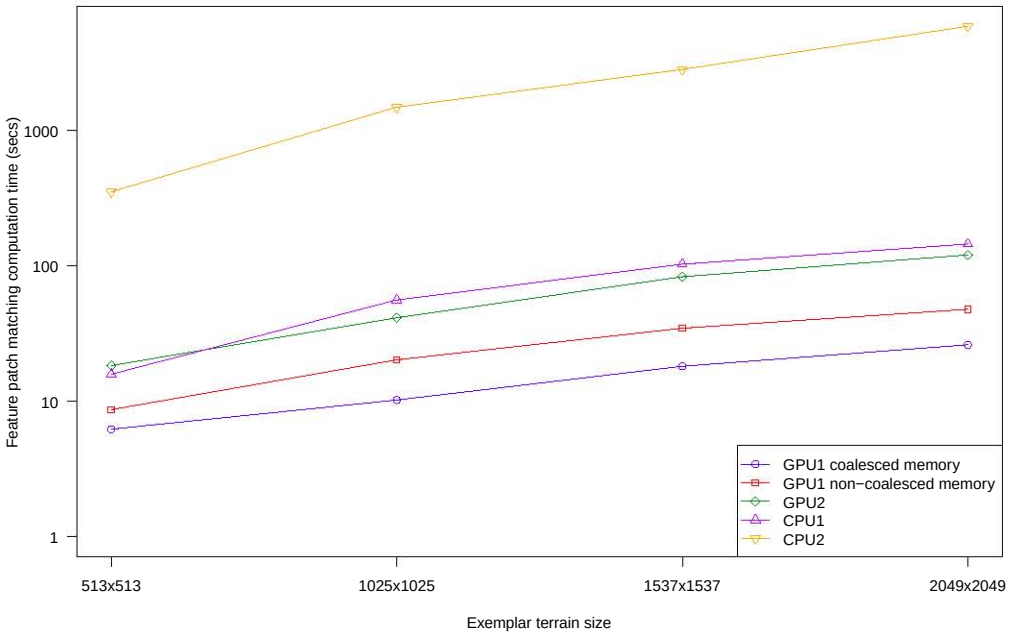


Figure 4.11: Performance analysis of feature patch matching. Execution times were obtained by adding up the time taken for matching 257 80×80patches containing target features.

6 times faster than CPU1 and twice as fast as the parallel algorithm based on texture memory (GPU2). GPU2 does not store patches in memory before the synthesis but directly access the exemplar in texture memory on a need-to basis. Thus, it does not have the memory issues of GPU1 C. Choosing one versus another is a tradeoff between computational cost and memory.

The GPU implementations presented so far can be improved in several ways. For large exemplars, terrain synthesis still takes more than a minute and therefore, slows down the design process of a landscape. The use of multiple GPUs will help mitigate memory issues and increase the speed-up by offering more memory space and cores. Furthermore, other candidate characteristics such as control points and noise variances can be placed in shared memory to take advantage of its fast memory transactions.

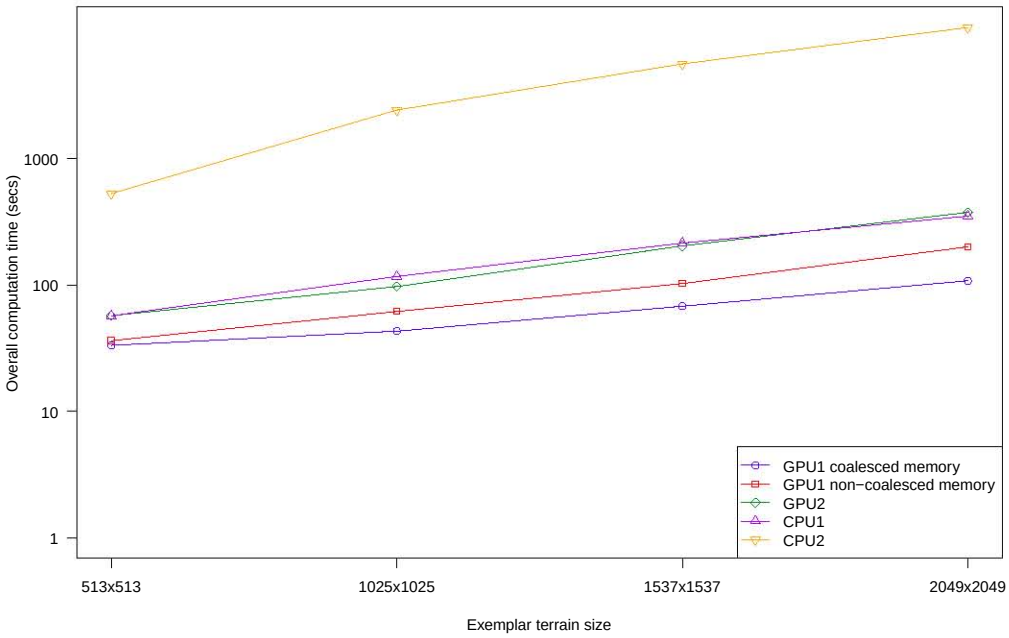


Figure 4.12: Performance analysis of the patch-based synthesis, with 25% of patches placed during feature patch matching

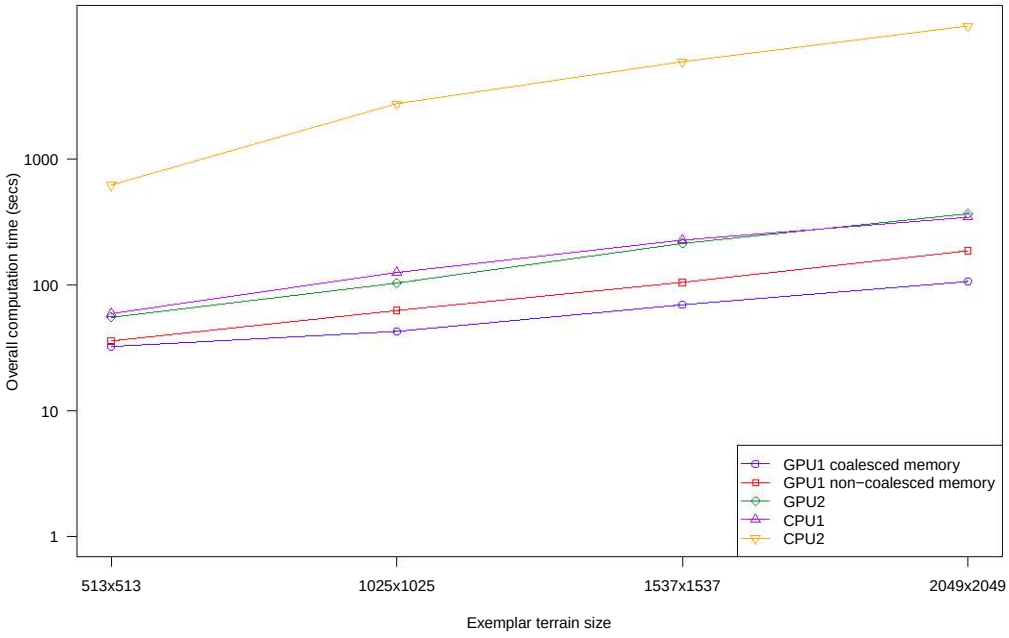


Figure 4.13: Performance analysis of the patch-based synthesis, with 50% of patches placed during feature patch matching

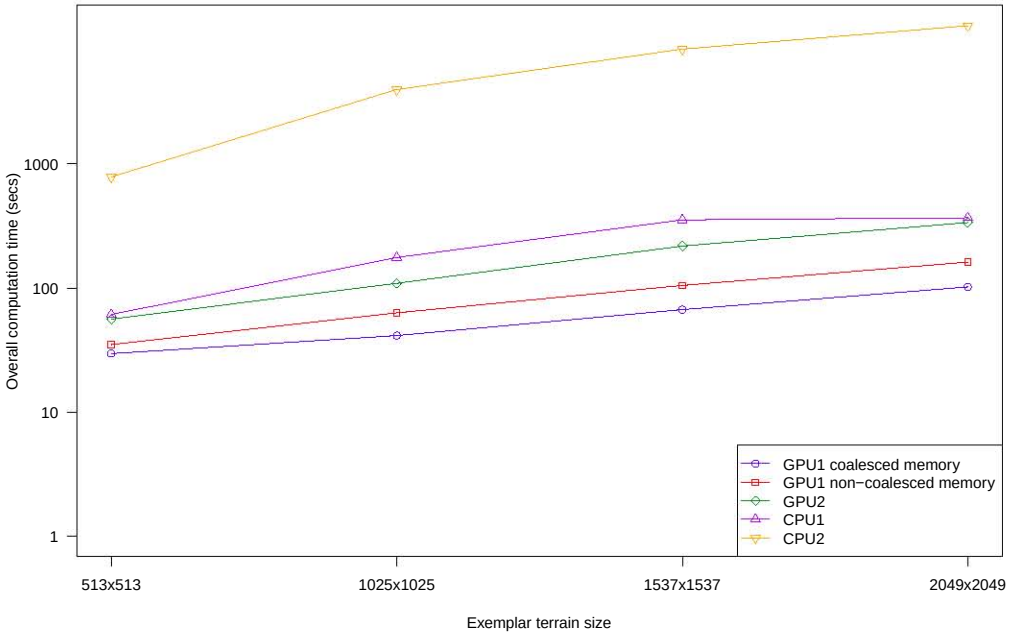


Figure 4.14: Performance analysis of the patch-based synthesis, with 75% of patches placed during feature patch matching

Chapter 5

Experiments and Results

This project has three goals. First, we aim to improve the interactive terrain sketching interface of Gain et al. (2009) by altering the terrain generation technique used, from a noised-based to a texture-based scheme. Secondly, since existing patch merging techniques are not suited to terrains, we seek an improved patch merging technique that overcomes these limitations. Finally, we propose a patch-based texture synthesis framework that improves on the quality of the terrains generated by Zhou et al. (2007). Specifically, we aim to enhance the realism of the synthesised landscapes and match user constraints. This chapter evaluates the success of these three goals.

We start by presenting the design, analysis and results of a user study (Section 5.1). Because most landscape generation systems are designed to produce terrains for environments such as games, movies and virtual training, where user immersion is essential, it is important to consider user opinions in assessing the success of the first two objectives. Then, a visual assessment of the terrains generated by our system is presented, as well as a comparison of our results against the test cases presented by Zhou et al. (2007).

5.1 Experimental design and data analysis

Two experiments were designed to determine the influence of the terrain synthesis framework and patch merging technique on the realism of the generated terrains. Landscape realism was evaluated by asking participants if the terrains (*stimuli*) presented to them were similar to real life landscapes. The purpose of these experiments was the investigation of the following areas of interest:

1. To determine the quality of the terrains generated by the patch-based terrain synthesis proposed in Chapter 3, when compared with both real landscapes and landscapes generated by the curve-based multi-resolution surface deformation used by Gain et al. (2009). It was expected that the terrains generated by our system would be more realistic than those produced by Gain et al. (2009) because they are derived from real landscapes. We also expected they would be less realistic than the real landscapes they are derived from.

2. To determine the success of our patch merging technique in removing boundary artifacts in terrains generated per patch, when compared with alternatives like the Poisson seam removal technique in Zhou et al. (2007) and the Shepard interpolation discussed in Chapter

3. It was expected that terrains produced using our patch merging technique would have the least artifacts, if any.

This section outlines the methodology used for the user study, presents an analysis of the data collected and an interpretation of the analysis results.

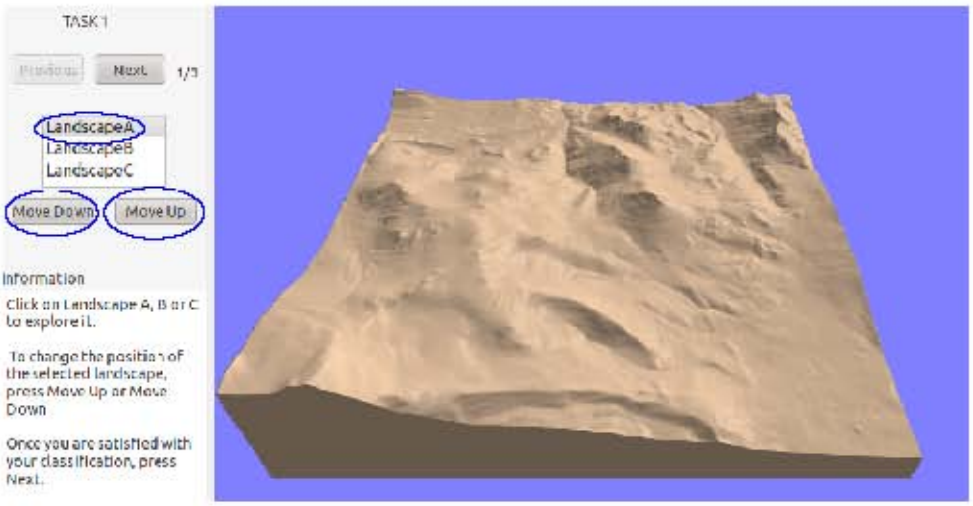
5.1.1 Methods

We used a *within-subjects design* in which every participant performed the same tasks on the same set of terrains during a single session. Greenwald (1976) extensively discusses the advantages and drawbacks of this design. A within-subjects design multiplies the number of observations and thus increases the statistical power of the study and the probability that significant results will appear if they are any. Furthermore, individual differences do not interfere with differences amongst conditions, which reduces the error variance. However, a "carry-over effect" can influence the results of the user study. Participation in one condition can affect subsequent performance due to fatigue or practice (referred to as the "learning effect"). This disadvantage is addressed by limiting the duration of each session to 50 minutes and randomly presenting the stimuli to the participants.

