Entertainment Computing 27 (2018) 128–136

Contents lists available at ScienceDirect

Entertainment Computing

journal homepage: www.elsevier.com/locate/entcom

Procedural feature generation for volumetric terrains using voxel grammars

Rahul Dey[a,b,*], Jason G. Doig[a], Christos Gatzidis[b]

[a] Sony Interactive Entertainment Euro R&D, 13 Great Marlborough Street, London W1F 7HP, UK
[b] Bournemouth University, Poole House, Talbot Campus, Poole, Dorset BH12 5BB, UK

ARTICLE INFO

2010 MSC:
00-01
99-00

Keywords:
Procedural generation
Terrain
Voxels
Grammar

ABSTRACT

Terrain generation is a fundamental requirement of many computer graphics simulations, including computer games, flight simulators and environments in feature films. There has been a considerable amount of research in this domain, which ranges between fully automated and semi-automated methods. Voxel representations of 3D terrains can create rich features that are not found in other forms of terrain generation techniques, such as caves and overhangs. In this article, we introduce a semi-automated method of generating features for volumetric terrains using a rule-based procedural generation system. Features are generated by selecting subsets of a voxel grid as input symbols to a grammar, composed of user-created operators. This results in overhangs and caves generated from a set of simple rules. The feature generation runs on the CPU and the GPU is utilised to extract a robust mesh from the volumetric dataset.

1. Introduction

Generation of terrains can be a particularly important process when creating realistic representations of virtual worlds, as found in computer graphics simulations, feature films and computer games with outdoor environments. So far, there has been a considerable amount of research in this domain, which ranges between fully automated and semi-automated methods.

While it is now feasible to create massive virtual worlds, the tasks of designing the terrain, populating the world with content, and, finally, ensuring it does not feel empty or barren, continue to be very time consuming processes. Procedural content generation (PCG) has many applications and has proven valuable to designers due to its ability to algorithmically produce content such as the generation of textures, geometry and animations [1] so it can greatly improve the cost efficiency of populating a virtual environment. PCG will be used in this research to assist designers and shorten the length of time to create large scale landscapes.

Traditionally, terrains are defined by their surface details using a texture-based approach representing a top-down, two-dimensional view, called a *heightmap*. However, the details beneath the terrain surface have a significant impact on how the terrain is formed and its eventual appearance. This research uses volumetric data to represent terrain. This is important as it provides meaning to the details that are not visible to the user. Various factors, such as soil type and material density, govern how terrains are created in the real world. This can be modelled accurately when a voxel-based approach is utilized. A further advantage of this approach is that both constructive and destructive methods to terrain creation can be adopted without being concerned about real-time polygon mesh editing, i.e. a surface can be extracted from the voxel data after the data has been constructed to the designer's liking.

This article proposes a procedural, voxel-based approach to assist users in the generation of key terrain features, such as overhangs and caves. The presented method expands the concept of *shape grammars* to a volumetric space and explains the process employed to create terrain features. We develop specific rulesets that are applied over a voxel dataset in order to create such features on the CPU. We also describe some good practices to be utilised when developing these rulesets. The final terrain mesh is generated at real-time frame rates by using our GPU-based surface nets algorithm. Furthermore, we present timings and memory usage from our results for the generation of the voxel data using different rulesets plus the performance statistics of our GPU surface extraction algorithm.

This research has been carried out in collaboration with Sony Interactive Entertainment Euro Research and Development (SIE Euro R &D), working with their proprietary game engine *PhyreEngine*™ [2]. We also thank *NVIDIA Corporation* for their hardware donation of a *Titan X* GPU.

2. Related work

In this section previous work related to the formation of volumetric terrain is briefly reviewed. The proposed method involves aspects of

* Corresponding author at: Bournemouth University, Poole House, Talbot Campus, Poole, Dorset BH12 5BB, UK.
E-mail address: rahul.dey@bournemouth.ac.uk (R. Dey).

https://doi.org/10.1016/j.entcom.2018.04.003
Received 7 December 2017; Received in revised form 11 April 2018; Accepted 24 April 2018
Available online 05 May 2018
1875-9521/ Crown Copyright © 2018 Published by Elsevier B.V. All rights reserved.

multiple domains, including terrain creation, data structures to manage voxel datasets and rule-based procedural generation methods.

2.1. Terrain

3D games and simulations set in an outdoors environment often need to represent terrain. Terrains for these applications need to be rendered at real-time frame rates as the user should be able to navigate the virtual world without waiting for the terrain to draw to the screen.

Modern GPU architectures have been designed to handle processing polygons quickly and, as such, most terrains are rendered using polygon meshes. However, terrain data can be represented in multiple ways, including the manipulation of vertices of a grid mesh directly (which this article refers to as polygonal terrain) and as volumetric terrain by storing and editing the type of terrain present at specific points in 3D space. It should be noted that volumetric terrain is still commonly rendered as a set of polygonal meshes generated by applying various surface extraction methods on the voxel representation of the terrain. This is necessary as volumetric data can become inordinately large at higher grid resolutions. A polygonal mesh that provides an accurate representation of the surface of the data and falls within the constraints of a GPU is important, especially for real-time rendering.

Heightmaps (and their extensions) have been used extensively to represent terrains as they provide a very intuitive method of designing outdoor environments. Heightmaps are two-dimensional textures that store height values at each texel which are, in turn, used to displace vertices on a pre-allocated polygonal grid mesh. A comprehensive survey of such approaches can be found in [3].

While heightmaps are relatively straightforward and intuitive to create terrains, some features of real-world landscapes can be more complicated to generate using heightmaps alone. Features such as caves and overhanging cliffs cannot be represented by single height values and thus call for a different method of creation. One workaround was presented by [4], where an existing heightmap was warped using a flow field. This resulted in overhanging terrain being generated in a simple manner. However, the main limitation of this method is that a specific vector field needs to be defined before the deformation takes place and this can be a complex process.

