Procedural Generation of Terrain within Highly Customizable JavaScript Graphics Utilities for WebGL

T.H.McMullen and K.A. Hawick
Computer Science, Massey University, North Shore 102-904, Auckland, New Zealand
email: timmy361@gmail.com k.a.hawick@massey.ac.nz
Tel: +64 9 414 0800 Fax: +64 9 441 8181

June 2013

ABSTRACT

Modelling realistic scenes and rendering them appropriately are two key aspects of modern computer games. Scenes need to be detailed, realistically non-repetitive and computationally feasible. Procedural generation involves encoding a game scene as a recipe or procedure that can be generated and regenerated at run time, rather than just loaded from file or network server. Procedural generation in the context of web games and systems is particularly powerful in reducing bandwidth transfer requirements. We describe experiments to implement a framework for procedural generation using JavaScript and modern web client software systems such as WebGL and OpenGL shader language. Our system is able to exploit available Graphical Processing Units(GPU) and is aimed at supporting existing web based graphics engines. We present some graphical results and discuss future performance and scalability issues.

KEY WORDS
computer games; scene generation; procedural generation; spatial structure; fractals.

1 Introduction

Computer games [20] make heavy use of scene modelling [19], generation and rendering. While OpenGL [4, 6, 26] software has become the *de facto* industry standard for rendering scene, a modern generation of network and mobile games make use of web clients to run and render the system. WebGL [1, 3, 16] is an excellent bridging technology, because it allows many sophisticated scene and game rendering tasks to work within a web client context and supports game interface design [24] in a platform independent manner [18].

The advent of WebGL means that many web-based graphics utilities have been created to support simple graphical functions. We aim to improve upon the current graphic engines by allowing for a more customizable utility to be used, with functionality focused on procedural generation [7, 10]. It is the aim of this research to maximise the use of procedural generation to create different landscapes in different levels of detail. Implementation plays a large role in these tools, and so we use the Require.js module loader. With the created library it is hoped that the resulting programs, and applications will run on a range of devices. This means that several optimizations need to be researched to achieve this.

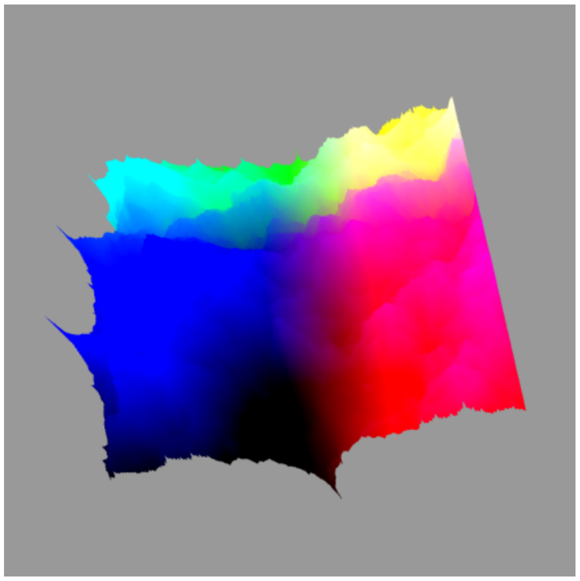
Figure 1: **Terrain Generation**: Landscape produced using the Diamond Square algorithm in WebGL. Map size is 257x257

Procedural generation is a method of using algorithms to create content. Taking this approach to creating content allows for unique areas, models and other objects to be produced. The idea behind this research is simplifying the process of creating content by utilising procedural generation. This means that if an object is created several times, they will differ slightly each time they are rendered. Procedural generation fundamentally changes the way in which a model is created. The model is generated at run time and with unique customisations that are made possible, rather than loaded from a static set of pre-generated options from a server.

Procedural generation has been used for a variety of scene aspects including village [5] or urban [25] architectural components, interior architectures [9], and organic materials such as trees [11, 13, 21]. Terrain and organic materials the generated patterns are often not simple geometric ones but are fractal in nature. This requires a more complex approach using fractal algorithms [15] such as L-Systems [23]. Various approaches to procedural generation software are possible, including the use of domain specific languages [8] to support customization. Some work using procedural generation for mobile systems has been reported on [14] but little work is available on its use in web based systems where data compression and minimizing data transfer [17] is important.

