Chapter 4
Fractals, noise and agents with applications to landscapes

Noor Shaker, Julian Togelius, and Mark J. Nelson

Abstract Most games include some form of terrain or landscape (other than a flat floor) and this chapter is about how to effectively create the ground you (or the characters in your game) are standing on. It starts by describing several fast but effective stochastic methods for terrain generation, including the classic and widely used diamond-square and Perlin-noise methods. It then goes into agent-based methods for building more complex landscapes, and search-based methods for generating maps that include particular gameplay elements.

4.1 Terraforming and making noise

This chapter is about terrains (or landscapes—we will use the words interchangeably) and noise, two types of content which have more in common than might be expected. We will discuss three very different types of methods for generating such content, but first we will discuss where and why terrains and noise are used.

Terrains are ubiquitous. Almost any three-dimensional game will feature some ground to stand or drive on, and in most of them there will be some variety such as different types of vegetation, differences in elevation etc. What changes is how much you can interact directly with the terrains, and thus how they affect the game mechanics.

At one extreme of the spectrum are flight simulators. In many cases, the terrain has no game-mechanical consequences—you crash if your altitude is zero, but in most cases the minor variations in the terrain are not enough to affect your performance in the game. Instead, the role of the terrain is to provide a pretty backdrop and help the player to orientate. Key demands on the terrain are therefore that it is visually pleasing and believable, but also that it is huge: airplanes fly fast, are not hemmed in by walls, and can thus cover huge areas. From 30,000 feet one might not be able to see much detail and a low-resolution map might therefore be seen as a solution, but preferably it should be possible to swoop down close to the ground

and see hills, houses, creeks and cars. Therefore, a map where the larger features were generated in advance but where details could be generated on demand would be useful. Also, from a high altitude it is easy to see the kind of regularities that result from essentially copying and pasting the same chunks of landscape, so reusing material is not trivial.

In open-world games such as *Skyrim* and the *Grand Theft Auto* series, terrains sometimes have mechanical and sometimes aesthetic roles. This poses additional demands on the design. When driving through a landscape in *Grand Theft Auto*, it needs to be believable and visually pleasing, but it also needs to support the stretch of road you are driving on. The mountains in *Skyrim* look pretty in the distance, but also function as boundaries of traversable space and to break line of sight. To make sure that these demands are satisfied, the generation algorithms need a high degree of controllability.

At the other end of the spectrum are those games where the terrain severely restricts and guides the player's possible course of actions. Here we find first-person shooters such as those in the *Halo* and *Call of Duty* series. In these cases, terrain generation has more in common with the level-generation problems we discussed in the previous chapter.

Like terrains, noise is a very common type of game content. Noise is useful whenever small variations need to be added to a surface (or something that can be seen as a surface). One example of noise is in skyboxes, where cloud cover can be implemented as a certain kind of white-coloured noise on a blue-coloured background. Other examples include dust that settles on the ground or walls, certain aspects of water (though water simulation is a complex topic in its own right), fire, plasma, skin and fur colouration etc. You can also see minor topological variations of the ground as noise, which brings us to the similarity between terrains and noise.

4.1.1 Heightmaps and intensity maps

Both noise and most aspects of terrains can fruitfully be represented as two-dimensional matrices of real numbers. The width and height of the matrix map to the x and y dimensions of a rectangular surface. In the case of noise, this is called an *intensity map*, and the values of cells correspond directly to the brightness of the associated pixels. In the case of terrains, the value of each cell corresponds to the height of the terrain (over some baseline) at that point. This is called a *heightmap*. If the resolution with which the terrain is rendered is greater than the resolution of the heightmap, intermediate points on the ground can simply be interpolated between points that do have specified height values. Thus, using this common representation, any technique used to generate noise could also be used to generate terrains, and vice versa—though they might not be equally suitable.

It should be noted that in the case of terrains, other representations are possible and occasionally suitable or even necessary. For example, one could represent the terrain in three dimensions, by dividing the space up into *voxels* (cubes) and

computing the three-dimensional voxel grid. An example is the popular open-world game *Minecraft*, which uses unusually large voxels. Voxel grids allow structures that cannot be represented with heightmaps, such as caves and overhanging cliffs, but they require a much larger amount of storage.

4.2 Random terrain

Let's say we want to generate completely random terrain. We won't worry for the moment about the questions in the previous chapter, such as whether the terrain we generate would make a fair, balanced, and playable RTS map. All we want for now is random terrain, with no constraints except that it looks like terrain.