Volumetric terrains can remove the requirement for workarounds to create caves and overhangs and recreate these features effectively with greater control over the final result. Many volumetric terrains are stored and accessed as chunked data structures, seen in [5]. Storing the data in this way splits it effectively and allows for parallelism between generation of chunks, dividing the workload of terrain generation among several threads.

The work of [6] presents a method of combining the voxel-based terrain as well as storing a signed distance field. This means that sculpting and approximating erosion can be trivially and efficiently implemented to model more realistic terrains. The authors also introduce a highly user-directed process, where procedural methods (i.e. the erosion algorithms) are applied based on initial user input. Another hybrid approach is presented in [7] which combines voxels and heightmaps. This approach creates a coarse voxel grid that can represent caves and overhangs and splits the entire voxel grid into patches. The authors then extract the surface to form the polygonal geometry required for rendering. The resulting geometry is subsequently displaced by heightmaps assigned to each patch. The process allows them to represent terrains with volumetric features, whilst maintaining a low voxel grid resolution.

Signed distance functions (SDFs) can also be used for volumetric rendering by storing functional representations of geometry at points in 3D space. This can offer a wide degree of flexibility and can scale well due to its resolution independence. The work of [8] decomposes volumes into a hierarchical tree data structure containing partial SDFs at each tree node. Constructing a suitable SDF for complex volumetric features requires either the use of well-developed tooling solutions or mathematically proficient users. Furthermore, constructing robust meshes from SDFs can be a difficult and slow process to perform on GPUs [9]. Due to these complexities, we represent terrains using voxels and have the advantage of direct and fast access to the underlying data.

2.2. Voxel management

A typical method of representing volumetric data is by utilising a dense grid of voxels. However, terrains tend to model large scale landscapes and representing them with a dense grid quickly becomes prohibitively expensive in terms of memory usage. Therefore, it is prudent to make use of data structures designed to ameliorate this limitation. There are several sparse data structures available, which have been successfully used in a variety of applications, such as fast global illumination approximations [10] and fluid simulation [11].

A simple, sparse grid structure is found in [12] that separates a large voxel grid into a set of blocks containing a fixed number of voxels. When a voxel is inserted into the grid, the corresponding block is inspected. If the block is empty, then the algorithm allocates memory for the entire block of voxels before inserting the voxel data. This scheme can substantially save memory, particularly when the majority of the voxel grid consists of empty space. However, this can also needlessly allocate large amounts of memory when voxels are very sparsely distributed within the entire voxel grid, i.e. there are very few voxels required per block. This method is highly dependent on the type and distribution of the input volumetric data and therefore requires some experimentation to achieve optimal block sizes to minimize wasted space.

A data structure for voxel storage that has come into fairly common usage is the *Sparse Voxel Octree (SVO)*. SVOs are used by [13] to create a fast approximation to global illumination in real time scenes by constructing a two-part octree consisting of a *node pool* and a *brick pool*. The node pool linearly stores the structure of the octree in contiguous memory in order to maximize cache coherency when querying the data structure. The brick pools contain bricks of voxels (sets of 3^3 voxels) stored in 3D textures. Storing the voxel data in GPU texture resources enables the algorithm to make use of hardware filtering on the voxel data to improve lighting quality. However, as the construction of the data structure can be a somewhat slow process, scenes that require considerable dynamic movement or regeneration are not handled particularly well with this method. As such, it is primarily suitable for static scenes.

There are also a number of variations for SVOs, including a different decomposition of the resulting structure. The work of [14] achieves this by storing child descriptors that contain index offsets to memory locations in order to allow for quick access to each node's children. The final memory footprint of an SVO has also been reduced by [15], by removing redundant relationships between nodes in the tree and constructing the data as a directed acyclic graph. However, both of these methods still have costly construction times and are therefore not suitable for scenes containing regeneration of geometry or other dynamic properties.

A different sparse data structure that can be used to store volumetric data is the *brickmap*. Brickmaps have been used for raytracing, in both offline renderers [16] and, more recently, real-time rendering [17]. Brickmaps contain a sparse list of *bricks* where each brick contains volumetric information for a small set of voxels. The key difference between an SVO and a brickmap structure used for real-time raytracing is that there are only two discrete levels used in a brickmap, a sparse map for looking up bricks and a finer resolution voxel grid, found in each brick's dataset. It is an efficient data structure that can be quickly rebuilt and is therefore useful for more dynamic scenes. This design however does not save as much memory as a standard SVO.

Recently, a new sparse data structure, named *VDB (Volumetric, Dynamic B+) trees*, was introduced [18]. It offers a number of key advantages to volumetric representations, such as good cache

coherence and fast random access capabilities, including insertion, traversal and deletion operators. These features in particular render it appropriate for dynamically changing volumes.

2.3. Rule-based procedural generation

Procedural content generation (PCG) refers to the process of computationally constructing assets in games and simulations, either from the ground up or by offering variations on an existing template. Many PCG methods are noise-based and utilise predictable pseudorandom number generation to produce varied results [1,19,20]. However, noise-based PCG is usually a fully automated process which offers designers little control over how the generated terrain is structured. Other PCG methods make use of a rule-based approach that consists of using an initial input and a ruleset to generate diverse content, whilst still constrained within the user defined parameters. Rule-based systems can be used to counteract the diminished control present in noise-based methods.

Existing methods use the concept of grammars to govern the generation of content. A formal grammar consists of a set of axioms and rules that recursively rewrite the initial state until a termination condition is met. This may be until the generation reaches a predetermined terminal state or runs for a set number of iterations. This results in a structured and deterministic result so long as the initial state, ruleset and random seed (for rule selection) remain the same. Variations can be generated by either altering the input state or the rules.