WebGL is an implementation of OpenGL designed to be run in the a web browser, using a JavaScript API [22]. It uses OpenGL Shader Language (GLSL) [2] to create and compile shaders to be run on a graphics processing unit(GPU) [12], to allow for highly dynamic and customizable graphical programming to be completed. As WebGL is web based it does come with some limitations. The limitations include restrictions imposed by bandwidth restraints, and JavaScript being interpreted code as opposed to compiled code. These limitations can be improved upon by minimising the amount of data which will need to be transferred, as well as improving the processing of data with the use of GLSL. The use of shader language means an application is able to offload some parts of the program to the GPU to be computed rather than the CPU.

As this research aims to create a highly customizable set of utilities, implementation needs to be as simplistic as possible. To help achieve this the Require.js script is used. This allows a web based application to be spread across multiple files, with each part accessible. By doing this, it helps to greatly simplify the process of creating and including code, into an web based environment.

This technology can be used to help improve upon some limitations of web based graphic engines by allowing for large landscapes to be generated, as opposed to being loaded as a file. The file is then created locally rather than being transferred from a server, which saves on bandwidth. This does create the need to generate the content rather than reading in a file, the benefits of which are discussed later on in this paper.

There are other implementations of procedural generation used within many graphical system, however ours has focused on the use of platform independence. This means that various limitations impact the design of the set of utilities. One such limitation is the maximum number of vertices which are able to be drawn in one draw call. This in turn creates the need to split up the scene and make multiple draw calls for one frame if necessary. Additional limitations are discussed throughout this paper.

Figure 1 shows a procedurally generated scene of a landscape, which was generated using the framework and utilities we report in this paper. The remainder of our article is structured as follows: In Section 2 we discuss to use of procedural generation, with a focus on the Diamond Square algorithm. Section 3 covers results from this research, while Section 4 talks about what we found, and problems were encountered. The final section, talks about what we have completed and where this research will lead to in the future.

2 Procedural Generation

Procedural generation of a landscape allows for a created environment to be based on a set of pseudo random numbers. Creating content in this manner allows for a developer to simply create unique or predefined content. The significant advantage of generating content this way, is that it allows for additional data to be assigned to each point. An example of this would be what kind of terrain it is, or if you want to have something else spawn at that location. The Diamond Square method combined with our created utilities means the Z value is the only one affected in our mesh, leaving the original environment dimensions with a minimum amount of change.

The Diamond Square implementation is a method of creating and changing the height value of a location in a square map. This method works by taking two steps, the diamond and square steps. For the Diamond step four points of a square are use to find the center point, this point has its elevation changed, in our case the Y value. The change in value is based on an offset from a random number, followed by using the average of the original four corner points. The square step involves taking the center point and using that to allow for the subdivision of the main map area, each segment being a quarter of the original size. This process is repeated till the map is unable to be divided anymore. Using the Diamond Square algorithm allows for a set of predefined pseudo random numbers to be provided to recreate a previously created environment.

Within the process of creating a procedurally generated terrain we take in the information of an area surrounding a point within the mesh, and based on that are able to define certain aspects for that point. Creating additional properties within a generated terrain like this allows for areas to have more details applied to them, and to influence surrounding areas. An example of this would be to have a tree spawn if a set condition was met, then having that tree at that location would increase the chances of the neighbouring tiles also having a tree or some other previously specified object.

To create a mesh which is able to store these details normally we would need to use a 2D array. As JavaScript does not currently have native support for 2D arrays this left two options for storing the data. The first option was to use a single array, and manipulate that to work as it would if it was a 2D array. This is done by adding supplementary information to the index when searching for a data point, such as line size, and height. The other option for creating a 2D array was to create an array, then append another array to each element within the original array. This latter method did create some unique problems when trying to access an element, but proved to be simpler in the conceptual stages of this research.

The utility uses pseudo random numbers to allow for a created scene to be reproduced. Pseudo random numbers, as the name suggests are random numbers which are not truly random. While the process of generation allows for seemingly random numbers to be produced, they are all based upon the originally selected numbers (seeds). Using this method allows for a scene which has been produced to be recreated by reusing the same base numbers. These utilities allow for the passing of predefined seeds, but if none are passed then a seed based on time is produced.