If we encode terrain as a heightmap, then it's represented by a two-dimensional array of values, which indicate the height at each point. Can generating random terrain be as simple as just calling a random-number generator to fill each cell of the array? Alas, no. While this technically works—a randomly initialized heightmap is indeed a heightmap that can be rendered as terrain—the result is not very useful. It doesn't look anything like random terrain, and isn't very useful as terrain, even if we're being generous. A random heightmap generated this way looks like random *spikes*, not random terrain: there are no flat portions, mountain ranges, hills, or other features typically identifiable on a landscape.

The key problem with just filling a heightmap with random values is that every random number is generated independently. In real terrain, heights at different points on the terrain are not independent of each other: the elevation at a specific point on the earth's surface is statistically related to the elevation at nearby points. If you pick a random point within 100 m of the peak of Mount Everest, it will almost certainly have a high elevation. If you pick a random point within 100 m of central Copenhagen, you are very unlikely to find a high elevation.

There are several alternative ways of generating random heightmaps to address this problem. These methods were originally invented, not for landscapes, but for textures in computer graphics, which had the same issue [3]. If we generate random graphical textures by randomly generating each pixel of the texture, this produces something that looks like television static, which isn't appropriate for textures that are going to represent the surfaces of "organic" patterns found in nature, such as the texture of rocks. We can think of landscape heightmaps as a kind of natural pattern, but a pattern that's interpreted as a 3D elevation rather than a 2D texture. So it's not a surprise that similar problems and solutions apply.

4.2.1 Interpolated random terrain

One way of avoiding unrealistically spiky landscapes is to require that the landscapes we generate are smooth. That change does exclude some realistic kinds of

landscapes, since discontinuities such as cliffs exist in real landscapes. But it's a change that will provide us with something much more landscape-like than the random heightmap method did.

How do we generate smooth landscapes? We might start by coming up with a formal definition of *smoothness* and then develop a method to optimise for that criterion. A simpler way is to make landscapes smooth by construction: fill in the values in such a way that the result is less spiky than the fully random generator. *Interpolated noise* is one such method, in which we generate fewer random values, and then interpolate between them.

With interpolated noise, instead of generating a random value at every point in the heightmap, we generate random values on a coarser *lattice*. The heights in between the generated lattice points are interpolated in a way that makes them smoothly connect the random heights. Put differently, we randomly generate elevations for peaks and valleys with a certain spacing, and then fill in the slopes between them.

That leaves one question: how do we do the interpolation, i.e. how do we connect the slopes between the peaks and valleys? There are a number of standard interpolation methods for doing so, which we'll discuss in turn.

4.2.1.1 Bilinear interpolation

A simple method of interpolating is to calculate a weighted average in first the horizontal, and then the vertical direction (or vice versa, which gives the same result). If we choose a lattice that's one-tenth as finely detailed as our heightmap's resolution, then $height[0,0]$ and $height[0,10]$ will be two of the randomly generated values. To fill in what should go in $height[0,1]$, then, we notice it's 10% of the way from $height[0,0]$ to $height[0,10]$. Therefore, we use the weighted average, $height[0,1] = 0.9 \times height[0,0] + 0.1 \times height[0,10]$. Once we've finished this interpolation in the x direction, then we do it in the y direction. This is called *bilinear interpolation*, because it does linear interpolation along two axes, and is both easy and efficient to implement.

While it's a simple procedure, coarse random generation on a lattice followed by bilinear interpolation does have drawbacks. The most obvious one is that mountain slopes become perfectly straight lines, and peaks and valleys are all perfectly sharp points. This is to be expected, since a geometric interpretation of the process just described is that we're randomly generating some peaks and valleys, and then filling in the mountain slopes by drawing straight lines connecting peaks and valleys to their neighbours. This produces a characteristically stylized terrain, like a child's drawing of mountains—perhaps what we want, but often not. For games in particular, we often don't want these sharp discontinuities at peaks and valleys, where collision detection can become wonky and characters can get stuck.

4.2.1.2 Bicubic interpolation

Rather than having sharp peaks and valleys connected by straight slopes, we can generate a different kind of stylized mountain profile. When a mountain rises from a valley, a common way it does so is in an S-curve shape. First, the slope starts rising slowly. It grows steeper as we move up the mountain; and finally it levels off at the top in a round peak. To produce this profile, we don't want to interpolate linearly: when we're 10% of the way between lattice points, we don't want to be 10% of the way up the slope's vertical distance yet.

