Controlled Procedural Terrain Generation Using Software Agents

Jonathon Doran and Ian Parberry

***Abstract*—Procedural terrain generation is used to create landforms for applications such as computer games and flight simulators. While most of the existing work has concentrated on algorithms that generate terrain without input from the user, we explore a more controllable system that uses intelligent agents to generate terrain elevation heightmaps according to designer-defined constraints. This allows the designer to create procedural terrain that has specific properties.**

***Index Terms*—Agents, procedural content generation, terrain.**

I. Introduction

PLAYER demand for more content is increasing as games grow in complexity and scope. Content such as terrain plays a fundamental role in certain types of games [1], and contributes greatly to replayability [2]. Traditionally, developers have handcrafted content for games, requiring a substantial investment of time and resources. Procedural content generation is the automated generation of these assets for games [3]. The desire for automatic terrain generation stems from the goal of providing the player with novel content without a large investment of developer resources. Prior work has shown that this generation is feasible (see, for example, [4]–[6]).

We feel that procedurally generated content should ideally have the following properties:

- novelty: contains an element of randomness and unpredictability;
- structure: is not merely random noise, but contains larger structures;
- interest: has a combination of randomness and structure that players find engaging;
- speed: can be quickly generated;
- controllability: can be generated according to a set of natural designer-centric parameters.

With this in mind, we present a terrain generator based on software agents. This generator creates heightmaps that are grids containing elevation data at regular points on the map. While our generator could generate arbitrary size maps, the examples presented in this paper were all 262 144-vertex, 512 $\times$ 512 vertex maps. The size of a rendered scene depends on the distance between grid points, and in our case, we scaled the terrain so that grid points were approximately 1 m apart, giving a total area of about 1/4 km^2.

Novelty in generated terrain leads to enhanced gameplay, since the player will need to explore the terrain [2]. As terrain agents make use of a pseudorandom number generator, each seed provided to the generator will correspond to a distinct map that adheres to the constraints provided by the designer. The generator may be executed again with the same seed at a later time and will produce the same heightmap, or another seed that will produce a different heightmap with similar features. This repeatability is desirable when developing applications, and the seed also provides an ultra compact representation of the heightmap. The heightmaps generated from different seeds will appear significantly different to the player to the degree that they will feel that they have not seen this area before.

While a procedural terrain generator should create unique maps, which means using enough randomness to avoid duplicating identifiable map features, at the same time this randomness should not lead to a loss of internal *structure*. Structure in terrain means that the map makes sense and is internally consistent, as opposed to looking as if a random set of features were placed on the map. Our hypothesis is that this can be achieved by requiring agents to work within constraints, rejecting any putative changes that fail to meet these constraints.

The definition of *interesting* is another loosely defined term that is related to but not necessarily identical to novelty and structure. A map may be novel and structured but fail to inspire the player, and likewise a map may be interesting but not novel. While we make no attempt to define what makes a map interesting, we note that this is an emergent property that can be influenced by the parameters and constraints provided to the agents. We suggest that one's interest in a heightmap is a function of anticipated utility, as well as variety of landscape features.

The idea of terrain generation *speed* brings to mind the definition of real time. As with a real-time system, we define "quickly" as "fast enough that the results are ready when needed." For some applications, this may require that terrain be generated in a fraction of a second, while in other applications having terrain ready in a minute or two is sufficient. Our example uses 512 $\times$ 512 heightmaps, which took on the order of 20 s to create using a 3.2-GHz Pentium 4.

Finally, there is the issue of *controllability*. A terrain generator with natural feature-based parameters will enable game companies to generate terrain using an employee who is an artist or designer, and not necessarily an engineer or a programmer. One can posit a simple design tool with tabs and sliders that

Manuscript received April 23, 2009; revised August 24, 2009; accepted April 13, 2010. Date of publication April 26, 2010; date of current version June 16, 2010.

The authors are with the Department of Computer Science and Engineering, University of North Texas, Denton, TX 76203-5017 USA (e-mail: jhd@unt.edu; ian@unt.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCIAIG.2010.2049020

allows the designer to create random terrains with a specific look and feel for a particular game or game genre. The designer should not necessarily need to understand the underlying mathematics of fractals to produce a range of custom terrains.

The main body of this paper is divided into three sections. Section II consists of a short literature survey. Section III begins with discussion of the agent-based approach in general, our use of agents to generate different terrain features, and has subsections describing the five different types of terrain agent that we used in our proof-of-concept implementation: coastline, smoothing, beach, mountain, and river agents. Section IV describes our implementation and discusses our evaluation methods.

II. Previous Work

Previous work on generating terrain has mostly been based on the use of fractal techniques. Most of the existing systems have a very small number of designer selectable parameters, and as a result provide the designer with limited opportunities to control what type of terrain is generated. The aim of this paper is to allow the designer more (and more natural) control over the generated terrain without sacrificing too much of the other four desirable attributes of novelty, structure, interest, and speed discussed in Section I.

Olsen [7] discusses erosion algorithms that use fractal Brownian motion ($1/f$ noise), and perturbed Voronoi diagrams. He provides some metrics for evaluating terrain (such as low average height and a high standard deviation for slope) which are used to compute a game suitability score used to evaluate generated terrains. Musgrave *et al.* [8] use noise synthesis to create eroded fractal terrain. Control of this algorithm appears to be limited to modulation of the noise frequency, in particular, scaling and translation of values. Control does not appear to be feature based, that is, designers do not express their desires in terms of terrain features. Evaluation of their technique was limited to the efficiency of generation, not the quality or ease of use.