Additionally as these utilities aim at being platform independent the GPU on the device is able to help improve performance in several key ways. Firstly we are able to have the GPU colour parts of the environment based on height and location information which is passed in as the x, y, and z coordinates of a point. Using the GPU to do these calculations reduces the quantity of data that needs to be passed from the CPU to the GPU at any given time, as well as freeing up the CPU for other calculations. Another advantage of using the GLSL with the GPU is that the utilities designed allow for data which has been processed on the GPU to then be returned and worked on again. This in turn allows for a cyclic flow of information.

Algorithm 1 Diamond Square Algorithm, Square Step.

```
declare $x, y, size, half, offset, avg, vertices[]$
half = size/2
point1 = vertexes[x - half][y - half]
point2 = vertexes[x + half][y - half]
point3 = vertexes[x - half][y + half]
point4 = vertexes[x + half][y + half]
avg = (point1 + point2 + point3 + point4)/4
vertices[x][y] = avg + offset
```

Algorithm 1 shows the Square step in the Diamond Square algorithm, which is taking the four surrounding corners of a point, and their corresponding value to find an average value. Then an offset is added to find average, and it is applied to our central point.

Algorithm 1 shows the Diamond step in the Diamond Square algorithm which is taking the top, bottom, left and right points surrounding a central point, and their corresponding values to find an average value. Then an offset is added to find average, and it is applied to our central point.t.

Algorithm 2 Diamond Square Algorithm, Diamond Step.

```
declare $x, y, size, half, offset, avg, vertices[]$
half = size/2
point1 = vertexes[x - half][y]
point2 = vertexes[x + half][y]
point3 = vertexes[x][y + half]
point4 = vertexes[x][y + half]
avg = (point1 + point2 + point3 + point4)/4
vertices[x][y] = avg + offset
```

The algorithms above are the key steps within this procedural generation implementation. The overall system works using a 2D array that is recursively subdividing itself by taking a point and initially using the Diamond step to assign a value to the center point. The Square step used next involves initialising the points above, below and to the left and right of the center point. Once this is done, the area is split into quarters, and the steps repeat, but now in a smaller section using a reduced size value. This process will repeat until all the points within the 2D array have become initialized.

Figure 2 shows the process of moving through a 2D mesh, using the Diamond Square algorithm. We can see in the first image the selection of the four corners, from this we move on to initialize the center point shown in tile to the right - this part was the Diamond step. Next we apply the Square step to the mesh, filling in the data for the center of each edge. Once we have completed both steps the area which we work in is reduced, and then the steps are repeated. The last part of figure 2 shows the final step; the Square step. Once applied, the whole grid has been initialized with a value which can then be used for the height of a location in the mesh.

3 Experimental Results

The system created is built up of several key parts; firstly the implementation of the Diamond Square algorithm to build the procedural generation for the utilities. This included simplifying the process of creating a scene, by using created functions. The optimization of code, and utilisation of GPUs have played a role in improving the system in speed and in bandwidth consumed. Finally the limitation of vertices able to be rendered in one draw call needed to be addressed to allow for larger terrains. Each of these parts played a vital role in creating this system, and will be expanded upon below. A performance review is also included so as to see the resulting improvements over other methods.

Implementing the Diamond Square algorithm required the use of several steps as explained in figure 2. This research aimed at producing a simplified method of implementing