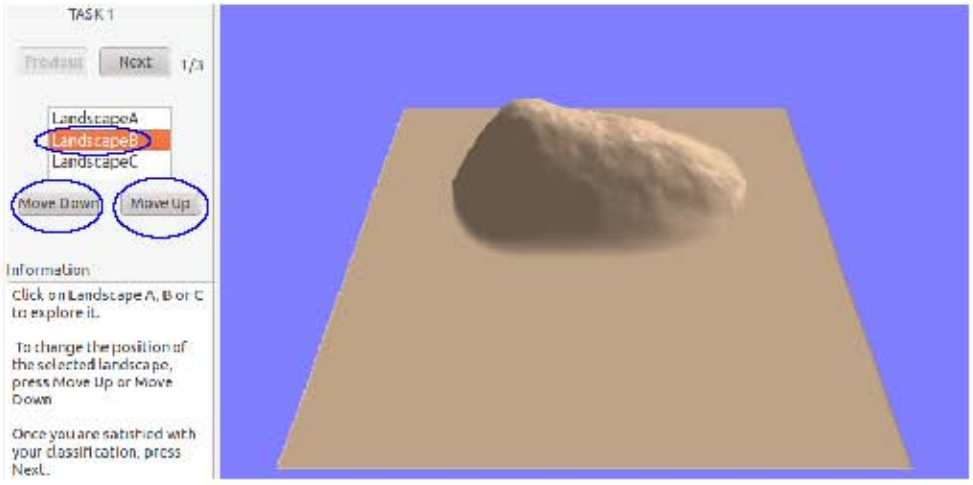
The study was advertised to University of Cape Town students via departmental mailing lists, notifications on Facebook and word of mouth. Volunteers then signed up for individual time slots. Normally, the choice of sample size for a statistical experiment is determined by the known variance of the population but this information was not available. Thus, twenty (20) participants were selected so as to increase the statistical power of the study. All participants were between 20 and 48 years old, undergraduate or graduate students and came from the Science, Commerce, Humanities and Electrical engineering faculties. 65% of the participants had at least an average level of experience with playing video games containing outdoor environments.

Every session was conducted in an experiment room with one participant at a time. A QT/OpenGL-based terrain viewer was deployed on a laptop with a 2.00GHz Intel CPU, 2 GB RAM, a 15" LCD monitor and a wheel mouse. The terrain viewer enabled participants to rotate, translate, zoom in or out and mark or unmark regions on the terrain displayed on the screen. The two experiments were computerized in order to prevent interruptions between tasks, that will distract participants. Task completion was confirmed by pressing a button and the system automatically moved to the next task. A quick tutorial on using the terrain viewer was provided and explanations on specific tasks were displayed on the screen at all times.

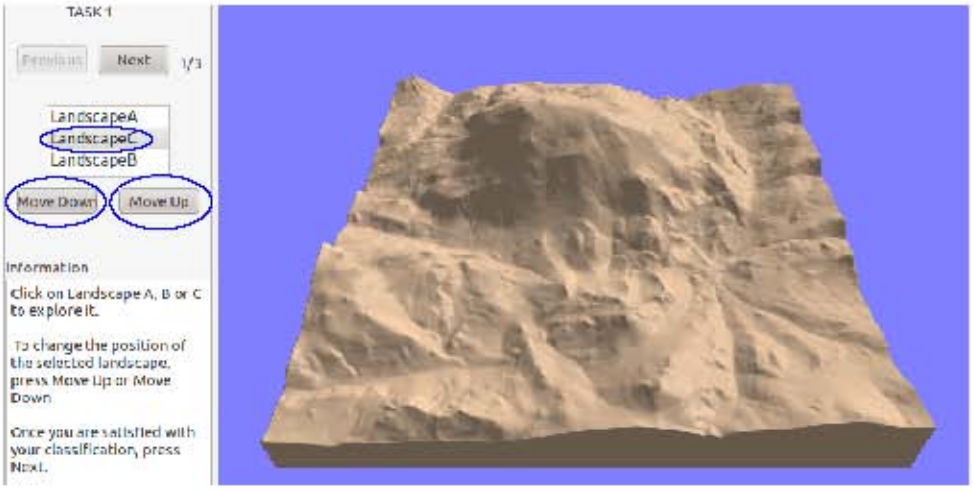
The user study consisted of twenty sessions spread out over three days at a rate of 6 to 7 sessions per day. During a session, the participant started by reading the tutorial and getting comfortable with 3D navigation in the terrain viewer. The interface was simple and intuitive (drag the mouse to translate, move the mouse to rotate, scroll for zooming and double click to mark a region or unmark a previously marked region). Once the participant was familiar with the test environment, the experiment began.



(a) Real landscape

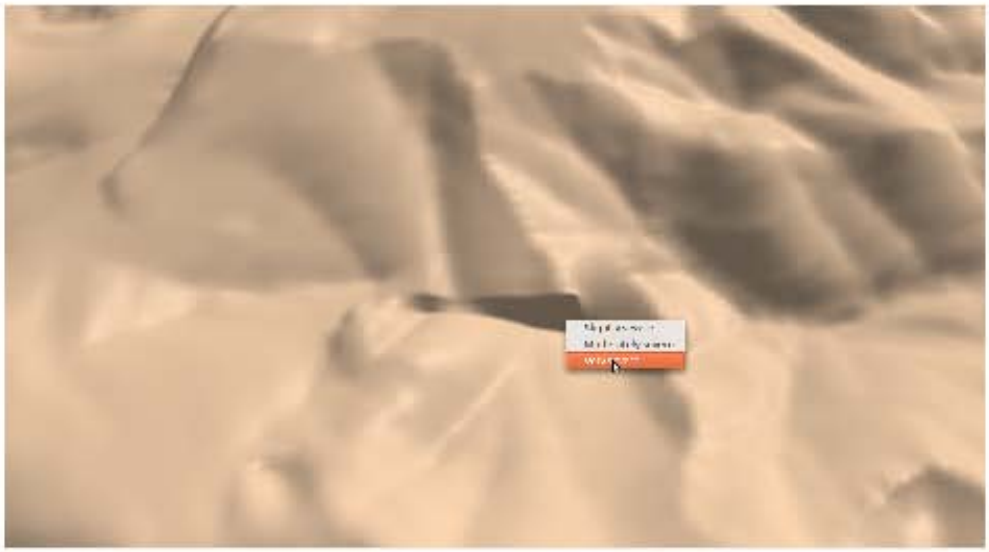


(b) Deformed landscape

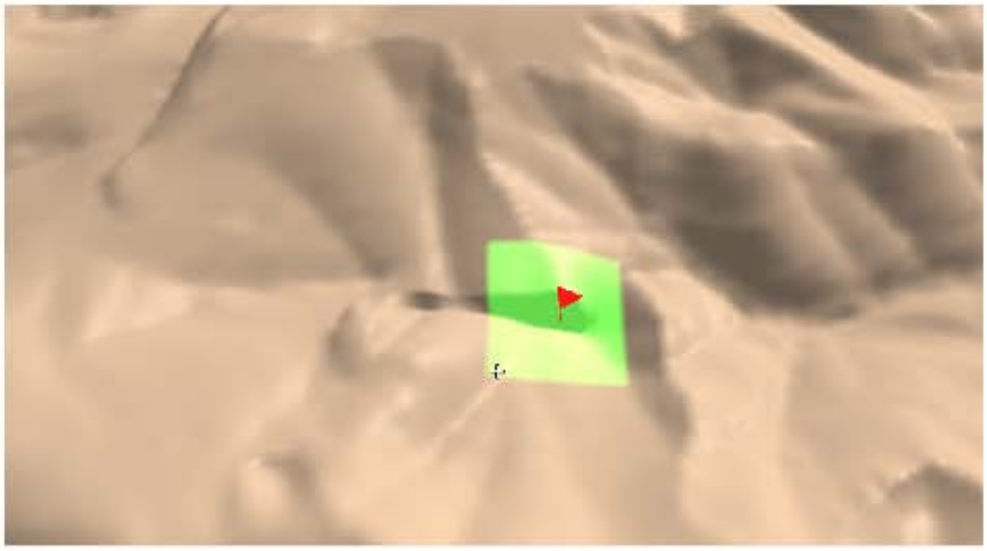


(c) Landscape generated by our system

Figure 5.1: Experiment 1 Procedure. The participant clicks on A, B or C to display the associated terrain, unaware of its source, and rates the realism of the landscape displayed by pressing the "Move Down" or "Move Up" buttons.



(a)



(b)

Figure 5.2: Experiment 2 Procedure. (a) Double-clicking on an artifact of the terrain pop-ups a selection menu that allows the participant to specify the severity level of the artifact. (b) The participant receives feedback from the system in the form of a red flag placed at the marked position and a square area around the flag is highlighted in green. The value of the green component used depends on the severity level selected: the higher the severity, the larger the value of the green component.

5.1.2 Experiment 1 – Comparing the realism of terrains from Gain et al. (2009) and the proposed framework against real-life landscapes.

This experiment asked users to compare three types of terrains randomly presented to them. Each participant was asked to performed three subtasks each consisting of ranking a set of three terrains or stimuli. In the rest of this chapter, we refer to a real landscape as a terrain of type T_{real}, a terrain from our system is of type T_{sys}, and a terrain deformed in the original sketching interface

(Gain et al., 2009) is of type T_{def}. The hypotheses tested in Experiment 1 are the following:

- A T_{real} terrain sourced by scanning real landscapes is superior to T_{sys} or T_{def} procedurally generated terrains.
- A T_{sys} terrain, created from patches of real terrain appears more realistic than a T_{def} landscape.

For each subtask, a T_{real} landscape was obtained from a database of real-life heightmaps (US Geological Survey, 2011) and the T_{def} terrain was obtained by drawing curves in the sketching interface to deform a flat terrain. The T_{sys} landscape was generated by our patch-based terrain synthesis framework using the same curves as the T_{def} heightfield and the T_{real} heightfield as the exemplar. The terrains were presented in a random order and participants were asked to rank them in order of realism, from the landscape that looked the most realistic to the least realistic. The three heightmaps were presented as A, B and C and the task was to rearrange them by selecting a letter and moving it up or down. Clicking on a letter changed the terrain currently displayed on screen to the terrain corresponding to the letter selected, with all the transformations such as scale, translation and rotation preserved. Figure 5.1 shows the interface used for the experiment and the different steps of the first subtask. Once satisfied with the rearrangement, the participant validated it by pressing the "Next" button, and the subsequent subtask was automatically presented.

Every validated arrangement was automatically saved to an external file created when the user started the experiment. The data collected from the first experiment was a list of 60 (20×3) ratings of terrains of type T_{real}, T_{sys} and T_{def}.

5.1.3 Experiment 2 – Comparing the frequency of artifacts in patch-based terrains generated using Poisson seam removal, Shepard Interpolation and our proposed merging method.

We denote landscapes generated with our proposed merging technique by the term M_{New}, terrains synthesised using Shepard Interpolation as $M_{Shepard}$ and those generated with Poisson seam removal are of type $M_{Poisson}$. The hypotheses tested in Experiment 2 are:

- M_{New} terrains have the least number of artifacts compared to $M_{Shepard}$ and $M_{Poisson}$ terrains.
- $M_{Poisson}$ landscapes have the highest rate and the most severe artifacts.

During this experiment, 18 terrains were presented to the participant, consisting of 6 sets of M_{New}, $M_{Shepard}$ and $M_{Poisson}$ heightmaps. Terrains within a set were generated using the same input data (the example terrain and the target terrain) and only differed by the patch merging process used. None were deformed after patch-based texture synthesis. All 18 terrains were presented in a random order and participants were asked to mark regions they believed had artifacts and specify the severity of each artifact as slightly severe, moderately severe or very severe. Marking a region was achieved by carefully navigating over the terrain and double-clicking on

found artifacts. Figure 5.2 illustrates the marking of an artifact and the visual feedback of the experimental terrain viewer. A flag was placed at the centre of every selected artifact. Mistakenly marked regions could be unmarked by double clicking on the associated flag. Every participant validated the artifacts for a specific terrain by pressing the Next button. The system saved the marked artifact positions and severity to an external file and moved to the next terrain. The number of artifacts per terrain and the average severity were analysed in order to compare the quality of the three different patch merging methods.

5.1.4 Data analysis and results

Experiment 1 – Comparing the realism of terrains from (Gain et al., 2009) and the proposed framework against real-life landscapes

Figure 5.3 shows the means plot of the rankings of three types of terrains investigated: T_{real}, T_{sys} and T_{def}. T_{def} terrains were classified almost universally as the least realistic. The plot also suggests that participants thought that T_{real} terrains were as realistic as T_{sys} terrain. We use statistical tests to confirm there is a significant difference and to determine which terrain types (or groups) are the cause. Analysis of Variance (ANOVA) is the standard test generally used to compare multiple groups. The test assumes that the population follows a normal distribution and that all the groups have the same variance. We performed the Shapiro Wilk test on each group to confirm their normality (Shapiro and Wilk, 1965). The results revealed that none of the groups follow a normal distribution (p-value<0.001). Thus, we use the Friedman test, a non parametric statistical test equivalent to a one-way repeated measures analysis of variance by ranks (Friedman, 1937). The test rejects the hypothesis that there is no significant difference between the three groups (p-value<0.001).

Turkey Honestly Significant Difference test (HSD) is used to compare each pair of terrain type. At a 5% level of significance, real terrains and landscapes synthesised by our system are more realistic than deformed terrains (p-value<0.001). This confirms our expectation that terrains generated with the proposed framework are superior to terrain synthesized by the original sketching interface in terms of realism. However there is no significant evidence that real terrains are superior to our synthesis (p-value=0.981). We, therefore, conclude that the realism of the terrains generated by the framework is not dissimilar from that of real-life landscapes.

Experiment 2 – Comparing the frequency of artifacts in patch-based terrains generated using Poisson seam removal, Shepard Interpolation and our proposed merging method

We analyse the data collected in the second experiment by investigating the number of artifacts and the average artifact severity specified by the participants. Figure 5.4 and 5.5 shows the means plot of the artifacts frequency and severity for each terrain type. Both plots suggest that $M_{Poisson}$ terrains have the highest artifacts in terms of both frequency and severity.