Szeliski and Terzopoulos [9] address fractal terrain generation in one section of their paper. Their approach uses real digital elevation data which they perturb using splines. Fractals are used to add in detail to the resulting heightmaps. The character of the output can be controlled by the initial elevation data, the functions used to determine the spline shapes, and the amount of noise used by the fractal generation. Van Pabst and Jense [10] generate terrain by creating a multifractal field based on four parameters. A separate utility is used to analyze existing digital elevation data and extract these four parameters. Belhadj and Audibert [11] create ridge and river networks and then use fractals to transform this data into a complete heightmap. The ridge and river network are created by randomly depositing particles and allowing them to interact with each other and the terrain. As with most fractal-based approaches, their algorithm does not appear to be controllable. Pi *et al.* [12] create fractal landscapes using Perlin noise. They do not indicate any method of controlling the quality of the generated terrain other than a scaling term applied to the noise.

Ong *et al.* [13] apply genetic algorithms to terrain generation, and also focus on controllability. Their approach uses a sketch of the boundary which could be either designer provided or machine generated. A database of representative heightmap samples appears to be the main form of control for their algorithm. Frade *et al.* [14] present another genetic technique which they claim is more controllable than fractal techniques.

Li *et al.* [15] use machine learning to model example heightmaps and then later use these models to synthesize new terrain. This requires that a suitable set of training examples be created for each desired output archetype. One issue with machine learning is that it is often not clear what is being learned, that is to say that one does not necessarily have control over what features in the training data are important.

Kamal and Uddin [16] present a technique that is minimally feature controllable. This technique repeatedly creates random polygons using a series of randomly placed lines and performs random walks that raise the points in these polygons. We believe this technique can be equated to Brownian motion, with the main difference being the order in which the individual points are changed. This technique offers limited control, as one is able to create multiple mountains and mountain ranges.

Lechner *et al.* [17] use an agent-based approach to determine urban land use in a growing city. They share with us a focus on producing a tool for the nontechnical designer, in this case artists and geographers. Their solution was controllable, but the parameter choices seemed rather abstract and suggested that designers would need to learn the effects through experimentation rather than intuition.

Our approach differs from these papers in that the input to our generator is a set of parameters that describe the quantity and quality of familiar terrain features, such as mountains and rivers. Once such a set of parameters is defined, the generator can create variations on that theme without further input from the designer. Only one of the above papers (see [16]) has any type of feature-level control, and some of the previous work (including [10], [13], and [15]) was example based, requiring the designer to provide examples of the types of output desired.

III. Software Agents

While there currently is no consensus on what a software agent is (see, for example, [18]), we may start with Russell and Norvig's definition [19] of an agent that perceives its environment through sensors and acts on it through effectors. All agents are given autonomy, other than some constraints on what the agent is to produce. Using Russell and Norvig's list of agent properties, our agents: are accessible, are deterministic in the results of their effectors, are nonepisodic, work in a dynamic environment, and work in a discrete environment.

Our agents run in three distinct phases, which can be loosely summarized as *coastline*, *landform*, and *erosion*. In the first or coastline phase, a large number of agents work to generate the outline of a landmass, possibly surrounded by water. In the second or landform phase, a larger assortment of agents work to define the features of the map (raising mountains, shaping the lowlands, creating beaches). In the third or erosion phase, rivers are added by eroding parts of the terrain. Agents run concurrently and asynchronously within each phase.

Agents are allowed to see the current elevation of any point on the map, and are allowed to modify these points at will.

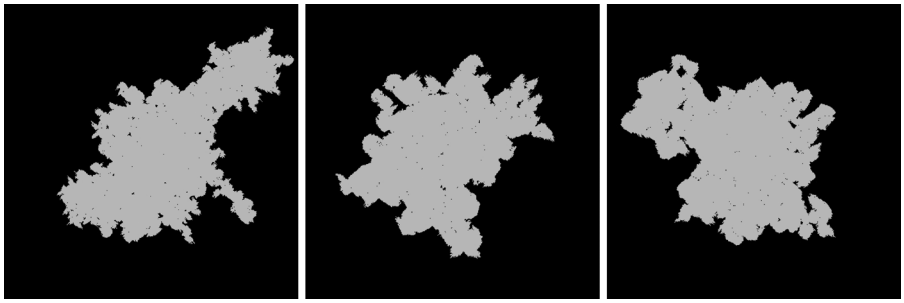

Fig. 1. Coastlines produced by coastline agents with (left to right) small, medium, and large action sizes.

The presence of other agents will likely cause the environment to change around them without warning. Observing the height (and derived values such as being on a coastline) and modifying heights are the extent of an agent's actions and percepts. These observations are analogous to sensory perceptions, and the modification is analogous to the use of effectors.

Each agent when it is created receives a number of tokens that it will consume as discrete actions are taken. This limits the lifetime of an agent, and is one way in which the terrain generation can be influenced by the designer. Macro features of the map may be controlled by the designer by specifying the number of each agent type to run and the number of tokens each agent receives. For example, since each river agent produces one river the number of rivers is altered by raising or lowering the number of river agents. We provide a default set of agents and tokens based on the map size. This default set can be overridden by the designer to create different types of terrain.