Therefore we don't want to do a weighted average between the neighbouring lattice points according to their distance, but according to a nonlinear function of their distance. We introduce a slope function, $s(x)$, specifying how far up the slope (verically) we should be when we're x of the way between the lattice points, in the direction we're interpolating. In the bilinear interpolation case, $s(x) = x$. But now we want an $s(x)$ whose graph looks like an S-curve. There are many mathematical functions with that shape, but a common one used in computer graphics, because it's simple and fast to evaluate, is $s(x) = -2x^3 + 3x^2$. Now, when we are 10% of the way along, i.e. $x = 0.1$, $s(0.1) = 0.028$, so we should be only 2.8% up the slope's vertical height, still in the gradual portion at the bottom. We use this as the weight for the interpolation, and this time $height[0,1] = 0.972 \times height[0,0] + 0.028 \times height[0,10]$.

Since the $s(x)$ we chose is a cubic (third-power) function of x, and we again apply the interpolation in both directions along the 2D grid, this is called *bicubic interpolation*.

4.2.2 Gradient-based random terrain

In the examples so far, we've generated random values to put into the heightmap. Initially, we tried generating all the heightmap values directly, but that proved too noisy. Instead, we generated values for a coarse lattice, and interpolated the slopes in between the generated values. When done with bicubic interpolation, this produced a smooth slope.

An alternate idea is to generate the slopes directly, and infer height values from that, rather than generate height values and interpolate slopes. The random numbers we're going to generate will be interpreted as random *gradients*, i.e. the steepness and direction of the slopes. This kind of random initialization of an array is called *gradient noise*, rather than the *value noise* discussed in the previous section. It was first done by Ken Perlin in his work on the 1982 film *Tron*, so is sometimes called *Perlin noise*.

Generating gradients instead of height values has several advantages. Since we're interpolating gradients, i.e. rates of change in value, we have an extra level of smoothness: rather than smoothing the change in heights with an interpolation method, we smooth the *rate of change* in heights, so slopes grow shallower or steeper smoothly. Gradient noise also allows us to use lattice-based generation

(which is computationally and memory efficient) while avoiding the rectangular grid effects produced by the interpolation-based methods. Since peaks and valleys are not directly generated on the lattice points, but rather emerge from the rises and falls of the slopes, they are arranged in a way that looks more organic.

As with interpolated value-based terrain, we generate numbers on a coarsely spaced lattice, and interpolate between the lattice points. However, we now generate a 2D vector, (d_x, d_y), at each lattice point, rather than a single value. This is the random gradient, and d_x and d_y can be thought of as the slope's steepness in the x and y directions. These gradient values can be positive or negative, for rising or falling slopes.

Now we need a way of recovering the height values from the gradients. First, we set the height to 0 at each lattice point. It might seem that this would produce noticeable grid artifacts, but unlike with value noise, it doesn't in practice. Since peaks and valleys rise and fall to different heights and with different slopes away from the $h = 0$ lattice points, the zero value is sometimes midway up a slope, sometimes near the bottom, and sometimes near the top, rather than in any visually regular position.

To find the height values at non-lattice points, we look at the four neighbouring lattice points. Consider first only the gradient to the top-left. What would the height value be at the current point if terrain rose or fell from $h = 0$ only according to that one of the four gradients? It would be simply that gradient's value multiplied by the distance we've traveled along it: the x-axis slope, d_x, times the distance we are to the right of the lattice point, added to the y-axis slope, d_y, times the distance we are down from the lattice point. In terms of vector arithmetic, this is the *dot product* between the gradient vector and a vector drawn from the lattice point to our current point.

Repeat this what-if process for each of the four surrounding lattice points. Now we have four height values, each indicating the height of the terrain if only one of the four neighbouring lattice points had influence on its height. Now to combine them, we simply interpolate these values, as we did with the value-noise terrain. We have four surrounding lattice points that now have four height values, and we have already covered, in the previous section, how to interpolate height values, using bilinear or bicubic interpolation.