Lindenmayer systems (L-systems) have been frequently used to generate smaller objects such as trees and foliage [21]. They have since been extended to generate road networks for urban environments and entire cities [22]. Further research into L-systems investigated a more efficient design by utilising multithreading. Multi-threaded applications require threads to be synchronised well to prevent errors such as race conditions. Synchronisation can be achieved by introducing locks, such as mutexes, as part of the algorithm's execution. However, locks tend to come with performance implications. Thus, [23] introduced a multi-threaded L-system that is completely lock-free. This extension can therefore be deployed to massively parallel architectures, such as GPUs, so a result using a large ruleset can be generated quickly and efficiently.

Shape grammars are another form of grammar that has been adopted for procedural generation [24]. Shape grammars function by recursively applying rules that govern transformations to an initial shape. An extension to this introduced the concept of a split grammar, where rules are composed of basic shapes and more detailed decompositions of the basic shapes [25]. This has been used for the generation of procedural architecture [26]. Shape grammars have also been modified to run on massively parallel architectures. A GPU-based shape grammar algorithm was created by [27] in order to create infinitely large cityscapes. They utilised the parallel algorithm found in [28] and a set of rules to generate geometry for realistically planned cities. When rendering scenes, the visibility of geometry can be handled by frustum culling, occlusion culling and the use of spatial databases, such as octrees or binary space partition trees. This makes the assumption that the entirety of the geometry data is available for rendering, if it is required. One of the concepts [27] introduced was their method of *visibility pruning*. This completely skipped evaluation of any rules that would create buildings outside the view frustum of the current camera, eliminating the generation of any redundant geometry and reducing GPU memory usage.

3. Voxel grammars

This section describes how the concept of voxel grammars was formulated in this article. The individual components that comprise a grammar are detailed, as well as the process of how they are used during the generation phase. Voxel grammars have been inspired from voxel space automata [29], where voxels are generated via a set of predefined rules. Greene primarily uses this method to simulate plant growth. However, adding detail to existing geometry is a further application of this approach. Our method extends the concept of recursively manipulating volumetric data governed by a set of rules to the formation of specific features found in real-world terrains. The method operates on a voxel grid that defines the terrain boundary. Each voxel is represented by a density value of the terrain material contained within it. Voxels are generated within the grid to create an initial state. The initial state in this work was constructed by voxelizing a heightfield generated with 3 octaves of 2D Perlin noise, as it is a common way of generating plausible procedural heightmap-based terrain. We use multiple octaves at differing frequency values to ensure the initial terrain contains a sufficient balance between low-frequency and high-frequency details so that it is not too smooth or too noisy, respectively. The populated voxel grid is then derived using a *sliding window* approach, checking whether the subset of voxels within the window satisfies any rule criteria. If a matching rule is found, then its respective transformation is performed on the voxels. The window is subsequently repeatedly offset by a user-defined stride parameter. The derivation process is then repeated for a number of iterations (exposed as a parameter to the user). Furthermore, the user can define a start and end position within the voxel grid to determine which section of the grid the grammar is applied to. The combination of bounding values, the sliding window stride and symbol grid sizes can reduce the state space for a large voxel dataset.

3.1. Rules

The rules within the class of grammars that we have developed consist of three components – *symbols*, *transformations* and *weights*. Symbols consist of a list of predicates to satisfy and transformations contain a list of mutations to apply to a subset of voxels. This can be seen as akin to a rudimentary programming language to a certain extent, as symbols and transforms are, respectively, analogous to conditional statements and intrinsic functions. Rules also possess a priority value that governs the probability that the rule will be selected for execution in the rule matching stage of the algorithm.

This can be formalised using L-system notation as follows: Let voxel grammar $\mathbf{G} = \langle V, \omega, P \rangle$, where ω is the initial state of the voxel grid and P is a set of rules that transform subsets of voxels in the form (s,t). $s \in V$ is the list of predicates, which we refer to as a *symbol* and t is the *transform* to apply to the subset of voxels. V is the set of all tensors of size $I \times J \times K$ of predicates. The predicates operate over the domain of real values and thus effectively enable the use of a theoretically infinite alphabet. As the values in a voxel grid in our implementation are stored as a single 32-bit floating point value, realistically the expressive range of the grammar is subject to the size and precision of the voxel values stored by the underlying voxel engine.

3.1.1. Symbols

A rule's symbol is a list of conditions in the form of an $I \times J \times K$ array that determines whether the rule's transformation will be executed. Each symbol entry in the array consists of an *operator* and a *data value*. The operator refers to the type of condition being checked and uses the data value as a comparator to the input voxel value. In order to ensure flexibility, there are a number of operators that have been implemented and their descriptions can be found in Table 1.

3.1.2. Transforms

A transformation consists of a list of manipulations in an array with the same dimensions as the rule's symbol and is only applied to the voxel grid when the symbol's criteria have been fulfilled. Similarly to symbols, transformations also consist of an *operator* and a *data value*. The selected voxel's density is manipulated in a way determined by the type of operator being used plus the data value. Descriptions of the operator types can be found in Table 2 and Fig. 1 demonstrates a simple

Table 1
List of symbol operators.

Symbol Operator	Description
IGN	Ignore – Always returns true
PRES	Present – Passes if voxel density value is greater than 0
ABS	Absent – Passes if voxel density value is 0
EQ	Equals – Passes if voxel density value is equal to the symbol value
NEQ	Not Equals – Passes if voxel density value is not equal to the symbol value
LT	Less Than – Passes if voxel density value is less than the symbol value
LEQ	Less Than or Equal – Passes if voxel density value is less than or equal to the symbol value
GT	Greater Than – Passes if voxel density value is greater than the symbol value
GEQ	Greater Than or Equal – Passes if voxel density value is greater than or equal to the symbol value

Table 2
List of transform operators.