First, we investigate the difference in artifact frequencies. Outliers (artifacts frequency higher than 20 and cases where the number of artifacts for each group is zero), which constitute 20% of the collected data, are removed from the collected sample so that the results of the statistical analysis are unbiased. The normality test rejects the hypothesis that this data follows a normal distribution (p-value<0.001). In fact, the data follows a negative binomial distribution and cannot be normalized using standard transformation due to the presence of many zeros. The non para-

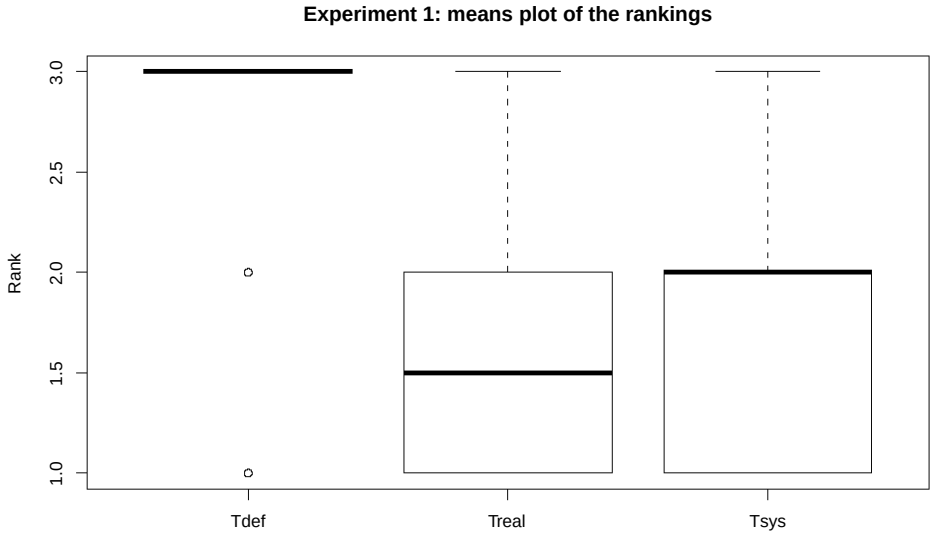


Figure 5.3: Box plot of Experiment 1. The plot depicts the minimum, lower quartile, median, upper quartile, and the maximum of the rankings of each group.

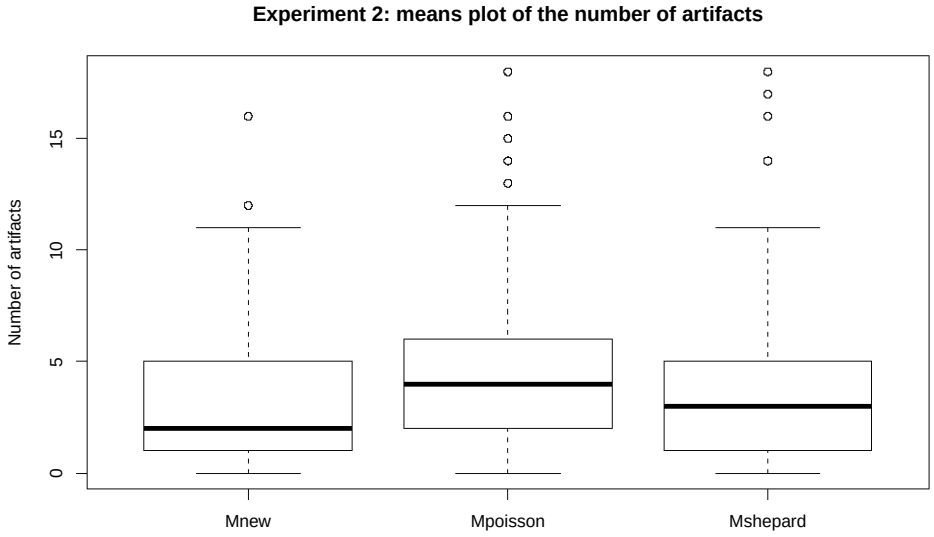


Figure 5.4: Box plot of Experiment 2. The plot depicts the minimum, lower quartile, median, upper quartile, and the maximum of artifact frequencies in each group.

metric Friedman test confirms that there are significant differences in the artifact frequencies of M_{new},$M_{Shepard}$ and $M_{Poisson}$. Post-hoc analysis reveals that $M_{Poisson}$ terrains have a higher artifact frequency than both M_{New} and $M_{Shepard}$ terrain. However, there is no significant evidence on the relationship between $M_{Shepard}$ and M_{New} terrains. Similarly, an analysis of the artifact severity level shows that there is significant difference in the severity level, due to the fact that

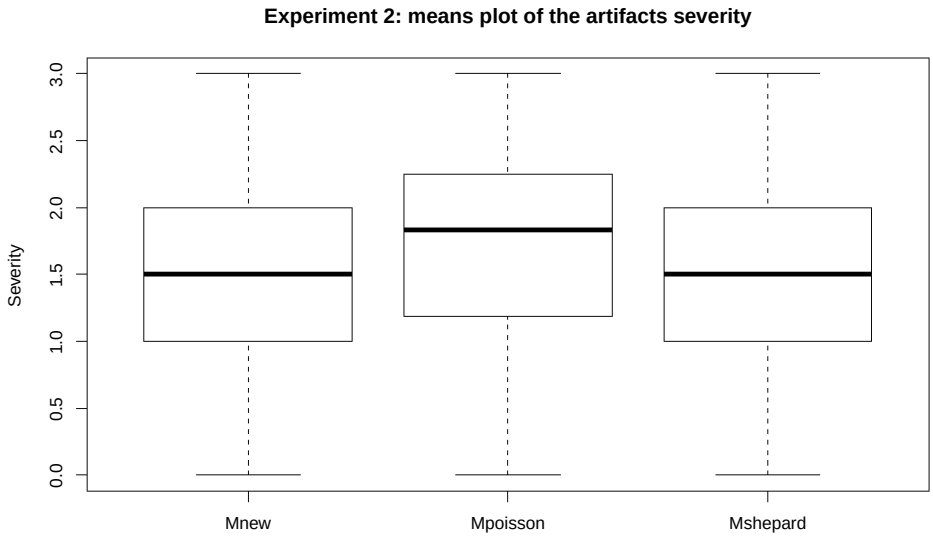


Figure 5.5: Box plot of Experiment 2. The plot depicts the minimum, lower quartile, median, upper quartile, and the maximum of average artifact severity in each group.

$M_{Poisson}$ terrain artifacts are worse than the artifacts in the other two terrain types. There is no statistically significant information to suggest that the severity levels of $M_{Shepard}$ and M_{New} terrains are different. Scaling the artifacts frequency by the severity produces similar results.

The non-normality of the data collected in the two experiments arises from the fact that, in both situations, the data tends towards a natural limit (0 for artifacts frequencies and severities). Using non parametric test such as the Friedman test and HSD over ANOVA and paired t-tests lessens the statistical power of the data analysis but does not require that the data follows a specific distribution.

An analysis of which artifacts the participants most agreed on reveals that their selection was dependent on the overall quality of the terrain. Terrains of type $M_{Poisson}$ whose artifacts were mostly straight lines on some areas of the landscape were more noticeable and participants easily agreed on those. As shown in Figure 5.7 and 5.6, a higher percentage of subjects agreed they were artifacts. Artifacts in $M_{Shepard}$ terrains on the other hand were spread on the landscape and participants opinions diverged over whether these were indeed artifacts or simply characteristics of the terrain topography. This is caused by the fact that artifacts produced by Shepard Interpolation occur every time a new patch is placed and can be mistaken as part of the terrain topography. Terrains of type M_{New} had the least average number of artifacts but whenever an artifact was visible, due to the bad matching, it was clearly noticeable.

Bad matching occurs when a selected patch differs significantly from already placed pixels on the overlap region. For example, a combination of a high overlapping area cost and low scores in other matching criteria may place a patch with large height values next to an area with low elevation values. These situations are very rare and but when they do occur, the large discrepancies in the pixel values cannot be completely removed by patch merging. Furthermore, the newly

placed patch may only share a corner or boundary pixels with already placed patches. Then, the overlapping region is too small for an efficient graph cut and if the differences in height values in the overlap are large, the Poisson equation solver will introduce artifacts (Figure 5.7(b)).

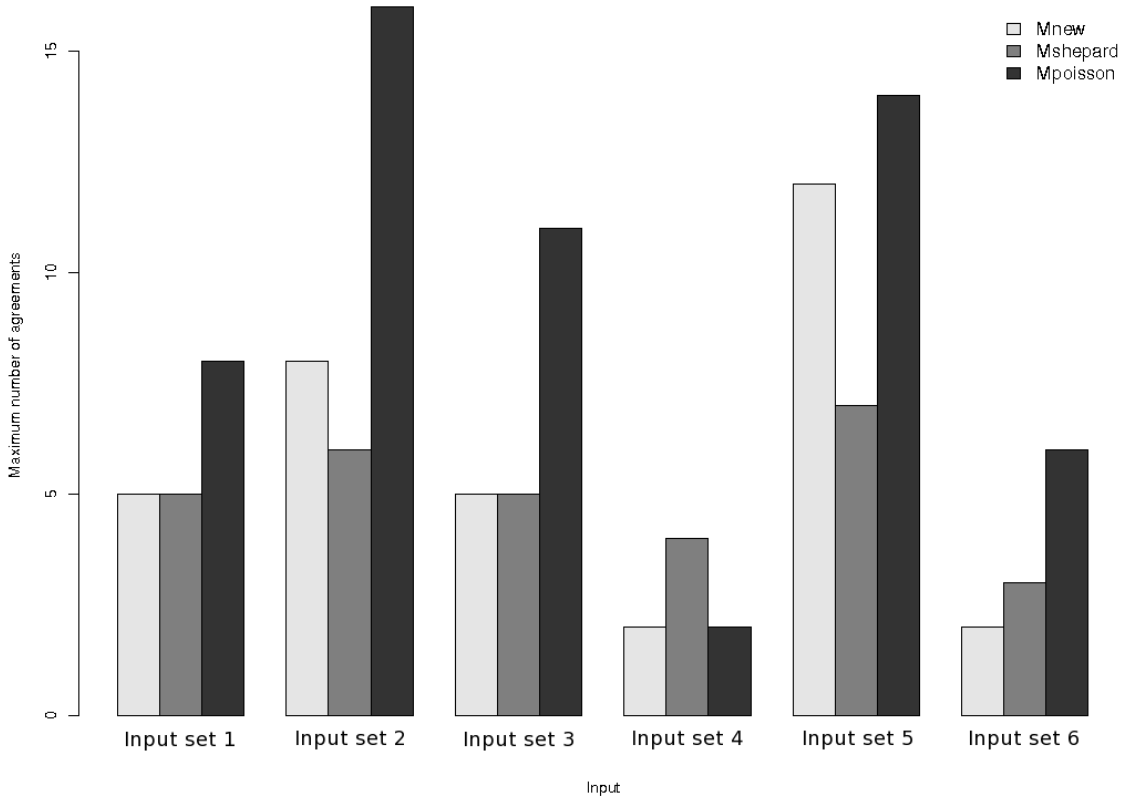
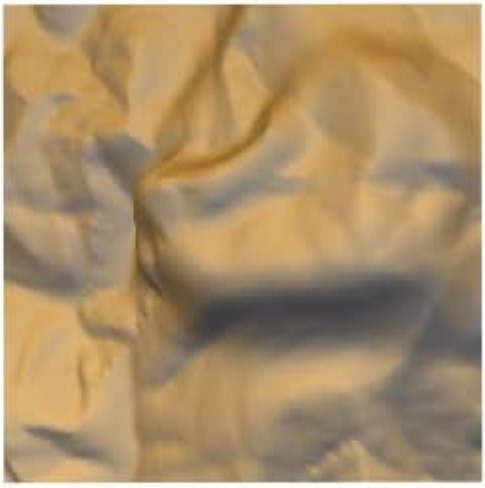


Figure 5.6: Barplot of participant agreements on selected artifacts. For each of the 18 terrains displayed to participants, the plot shows the maximum number of agreements on artifacts. User mostly agreed on artifacts in $M_{Poisson}$ terrains, followed by M_{new} terrains and then $M_{Shepard}$ terrains.

(a) Type M_{New}, selected by 60%

(b) Type M_{New}, selected by 40%

(c) Type $M_{Shepard}$, selected by 30%



(d) Type $M_{Shepard}$, selected by 35%

(e) Type $M_{Poisson}$, selected by 80%

(f) Type $M_{Poisson}$, selected by 70%

Figure 5.7: Examples of artifacts selected by the user study participants

5.2 Visual assessment

5.2.1 Comparison with Zhou et al. (2007)'s algorithm

We compare several landscapes generated by our framework against the results of Zhou et al. (2007)'s algorithm, using the same exemplar and target terrains. Figure 5.8 and 5.9 present a terrain generated with our framework, using the Grand Canyon and Mount Jackson, respectively, as the exemplar and the half-life symbol as the target. Both synthesis results are a better match of the targets compared to the results of Zhou et al. (2007)'s algorithms. Our framework was able to match the hooks of the half-life symbol and the circle around it so that they are well-defined in the synthesised heightmaps.

Figure 5.10 showcases the effects of extending linear features placed by feature matching during the non-feature matching process. An extra valley is added even though it is not in the target heightmap. The best-first filling order used in non-feature patch matching is designed to propagate structural information so that regions around large-scale features do not appear unrealistically flat. But in some few situations, the filling approach adds large features that were not specified by the user.

Figure 5.11 shows a failed case appearing on Zhou et al. (2007)'s website. The output terrain is generated from patches of Cape Girardeau (KY, USA). It fails to match the target features due to poor feature matching. The output of our framework successfully matches the sketch map and regions where no features were specified have a lower level of detail. Because the feature dissimilarity cost takes into account the height values along the the target features as well as the height profiles along paths perpendicular to those features, our system offers better feature patch matching. In Figure 5.11(b), the exemplar is dominated by the presence of multiple branches and a large number of small features. Only considering the height values along paths perpendicular to these features, as is the case in Zhou et al. (2007), is not enough to select patches that best match the target.

5.2.2 Terrain results with user sketched-curves

When combined with the terrain sketching interface, our framework supports multiple sketched curves and deformations. Figure 5.12 shows a terrain deformed by four sets of constraint curves before a patch-based synthesis is applied to enhance realism. Instead of a smooth terrain, the final landscape is less homogeneous and exhibits erosion effects such as drainage patterns.