4.3 Fractal terrain

While gradient noise looks more organic, there is still a rather unnatural aspect to it when treated as terrain: terrain now undulates at a constant frequency, which is the frequency chosen for the lattice point spacing. Real terrain has variation at multiple scales. At the largest scale (i.e. lowest frequency), plains rise into mountain ranges. But at smaller scales, mountain ranges have peaks and valleys, and valleys have smaller hills and ravines. In fact, as you zoom in to many natural phenomena, you see the same kind of variation that was seen at the larger scale, but reproduced at

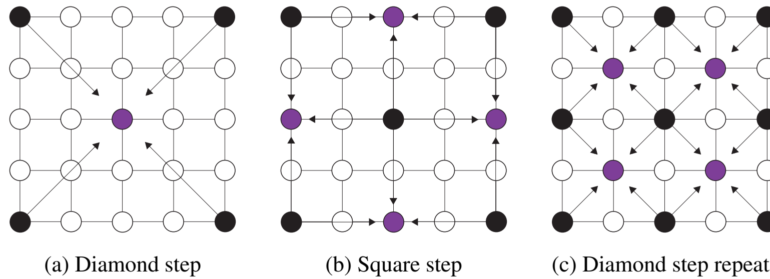

Fig. 4.1: The diamond-square algorithm. (Illustration credit: Amy Hoover)

a new, smaller scale [10]. This self-similarity is the basis of *fractals*, and generated terrain with this property is called *fractal terrain*.

Fractal terrain can be produced through a number of methods, some of them based directly on fractal mathematics, and others producing a similar effect via simpler means.

A very easy way to produce fractal terrain is to take the single-scale random terrain methods from the previous section and simply run them several times, at multiple scales. We first generate random terrain with very large-scale features, then with smaller-scale features, then even smaller, and add all the scales together. The larger-scale features are added in at a larger magnitude than the smaller ones: mountains rise from plains a larger distance than boulders rise from mountain slopes. A classic way of producing multi-scale terrain in this way is to scale the generated noise layers by the inverse of their frequency, which is called $1/f$ noise. If we have a single-scale noise-generation function, like those in the previous section, we can give it a parameter specifying the frequency; let's call this function $noise(f)$. Then starting from a base for our lowest-frequency (largest-scale) features, f, we can define $1/f$ noise as

$$noise(f) + \frac{1}{2}noise(2f) + \frac{1}{4}noise(4f) + \ldots$$

There are many other methods for fractal terrain generation, most of which are beyond the scope of this book, as there exist other textbooks covering the subject in detail [3]. Musgrave et al. [11] group them into five categories of technical approaches, all of which can be seen as implementation methods for the general concept of fractional Brownian motion (fBm). In fBm, we can conceptually think of a terrain as being generated by starting from a point and then taking a random walk following specific statistical properties. Since actually taking millions of such random walks is too computationally expensive, a similar end result is approximated using a variety of techniques. One that is commonly used in games, because it is relatively simple to implement and computationally efficient, is the *diamond-square* algorithm, illustrated in Figure 4.1.

In the diamond-square algorithm, we start by setting the four corners of the heightmap to seed values (possibly random). The algorithm then proceeds as fol-

lows. First, find the point in the center of the square defined by these four corners, and set it to the average of the four corners' values plus a random value. This is the "diamond" step. Then find the four midpoints of the square's sides and set each of them to the average of three values: the two neighbouring corners and the middle of the square (which we just set in the last step)—again, plus a random value. This is the "square" step. The magnitude of the random values we use is called the *roughness*, because larger values produce rougher terrain (likewise, smaller values produce smoother terrain). Completing these two steps has subdivided the original square into four squares. We then reduce the roughness value and repeat the two steps, to fill in these smaller squares. Typically the process repeats until a specified maximum number of iterations have been reached. The end result is an approximation of the terrain produced by fBm.

4.4 Agent-based landscape creation

In Chapter 1, we discussed the desired properties of a PCG algorithm. The previously discussed methods satisfy most of these properties, however they suffer from uncontrollability. The results delivered by these methods are fairly random and they offer very limited interaction with designers, who can only provide inputs on the global level through modifying a set of unintuitive parameters [14]. Several variations of these methods have been introduced that grant more control over the output [7, 1, 13, 16].

The main advantage of software-agent approaches to terrain generation over fractal-based methods is that they offer a greater degree of control while maintaining the other desirable properties of PCG methods. Similarly to the agent-based approaches used in dungeon generation (Section 3.3), agent-based approaches for landscape creation grow landscapes through the action of one or more software agents. An example is the agent-based procedural city generation demonstrated by Lechner et al. [9]. In this work, cities are divided into areas (such as squares, industrial, commercial, residential, etc.) and agents construct the road networks. Different types of agents do different jobs, such as *extenders*, which search for unconnected areas in the city, and *connectors*, which add highways and direct connections between roads with long travel times. Later versions of this system introduced additional types of agents, for tasks such as constructing main roads and small streets [8].