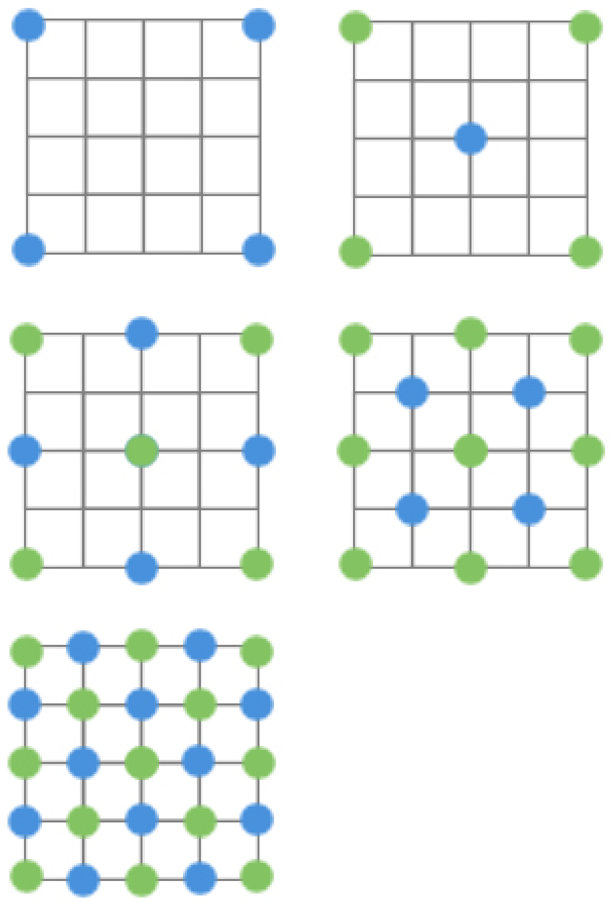

Figure 2: The Process of the Diamond Square Algorithm on a 5x5 grid

these, along with allowing for customisation of additional data points. This was achieved by creating several functions which would set up the scene, and create the terrain. The produced environments can be based on various map sizes, and heights, along with various colors or textures if required. This resulted in allowing for a scene to be set up and rendered using fewer lines of code, as well as allowing for more complex environments to be produced if required.

For optimizing the algorithms and effects applied much of the work was passed to the GPU of the device. When a scene is created, based on the required setup, the GPU will use different shader code, producing different effects. The simplest and greatest improvement was achieved by having the coloring of an environment based on the x, y and z coordinates of the vertex. The result of this style of processing helped to minimise the data which needed to be passed between the CPU and GPU, along with reducing the work carried out on the CPU. By comparison, other implementations require that additional texture information be passed to the GPU, including imaging and coordinate data. These techniques, though they can lead to some visually creative effects, do not fit well with the idea of procedural generation.

With WebGL we are limited by the number of vertices that are able to be drawn in a single GL.DrawElements function, some steps are needed to overcome this. The limitation occurs due to the indices using 16 bit for each data point, causing wrap around and other ill effects when trying to render over the limit of 65000 points. To work around this requires the use of multiple draw calls, based on multiple index arrays. This method splits up meshes which are over this limit into several smaller ones, and creates relating indices for each, along with using a repetitive draw function which will loop through each draw call and thus mesh.

The performance of this research plays a large role. To check this, an obj file was produced using the Diamond Square algorithm. For a mesh the size of 256 x 256, a file of 3.5 mb was produced. This would take several minutes to load in the file and set up the arrays in order to be rendered. Comparatively, a landscape created in JavaScript using the same algorithm was able to be loaded and set up within a matter of milliseconds. This is because when loading in the file it would need to be read, and parsed from a string to float. As seen in figure 3, it can be seen that a mesh of 256x256 would only take 34.8 ms to generate and load. This shows that a landscape which is generated is able to have a much larger scope, due to improved performance. Another upside of this method is that it saves in bandwidth as a large file does not need to be passed as only a small set of numbers are used to produce the area.

With these results it is clear to see the improvements made to generating terrain with WebGL for three reasons. Firstly, the speed in which a scene can be set up and rendered has been improved upon. Secondly, the limitations in the maximum numbers of vertices able to be rendered in one draw call have been worked around. Finally, we include optimizing the use of the GPU within this area of study.

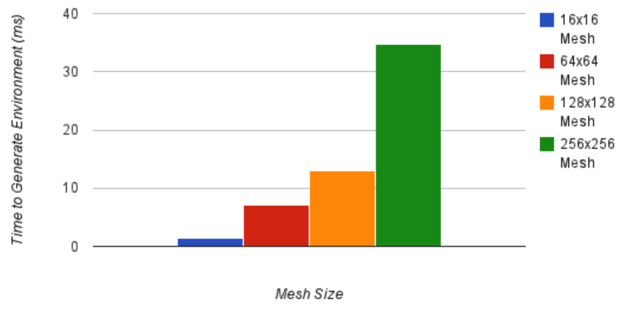

Figure 3: **Terran Generation 64x64**: Time taken to generate environments in various sizes

Figure 3 Shows the time taken for meshes of various sizes to be rendered. A mesh of 16 x 16 takes on average 1.5 milliseconds, while one of 64 x 64 takes 7.1 milliseconds.

A larger area such as 256 will still only take part of a second to create, with a timing of 34.7 milliseconds to generate the terrain.