Transform Operator	Description
NOP	Does nothing to the value of the selected voxel
SET	Sets the selected voxel value to the transform value
ADD	Adds the transform value to the selected voxel value

example of a transformation.

3.1.3. Rule selection

As grammars become larger and more complex, there can be times where a set of input voxels can satisfy the conditions for multiple rules. In our implementation, each rule contains a priority value as part of its parameters. When there are multiple matching rules, the priorities are sorted and the rule with the highest priority value is selected. If the priorities match, the selected rule is chosen stochastically from the matches. The greatest priority is used in order to simplify the creation of grammars, as this allows designers to create rules using a more intuitive approach by introducing more control in the variations, instead of relying on a probabilistic method.

3.2. Grammar construction

During the course of constructing grammars, some observations were made about the effects that certain components of rules had on the resulting voxels. When some rules were matched with the voxel grid, an issue that arose was the repeating, uniform patterns. For natural looking terrains, this symptom is usually undesirable. One method of alleviating this is to introduce some variance to the rules. This can be achieved by creating a rule with the same symbol values and the same rule weight, but with a transformation consisting entirely of *NoOp* operators. This method exploits the mechanisms of our rule-matching system: as the weight remains the same, when the appropriate symbol is matched, the rule selector will either apply a transformation or do nothing based on a random probability.

When developing grammars, it is also prudent to be wary of overuse of the *Ignore* symbol operator. A symbol consisting entirely of these operators should almost always be avoided, as it matches with the entire voxel grid. This can result in severe consequences, where transformations are applied globally to the entire voxel grid and is typically not what the designer intended.

3.3. Derivation process

The final grammar processing method occurs in two stages: *rule matching* and *replacement*. The rule matching stage iterates through the ruleset and checks whether the rule's symbol matches the group of currently selected voxels. Firstly, the rule with the largest dimension symbol is found and a sliding window of this size is passed along each axis. The voxels contained within this window are the currently selected voxels. These are used to compare against the symbol of all rules that have the same symbol dimensions. If there are matches, then the replacement stage occurs. This process continues until all rules have been queried. Pseudocode for this process can be found in Fig. 2. Replacement is simply the process of transforming the set of matching voxels with the selected rule's transformation.

4. Method and implementation

This section discusses how grammars were designed for use with the proposed method. Explanations are provided as to why each rule was constructed. The topologies resulting from cave and overhang formation tend to be very different and, as a result of this, two grammars were constructed with different rulesets to enable the creation of these terrain features. Each rule in both rulesets is given the same priority weight as all of their respective symbols are different. Thus, there are no other rules to choose between when the rule selection portion of the derivation process takes place.

4.1. Cliffs and overhangs

A feature found in rich terrains is the formation of naturally occurring cliffs and overhangs. Similar to caves, they are formed due to erosion from water and weathering effects. Assuming an input of a solid, vertical wall of voxels, the ruleset contains one rule to remove voxels towards the base of the wall and another to add some detail. On their own, these rules produce some unnatural looking features (regularly spaced voxels) and some impossible features (floating segments of terrain). In order to mitigate this, another rule was added that acted as a "clean up" action. The overhang ruleset can be found in Table 3.

4.2. Caves

The formation of caves is governed by the erosion of the different types of rock. Existing fractures already within the terrain structure are

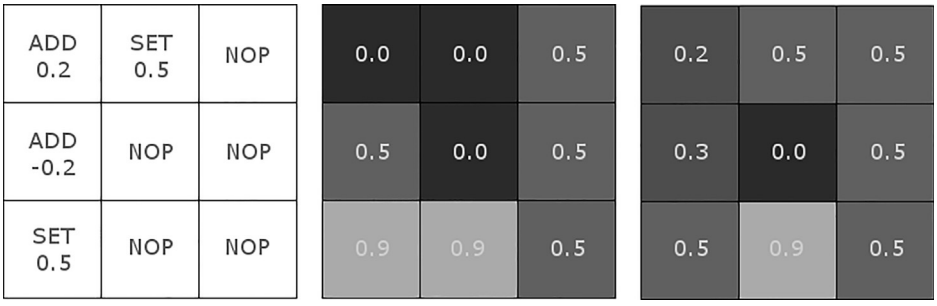

Fig. 1. Example of a simple two dimensional transformation. Rule transform (left), input voxels (centre), output voxels (right).

```
sort rules in descending order of symbol dimensions

foreach iteration
    foreach rule in rules:
      voxelSubset = select voxels of size
                    rule.symbol.dimensions

      while voxelSubset not reached voxel grid bounds:
        if voxelSubset satisfies rule.symbol:
            add rule to matchArray
        move voxelSubset along axis

    if matchArray is not empty:
        rule = select from matchArray
        execute rule.transform
```

Fig. 2. Pseudocode for deriving the voxel data.

Table 3
Ruleset for generating overhangs.