But since this chapter is about terrain generation, we'll look now at work on agent-based terrain generation by Doran and Parberry [2], which focuses primarily on the issue of controllability, especially on providing more control to a designer than the dominant fractal-based terrain generation methods do. Because of the lack of input and interaction with designers, fractal-based methods are usually evaluated in term of efficiency rather than the aesthetic features of the terrains generated [2]. Agent-based approaches, on the other hand, offer the possibility of defining more fine-grained measures of the *goodness* of the terrains according to the behaviour of

the agents. By controlling how and how much the agent changes the environment, one can vary the quality of the generated terrains.

4.4.1 Doran and Parberry's terrain generation

Doran and Parberry's terrain-generation approach starts with five different types of agents that work concurrently in an environment to simulate natural phenomena. The agents are allowed to sense the environment and change it at will. Designers are provided with a number of ways to influence terrain generation: controlling the number of agents of each type is one way to gain control, another is by limiting the agent lifetime using a predefined number of actions that the agent can perform. After the number of steps is consumed, the agent becomes inactive.

The agents can modify the environment by performing three main tasks:

- Coastline: in this phase, the outline or shape of the terrain is generated using multiple agents.
- Landform: the detailed features of the land are defined in this phase employing more agents than were used in the previous phase. The agents work simultaneously on the environment to set the details of the mountains, create beaches and shape the lowlands.
- Erosion: this is the last phase of the generation and it constitutes the creation of rivers through eroding the previously generated terrain. The number of river to create is determined by the number of agents defined in this phase.

According to these phases, several types of agents can be identified to achieve the several tasks defined in each phase. The authors focused their work on five different types:

1. Coastline agents: these agents work in the coastline phase before any other agents, to draw the outline of the landscape. The map is initially placed under sea level and the agents work by raising points above sea level. The process starts with a single agent working on the entire map. Depending on the size of the map, this agent multiplies by creating many other coastline agents which subdivide themselves in turn until each agent is assigned a small part of the map. The process undertaken by each agent to generate the coastline can be described as follows:

    - Each agent is assigned a single seed point at the edge of the map, a direction to follow and a number of tokens to consume.
    - The agent checks its surroundings and if it is already land (this might happen since all the agents are working simultaneously on the map) the agent starts searching in the assigned direction for another appropriate starting point.
    - Once the starting point is located, the agent starts working on the environment by changing the height of the points. This is done by

a. generating two points at random in different directions: one works as an attractor and the other as a repulser.
b. identifying the set of points for elevation above the sea level.
c. scoring the points according to their distance from the attractor and the repulser points. The ones closer to the attractor are scored higher.
d. the point with the highest score is then elevated above sea level and it becomes part of the coastline.
e. the agent then continues by moving to another point in the map.

This method allows multiple agents to work concurrently on the map while preserving localization since each agent moves in its surroundings and has a predefined number of tokens to consume. The number of tokens given to each agent and the number of agents working on the map are directly related. The smaller the number of tokens, the larger the number of agents since more agents will be required to cover the whole map. These parameters also affect the level of detail of the coastline. A map generated with a small number of tokens will feature more fine details than one with a large number, since in the first case more agents will be created, each influencing a small region.

2. Smoothing agents: after the shape of the landscape has been defined by the coastline agents, smoothing agents operate on the map to eliminate rapid elevation changes. This is done by creating a number of agents each assigned a single parameter specifying the number of times that agent has to revisit its starting point. The more visits, the smoother the area around this point.
   The agents are scattered around the map, they move randomly and while wandering they change the heights of arbitrary points according to the heights of their neighbours. For each point chosen, a new height value is assigned taking the weighted averages of the heights of its four orthogonal surrounding points and the four points beyond these.
3. Beach agents: after the smoothing phase, the landscape is ready for the creation of sandy beaches. This is the work assigned to beach agents. These agents traverse the shoreline in random directions creating sandy areas close to water. Beach generation is controlled by adjusting the agents' parameters. These include the depth of the area the agent is allowed to flatten, the total number of steps the agents can move, the altitude under which the agents are permitted to work and the range of height values they can assign to the points they affect.
   The agents are initially placed in a coastline area where they work on adjusting the height of their surrounding points by lowering them as long as their height is below the predefined altitude. This prevents the elevation of mountain areas located close to the sea. The new values assigned to the points are randomly chosen from the designer-specified range. This allows the creation of flat beaches if the range is narrow and more bumpy beaches when the range is high.
4. Mountain agents: The coastline agents elevate areas of the map above sea level. These areas are then smoothed by the smoothing agents and beaches along the shoreline are then flattened via the beach agents. Regions above a certain thresh-

old are kept untouched by the beach agents, and these are then modified by mountain agents.