Figure 5.13 shows another sketched patch-based terrain derived from a mostly flat terrain. Even though the patch-based synthesis produces a flat terrain, the deformation process applied to the terrain afterwards still delivers a landscape that fits the user requirement. This introduces the same realism issues that the original sketching interface has: the output terrain is smooth in the deformation area, a phenomena that is very rare in real landscapes. Better procedurally generated landscapes are produced when the exemplar terrain has the desired features in the target.

Combining a sketching interface with an example-based method allows users to benefit from an intuitive and precise level of control (adding features such as volcanoes or removing side of a mountain) along with higher quality terrains. It supports both users with no experience in

designing landscape terrain models and professional 3D terrain modellers, in the sense that the level of control is completely dependent of the user.

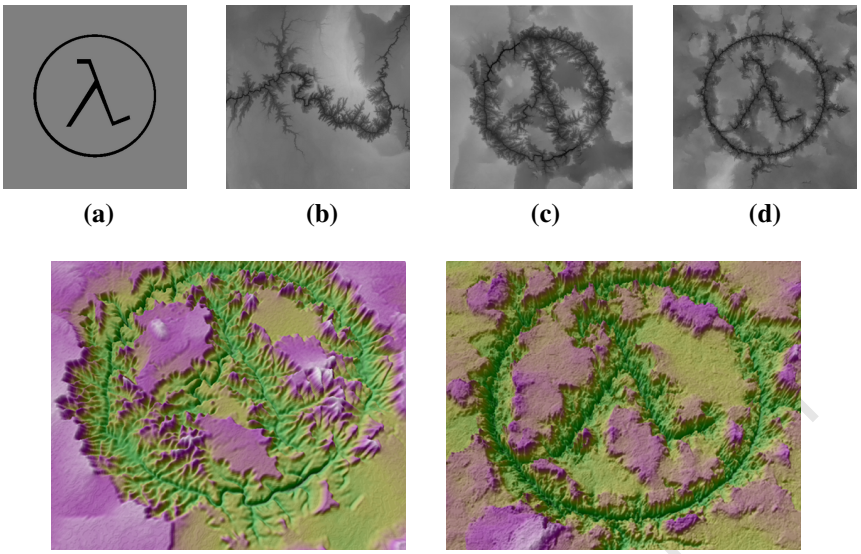


Figure 5.8: Grand Canyon synthesis result. (a) Target terrain (2000×2000). (b) Grand Canyon exemplar (2040×1380). (c) Terrain generated by Zhou et al. (2007)'s algorithm. (d) Landscape generated by our framework in 4.8 minutes. (e) A 3D rendering of the Zhou et al. (2007)'s output. (f) A 3D rendering of our framework's output.

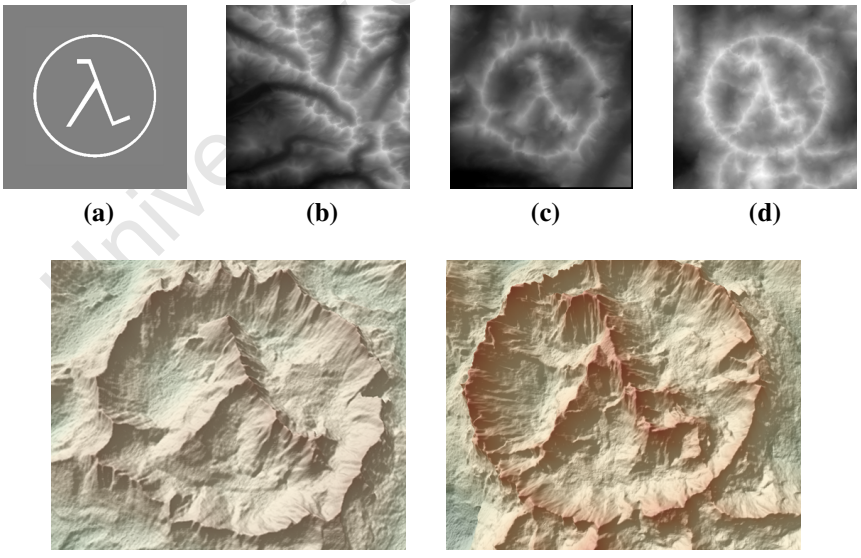


Figure 5.9: Mount Jackson synthesis result. (a) Target terrain (1000×1000). (b) Mount Jackson exemplar (1620×1620). (c) Terrain generated by Zhou et al. (2007)'s algorithm. (d) Landscape generated by our framework in 1 minute. (e) A 3D rendering of the Zhou et al. (2007)'s output. (f) A 3D rendering of our framework's output.

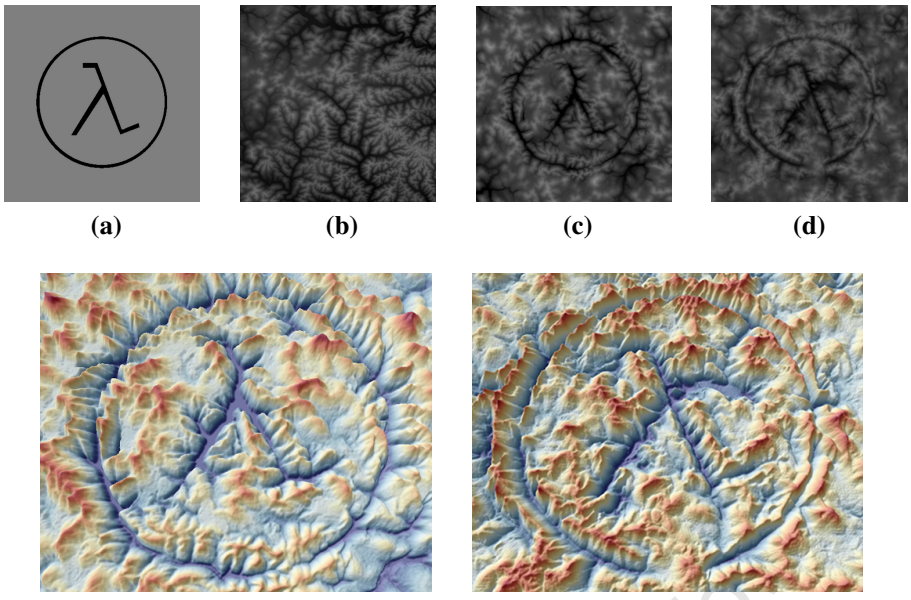


Figure 5.10: Mount Vernon synthesis result. (a) Target terrain (1000×1000). (b) Mount Vernon exemplar (1200×1200). (c) Terrain generated by Zhou et al. (2007)'s algorithm. (d) Landscape generated by our framework in 1 minute. (e) A 3D rendering of the Zhou et al. (2007)'s output. (f) A 3D rendering of our framework's output.

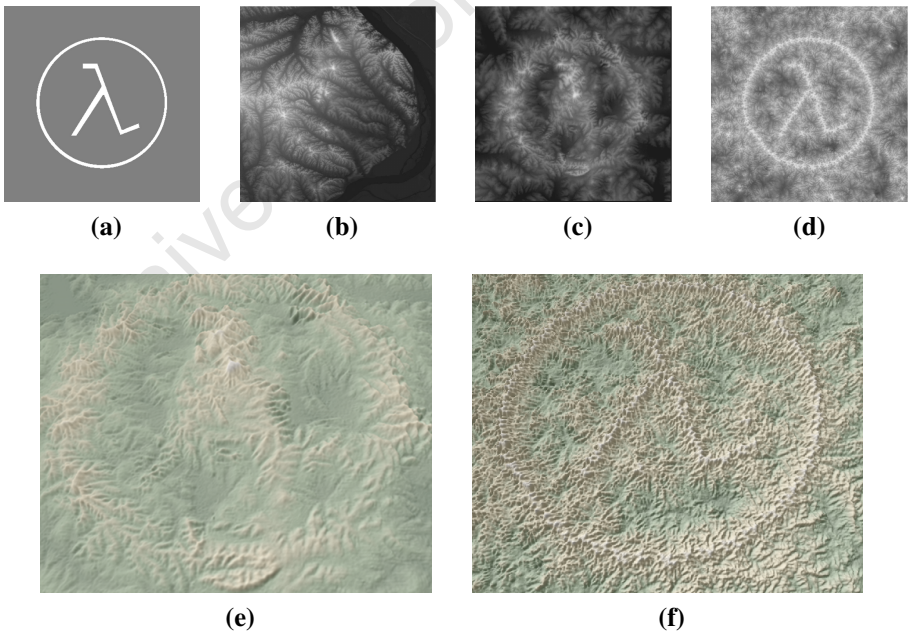
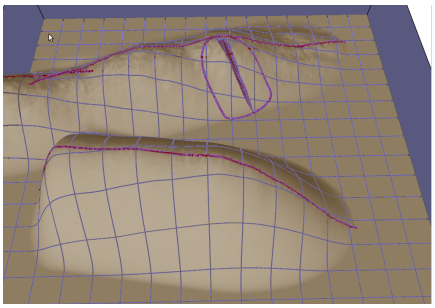
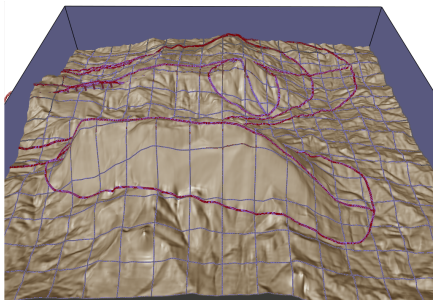


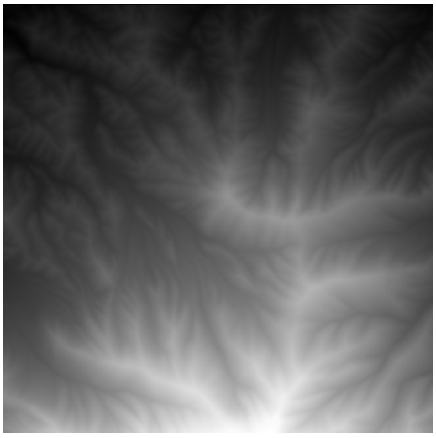
Figure 5.11: Cape Giradeau synthesis result. (a) Target terrain (2000×2000). (b) Cape Girardeau exemplar (1200×1200). (c) Failed result generated by Zhou et al. (2007)'s algorithm. (d) Landscape generated by our framework in 5.6 minutes. (e) A 3D rendering of the Zhou et al. (2007)'s output. (f) A 3D rendering of our framework's output.



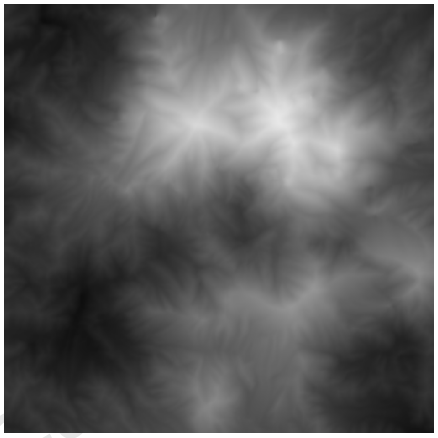
(a) User sketched curves



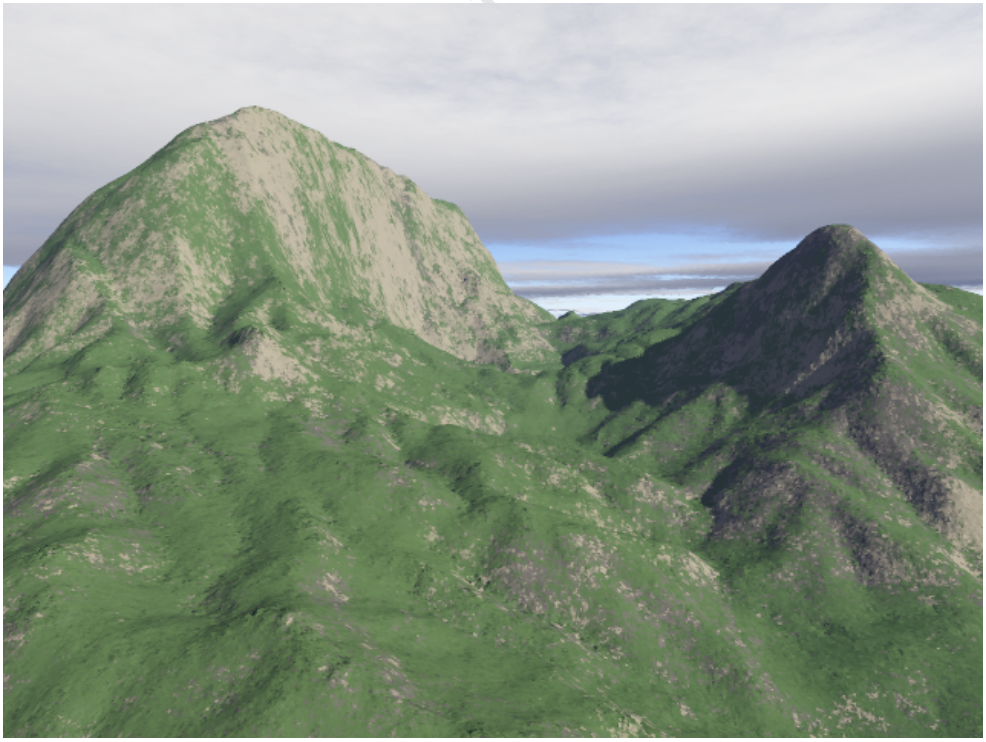
(b) Final result



(c) Exemplar (512×512)

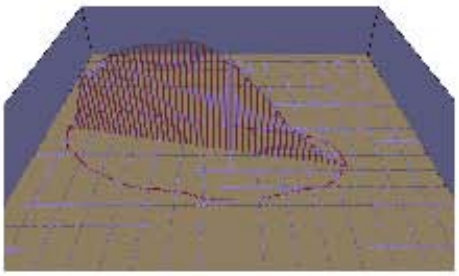


(d) Patch-based synthesis

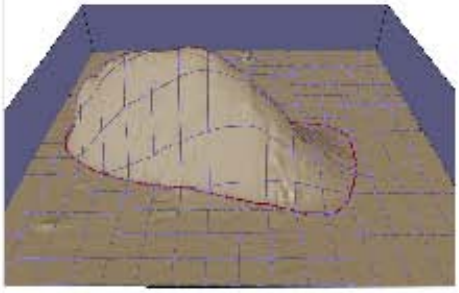


(e) Rendered output

Figure 5.12: Multiple sketched-curves. Both target and example terrains are $512x512$. Generation time: 20 seconds.