As a proof-of-concept, we employ five different types of agents, although it should be kept in mind that many more agent types are possible. Our five agent types are as follows:

1) *coastline agents* create a landmass, possibly surrounded by water;
2) *smoothing agents* perform random walks, averaging the height of nearby points;
3) *beach agents* produce flat areas on parts of the coastline;
4) *mountain agents* raise mountain chains;
5) *river agents* erode terrain producing rivers that run from mountains to the ocean.

Each of these agent types is treated in a separate subsection below.

A. Coastline Agents

Coastline agents create the outline of the landmass before any heightmap data are calculated. Coastline generation starts with a single agent that is responsible for the entire landmass. This agent subdivides the task by creating multiple subordinate coastline agents and assigning them part of the landmass to work on. These in turn subdivide, and the process repeats until each agent is working with a small number of vertices (on the order of 1000). Each of these agents works independently of other agents, but all agents have access to the environment. In this case, the environment consists of the landmass in whatever state of development it may be in when the agent is running. Agents modify the environment by raising points above sea level (initially the entire map is below sea level).

Each agent starts with a single seed point (on the edge of the landmass), a preferred direction, and the number of points the agent is expected to generate. If for some reason the seed point is surrounded by land by the time the agent begins executing, the agent begins searching in its preferred direction for a coastline point. Once the agent places itself on the coastline it creates two points at random: one is an attractor point and the other is a repulsor. These must be in different directions. When the agent evaluates candidate points for elevation above sea level, points closer to the attractor point are scored higher by the agent, and points closer to the repulsor are scored lower.

The score for a point p is given by $d_r(p) - d_a(p) + 3d_e(p)$, where $d_a(p)$ is the square of the distance from p to the attractor, $d_r(p)$ is the square of the distance from p to the repulsor, and $d_e(p)$ is the square of the closest distance from p to the edge of the map. These terms encourage an agent to move towards the attractor, move away from the repulsor, and to avoid the map edges as much as possible. Since each agent randomly selected its attractor and repulsor points, each agent would score a given point differently.

The agent expands the landmass by adding points to the edges of the mass. It calculates a score for all surrounding points that are not part of the landmass, and moves to the highest scoring point. This point is then elevated above sea level, becoming part of the coastline.

Note that multiple agents are running concurrently, and that each agent will have unique goals and scoring biases. Agents will elevate one point above sea level, and then warp to another part of the coastline they are working on. This causes agents to operate at random, but in a local area of the map. The goal is to give map features some localized consistency, rather than adding points at random to the coastline. This effect is seen by the presence of capes that have smaller capes. The higher level agent will bias its children, who will in turn bias their children to create smaller features.

Each coastline agent has two configurable parameters that collectively define the range of vertices where an agent will be elevating points rather than subdividing its task. Fig. 1 shows the shape of three landmasses created by varying these parameters. We used the same seed for the random number generator in each of these three runs so that variations in coastline shape are due only to changes in agent actions. We requested 150–220 vertex actions for the landmass on the left, 500–550 vertex actions for the landmass in the middle, and 950–1000 vertex actions for the landmass on the right. In general, a lower vertex range will

result in more agents operating on smaller regions, and as a result the coastline will have more fine detail than a landmass that had a larger vertex range. A landmass created with a high vertex range will have larger, wider features and will tend to be more circular, as is the case with the landmass on the right of Fig. 1.

COASTLINE-GENERATE*(agent)*

```
1 if tokens(agent) ≥ limit
2   then
3     create two child agents
4     for each child
5       do
6         child ← a random seed point on parent's border
7         child ← 1/2 of the parent's tokens
8         child ← a random direction
9         COASTLINE-GENERATE(child)
10  else
11    for each token
12      do
13        point ← random border point
14        for each point p adjacent to point
15          do
16            score p
17        fill in the point that has the highest score
```

B. Smoothing Agents

Smoothing agents make random walks around the map adjusting the height of an arbitrary point p to be the average of points in an extended von Neumann neighborhood of p consisting of the four orthogonal map points surrounding p on the elevation grid and the four points beyond these (see [20]). A weighted average height is calculated, with the center point given three times the weight of the other points. Therefore, nine points with a total weight of 11 are used. This provides some inertia to prevent elevations from rapidly changing. We believe that the extended neighborhood is responsible for the emergence of "interesting" curved features on the map. Use of an 8-cell Moore neighborhood resulted in less "interesting" results.

Each smoothing agent returns to its point of origin periodically. This encourages smoothing agents to operate in a local area, which is useful when certain features of the map need more smoothing than others.

The only configurable parameter for smoothing agents is the number of times that the agent will return to its start point. Setting this number to a large value causes the agent to spend most of its time near the start point. This provides a great deal of smoothing for that area, rather than less smoothing spread over a larger area in the case where the agent is allowed to wander further away.

SMOOTH*(starting point)*

```
1 location ← starting point
2 for each token
3   do
4     height_location ← weighted average of neighborhood
5     location ← random neighboring point
```

C. Beach Agents

Beach agents create flat sandy areas next to the main coastline after the coastline agents have finished. Before they begin, points on the main coastline are identified using breadth-first search. Beach agents then use these points to place themselves on the coastline. They then perform random walks flattening areas of beach, following the shoreline. Beach agents adjust the height of the beach to allow random fluctuations in elevation so the beach is not a uniform flat space. After moving to a spot on the coastline, the agent will lower the nearby points and jump inland to perform a random walk. This creates variable sized sandy areas that can extend a short distance from the water. After the random walk is complete, the agent returns to the coastline and continues to walk along the shore. If an agent becomes stuck (for example, running into a mountain range) and is unable to continue its walk, it moves to another randomly chosen point on the main coastline and continues. Beach agents avoid high areas, so any mountains that are next to the ocean are left alone.