The agents are placed at random positions in the maps and are allowed to move in random directions. While moving, if a V-shaped wedge of points is encountered, the wedge is elevated, creating a ridge. Frequently, the agents might decide to turn randomly within 45 degrees of their initial course, resulting in zigzag paths. Mountain agents also periodically produce foothills perpendicular to their movement direction.

The shape of the mountains can be controlled by designers via specifying the range of the rate at which slopes can be dropped, the maximum mountain altitude and the width and slope of the mountain. Designers can also determine the number of agents, the number of steps each one can perform, the length of foothills and their frequency.

After mountain generation, a smoothing step is followed to blend nearby points. This step is further followed by an addition of noise to regain some of the details lost while smoothing.

5. Hill agents: these agents work in a similar way to the mountain agents but they have three distinctive characteristics: they work on a lower altitude, they are assigned smaller ranges, and they are not allowed to generate foothills.
6. River agents: in the final phase of terrain generation, river agents walk through the environment digging rivers near mountains and the ocean. To resemble natural rivers, a river agent works in the following steps:

    a. initiate two random points, one on the coastline and another on the mountain ridge line.
    b. starting at the coastline, the agent moves uphill towards a mountain, guided by the gradient. This determines the general path of the river.
    c. as the agent reaches the mountain, it starts moving downwards while digging the river. This is done by lowering a wedge of terrain, following a similar method to the one implemented by mountain agents.
    d. the agent increases the width of the wedge as it moves back towards the ocean.

    Designers specify the initial width of the river, the frequency of widening and the downhill slope. Designers also determine the shortest length possible for a river. A river agent might make several attempts to place its starting and ending positions before it satisfies the shortest-length threshold. If this condition is not met after several attempts, the river will not be created.

The method followed for defining the agents and their set of parameters allows the generation of endless variations of terrains through the use of different random seed numbers. The technique can be used to generate landscapes on the fly, or it can be employed by designers who can investigate different setups and tweak the system's parameters as desired.

4.5 Search-based landscape generation

We have seen two families of methods for terrain and noise generation, which both have several benefits. However, at least in the form presented here, these methods suffer from a certain lack of controllability. It is not easy to specify constraints or desirable properties, such that there must be an area with no more than a certain maximum variation in altitude, or that two points on a terrain should be easily reachable from each other. This form of controllability is one of the strengths of search-based methods. Unsurprisingly, there have been several attempts to apply search-based methods to terrain generation.

4.5.1 Genetic terrain programming

Frade et al. developed a concept called genetic terrain programming (GTP). This is a search-based method with an indirect encoding, where the phenotype representation is a heightmap but the genotype representation is an *expression tree* evolved with genetic programming [5, 6, 4].

Genetic programming is a method for creating runnable programs using evolutionary computation [12]. The standard program representation in genetic programming is an expression tree, which in the simplest case is nothing more than an algebraic expression in prefix form such as $(+3(*52))$ (written in infix form as $3+5*2$). This can be visualized as a tree with the $+$ sign as the root node, and the 3 and $*$ in separate branches from the root. The plus and multiplier are arithmetical functions, and the constants are called *terminals*. In genetic programming, a number of additional functions are commonly employed, including if-then-else, trigonometric functions, max, min etc. Additional types of terminals might include external inputs to the program, random-number generators etc. The evolutionary search proceeds through adding and exchanging functions and terminals, and by recombining parts of different trees.

In GTP, the function set typically includes arithmetical and trigonometric functions, as well as functions for exponentiation and logarithms. The terminal set includes x and y location, standard noise functions (such as Perlin noise) and functions that are dependent on the distance from the centre of the map.

The core idea of GTP is that in the genotype-to-phenotype mapping, the algorithm iterates over cells in the (initially empty) heightmap and queries the evolved terrain program with the x and y parameters of each cell as input to the program. This is therefore a highly indirect and compact representation of the map. The representation also allows for infinite scalability (or zooming), as increasing the resolution or expanding the map simply means querying the program using new coordinates as inputs.