(a) User sketched curves



(b) Final result

(c) Exemplar (512×512)

(d) Patch-based synthesis (512×512)

(e) Rendered output

Figure 5.13: Results with a mostly flat exemplar. Both target and example terrains are $512x512$. Generation time: 18 seconds.

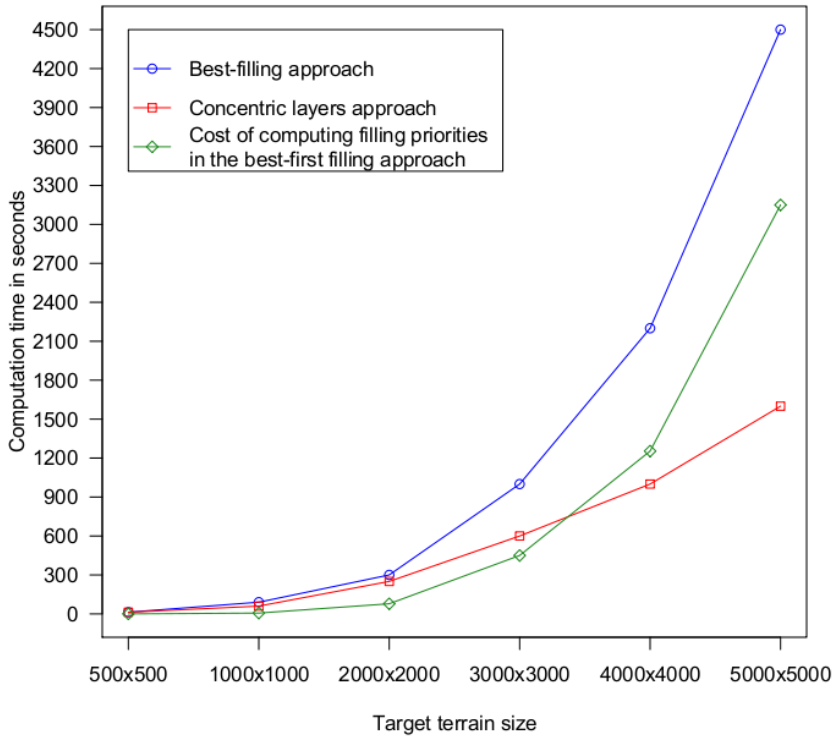


Figure 5.14: Performance comparison of the best-first filling approach and the concentric layers filling method. (a) Percentage of the overall computational time spent on determining where the next patch will be placed. (b) Overall synthesis time.

5.2.3 Influence of patch filling order

The patch filling order used in the non-feature matching phase of the proposed texture synthesis framework propagates the linear structures in the patches placed during feature matching. However, the cost of constantly updating the boundary of the unknown region and computing the filling priorities of the pixels on the boundary quickly becomes computationally expensive as the size of the target terrain increases. Figure 5.14 illustrates the amount of time spent on computing and sorting filling priorities. The percentage of time spent on determining where the next patch should be placed increases with the size of the target terrain and can reach up to 70% of the overall computation time.

An alternative solution to a feature-based filling order is to place patches in concentric layers starting from the already filled pixels. This technique is referred to as concentric layer filling (or the onion peel) and is used in Efros and Leung (1999)'s pixel-based texture synthesis. The disadvantage here is that the portion of known pixels or the edge information around pixels are not taken into account. Thus, the non feature matching phase gains in performance but the surface of the terrain is flatter around the large scale features compared to the best-first filling approach. Figure 5.15 provides an example of two different outputs based on the two different filling meth-

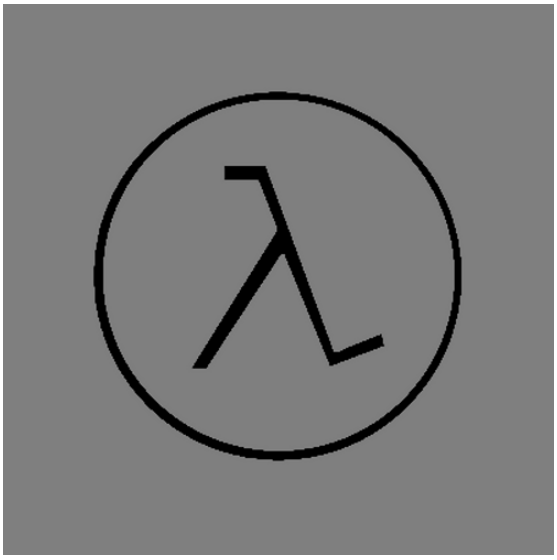
ods. The output based on the best-first filling order has small additional characteristics that are extensions of the features placed during feature patch matching. The synthesised terrain based on the concentric layer filling method is mostly flat around the large scale features. We argue that although flat regions surrounding large scale features are uncommon in real life, it matches the target terrain so that the prominent features in the output are user-specified. The correct filling order is then based on a trade-off between user-conforming output, realism and computational cost. All the terrain output presented so far was generated using the best-first filling order, but, depending on the level of control and detail desired by the user, either of the discussed filling methods can be used.

5.2.4 The effect of user-defined patch and terrain sizes on the synthesised terrain

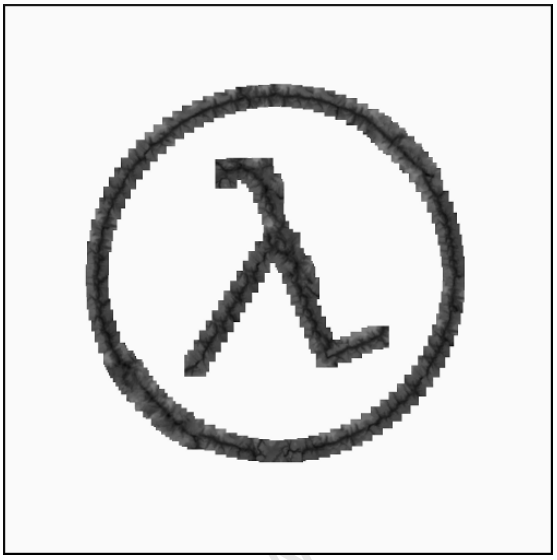
Patch size As with all patch-based texture synthesis methods, patch size significantly influences the quality of the synthesized result. It determines how well the output terrain captures the local features in the exemplar. A very small patch size will produce a landscape with very little structural information. Figure 5.17 illustrates the effects of the patch size on the generated landscape. To address this issue, Liang et al. (2001) algorithmically compute the patch size w with the formula $w = \lambda min(width(T_{exm}), height(T_{exm})$. This does not take into account the nature of the example terrain nor the scale of the features it contains. Texture analysis methods exist that find the scale of local characteristics in the exemplar (Narkdej and Kanongchaiyos, 2009) but these techniques have a very high computational cost that often surpass that of the synthesis. In our proposed framework, the patch size is selected by the user. However, an optimized pre-preprocessing step for determining the optimal block size is an approach worth investigating in future work.

Terrain size With a typical patch-based synthesis, the number of patches repeated in the output scales as the difference in size between the target and exemplar increases. However our technique multiplies the number of candidates and the patch merging makes the repetition less visible if the repeated patches are placed within a close proximity. Figure 5.16 shows an output terrain synthesized using a large 3500×3500 target and a 1025×1025 exemplar. In rare cases, if target features are very similar and close to each other, as it the case in 5.16, then similar or identical patches are selected from the exemplar and placed in the output. This often occurs if a target ridge or valley is represented by a straight line and the features along it are identical. This can be fixed by keeping track of the most recently placed patches and ensuring that none of those patches are placed in the next iteration.

Our framework supports a variety of target terrains and its enhanced patch matching generates landscapes that matches the user constraints as closely as possible. The synthesis illustrated in Figure 5.19 is heavily constrained by a concentric circles that covers a large portion of the target. The generated mountain range exhibits the same concentric circles in the form of ridges copied from a real landscape. A smaller patch size, 60×60 is used to ensure that each circle of ridges appears in the output. A real landscape can also be used as the target terrain, if the user wishes to reconstruct the features in one real terrain with regions from another. Figure5.20 shows a terrain generated by using the Grand Canyon as the target and Mount Vernon as the exemplar. The synthesised heightmap has large scale valley features similar the Grand Canyon



(a) Target (2000×2000)



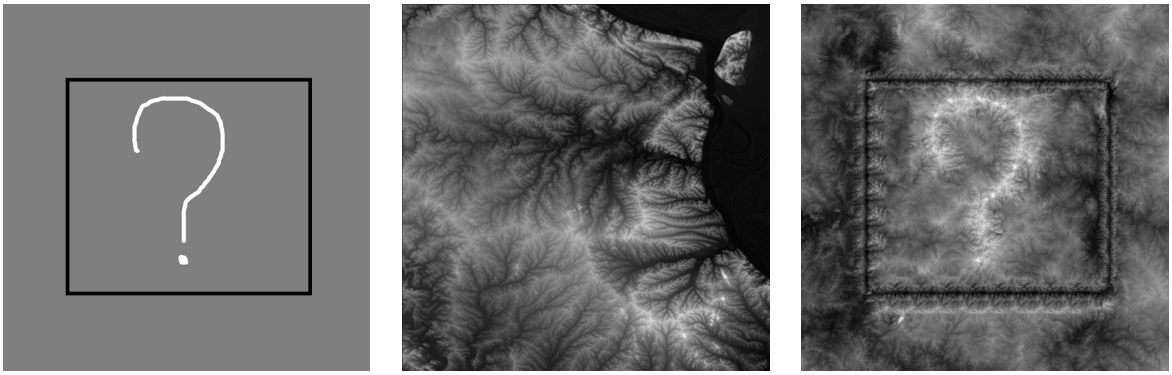
(b) Feature patch matching

(c) Best-first filling

(d) Concentric layer filling

Figure 5.15: Visual assessment of the best-first filling method compared to the concentric layers filling order. (a) The terrain was generated in 211.2 secs. Note how the best-first filling approach add more details by propagating the large scale features. (b) Although the concentric layers filling method speeds-up the synthesis (the terrain was generated in 134.03 secs), the regions around the large-scale valleys are very smooth, which decreases the realism of the output.

valley range and the small characteristics of Mount Vernon. Using a real heightmap as the target works particularly well when the heightmap has one dominant kind of features (ridge or valley).



(a) Target (1000×1000) **(b)** Exemplar 1025×1025 **(c)** Final output

Figure 5.16: Synthesising a terrain with a group of identical features. Features along the top, left and bottom sides of the rectangle, representing the terrain valleys, have similar or identical matches from the exemplar. The patch merging deforms each patch placed in the output according to already filled regions and thus identical patches that are placed far apart from each other are difficult to spot. However, these repetitions are more visible in the feature matching step, due to their locality.

5.3 Limitations

The patch-based terrain synthesis framework presented in this thesis improves on previous texture-based landscape generation methods and becomes more powerful when combined a terrain sketching interface. However, the framework fails to produce a terrain that fits the user constraints in the following cases:

- The features requested by the user are not present in the exemplar and thus, cannot be placed in the output. Areas in the target with features that are not available in the example terrain are treated as regions with no dominant features. If a sketch interface is used for user control, the interface will deform the generated terrain by the patch-based synthesis to fit user constraints. If regions deformed are flat, then they will appear smooth and diminish the realism of the overall landscape.

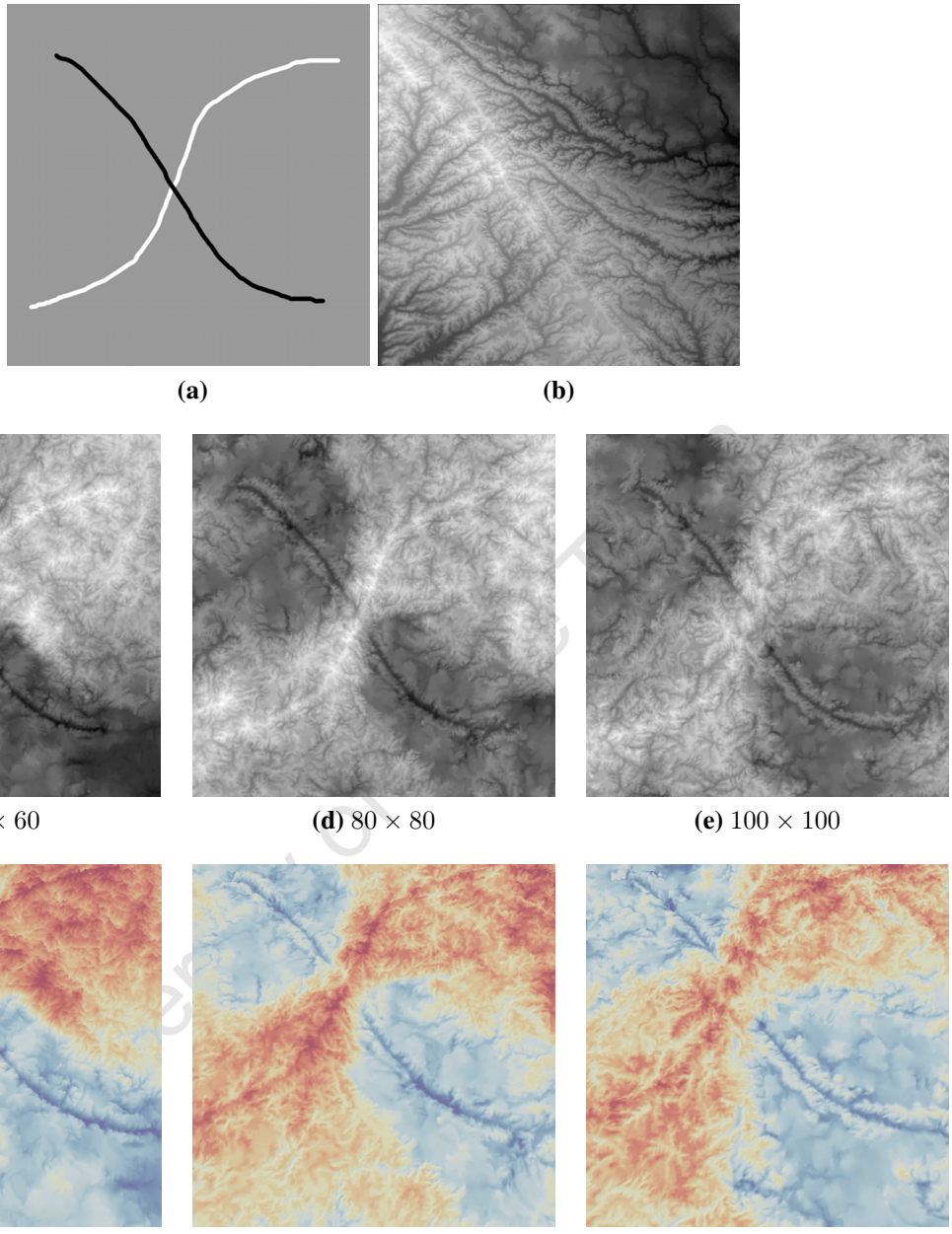
- The patch size is too small and thus introduces very small random features in the output or the patch size is too large and fails to finds good matches from the exemplar that capture the features specified in the target.