Rule 1						
Size: [3, 3, 1], Weight: 1.0						
Symbol				Transform		
GEQ 0.5	LT 0.5	IGN		NOP	NOP	NOP
GEQ 0.5	LT 0.5	IGN		NOP	NOP	NOP
GEQ 0.5	LT 0.5	IGN		ADD -0.5	NOP	NOP

Rule 2						
Size: [3, 3, 1], Weight: 1.0						
Symbol				Transform		
GEQ 0.5	GEQ 0.5	IGN		NOP	NOP	NOP
GEQ 0.5	LT 0.5	IGN		ADD -0.5	NOP	NOP
GEQ 0.5	GEQ 0.5	IGN		NOP	NOP	NOP

Rule 3						
Size: [3, 3, 1], Weight: 1.0						
Symbol				Transform		
GEQ 0.5	GEQ 0.5	GEQ 0.5		NOP	NOP	NOP
GEQ 0.5	LT 0.5	IGN		NOP	ADD 0.5	NOP
IGN	LT 0.5	IGN		ADD -0.5	NOP	NOP

Rule 4						
Size: [3, 3, 1], Weight: 1.0						
Symbol				Transform		
GEQ 0.5	GEQ 0.5	IGN		NOP	NOP	ADD 0.5
GEQ 0.5	GEQ 0.5	IGN		NOP	NOP	ADD 0.5
GEQ 0.5	LT 0.5	IGN		NOP	NOP	NOP

eroded by groundwater. In speleological literature, cave topology can be defined as *vadose* or *phreatic* and is determined by the proximity to the groundwater source [30]. This research emulates the formation of such types of cave by finding a suitable approximation that can be decomposed into a grammar. The resulting grammar can be found in Table 4. The first rule is designed to create an initial starting point for the cave to be generated, which emulates the initial entry point for groundwater to begin creating caves within a section of the terrain. Gravitational forces and further groundwater erosion are emulated using the second and third rules, which lower the density of the lowest vertical point within a cavity in the terrain. The fourth rule is designed to widen existing cavities in the terrain in order to provide the resulting cave with more internal space, as well as offer some variation to the cavern's internal structure.

4.3. Surface extraction

The polygonal mesh is extracted from the voxel data using the surface nets method [31], where the global energy minimization strategy used for creating the surface is defined by the centre of mass of each voxel's edge intersections. This is more commonly referred to as the *naive surface nets* variant [32].

This article utilises a version of the naive surface nets algorithm that executes entirely on the GPU. The algorithm is capable of high parallelism, therefore executing *compute shaders* on the voxel data can be greatly beneficial in terms of performance, as a large number of threads can be launched to work on individual segments of the data concurrently. The method we have developed executes several compute shaders:

1. Compute the centres of mass (*ComputeCOM*).
2. Construct a vertex buffer (*ConstructVertexBuffer*).
3. Construct an index buffer (*ConstructIndexBuffer*).
4. Calculate the vertex normals of the mesh (*ComputeNormals*).

The *ComputeCOM* shader writes to a linear array of 3D position vectors. This array represents the dual mesh of the input voxel grid. Each thread in the shader reads a section of voxel data in 2^3 groups. It then sets the initial position to be the centre of this group and interpolates the position using the direction and densities for each voxel, which is written to the final centre-of-mass buffer.

Construction of the vertex and index buffers are relatively straightforward. Each vertex in the vertex buffer corresponds to each individual element in the centre-of-mass buffer. When an element is added to the vertex buffer, its array index is written to a 3D array of integers that is used as a lookup table. After this process has completed, the construction of the index buffer takes place. The index of the thread in the *ConstructIndexBuffer* shader is used as an index into the lookup table. The thread subsequently queries the neighbours of this index to see if values exist in the table to create a triplet of indices. The triplet is then added to the index buffer. Pseudocode of the process is shown in Fig. 3.

The vertex normals of the mesh are computed as a separate shader after the GPU buffers have been constructed successfully. The *ComputeNormals* shader uses the index buffer and a cross product operation to calculate the normals for each vertex in the vertex buffer.

5. Results

This section discusses the results obtained using the proposed method. Its effectiveness is demonstrated by the examples of caves and overhangs generated by the process. The results have been obtained by deriving voxel grids of several resolutions (32^3, 64^3, 96^3 and 128^3). The

Table 4
Ruleset for generating caves.

Rule 1					
Size: [3, 3, 1], Weight: 1.0					
Symbol			**Transform**		
GEQ 0.5	GEQ 0.5	LT 0.5	NOP	NOP	NOP
GEQ 0.5	GEQ 0.5	LT 0.5	NOP	ADD -0.5	NOP
GEQ 0.5	GEQ 0.5	LT 0.5	NOP	NOP	NOP

Rule 2					
Size: [3, 3, 1], Weight: 1.0					
Symbol			**Transform**		
GEQ 0.5	GEQ 0.5	GEQ 0.5	NOP	ADD -0.5	ADD -0.5
GEQ 0.5	GEQ 0.5	LT 0.5	NOP	ADD -0.5	ADD -0.5
GEQ 0.5	GEQ 0.5	GEQ 0.5	ADD -0.5	ADD -0.5	NOP

Rule 3					
Size: [3, 3, 1], Weight: 1.0					
Symbol			**Transform**		
LT 0.5	LT 0.5	GEQ 0.5	NOP	NOP	ADD -0.5
GEQ 0.5	GEQ 0.5	GEQ 0.5	ADD -0.5	ADD -0.5	ADD -0.5
LT 0.5	GEQ 0.5	GEQ 0.5	NOP	ADD -0.5	NOP

Rule 4							
Size: [4, 4, 1], Weight: 1.0							
Symbol				**Transform**			
LT 0.5	LT 0.5	GEQ 0.5	IGN	NOP	NOP	ADD -0.5	NOP
LT 0.5	GEQ 0.5	GEQ 0.5	IGN	ADD -0.5	ADD -0.5	ADD -0.5	NOP
GEQ 0.5	GEQ 0.5	GEQ 0.5	IGN	NOP	ADD -0.5	ADD -0.5	NOP
IGN	IGN	IGN	IGN	ADD 0.5	NOP	NOP	NOP

final surface meshes are rendered using a deferred renderer in *DirectX 11* [33] on a PC equipped with a 3.20 GHz quad-core *Intel*® CPU, 16 GB of RAM and an *NVIDIA*® *Titan X* GPU.