One of the more important parameters for the beach agent is the altitude limit, above which the agent abandons an area and moves elsewhere. When the altitude limit is low, the raised area near the middle of the beach is allowed to remain. When the altitude limit is high, the agent is able to continue its work in this area and flattens the mound.

Beach agents set the height of the beach to random values within a specified range specified by the designer. When this range is narrow, flat beaches are created. When it is raised a bit, we see more bumps. The designer can also control the width of a beach by indicating how far inland the beach agents should begin flattening, and how long their random walk should be. Fig. 2 shows the effects of varying the width of a beach.

BEACH GENERATE*(starting point)*

```
1 location ← starting point
2 for each token
3   do
4     if height_location ≥ limit
5       then
6         location ← random shoreline point
7     flatten area around location
8     smooth area around location
9     inland ← random point a short distance inland from
      location
10    for i ← 0 to size(walk)
11      do
12        flatten area around inland
13        smooth area around inland
14        inland ← random neighboring point
15    location ← random point adjacent to location
```

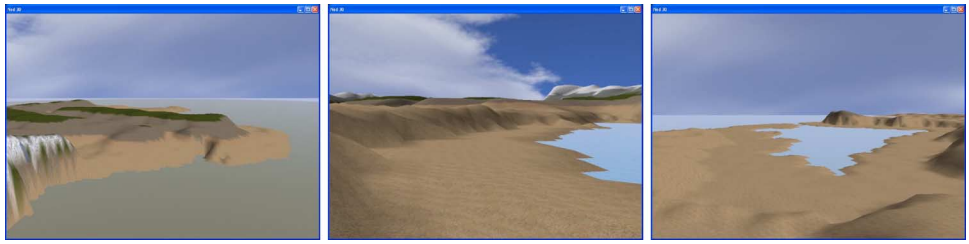

Fig. 2. Beaches produced by beach agents with (left to right) small, medium, and large beach width.

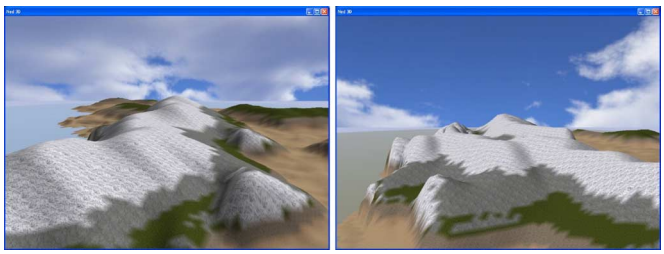

Fig. 3. Mountains with (left) narrower features and (right) with wider features.

D. Mountain Agents

Mountain agents raise mountain ranges. Each starts at a random point on land and selects a preferred direction of travel. As a mountain agent moves in this direction it raises an inverted V-shaped wedge of points with the center line becoming the ridge line. The agent will move along this ridge and will periodically decide to change direction within a 90° cone from its original direction. The effect is that the agent zig-zags but heads generally in the same direction. If an agent runs into the ocean or the map edge, it changes direction to avoid this obstacle.

The width of the V-shaped wedge determines the general width of the mountains, and to a large degree the slope of the mountain sides. The rate at which the slope drops in elevation is randomly determined for each wedge (within a designer-specified range), which produces some interesting features on the sides of the mountains. Mountain agents also periodically create foothills running perpendicular to the mountain range axis. Smoothing is performed on the mountain after the wedge is raised, blending the heights and leaving gentler transitions between nearby points.

Prior terrain generators have used other techniques for creating mountains, such as fault generation [21], [22], fractal midpoint displacement [23], and point deposition [7], [24], [25]. While we make no explicit attempt to simulate faults, our mountain agent's terrain elevation is similar, with the major difference being that the mountain agent determines its path as it operates, avoiding obstacles in its way, whereas fault simulators determine the fault's position prior to modifying the landscape.

The mountain agent's simplistic wedge raising produces acceptable results, mainly due to the interaction of the smoothing agents that are making random walks over the terrain. Fig. 3 shows the effects of widening a mountain and increasing its foothill length.

Mountain agents are the most configurable of all agents, as they introduce most of the interesting features on a landscape. Without them the heightmap would be mostly flat. The designer determines the number of mountain agents that will run, and specifies how many tokens each mountain agent will receive. A single mountain agent will randomly position itself on the map, decide on direction, and begin elevating terrain. It stops when it runs out of tokens or is unable to proceed due to some obstacle. Mountain agents attempt to turn to avoid obstacles, but this ability is limited to ensure that agents do not randomly wander the map.

Mountain agents are given a maximum altitude, and vary the generated height within a specified range below this height. Mountain agents may also be assigned a width and a slope, which allows them to either spread out, or to create tall narrow ranges. While mountain agents perform smoothing, they also follow this up by adding noise to restore some of the character lost during smoothing. This noise is specified by a probability of altering a point's altitude, and a variance. When a point's altitude is modified during this roughening phase, a random value up to the variance parameter is either added or subtracted from the point's current altitude.