- During feature patch matching, similar or identical patches from the exemplar are placed in the output in close proximity, making the repetition noticeable if the features they were matched against are very similar or identical. This occurs when the target contains a long line with identical features (see Figure 5.16).

The shortcomings of our framework are fundamentally related to the input provided by the user. Although little can be done about real landscapes that do not have a specific feature (ridge or valley) desired by the user, an automatic search of the optimal patch size can be used to address the limitations related to patch size in future extensions of the patch-based synthesis.

5.4 Summary

We have established that our framework combines and improves on Zhou et al. (2007) and Gain et al. (2009). Furthermore, we have demonstrated that our novel gradient-based patch merging technique produces better results than Poisson seam removal. No statistically significant evidence, based on the experiment design presented in Section 5.1, was found on the relationship between the proposed merging method and the Shepard Interpolation approach. Artifacts introduced by Shepard Interpolation are spread over the terrain and during the user study, they were often mistaken for part of the terrain topography. Finally, we presented examples of terrains generated by the proposed framework and a visual assessment of our results. It was established that the proposed framework is able to generate more realistic landscapes that better fit the user requirements. We discussed the shortcomings that parameters such as patch size and the size and nature of the input terrains can have on the end result. These shortcomings are inherent in patch-based texture synthesis approaches. Unfortunately, similarly to most patch-based methods, the user addresses these shortcomings by choosing appropriate parameters values. Chapter 6 presents a few ideas on how some of these issues can be directly addressed by the framework.

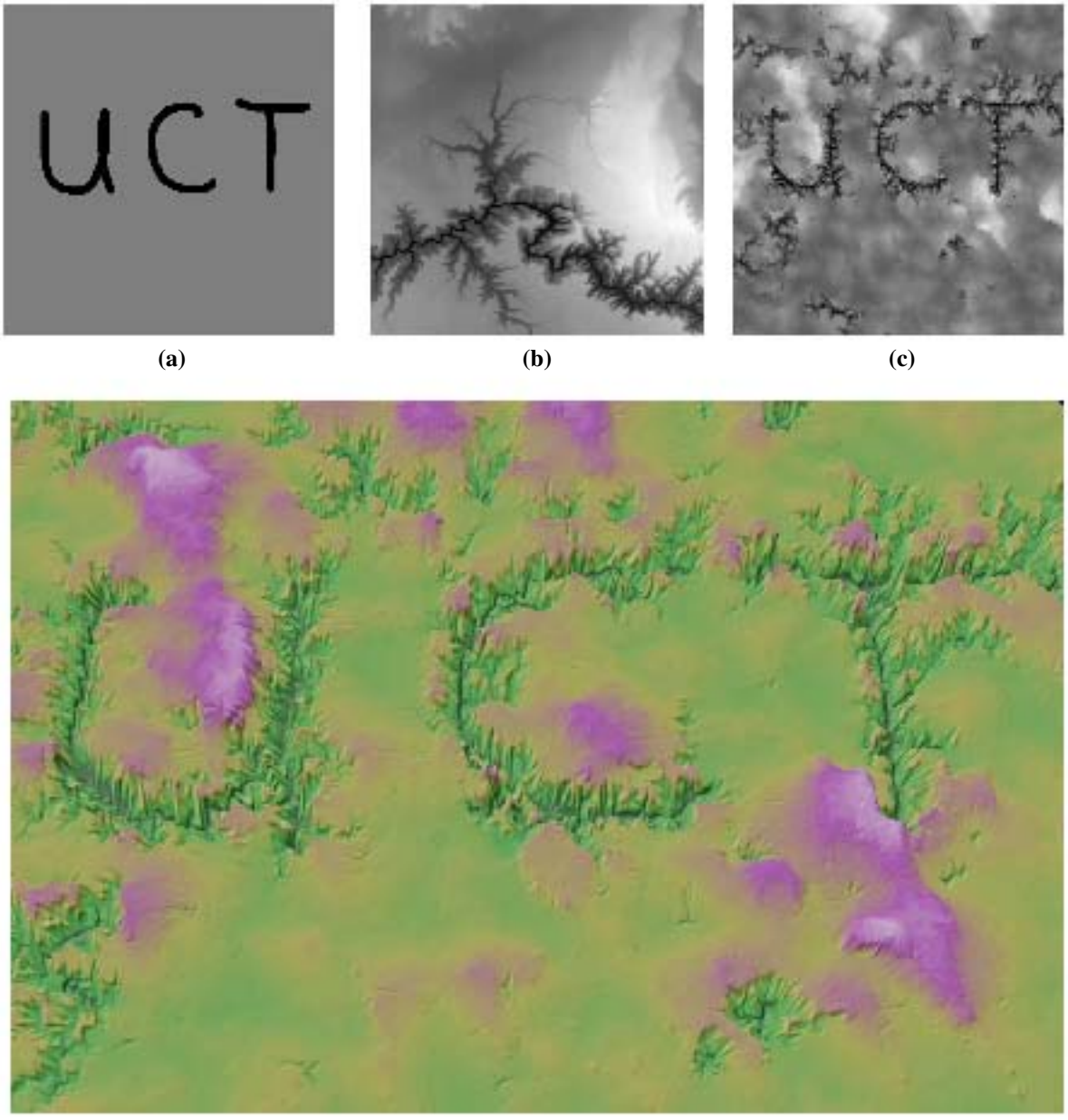


(a) **(b)**

(c) 60×60 **(d)** 80×80 **(e)** 100×100

(f) 60×60 **(g)** 80×80 **(h)** 100×100

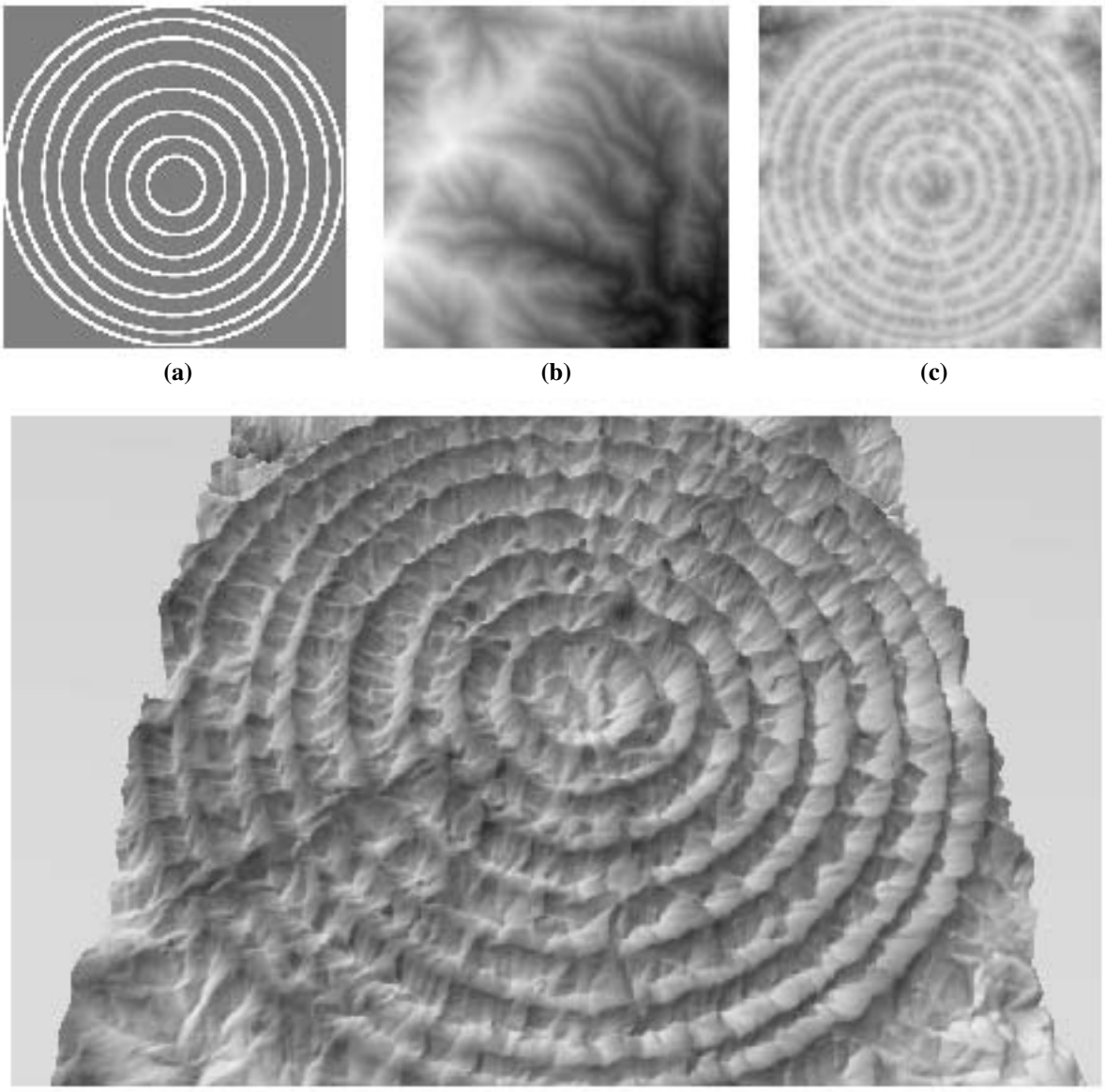
Figure 5.17: Effect of patch size on terrain synthesis. (a) Target terrain (1025×1025). (b) Exemplar terrain (1025×1025) (c) The small patch size 60×60 produces a terrain (generated in 55 seconds) with little structural information around the large scale features. Moreover, the small ridges appear random. (d) The output (generated in 48 secs) matches the target landscape, has more structural information and the small characteristics appear less random. (e) A large patch size reduces the number of possible candidates and decreases the quality of patch matching. The landscape was generated in 51 seconds. (f), (g), and (h) are renderings of (c), (d), and (e), respectively.



(a) **(b)** **(c)**

(d)

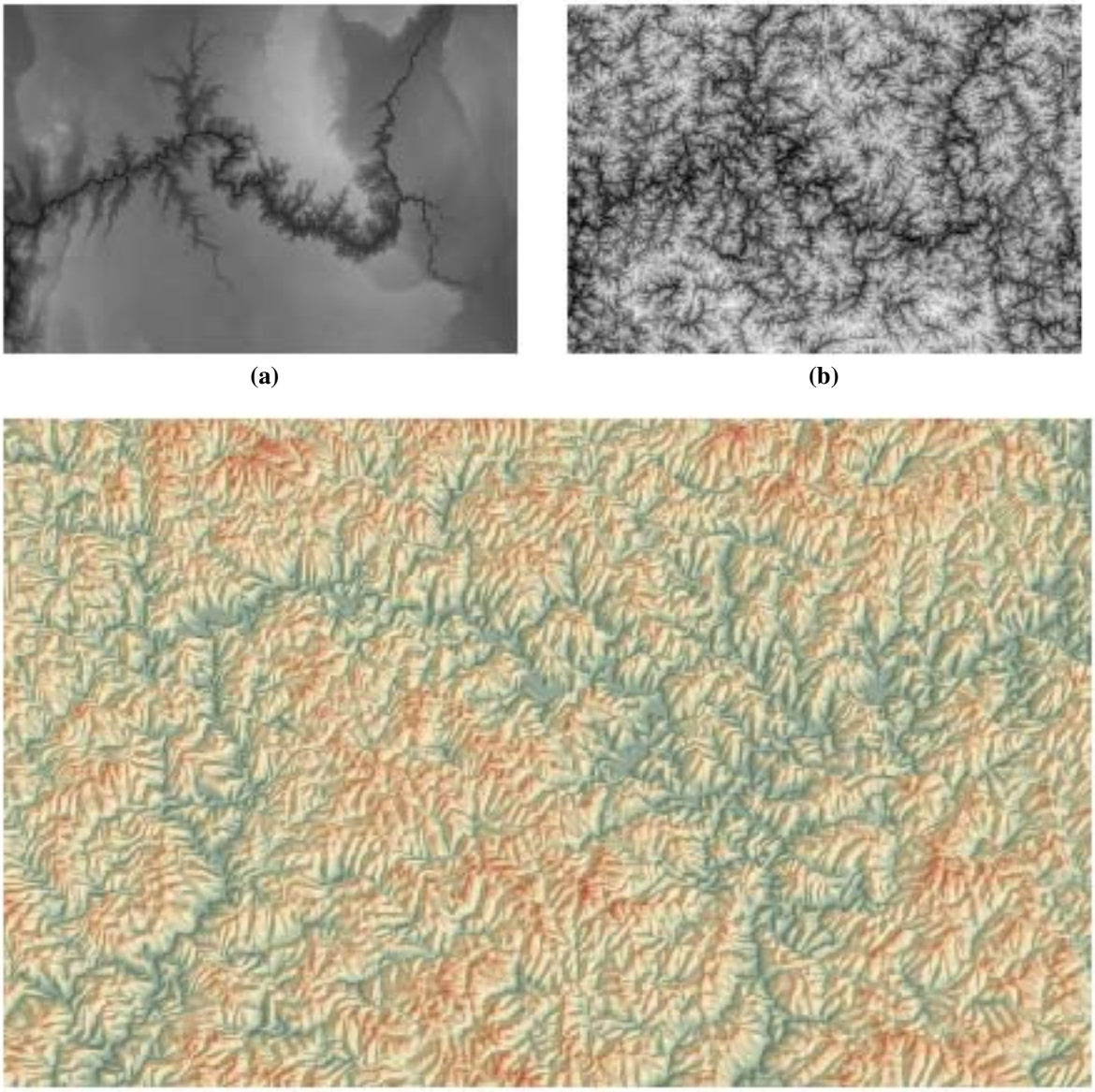
Figure 5.18: Synthesising a large terrain. (a) Target terrain (3500×3500). (b) Grand canyon exemplar (1025×1025). (c) Landscape generated in 15 minutes. (d) A 3D rendering of the generated terrain.