When rendering the mesh, triplanar texturing is used to effectively blend multiple textures [34]. The weighting of each texture used at a point on the surface is determined by the dominant axis of the surface

```
ComputeCOM(voxels):
  foreach v in voxels:
    foreach neighbour in neighboursOf(v):
      mass += neighbour.value
      massCentre += mass * neighbour.voxelPosition
    massCentre /= mass
    massCentres.append(massCentre)
  return massCentres

ConstructVBuffer(massCentres):
  foreach massCentre in massCentres:
    vbuffer.append(massCentre)
    insertIntoVertexLUT(vbuffer.indexOfLastElement)
  return vbuffer

ConstructIBuffer(vertexLUT):
  foreach index in vertexLUT:
    if index is valid and neighbours of index are valid:
      ibuffer.append(index)
      ibuffer.append(validNeighbours)
  return ibuffer

SurfaceExtract(voxels):
  massCentres := ComputeCOM(voxels)
  vbuffer, vertexLUT := ConstructVBuffer(massCentres)
  ibuffer := ConstructIBuffer(vertexLUT)
```

Fig. 3. Pseudocode for extracting the surface of the voxel data.

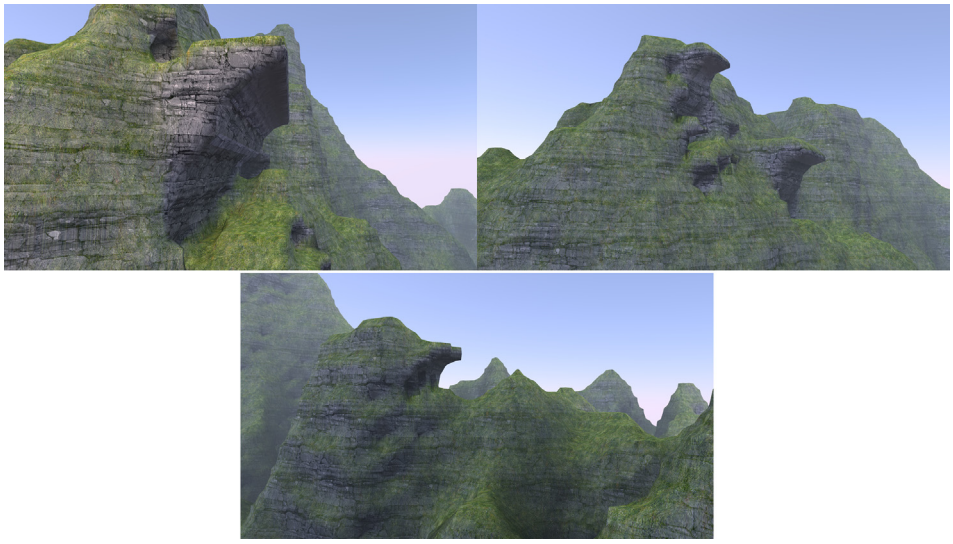

Fig. 4. Examples of generated overhangs using the ruleset found in Table 3 (Voxel Grid Resolution: 128^3).

normal and the texture coordinate is calculated by the fractional part of the point's world space coordinate.

Examples of overhangs generated with the voxel grammar can be seen in Fig. 4. The first example demonstrates an overhanging ledge protruding from the terrain with a plateau which appears naturally embedded within the terrain. In a heightmap-based approach a ledge such as this would either have to be stitched to the mesh to make a single mesh, or be rendered as a separate object. As our method integrates the features directly into the mesh it avoids both of these options, so that the terrain can be treated as a unified entity. The second example shows multiple overhangs being created within a region of space in the terrain by increasing the range of the generation bounding box and offers a flexible option of being able to generate repeating volumetric features within a space. The third example presents another single overhang, this time without a plateau.

Caves generated with our method can be found in Fig. 5. The first example shows a wide-mouthed cave constructed with our grammar, further demonstrating the benefits of editing volumetric data directly. Achieving this effect in a heightmap-based terrain would be a difficult task, as many surfaces would have to be edited to recreate the concavity presented in this image. The second example shows the grammar being applied to a smaller region of the terrain, where the surface on the side has been eroded to present a small network of caverns. The third image shows two medium-sized caves that have formed next to each other. This has been achieved by modifying the stride of the sliding window in the grammar generation parameters.

Table 5 shows the timings to generate cave and overhang examples in different voxel grid resolutions. When timing the data, the voxel grammar was set to derive the entirety of the voxel grid and the stride of the sliding window was set to $1 \times 1 \times 1$ to ensure that the worst-case performance statistics were recorded.

The generation of voxels is the most expensive operation in the process and this is to be expected as it currently does not utilise the GPU. Instead, the voxel grid is derived on a single thread by the CPU and passed to the surface extraction functions. However, even at the largest resolution the grid was derived and voxels were generated in under a second. Generating voxels for overhangs was consistently faster than for caves (between 2.6 and 2.9 times faster). The cave grammar's last rule derives groups of 16 voxels at a time and this would account for the increase in processing time.

Our compute shader variant of the surface nets algorithm has been able to extract the surface of the volumetric data at satisfactorily fast rates. Meshes were created within 1 ms and this makes it particularly suitable for dynamic editing of the voxel data, as the meshes can be

Fig. 5. Examples of generated caves using the ruleset found in Table 4 (Voxel Grid Resolution: 128^3).

Table 5
Times taken to generate voxels from rulesets, extract surfaces and render the meshes (in ms).

Grid Resolution	Generation		Surface extraction		Rendering	
	Overhang	Cave	Overhang	Cave	Overhang	Cave
32^3	3.477	9.204	0.125	0.117	0.778	0.777
64^3	32.008	89.059	0.271	0.281	0.915	0.919
96^3	111.331	314.384	0.578	0.604	1.126	1.129
128^3	269.121	770.208	1.072	1.130	1.434	1.441

regenerated at real-time rates. Mesh construction for overhangs and caves both perform at similar speeds as the algorithm is primarily bound by the voxel grid resolution and is independent of the type of features being created in the terrain.