Several different evaluation functions were tried. In initial experiments, interactive evaluation was used: users selected which of several presented maps should be used for generating the next generation. Later experiments explored various direct

evaluation functions. One of these functions, accessibility, was motivated by game design: the objective was to maximise the area which is smooth enough to support vehicle movement. To avoid completely flat surfaces from being evolved, the accessibility metric had to be counterbalanced by other metrics, such as the sum of the edge length of all obstacles in the terrain. See Figure 4.2 for some examples of landscapes evolved with GTP.

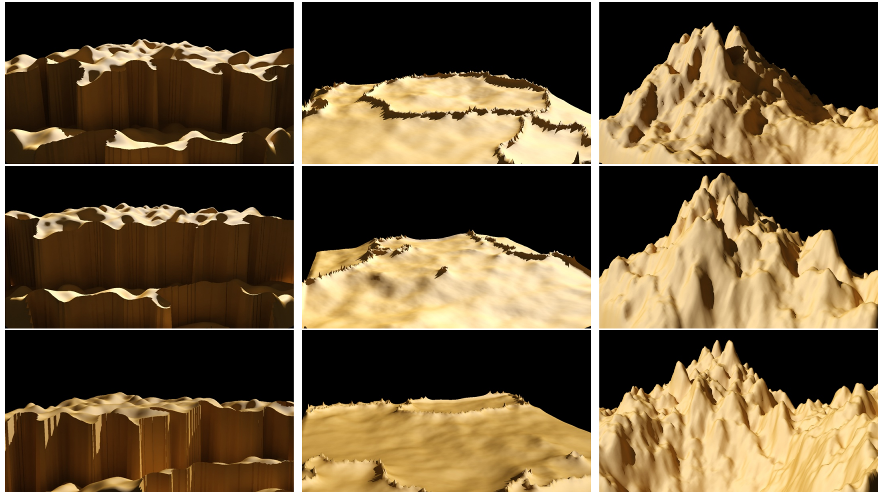

Fig. 4.2: Landscapes generated by genetic terrain programming. From left to right: cliffs, corals and mountains. Adapted from [4]

4.5.2 Simple RTS map generation

Another search-based landscape generation method was described by Togelius et al. [15], to produce a map with smoothly varying height for a real-time strategy game. The phenotype in this problem consists of a heightmap and the locations of resources and base starting locations.

The representation is rather direct. Base and resource locations are represented directly as polar coordinates (ϕ and θ coordinates for each location). The heightmap is initially flat, and then a number of hills is added. These hills are modelled as simple Gaussian distributions, and encoded in the phenotype with their x and y positions, their heights z, and their standard distributions σ_x and σ_y (i.e. their widths). Ten mountains were used in each run.

Three different evaluation functions were defined. Two of them relate to the placements of bases and resources to create a fair game, whereas the third is the topological asymmetry of the map. This is because the simplest way of satisfying

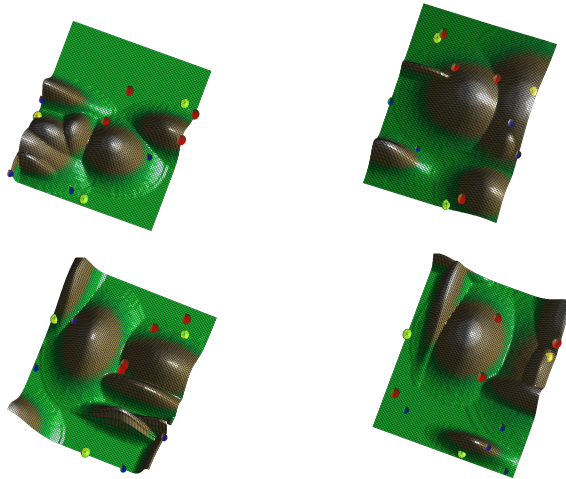

Fig. 4.3: Four maps generated using search-based methods with the heightmap determined by hills represented by Gaussians. The coloured dots represent locations of resources and bases in an RTS game. Adapted from [15]

the first two evaluation functions is to create a completely symmetric map, but this would be visually uninteresting for players. Given that the three fitness functions are in partial conflict, a multiobjective evolutionary algorithm was used to optimise all three evaluation functions simultaneously. Figure 4.3 show three different terrains that resulted from the same evolutionary run.