(a) **(b)** **(c)**

(d)

Figure 5.19: Heavily constrained terrain synthesis. (a) Target terrain (1000×1000). (b) Exemplar terrain (512×512). (c) Landscape generated in 46 seconds. (d) A 3D rendering of the generated terrain.



(a) **(b)**

(c)

Figure 5.20: Using the Grand Canyon as the synthesis target. (a) Target terrain (2040×1380). (b) Landscape generated from 1025×1025 Mount Vernon exemplar in 3.5 minutes. (c) A 3D rendering of the generated terrain.

Chapter 6

Conclusion

6.1 Summary

This thesis presents a parallel terrain generation method that creates new and realistic landscapes from real-life terrains in a patch-based texture synthesis guided by a user-provided target heightmap. Our results indicate the following:

- Combined with a sketching interface Gain et al. (2009), the proposed framework provides a powerful terrain modelling tool controlled by $2\frac{1}{2}D$ user-sketched curves that indicates the features the generated terrain must have and the height values along these features. The proposed system generates a terrain that fits user constraints and has the characteristics of a real landscape. A user study confirms that terrains generated by the framework are more realistic than those created by a multiresolution deformation of a flat terrain to fit user requirements Gain et al. (2009). Moreover, the study shows that there is no significant difference in realism between the framework's results and real landscapes, indicating the high level of realism of our results.

- Our system improves on Zhou et al. (2007)'s algorithm by increasing the number of candidate patches from the exemplar by a factor of 10 and enhancing the matching process though the addition of criteria such as noise variance and height values along user-specified features. A visual comparison of landscape generated by the proposed framework against Zhou et al. (2007)'s results shows that in most situations, our results show a better match for the user target.

- Our patch merging technique based on the graphcut algorithm, Shepard Interpolation and Poisson equation solver succeeds in eliminating the boundary artifacts created by overlapping patches. The current approach to patch merging, a combination of graphcut and Poisson seam removal (Zhou et al., 2007), is not suited to terrains because it introduces discontinuities in the second order derivatives of the output terrain. These discontinuities are not visible when the terrain is viewed as a $2D$ heightmap but become immediately noticeable in 3D under lighting. A user study confirms that our merging technique is superior to the Poisson seam removal approach. However, the study does not provide any information on the relationship between our patch merging method and an approach that uses

Shepard Interpolation to remove optimal cuts found by graphcut. This approach introduces artifacts in the terrain that, for the non expert eye, may appear to be part of the landscape topography.

- Two separate GPU solutions, one based on coalesced global memory and memoization (GPU1C) and the other based on device texture memory (GPU2), accelerate the patch-based synthesis. A performance analysis of GPU1C, GPU2 patch matching and their sequential versions CPU1 and CPU2, shows that GPU2 is 30 times faster than CPU2 for exemplars of size 2049×2049. In addition, GPU1C is 6 times faster than CPU1 and GPU2. GPU1C patch matching offers the best performance but is limited by the size of the device global memory. For instance, the device on which GPU1C was tested has a 1GB memory capacity and can only accommodate up to $26,060$ candidate patches of size 80×80 . Our system is thus set up to use GPU1C if the number of candidates is small enough for the device memory and GPU2 otherwise. Given a 1025×1025 exemplar, the framework is able generate to a 1000×1000 landscape in a minute and a 3500×3500 terrain in 15 minutes.

These promising results suggest that texture-based terrain synthesis is an attractive alternative to fractal-based and physics-based terrain synthesis techniques. There are still some challenges in making patch-based terrain synthesis methods fast enough to be an alternative to fractal-based generation for real-time applications.

6.2 Future work

The patch-based synthesis could be extended by using multiple example terrains to produce landscapes with a visual appearance combining all the exemplars. This will increase the range of possible output terrains. Wei (2003) proposes a pixel-based texture synthesis method from multiple sources $\{S_i\}$ that finds the best set of input pixels $\{q_i\}$ for an output pixel p by minimizing the error function

$$E(p, \{q_i\}) = \sum_i w_i \times (\|p - q_i\|^2 + \|N(p) - N(q_i)\|^2) \tag{6.1}$$

where i loops through all the input textures, $N(p)$ and $N(q_i)$ are the neighbourhoods of p and q_i, w_i specifies the influence of the input texture S_i, and $\|.\|$ is the L_2 norm.

Minimizing this error function picks the value of the output pixel p so that local similarity with all the exemplars is preserved simultaneously. This pixel-based algorithm could be adapted to patch-based synthesis by using a modified version of Equation 6.1 for patch matching:

$$E^{'}(P, \{Q_i\}) = \sum_i w_i \times (\alpha_o \|P - Q_i\|^2 + \alpha_n \|noisevar(P) - noisevar(Q_i)\|^2) \tag{6.2}$$

where $noisevar(P)$ is the noise variance of the patch P at multiple levels of resolution , α_o is the weight of height differences and α_n determines the weight of the noise term.

Multiple GPUs increase the available computational power and memory bandwidth, and their simultaneous use is supported by NVIDIA's CUDA. The proposed framework could be further accelerated by implementing patch matching and merging on multiple devices. A GPU could be allocated to each exemplar, if multiple exemplars are used, or kernel threads could be divided into groups and each group allocated to a GPU.

Thus far, parameter values such as the patch size are user-specified and significantly influence the appearance of the output. Automatically selecting the optimal patch size in a preprocessing step will not only reduce parameter manipulation but also improve the quality of the terrain generation. Narkdej and Kanongchaiyos (2009) propose an algorithm for automatically finding the best patch size and extent of overlap in patch-based texture synthesis. Unfortunately, the analysis needed for estimating these parameters is slower than the synthesis itself. Implementing this algorithm on the GPU might make it fast enough to be usable in our framework.

Bibliography

ADELSON, E. H., ANDERSON, C. H., BERGEN, J. R., BURT, P. J., AND OGDEN, J. M. 1984. 1984, Pyramid methods in image processing. *RCA Engineer 29,* 6, 33–41.

ALIAGA, D. G., VANEGAS, C. A., AND BENEŠ, B. 2008. Interactive example-based urban layout synthesis. In *ACM SIGGRAPH Asia 2008 papers*. SIGGRAPH Asia '08. ACM, New York, NY, USA, 160:1–160:10.

ANH, N. H., SOURIN, A., AND ASWANI, P. 2007. Physically based hydraulic erosion simulation on graphics processing unit. In *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*. GRAPHITE '07. ACM, New York, NY, USA, 257–264.

ASHIKHMIN, M. 2001. Synthesizing natural textures. In *In ACM Symposium on Interactive 3D Graphics*. 217–226.

BANGAY, S., DE BRUYN, D., AND GLASS, K. 2010. Minimum spanning trees for valley and ridge characterization in digital elevation maps. In *Proceedings of the 7th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*. ACM, 73–82.

BELHADJ, F. 2007. Terrain modeling: a constrained fractal model. In *Proceedings of the 5th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*. AFRIGRAPH '07. ACM, New York, NY, USA, 197–204.

BENES, B., MARAK, I., AND SIMEK, P. 1997. Hierarchical erosion of synthetical terrains. In *In Spring Conference on Computer Graphics, Bratislava: Comenius University*. 93–100.

BOOKSTEIN, F. L. 1989. Principal warps: Thin-plate splines and the decomposition of deformations. *IEEE Transactions on Pattern Analysis and Machine Intelligence 11*, 567–585.

BOYKOV, Y. AND KOLMOGOROV, V. 2004. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Transactions Pattern Analysis and Machine Intelligence 26*, 1124–1137.

BROSZ, J., SAMAVATI, F. F., AND SOUSA, M. C. 2006. Terrain synthesis by-example. In *Proceedings of the first International Conference on Computer Graphics Theory and Applications*.

BRUUN, B. T. AND NILSEN, S. 2001. Multiscale representation of terrain models using average interpolating wavelets. In *ScanGIS 2001: Proceedings of the 8th Scandinavian Research Conference on Geographical Information Science*. 33–44.

BRYCE. 2010. www.daz3d.com/i.x/software/bryce.

BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. 2004. Brook for gpus: stream computing on graphics hardware. *ACM Transactions Graphics 23*, 777–786.

CHANG, Y.-C. AND SINHA, G. 2007. A visual basic program for ridge axis picking on dem data using the profile-recognition and polygon-breaking algorithm. *Computers and Geosciences 33*, 229–237.

CHANG, Y.-C., SONG, G.-S., AND HSU, S.-K. 1998. Automatic extraction of ridge and valley axes using the profile recognition and polygon-breaking algorithm. *Computer Geosciences 24*, 83–93.

CHANGZHEN, X., FENHONG, G., JIANCHENG, Z., AND DONGXU, Q. 2007. Patch map for fast texture synthesis. In *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*. PacRim 2007.

CHIANG, M.-Y., TU, S.-C., HUANG, J.-Y., TAI, W.-K., LIU, C.-D., AND CHANG, C.-C. 2005. Terrain synthesis: An interactive approach. In *International Workshop on Advanced Image Technology*.

CHIBA, N., MURAOKA, K., AND FUJITA, K. 1998. An erosion model based on velocity fields for the visual simulation of mountain scenery. *Journal of Visualization and Computer Animation 9*, 185–194.

CLARKSON, K. L. 2006. Nearest-neighbor searching and metric space dimensions. In *Nearest-Neighbor Methods for Learning and Vision: Theory and Practice*, G. Shakhnarovich, T. Darrell, and P. Indyk, Eds. MIT Press, 15–59.

COHEN, J. M., HUGHES, J. F., AND ZELEZNIK, R. C. 2000. Harold: A world made of drawings. In *Proceedings of the First International Symposium on Non Photorealistic Animation and Rendering (NPAR) for Art and Entertainment*. To be held in Annecy, France.

COOK, R. L. AND DEROSE, T. 2005. Wavelet noise. In *ACM SIGGRAPH 2005 Papers*. SIGGRAPH '05. ACM, New York, NY, USA, 803–811.

CRIMINISI, A., PEREZ, P., AND TOYAMA, K. 2004. Region filling and object removal by exemplar-based image inpainting. *Image Processing, IEEE Transactions on 13,* 9, 1200 – 1212.

DACHSBACHER, C. 2006. Interactive terrain rendering: Towards realism with procedural models and graphics hardware. Ph.D. thesis.

DORSEY, J., EDELMAN, A., JENSEN, H. W., LEGAKIS, J., AND PEDERSEN, H. K. 2006. Modeling and rendering of weathered stone. In *ACM SIGGRAPH 2006 Courses*. SIGGRAPH '06. ACM, New York, NY, USA.

EBERT, D. S., MUSGRAVE, F. K., PEACHEY, D., PERLIN, K., AND WORLEY, S. 2002. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann Publishers Inc.

EFROS, A. A. AND FREEMAN, W. T. 2001. Image quilting for texture synthesis and transfer. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '01. ACM, New York, NY, USA, 341–346.

EFROS, A. A. AND LEUNG, T. K. 1999. Texture synthesis by non-parametric sampling. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*. Vol. 2. 1033–1038.

FOURNIER, A., FUSSELL, D., AND CARPENTER, L. 1982. Computer rendering of stochastic models. *Communications ACM 25,* 6, 371–384.

FRIEDMAN, M. 1937. The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the American Statistical Association 32,* 200, pp. 675–701.

GAIN, J., MARAIS, P., AND STRASSER, W. 2009. Terrain sketching. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*. I3D '09. ACM, New York, NY, USA, 31–38.

GARCIA, V. AND NIELSEN, F. 2009. Searching high-dimensional neighbours: Cpu-based tailored data-structures versus gpu-based brute-force method. In *Computer Vision/Computer Graphics CollaborationTechniques*. Vol. 5496. Springer Berlin / Heidelberg, 425–436.

GEORGE, J. A. 1970. The use of direct methods for the solution of the discrete poisson equation on non-rectangular regions. Tech. rep., Stanford University, Stanford, CA, USA.

GREENWALD, A. G. 1976. Within-subjects designs: To use or not to use. *Psychological Bulletin 83*, 216–229.

HALF-LIFE 2. 2004. http://www.valvesoftware.com/games/hl2.html.

HAN, C., RISSER, E., RAMAMOORTHI, R., AND GRINSPUN, E. 2008. Multiscale texture synthesis. In *ACM SIGGRAPH 2008 papers*. SIGGRAPH '08. ACM, New York, NY, USA, 51:1–51:8.

HERTZMANN, A., JACOBS, C. E., OLIVER, N., CURLESS, B., AND SALESIN, D. H. 2001. Image analogies. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '01. ACM, New York, NY, USA, 327–340.

INDYK, P. AND MOTWANI, R. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. STOC '98. ACM, New York, NY, USA, 604–613.

KELLEY, A. D., MALIN, M. C., AND NIELSON, G. M. 1988. Terrain simulation using a model of stream erosion. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '88. ACM, New York, NY, USA, 263–268.

KRUSKAL, J. B. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. In *Proceedings of the American Mathematical Society*. Vol. 7. 48–50.

KWATRA, V., ESSA, I., BOBICK, A., AND KWATRA, N. 2005. Texture optimization for example-based synthesis. In *ACM SIGGRAPH 2005 Papers*. SIGGRAPH '05. ACM, New York, NY, USA, 795–802.