There is little difference between the time taken to render the meshes of overhangs or caves and, as is expected, render times increase as the resolution of the grid becomes larger. This is due to the dual mesh grid required for the surface extraction increasing in resolution and, therefore, increasing the vertex count of the final mesh. However, this could be reduced by utilising various mesh simplification methods since the size of the polygons all over the mesh is uniform. By simplifying the mesh, the triangle count of the final mesh would be reduced and the GPU would have less work to execute. Furthermore, level-of-detail methods can be applied to the mesh that would make polygons distant from the camera larger, so that the GPU does not spend time on rasterizing many small polygons far away from it that would produce no discernible visual difference. It should also be noted that the surface extraction creates the polygons in the index buffer in a non-deterministic manner. While the resulting mesh is correctly produced, it may benefit from some form of vertex cache optimization, such as in [35], in order to try and ensure that the data is in a more GPU-friendly format.

6. Conclusions

This article presented a method for generating complex terrain features using voxel grammars. Furthermore, it presented approaches on how such grammars can be created and what considerations should be taken into account when developing them.

The described methodology is effective in manipulating groups of voxels to create topologies of interesting terrain features. However, the main limitation of this method is the substantially manual nature of designing new grammars. While there are several aforementioned factors that can be taken into account when creating rulesets, the approach requires some considerable effort and experimentation. Symbols and transforms in rules are concretely defined as a set of operators, but it would be beneficial to create rules in more abstract terms to be more intuitive to use. For example, this could be achieved by defining a symbol in terms of the slopes and material types found in a region of voxels, instead of being composed of a specific permutation of voxel densities. Similarly, defining transforms to create a cliff or a cave at a location, instead of operating on individual voxels, would be more useful for generating desired features more quickly. As such, we have reserved future research to create more abstract methods of defining rules in a grammar.

Furthermore, many terrain features such as caves can be formed in a recursive manner, by modifying transform operator values at each recursion. Currently, it is difficult to design grammars that perform such operations effectively. In order to enable this functionality to the voxel grammar approach, rules that execute other rules with parameter deltas would need to be added.

As this approach works on singular density values found within the terrain, this implies the terrain is composed of a single material. However, real terrains consist of multiple materials at multiple densities and ideally these varying densities should be taken into account by adding a check for the material type as a symbol operator.

The presented method is a single-threaded CPU implementation. This will detrimentally impact on performance as the voxel grids the grammar is applied to increase in resolution. Therefore, it would be prudent to design a GPU-based method of executing the grammar on the voxel grid to maintain high performance in terms of time. For future research, we will fully utilise the massively parallel nature of modern GPUs. The algorithm presented for terrain generation is highly parallelizable as it has very few inter-resource dependencies, so we expect to obtain substantial improvements in performance with the addition of GPGPU methods.

The size of the voxel data also needs to be monitored so that it does not grow to too large a magnitude. We will also adopt a sparse data structure for voxel storage in order to scale to much larger dimensions of volumetric grids with limited memory resources. Sparse Voxel Octrees (SVOs) [10] and brickmaps [16] are two methods optimized for GPU usage that are appropriate for this task. Furthermore, support for terrains consisting of several materials would require multiple density values denoting the amount of each material. Extending this level of detail to a voxel increases its size and it would be prudent to use the work of [36] to compress the data further.

Another direction we will explore is a more automated process for generating the grammars themselves. In this regard, machine learning technologies, such as deep convolutional neural networks and evolutionary algorithms, appear to suit the underlying training requirements of generating a grammar. As there are multiple parameters per rule, this can form a high-dimensional problem which could be resolved using methods from approximate dynamic programming [37] and reinforcement learning literature [38].

References

[1] D.S. Ebert, F.K. Musgrave, D. Peachey, K. Perlin, S. Worley, Texturing and Modeling: A Procedural Approach (The Morgan Kaufmann Series in Computer Graphics), Morgan Kaufmann, 2002.
[2] SIE R&D West, Phyreengine, 2017. < http://develop.scee.net/research-technology/phyreengine/ > (Online; accessed 02-January-2017).
[3] R.M. Smelik, K.J. De Kraker, S.A. Groenewegen, T. Tutenel, R. Bidarra, A survey of procedural methods for terrain modelling, 3AMIGAS - 3D Advanced Media In Gaming And Simulation (June 2015) (2009) 25–34. http://dx.doi.org/10.1145/1814256.1814258.
[4] M.N. Gamito, F.K. Musgrave, in: Procedural Landscapes with Overhangs, vol. 2, 2001, p. 3.
[5] A. Santamaría-Ibirika, X. Cantero, M. Salazar, J. Devesa, I. Santos, S. Huerta, P.G. Bringas, Procedural approach to volumetric terrain generation, Visual Comput. 30 (9) (2013) 997–1007, http://dx.doi.org/10.1007/s00371-013-0909-y.
[6] A. Peytavie, E. Galin, J. Grosjean, S. Merillou, Arches: a framework for modeling complex terrains, Comput. Graph. Forum 28 (2) (2009) 457–467, http://dx.doi.org/10.1111/j.1467-8659.2009.01385.x.
[7] Ç. Koca, U. Güdükbay, A hybrid representation for modeling, interactive editing, and real-time visualization of terrains with volumetric features, Int. J. Geograph. Inform. Sci. 28 (9) (2014) 1821–1847, http://dx.doi.org/10.1080/13658816.2014.900560.
[8] L. Wang, Y. Yu, K. Zhou, B. Guo, Multiscale vector volumes, ACM Trans. Graph. 30 (6) (2011) 1, http://dx.doi.org/10.1145/2070781.2024201.
[9] M. Swoboda, Advanced Procedural Rendering in DirectX 11, 2012. < http://www.gdcvault.com/play/1015455/Advanced-Procedural-Rendering-with-DirectX > (Online; accessed 05-January-2017).
[10] C. Crassin, F. Neyret, S. Lefebvre, E. Eisemann, Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering, in: Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games, ACM, 2009, pp. 15–22.
[11] R. Setaluri, M. Aanjaneya, S. Bauer, E. Sifakis, SPGrid, ACM Trans. Graph. 33 (6) (2014) 1–12, http://dx.doi.org/10.1145/2661229.2661269.
[12] M. Wrenninge, Production Volume Rendering: Design and Implementation, CRC Press, 2012.
[13] C. Crassin, F. Neyret, M. Sainz, S. Green, E. Eisemann, Interactive indirect illumination using voxel cone tracing, Comput. Graph. Forum 30 (7) (2011) 1921–1930, http://dx.doi.org/10.1111/j.1467-8659.2011.02063.x.
[14] S. Laine, T. Karras, Efficient sparse voxel octrees, IEEE Trans. Visual Comput. Graph. 17 (8) (2011) 1048–1059.
[15] V. Kämpe, E. Sintorn, U. Assarsson, High resolution sparse voxel DAGs, ACM Trans. Graph. 32 (4) (2013) 1, http://dx.doi.org/10.1145/2461912.2462024.
[16] P.H. Christensen, D. Batali, An irradiance atlas for global illumination in complex production scenes, in: Proceedings of the Fifteenth Eurographics Conference on Rendering Techniques, EGSR'04, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2004, pp. 133–141. http://dx.doi.org/10.2312/EGWR/