Mountain agents periodically generate foothills perpendicular to the range axis. The lengths of these are randomly determined from a configurable range, as is the frequency at which these foothills are created.

MOUNTAIN-GENERATE*(starting point)*

```
1 location ← starting point
2 direction ← random direction
3 for each token
4   do
5     elevate wedge perpendicular to direction
6     smooth area around location
7     location ← next point in direction
8     every nth token
9       do
10          direction ← original direction ± 45°
```

E. Hill Agents

Hill agents are a special case of the mountain agent. As with a mountain agent, the designer specifies the number of tokens assigned to each hill agent, indirectly determining the size of each hill. Hill agents generally create very short mountain ranges with a lower altitude, and no foothills, as seen in Fig. 4. Hill agents may have their altitude range determined by specifying a maximum altitude and a variance. Since hill agents are a special

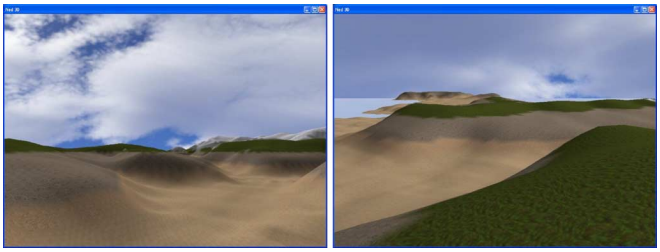

Fig. 4. Hill agents produce hills, similar to the way mountain agents produce mountains.

case of mountain agents, these parameters work exactly as they do for mountain agents.

HILL-GENERATE*(mountain)*

1 *location* ← random point at the base of *mountain*
2 *direction* ← direction away from the *mountain* centerline
3 **for each** *token*
4 **do**
5 $height_{\text{location}}$ ← weighted average of the neighborhood
6 raise a wedge perpendicular to *direction*
7 *location* ← next point in *direction*

F. River Agents

River agents create water channels between points near mountains and the ocean. Each agent creates one river. A river agent begins by choosing a random point on the coastline and a random point on a mountain ridge line. It begins at the ocean and moves in the general direction of the mountain point, following the elevation gradient uphill. This causes the river to meander, rather than creating a perfectly straight river. When the agent runs into a mountain it stops and begins to backtrack its path downstream. During this phase of the agent's operation it will lower a wedge of terrain, similar to the way mountain agents raise heights. The width of this wedge increases as the river moves downstream. This downhill phase stops if the river encounters another river, in effect making one of the rivers a tributary of the other.

The river agent avoids creating rivers that are too short, or which fail to reach the mountains. An agent makes multiple attempts to place a river, but will eventually give up if too many attempts are unsuccessful. As a result, it is possible to specify river agents and not have rivers placed. The primary cause of river agents failing to place a river is that every attempt to run a river uphill encounters a mountain before the river's minimum length is reached. This is more of a problem with smaller maps, since the mountains dominate small maps more than larger maps and are more likely to be found near coastlines. The most important parameter for a river agent is probably the minimum length. River agents attempt to place rivers on the terrain, running from a mountainous region to the ocean. The minimum length parameter causes the agent to abort attempts to place rivers smaller than this value, and to instead search for a place to create a longer river.

River length is a function of the coastal and mountain points selected, as well as the height of the terrain between these points. Rivers are created uphill, and will stop when the terrain becomes too mountainous, even if the preselected mountain point has not been reached. The alternative would be to allow the river agent to create a channel through one mountain to reach the other. The designer is able to specify the maximum altitude of the terrain allowed before the river terminates. Other altitude limits are the maximum allowed altitude on the shoreline (to prevent waterfalls at the coastline), and the minimum altitude of the mountain point (to prevent rivers from starting at foothills or on flat terrain). A minimum distance from the coastline to the mountain point may also be specified, which prevents rivers from starting near the coast but heading towards a more remote part of the coastline.

River agents require a *backoff* parameter that causes them to back away from mountains when the altitude limit is exceeded. This prevents rivers from climbing a cliff and then starting their downhill building phase. In effect, this type of behavior would allow rivers to start at the top edge of cliffs. When the agent's altitude limit is reached, the agent will back off a number of vertices along its path, and then begin cutting the river into the landscape.

River agents may optionally lower the terrain on the mountain side of the river. This causes a deeper channel to be formed, as opposed to gentler sloping river banks. Rivers are assigned an initial width, which then increases over time as the river runs downhill. This width, the frequency of widening, and the downhill slope of the cut channel may all be specified. Fig. 5 shows two generated rivers. The river on the left is a dry riverbed created by using a shallow initial altitude drop, and a shallow slope. The rivers on the right were created by specifying multiple river agents, plus a narrow target altitude range for the coastline.

RIVER-GENERATE()

1 *coast* ← random point on coastline
2 *mountain* ← random point at the base of a mountain
3 *point* ← *coast*
4 **while** *point* **not** at *mountain*
5 **do**
6 add *point* to path
7 *point* ← next point closer to *mountain*
8 **while** *point* **not** at *coast*
9 **do**
10 flatten wedge perpendicular to downhill direction
11 smooth area around *point*
12 *point* ← next point in path

IV. Implementation and Evaluation

We have implemented a framework that executes runnable agents in a random order, and for a random slice of time (within a window). We assume atomic locking exists at the vertex level. The first phase of agents, those that produce the coastline, do not share this scheduling system, but there is nothing in these agents which depend on other agents. The purpose of this framework

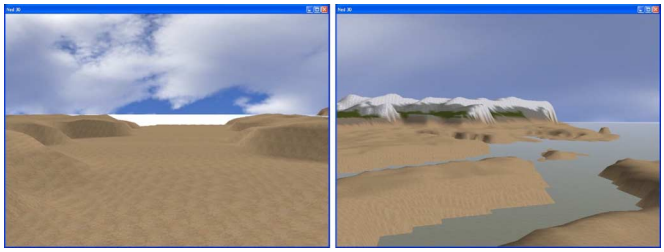

Fig. 5. (Left) River agents generated a dry river bed. (Right) Three rivers that meet at the ocean.