KWATRA, V., SCHÖDL, A., ESSA, I., TURK, G., AND BOBICK, A. 2003. Graphcut textures: image and video synthesis using graph cuts. In *ACM SIGGRAPH 2003 Papers*. SIGGRAPH '03. ACM, New York, NY, USA, 277–286.

LAGAE, A., LEFEBVRE, S., COOK, R., DEROSE, T., DRETTAKIS, G., EBERT, D., LEWIS, J., PERLIN, K., AND ZWICKER, M. 2010. State of the art in procedural noise functions. In *EG 2010 - State of the Art Reports*, H. Hauser and E. Reinhard, Eds. Eurographics, Eurographics Association.

LEE, T.-Y. AND YAN, C.-R. 2005. Feature-based texture synthesis. In *Computational Science and Its Applications ICCSA 2005*, O. Gervasi, M. L. Gavrilova, V. Kumar, A. LaganÃ , H. P. Lee, Y. Mun, D. Taniar, and C. J. K. Tan, Eds. Lecture Notes in Computer Science Series, vol. 3482. Springer Berlin / Heidelberg, 131–173.

LEFEBVRE, S. AND HOPPE, H. 2005. Parallel controllable texture synthesis. In *ACM SIGGRAPH 2005 Papers*. SIGGRAPH '05. ACM, New York, NY, USA, 777–786.

LEVIN, A., ZOMET, A., PELEG, S., AND WEISS, Y. 2004. Seamless image stitching in the gradient domain. In *In Eighth European Conference on Computer Vision*. ECCV 2004. Springer-Verlag, 377–389.

LEWIS, J. P. 1987. Generalized stochastic subdivision. *ACM Transactions on Graphics 6*, 167–190.

LIANG, L., LIU, C., XU, Y.-Q., GUO, B., AND SHUM, H.-Y. 2001. Real-time texture synthesis by patch-based sampling. *ACM Transactions on Graphics 20*, 127–150.

MANDELBROT, B. B. 1982. *The Fractal Geometry of Nature* 1 Ed. W. H. Freeman.

MILLER, G. S. P. 1986. The definition and rendering of terrain maps. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '86. ACM, New York, NY, USA, 39–48.

MILLIRON, T., JENSEN, R. J., BARZEL, R., AND FINKELSTEIN, A. 2002. A framework for geometric warps and deformations. *ACM Transactions on Graphics 21*, 20–51.

MUSGRAVE, F. K., KOLB, C. E., AND MACE, R. S. 1989. The synthesis and rendering of eroded fractal terrains. *SIGGRAPH Computer Graphics 23*, 41–50.

NAGASHIMA, K. 1998. Computer generation of eroded valley and mountain terrains. *The Visual Computer 13*, 456–464. 10.1007/s003710050117.

NARKDEJ, J. AND KANONGCHAIYOS, P. 2009. An efficient parameters estimation method for automatic patch-based texture synthesis. In *Proceedings of the 6th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*. AFRIGRAPH '09. ACM, New York, NY, USA, 91–99.

NEALEN, A. AND ALEXA, M. 2003. Hybrid texture synthesis. In *Proceedings of the 14th Eurographics workshop on Rendering*. EGRW '03. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 97–105.

NEIDHOLD, B., WACKER, M., AND DEUSSEN, O. 2005. Interactive physically based fluid and erosion simulation. In *Proceedings of Eurographics Workshop on Natural Phenomena*. 25–32.

NIE, D., MA, L., AND XIAO, S. 2006. Similarity based image inpainting method. In *Multi-Media Modelling Conference Proceedings, 2006 12th International*. 4 pp.

NVIDIA CORPORATION. 2010a. *NVIDIA CUDA Compute Unified Device Architecture Best Practices Guide*. NVIDIA Corporation.

NVIDIA CORPORATION. 2010b. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation.

OLSEN, J. 2004. Realtime procedural terrain generation - realtime synthesis of eroded fractal terrain for use in computer games. In *Department of Mathematics And Computer Science (IMADA)*. University of Southern Denmark.

ONG, T. J., SAUNDERS, R., KEYSER, J., AND LEGGETT, J. J. 2005. Terrain generation using genetic algorithms. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*. GECCO '05. ACM, New York, NY, USA, 1463–1470.

PAJAROLA, R., ANTONIJUAN, M., AND LARIO, R. 2002. Quadtin: quadtree based triangulated irregular networks. In *Proceedings of the conference on Visualization '02*. VIS '02. IEEE Computer Society, Washington, DC, USA, 395–402.

PANDROMEDA. 2010. Mojoworld. http://www.mojoworld.org/.

PÉREZ, P., GANGNET, M., AND BLAKE, A. 2003. Poisson image editing. In *ACM SIGGRAPH 2003 Papers*. SIGGRAPH '03. ACM, New York, NY, USA, 313–318.

PERLIN, K. 1985. An image synthesizer. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '85. ACM, New York, NY, USA, 287–296.

PERLIN, K. 2002. Improving noise. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '02. ACM, New York, NY, USA, 681–682.

PEYTAVIE, A., GALIN, E., MERILLOU, S., AND GROSJEAN, J. 2008. ModÃ©lisation de terrains complexes 3D. In *21e JournÃ©es Annuelles de l'Association FranÃ§aise d'Informatique Graphique*.

PRIM, R. C. 1957. Shortest connection networks and some generalizations. *Bell System Technical Journal 36*, 1389–1401.

ROUDIER, P., PEROCHE, B., AND PERRIN, M. 1993. Landscapes synthesis achieved through erosion and deposition process simulation. *Computer Graphics Forum 12,* 3, 375–383.

SAUPE, D. 1988. *Algorithms for random fractals*. Springer-Verlag New York, Inc., New York, NY, USA, 71–113.

SAUPE, D. 2003. Fractals. In *Encyclopedia of Computer Science*. John Wiley and Sons Ltd., Chichester, UK, 725–732.

SCHNEIDER, J., BOLDTE, T., AND WESTERMANN, R. 2006. Real-time editing, synthesis, and rendering of infinite landscapes on GPUs. In *Vision, Modeling and Visualization 2006*.

SEDGEWICK, R. 2001. *Algorithms in C, Part 5: Graph Algorithms*. Addison-Wesley, Massachusetts.

SHAPIRO, S. AND WILK, M. 1965. An analysis of variance test for normality (complete samples). *Biometrika 52*, 591–611.

SHEPARD, D. 1968. A two-dimensional interpolation function for irregularly-spaced data. In *Proceedings of the 1968 23rd ACM national conference*. ACM '68. ACM, New York, NY, USA, 517–524.

SHEWCHUK, J. R. 1994. An introduction to the conjugate gradient method without the agonizing pain. Tech. rep., Carnegie Mellon University, Pittsburgh, PA, USA.

SILVA, C. T., MITCHELL, J. S. B., AND KAUFMAN, A. E. 1995. Automatic generation of triangular irregular networks using greedy cuts. In *Proceedings of the 6th conference on Visualization '95*. VIS '95. IEEE Computer Society, Washington, DC, USA, 201–.

SINHA, S. N., FRAHM, J.-M., POLLEFEYS, M., AND GENC, Y. 2006. GPU-based Video Feature Tracking and Matching. In *In Workshop on Edge Computing Using New Commodity Architectures*.

STOLLNITZ, E. J., DEROSE, T. D., AND SALESIN, D. H. 1995. Wavelets for computer graphics: A primer, part 1. *IEEE Computer Graphics Applications 15*, 76–84.

TERRAGEN. 2010. Terragen by planetside software. http://www.planetside.co.uk/terragen.

US GEOLOGICAL SURVEY. 2011. http://seamless.usgs.gov/.

VINEET, V. AND NARAYANAN, P. J. 2008. Cuda cuts: Fast graph cuts on the gpu. *Computer Vision and Pattern Recognition Workshop 0*, 1–8.

VOSS, R. F. 1985. Random fractal forgeries. In *Fundamental Algorithms for Computer Graphics*, R. A. Earnshaw, Ed. Springer-Verlag.

WATANABE, N. AND IGARASHI, T. 2004. A sketching interface for terrain modeling. In *ACM SIGGRAPH 2004 Posters*. SIGGRAPH '04. ACM, New York, NY, USA, 73.

WEI, L.-Y. 2003. Texture synthesis from multiple sources. In *ACM SIGGRAPH 2003 Sketches & Applications*. SIGGRAPH '03. ACM, New York, NY, USA, 1–1.

WEI, L.-Y., LEFEBVRE, S., KWATRA, V., AND TURK, G. 2009. State of the art in example-based texture synthesis. In *Eurographics 2009, State of the Art Report, EG-STAR*. Eurographics Association.

WEI, L.-Y. AND LEVOY, M. 2000. Fast texture synthesis using tree-structured vector quantization. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '00. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 479–488.

WEI, L.-Y. AND LEVOY, M. 2002. Order-independent texture synthesis. Tech. rep., Stanford University.

WOODARD, T. 2005. Real-time gpu-based texture synthesis. In *Proceedings of IMAGE 2005*.

WORLD MACHINE 2. 2010. http://www.world-machine.com/.

WU, Q. AND YU, Y. 2004. Feature matching and deformation for texture synthesis. In *ACM SIGGRAPH 2004 Papers*. SIGGRAPH '04. ACM, New York, NY, USA, 364–367.

XIAO, C., LIU, M., YONGWEI, N., AND DONG, Z. 2010. Fast exact nearest patch match for patch-based image editing and processing. *Visualization and Computer Graphics, IEEE Transactions on PP,* 99, 1.

ZELINKA, S. AND GARLAND, M. 2002. Towards real-time texture synthesis with the jump map. In *Proceedings of the 13th Eurographics workshop on Rendering*. EGRW '02. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 99–104.

ZHANG, H., OUYANG, D., LIN, H., AND GUAN, W. 2008. Texture synthesis based on terrain feature recognition. In *Proceedings of the 2008 International Conference on Computer Science and Software Engineering*. Vol. 2. IEEE Computer Society, Washington, DC, USA, 1158–1161.

ZHANG, Z., WANG, W., AND JIN, W. 2010. Texture synthesis based on compute unified device architecture. In *Multimedia Technology (ICMT), 2010 International Conference on*. 1–4.

ZHOU, H., SUN, J., TURK, G., AND REHG, J. M. 2007. Terrain synthesis from digital elevation models. *IEEE Transactions on Visualization and Computer Graphics 13*, 834–848.

ZOU, K., XU, X., YANG, L., LI, Y., AND ZHANG, J. 2010. Graph cut-based fast texture synthesis with cuda. In *2010 International Conference On Computer Design And Appliations*. Vol. 5.

Appendix A

	T_{real}	T_{sys}	T_{def}		T_{real}	T_{sys}	T_{def}		T_{real}	T_{sys}	T_{def}
1	2	1	3	21	1	2	3	41	1	2	3
2	1	2	3	22	2	1	3	42	1	2	3
3	2	1	3	23	2	1	3	43	2	1	3
4	2	1	3	24	2	1	3	44	1	2	3
5	1	2	3	25	1	2	3	45	1	2	3
6	2	1	3	26	2	1	3	46	1	2	3
7	1	2	3	27	2	1	3	47	1	2	3
8	2	1	3	28	1	2	3	48	2	1	3
9	1	2	3	29	1	2	3	49	2	1	3
10	2	1	3	30	1	2	3	50	1	2	3
11	2	1	3	31	2	1	3	51	2	1	3
12	1	2	3	32	1	3	2	52	2	1	3
13	1	2	3	33	1	2	3	53	1	2	3
14	2	1	3	34	2	1	3	54	2	1	3
15	2	1	3	35	2	3	1	55	1	2	3
16	1	2	3	36	2	1	3	56	2	1	3
17	1	2	3	37	1	2	3	57	1	2	3
18	2	1	3	38	3	1	2	58	1	2	3
19	1	2	3	39	2	1	3	59	2	1	3
20	1	2	3	40	3	2	1	60	1	2	3

Table 6.1: Experiment 1: Ratings of terrain groups T_{def} (deformed), T_{real} (real) and T_{sys} (generated by the patch-based synthesis framework).

GPU1

Exemplar size	Feature extraction	Feature patch matching	Non feature patch matching	Patch merging	Computing filling priorities	Overall time
513^2	0.88	6.23	6.33	14.09	4.94	32.47
1025^2	4.35	10.23	8.15	14.65	5.47	42.85
1537^2	13.21	18.17	17.75	14.53	6.12	69.78
2049^2	26.66	26.19	32.38	14.55	7.08	106.86

GPU1 NC

Exemplar size	Feature extraction	Feature patch matching	Non feature patch matching	Patch merging	Computing filling priorities	Overall time
513^2	0.88	8.68	7.69	14.12	4.57	35.94
1025^2	4.37	20.27	18.36	14.51	5.26	62.77
1537^2	13.19	34.73	36.76	14.76	5.84	105.28
2049^2	26.76	47.92	91.19	14.48	7.04	187.39

GPU2

Exemplar size	Feature extraction	Feature patch matching	Non feature patch matching	Patch merging	Computing filling priorities	Overall time
513^2	0.89	18.39	17.86	13.94	4.58	55.46
1025^2	4.37	41.48	37.67	14.47	5.83	103.82
1537^2	13.28	83.3	96.67	14.63	6.37	214.25
2049^2	28.25	120.85	197.94	14.67	7.82	369.53

CPU1

Exemplar size	Feature extraction	Feature patch matching	Non feature patch matching	Patch merging	Computing filling priorities	Overall time
513^2	0.88	15.85	9.93	26.83	5.77	59.26
1025^2	4.37	56.13	32.9	27.89	4.59	125.88
1537^2	13.22	103.21	78.23	28.77	4.45	227.88
2049^2	26.81	145.43	141.61	28.06	4.48	346.39

CPU2

Exemplar size	Feature extraction	Feature patch matching	Non feature patch matching	Patch merging	Computing filling priorities	Overall time
513^2	0.88	353.15	239.13	26.91	4.53	624.6
1025^2	4.35	1491.29	1244.36	27.67	4.86	2772.53
1537^2	13.37	2835.85	3098.5	28.79	4.7	5981.21
2049^2	27.1	5906.7	5790.61	28.13	4.68	11737.22

Table 6.2: CPU and GPU performance (in seconds) as the exemplar size varies, with 50% of patches placed during feature patch matching.