EGSR04/133-141. < https://doi.org/10.2312/EGWR/EGSR04/133-141 > .
[17] M. Swoboda, Real Time Ray Tracing Part 2, 2013. < http://directtovideo.wordpress.com/2013/05/08/real-time-ray-tracing-part-2/ > (Online; accessed 02-April-2016).
[18] K. Museth, Vdb: high-resolution sparse volumes with dynamic topology, ACM Trans. Graph. 32 (3) (2013) 27:1–27:22, http://dx.doi.org/10.1145/2487228.2487235.
[19] S. Gustavson, Simplex Noise Demystified, Linköping University, Linköping, Sweden, Research Report.
[20] S. Worley, A cellular texture basis function, in: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, ACM, 1996, pp. 291–294.
[21] P. Prusinkiewicz, A. Lindenmayer, The Algorithmic Beauty of Plants, Springer Science & Business Media, 2012.
[22] Y.I.H. Parish, P. Müller, Procedural modeling of cities, in: 28th Annual Conference on Computer Graphics and Interactive Techniques (August), 2001, pp. 301–308. http://dx.doi.org/10.1145/383259.383292. < http://portal.acm.org/citation.cfm?doid=383259.383292 > .
[23] M. Lipp, P. Wonka, M. Wimmer, Parallel generation of L-systems, Vision, Model., Visual. (2009) 205–214.
[24] G. Stiny, Introduction to shape and shape grammars, Environ. Plann. B: Plann. Des. 7 (November) (1980) 343–351, http://dx.doi.org/10.1068/b070343.
[25] P. Wonka, M. Wimmer, F. Sillion, W. Ribarsky, Instant architecture, ACM Trans. Graph. 22 (3) (2003) 669, http://dx.doi.org/10.1145/882262.882324.
[26] P. Müller, P. Wonka, S. Haegler, A. Ulmer, L. Van Gool, Procedural modeling of buildings, ACM Trans. Graph. 25 (3) (2006) 614, http://dx.doi.org/10.1145/1141911.1141931.
[27] M. Steinberger, M. Kenzel, B. Kainz, P. Wonka, D. Schmalstieg, On-the-fly generation and rendering of infinite cities on the GPU, Comput. Graph. Forum 33 (2) (2014) 105–114, http://dx.doi.org/10.1111/cgf.12315.
[28] M. Steinberger, M. Kenzel, B. Kainz, J. Müller, W. Peter, D. Schmalstieg, Parallel generation of architecture on the GPU, Comput. Graph. Forum 33 (2) (2014) 73–82, http://dx.doi.org/10.1111/cgf.12312.
[29] N. Greene, Voxel space automata: modeling with stochastic growth processes in voxel space, Siggraph 23 (3) (1989) 175–184, http://dx.doi.org/10.1145/74334.74351.
[30] J.H. Bretz, Vadose and phreatic features of limestone caverns, J. Geol. 50 (6) (1942) 675–811 <http://www.jstor.org/stable/30060299> .
[31] S. Gibson, Constrained elastic surface nets: Generating smooth surfaces from binary segmented data, Medical Image Computing and Computer-Assisted Interventation-MICCAI'98 (1998) 888–898.
[32] M. Lysenko, Smooth Voxel Terrain (Part 2), 2012. < https://0fps.net/2012/07/12/smooth-voxel-terrain-part-2/ > (Online; accessed 30-August-2016].
[33] Microsoft, DirectX 11. < https://www.microsoft.com/en-gb/download/details.aspx?id=6812 > .
[34] R. Geiss, Generating complex procedural terrains using the GPU, GPU Gems 3 (2007) 7–37.
[35] T. Forsyth, Linear-speed Vertex Cache Optimisation, 2006.
[36] B. Dado, T.R. Kol, P. Bauszat, J.M. Thiery, E. Eisemann, Geometry and attribute compression for voxel scenes, Comput. Graph. Forum 35 (2) (2016) 397–407, http://dx.doi.org/10.1111/cgf.12841.
[37] W.B. Powell, Approximate Dynamic Programming: Solving the Curses of Dimensionality, John Wiley & Sons, 2007.
[38] R.S. Sutton, A.G. Barto, Reinforcement Learning: An Introduction vol. 1, MIT Press, Cambridge, 1998.