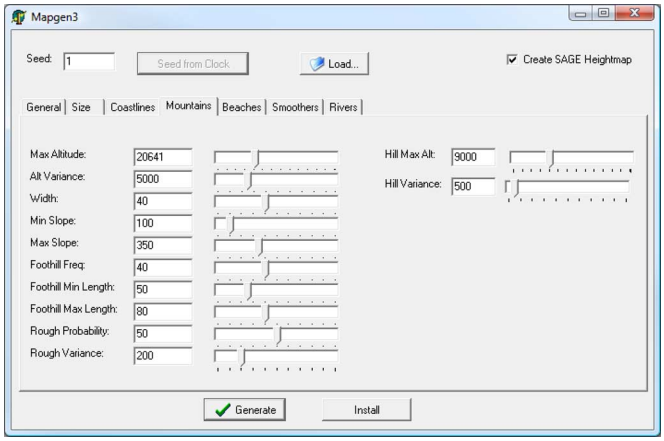

Fig. 6. Screen shot of our prototype designer interface.

is to demonstrate the independence of agents. The simplistic behavior of these agents results in complex interactions among agents, and that the terrain is an emergent result.

Our agent-based terrain generator lends itself to implementation in either a purely procedural environment, or in a designer-centric environment. In the former, a game could use our technique to generate terrain on-the-fly, guided by agent settings provided by the publisher in advance or in real time. In the latter environment, in which the publisher requires more control over the content, a designer could use our technique to generate terrains that are first screened and/or modified by a human being before being distributed. Note that unmodified terrains lend themselves to easy distribution since our terrains are uniquely specified by the parameters and the initial random number seed. We have created a prototype tool (see Fig. 6) that lets designers experiment with the various agent parameters and view the resulting terrain in 3-D using the SAGE engine [26], which was also used to create the screen shots used in this paper. Note that the parameters tend to be high-level designer-centric values that correspond in an intuitive way to terrain features.

We made no formal attempt to ask individuals to evaluate the quality of the generated terrain, as we felt that such a study was beyond the scope of this paper. We provided a list of attributes earlier which we believe any good procedural content generation system should have. We note that others have provided similar lists which we contrast with our own. For example, Ong *et al.* [13] suggested that the qualities should be as follows:

- adaptive;
- innovative;
- scalable;
- intuitive.

We believe our approach satisfies each of these traits. Ong's definition of *adaptive* is met as our use of agents allows different terrain types to be created, and new terrain types may be added by writing new agents. We believe that the *innovation* requirement is met since terrain features are an emergent property of the intra-agent interaction. Our system is *scalable* as the map size, the number of agents, and the number of tokens given to each agent may be changed. We believe our set of designer-specified parameters results in *intuitive* control of terrain generation. For example, using the graphical front end, the designer may manipulate sliders that correspond to terrain features instead of being asked to modify abstract parameters that do not directly correspond to the features of the generated terrain.

Saunders [27] suggested the following traits:

- require a low degree of human input;
- permit a high degree of human control;
- be completely intuitive to control;
- be able to generate a wide variety of recognizable terrain types and features, in believable relationship to one another;
- produce models at arbitrary levels of detail;
- run quickly enough to be used in real-type, dynamic applications;
- be extensible to support new types of terrain.

We believe our approach satisfies all but the real-time requirement. The designer is asked to select parameters for generation, and defaults exist for all parameters. As a result, the designer may generate multiple heightmaps with the same parameters without providing any further input. Since our parameters deal with geographic features rather than abstract concepts like noise, we feel that they are intuitive and accessible to a nontechnical designer. When the effect of a parameter is not obvious, we feel that it is easily learned by experimentation since the parameters are grouped by geographic feature type. Our agent-based technique allows the creation of arbitrary levels of detail by varying the size of the generated heightmap, although other parameters will need to be similarly scaled. The variety of agents in the current implementation allows for a variety of recognizable terrain types, and as new agents may be added the system is extensible.

We do not feel that real time is a necessary requirement for an ideal terrain generation system, unless one adopts a loose definition of real time. If an application has terrain available when it is needed, even if the terrain was generated offline, we believe the terrain generator has performed its job. While there are applications that require that terrain be generated as a player moves around the world, there are many successful commercial games that have shipped with pregenerated terrain. In order to preserve generality in our list of traits, we therefore dispense with a real-time requirement.