4.6 Lab session: Generate a terrain with the diamond-square algorithm

Implement the diamond-square method to generate terrain heightmaps. Have your function take three parameters: *seed*, which specifies the initial values at the corners; *iterations*, which specifies the number of diamond-square iterations to perform; and *roughness*, which specifies the magnitude of the random components added in the diamond and square steps.

Figure 4.4 presents three example heightmaps generated using different parameters.

Fig. 4.4: Three heightmaps generated using the diamond-square method. The parameters used are: iterations = 9 for all maps, seed = 12, 128, 128, and roughness = 256, 256, 128 for the first, second and third map, respectively

4.7 Summary

Terrain/landscape generation is a very important task for many games, and there are a number of methods that are commonly used for this. The most basic representation is the heightmap, where the number in each cell represents the height of the ground at the corresponding location. Maps can be generated very simply by randomizing these numbers, though this leads to unnatural and ugly maps. Interpolating between these numbers helps a lot. Many different interpolation techniques exist, and there is a tradeoff between the quality of the results and the computation time needed. Instead of generating the height values, another family of methods generates the gradients of slopes and then computes heights from those slopes. Fractal methods, including various types of noise, generate heights at several different scales or resolutions, leading to more natural-looking terrain. The diamond-square algorithm is a commonly used fractal method. For more complex environments, agent-based methods can be used to construct terrains that have multiple types of features. If there are constraints involved, for example having to do with traversability or other gameplay considerations, search-based methods might be useful as well.

References

1. Belhadj, F.: Terrain modeling: A constrained fractal model. In: Proceedings of the 5th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa, pp. 197–204 (2007)
2. Doran, J., Parberry, I.: Controlled procedural terrain generation using software agents. IEEE Transactions on Computational Intelligence and AI in Games **2**(2), 111–119 (2010)
3. Ebert, D.S., Musgrave, F.K., Peachey, D., Perlin, K., Worley, S.: Texture and Modeling: A Procedural Approach, 3rd edn. Morgan Kaufmann (2003)
4. Frade, M., de Vega, F.F., Cotta, C.: Modelling video games' landscapes by means of genetic terrain programming: A new approach for improving users' experience. In: Applications of Evolutionary Computing, pp. 485–490 (2008)
5. Frade, M., de Vega, F.F., Cotta, C.: Evolution of artificial terrains for video games based on accessibility. Applications of Evolutionary Computation pp. 90–99 (2010)
6. Frade, M., de Vega, F.F., Cotta, C.: Automatic evolution of programs for procedural generation of terrains for video games. Soft Computing **16**(11), 1893–1914 (2012)

7. Kamal, K.R., Uddin, Y.S.: Parametrically controlled terrain generation. In: Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia, pp. 17–23 (2007)
8. Lechner, T., Ren, P., Watson, B., Brozefski, C., Wilenski, U.: Procedural modeling of urban land use. In: ACM SIGGRAPH 2006 Research posters, p. 135. ACM (2006)
9. Lechner, T., Watson, B., Wilensky, U.: Procedural city modeling. In: Proceedings of the 1st Midwestern Graphics Conference (2003)
10. Mandelbrot, B.B.: The Fractal Geometry of Nature. W.H. Freeman (1982)
11. Musgrave, F.K., Kolb, C.E., Mace, R.S.: The synthesis and rendering of eroded fractal terrains. In: Proceedings of SIGGRAPH 1989, pp. 41–50 (1989)
12. Poli, R., Langdon, W.B., McPhee, N.F.: A Field Guide to Genetic Programming (2008). http://www.gp-field-guide.org.uk
13. Schneider, J., Boldte, T., Westermann, R.: Real-time editing, synthesis, and rendering of infinite landscapes on GPUs. In: Proceedings of the International Symposium on Vision, Modeling and Visualization, pp. 145–152 (2006)
14. Smelik, R.M., De Kraker, K.J., Tutenel, T., Bidarra, R., Groenewegen, S.A.: A survey of procedural methods for terrain modelling. In: Proceedings of the CASA Workshop on 3D Advanced Media in Gaming and Simulation (3AMIGAS) (2009)
15. Togelius, J., Preuss, M., Yannakakis, G.N.: Towards multiobjective procedural map generation. In: Proceedings of the 2010 Workshop on Procedural Content Generation in Games (2010)
16. Zhou, H., Sun, J., Turk, G., Rehg, J.M.: Terrain synthesis from digital elevation models. IEEE Transactions on Visualization and Computer Graphics **13**(4), 834–848 (2007)