V. Conclusion

The screen shots presented represent only a small fraction of the possible heightmaps that can be generated by this system. We feel that this agent-based terrain generation system produces novel content that is influenced by the wishes of the designer. They are able to envision a desired look, and by modifying agent constraints cause the system to generate heightmap

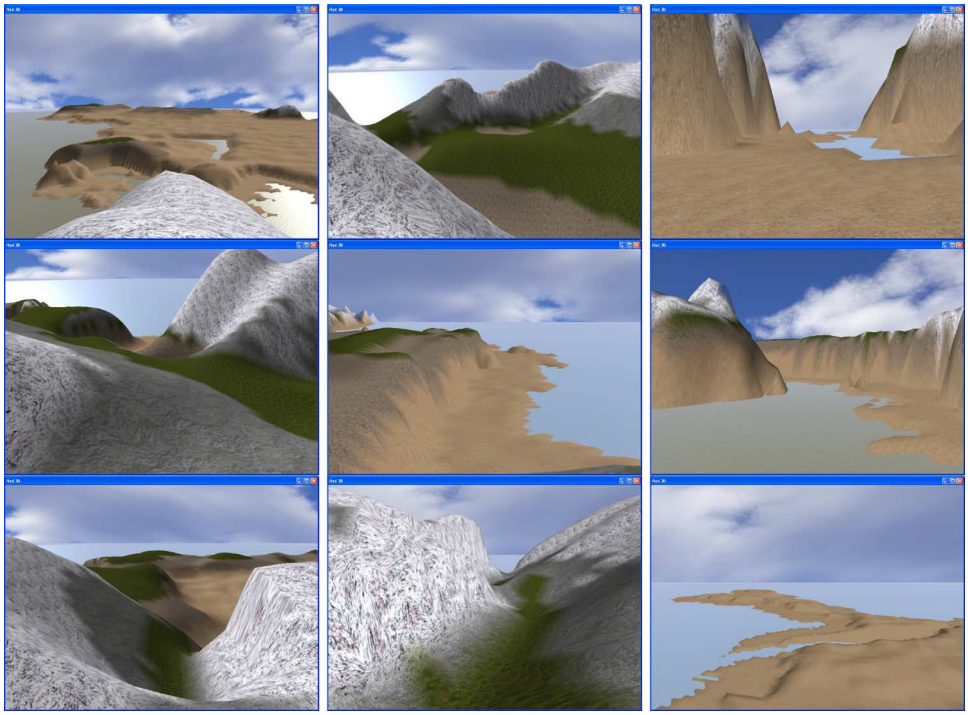

Fig. 7. Assortment of random landscapes.

after heightmap that conform to their wishes. Fig. 7 shows an assortment of random landscapes that differ considerably from each other. We are able to create flat areas, areas with gently rolling hills, and mountainous regions with passes and valleys. The coastlines have interesting shapes, and each one is unique. The reader who wishes to perform their own evaluation may download the source code from [28].

While the evaluation of generated content is subjective, and is best left to the player, we are able to make some judgments based on our experiments with this generator. Exploring a map feels like actual exploration, since we have no knowledge of what may lie around the corner. Curiosity about the environment is a factor in the motivation of players to play a game [4], and is therefore a desirable feature when creating game content.

There is a perhaps unmeasurable quality that causes us to want to play on the terrain. This may be partially a function of the landscape feeling like it belongs, and also a function of the variety of landscape features found on each map. If one wants to find an interesting bay or a hidden valley, there is likely one to be found. These features help to maintain a player's interest, and perhaps increase replayability in a game. Previous games that generated random terrain such as *Empire* or *Civilization* showed that the same set of rules when played on a different terrain can result in different gameplay. This sense of a mountain belonging on the map is an example of structural consistency. There is an overall structure to a heightmap, which the agents create and maintain.

The primary benefit of our agent-based terrain generation system is its intuitive controllability. Rather than running a generator and accepting whatever output it produces, we have introduced a high degree of controllability without requiring the designer to take an active role in the generation. One is able to dictate the qualities they desire in the terrain and have random heightmaps generated that correspond to their wishes.

The success of this approach comes not from designing highly complex agents, but rather by allowing them to interact in unpredictable, but controllable ways. Jennings and Wooldridge [29, p. 17] discuss this on a more general level, stating that "Because the behavior of individual agents is not uniquely determined at design time, the behavior of the system as a whole can only emerge at run time."

References

[1] K. Forbus, J. Mahoney, and K. Dill, "How qualitative spatial reasoning can improve strategy game AIs," *IEEE Intell. Syst.*, vol. 17, no. 4, pp. 25–30, Jul./Aug. 2002.

[2] D. Sampath, "ABRCon, Adaptive oBject Re-CONfiguration: An approach to enhance, repeat playability of games and repeat watchability of movies," in *Proc. ACM SIGCHI Int. Conf. Adv. Comput. Entertain. Technol.*, New York, 2004, pp. 313–316.

[3] M. Nelson and M. Mateas, "Towards automated game design," in *Lecture Notes in Computer Science*. Berlin, Germany: Springer-Verlag, 2007, vol. 4733, p. 626.

[4] S. Greuter, N. Stewart, and G. Leach, "Beyond the horizon," in *Proc. Image Text Sound Conf.*, 2004.

[5] M. Nitsche, C. Ashmore, W. Hankinson, R. Fitzpatrick, J. Kelly, and K. Margenau, "Designing procedural game spaces: A case study," in *Proc. FuturePlay*, 2006, pp. 10–12.

[6] J. Togelius, R. De Nardi, and S. Lucas, "Towards automatic personalised content creation for racing games," in *Proc. IEEE Symp. Comput. Intell. Games*, 2007, pp. 252–259.

[7] J. Olsen, "Realtime procedural terrain generation," IMADA, Univ. Southern Denmark, Odense, Denmark, Tech. Rep., 2004.

[8] F. Musgrave, C. Kolb, and R. Mace, "The synthesis and rendering of eroded fractal terrains," *ACM SIGGRAPH Comput. Graph.*, vol. 23, no. 3, pp. 41–50, 1989.

[9] R. Szeliski and D. Terzopoulos, "From splines to fractals," *ACM SIGGRAPH Comput. Graph.*, vol. 23, no. 3, pp. 51–60, 1989.

[10] J. Van Pabst and H. Jense, "Dynamic terrain generation based on multifractal techniques," in *Proc. Int. Workshop High Performance Comput. Comput. Graph. Vis.*, 1995, pp. 186–203.

[11] F. Belhadj and P. Audibert, "Modeling landscapes with ridges and rivers: Bottom up approach," in *Proc. 3rd Int. Conf. Comput. Graph. Interactive Tech. Australasia and South East Asia*, New York, 2005, pp. 447–450.
[12] X. Pi, J. Song, L. Zeng, and S. Li, "Procedural terrain detail based on patch-LOD algorithm," in *Lecture Notes in Computer Science*. Berlin, Germany: Springer-Verlag, 2006, vol. 3942, p. 913.
[13] T. Ong, R. Saunders, J. Keyser, and J. Leggett, "Terrain generation using genetic algorithms," in *Proc. Conf. Genetic Evol. Comput.*, New York, 2005, pp. 1463–1470.
[14] M. Frade, F. De Vega, and C. Cotta, "Breeding terrains with genetic terrain programming: The evolution of terrain generators," *Int. J. Comput. Games Technol.*, vol. 2009, 2009.
[15] Q. Li, G. Wang, F. Zhou, X. Tang, and K. Yang, "Example-based realistic terrain generation," in *Lecture Notes in Computer Science*. Berlin, Germany: Springer-Verlag, 2006, vol. 4282, p. 811.
[16] K. R. Kamal and Y. S. Uddin, "Parametrically controlled terrain generation," in *Proc. 5th Int. Conf. Comput. Graph. Interactive Tech. Australia and Southeast Asia*, New York, 2007, pp. 17–23.
[17] T. Lechner, P. Ren, B. Watson, C. Brozefski, and U. Wilenski, "Procedural modeling of urban land use," in *Proc. SIGGRAPH Res. Posters*, New York, 2006, p. 135.
[18] I. Rudowsky, "Intelligent agents," *Commun. Assoc. Inf. Syst.*, vol. 14, pp. 275–290, 2004.
[19] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Englewood Cliffs, NJ: Prentice-Hall, 1995.
[20] L. B. Kier, P. G. Seybold, and C.-K. Cheng, "Cellular automata," in *Cellular Automata Modeling of Chemical Systems: A Textbook and Laboratory Manual*. New York: Springer-Verlag, 2005, pp. 9–38.
[21] G. W. Lecky-Thompson, "Real-time realistic terrain generation," *Game Programming Gems*, vol. 1, pp. 484–498, 2000.
[22] J. Shankel, "Fractal terrain generation-fault formation," in *Game Programming Gems*. Brookline, MA: Charles River Media, 2000, pp. 499–502.
[23] J. Shankel, "Fractal terrain generation-midpoint displacement," in *Game Programming Gems*. Brookline, MA: Charles River Media, 2000, pp. 503–507.
[24] N. H. Anh, A. Sourin, and P. Aswani, "Physically based hydraulic erosion simulation on graphics processing unit," in *Proc. 5th Int. Conf. Comput. Graph. Interactive Tech. Australia and Southeast Asia*, New York, 2007, pp. 257–264.
[25] J. Shankel, "Fractal terrain generation-particle disposition," in *Game Programming Gems*. Brookline, MA: Charles River Media, 2000, pp. 508–511.
[26] I. Parberry, J. Nunn, J. Scheinberg, E. Carson, and J. Cole, "SAGE: A simple academic game engine," in *Proc. 2nd Annu. Microsoft Acad. Days Game Develop. Comput. Sci. Edu.*, 2007, pp. 90–94.
[27] R. Saunders, "Terrainosaurus: Realistic terrain synthesis using genetic algorithms," M.S. thesis, Dept. Comput. Sci. Eng., Texas A&M Univ., College Station, TX, 2007.
[28] Agent based terrain generation," 2010 [Online]. Available: http://www.eng.unt.edu/ian/research/terrain
[29] N. R. Jennings and M. Wooldridge, "Applications of intelligent agents," in *Agent Technology: Foundations, Applications, and Markets*, N. Jennings and M. Wooldridge, Eds. Secaucus, NJ: Springer-Verlag, 1998, ch. 1, pp. 3–28.

Jonathon Doran received the B.S. degree in computer science from the University of Colorado at Boulder, Boulder, in 1998. Currently, he is working towards the Ph.D. degree in computer science and engineering at the University of North Texas, Denton.

His research interests are procedural content generation, agent-based simulations, and AI techniques for role playing games.

Ian Parberry received the Ph.D. degree in computer science from the University of Warwick, Coventry, U.K., in 1984.

Currently, he is a Professor at the Department of Computer Science and Engineering, University of North Texas, Denton. He is a pioneer of game programming in academia, having taught game programming classes since 1993. He is the author of six books, three of them on game programming, and more than 70 articles on a wide range of computing subjects including (in addition to game programming) algorithms, complexity theory, parallel computing, and neural networks.