4 Discussion

This research has resulted in various improvements in generating unique scenes in web based applications using WebGL. These include; the improved performance of producing unique terrain within a WebGL application, the ability to reuse these improvements by using a created set of utilities in future areas of research, scaling an area based on a the performance of a device, and the required level of detail has being addressed, along with created applications being support across a range of devices. Various landscapes have being produced, and have created challenges rendering.

The ability to create dynamic and various environments simply by using different random numbers allows for a larger and more immersive scene. This has been achieved through the use of the Diamond Square algorithm, as seen in figures 4, 5 and 6. Figure 4 displays a smaller mesh which is 16 x 16, this was generated in 1.5 milliseconds, but evidently it is very jagged as it has a low polygon count. In Figure 5 it is clear that by increasing the size the mesh produces a more fluid landscape, and this can be seen improving across all the meshes produced, such as in figure 6 and figure 1. This allows for a more unique terrain to be created with WebGL.

As research is able to be integrated into other projects, it is necessary to allow for it to be built upon and customised. One feature is that each data point of the mesh can have additional associated values assigned to it. This allows for a graphic engine to know when and how to produce another object within a scene. To produce a project using these utilities helps remove the need to manually set up a WebGL environment as this has been replaced with smaller functions to work with the other functions within the utility.

As a range of devices are able to run applications based on this research it is necessary to allow for the environment to be generated based on the device. The range in support hardware lead to the need to scale an area based on the processing power available. Conventional PCs and laptops ran the application smoothly across all ranges tested, but the larger terrains were slower to render on tablets and smart-phones, due to the limitations within the GPU. With the rendering side being affected it is necessary to, in this case, optimise the application to maximise the use of the CPU rather than the GPU. The process of creating the environment was slightly slower, but did not affect the performance of the application as greatly as the rendering. To overcome this we used smaller meshes with a smaller height limitation. This meant larger areas needed to be covered by each produced triangle, but with a smaller rise, which produced a smoother effect, and allowed for lower end devices to run with noticeably improved performance.

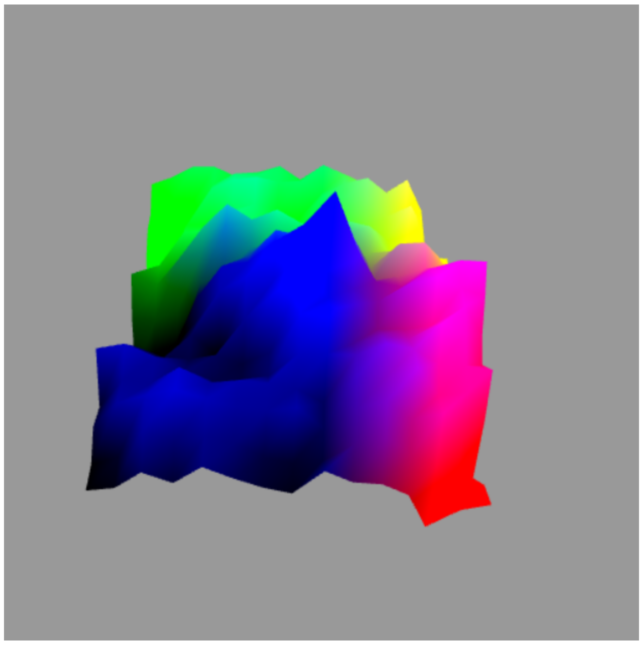
Figure 4: **Terrain Generation 16x16**: Landscape produced using the Diamond Square algorithm in WebGL. Map size is 16x16, this does correctly create a environment, but the edges are sharp, and could easily be improved on.

With many of the produced landscapes exceeding the limitations of a single draw call within WebGL, this issue needed to be addressed. The cause was found to be the use of a 16 bit short to store the indices. This is intended to allow for applications to be cross platform, and work on a range of hardware. To overcome this, when necessary, once the mesh is created it will be split up, and each part draw in a different draw call. This is completed by creating a class which stores the numbers of meshes, along with each mesh within it. When the draw cycle begins, it is looped through using a different set of indices each time, for each mesh.

5 Conclusions

This research has produced a set of utilities to improve upon the limitations of WebGL and platform independent based graphics. The use of procedural generation enables the process of creating unique environments. Optimizations were made to maximise the use of the GPU, along with minimising bandwidth. The speed at which an environment is created has greatly increased, as the scene can be created without the need to load in a new file. The ability to run applications using this is also taken into consideration with the use of various techniques to create a smooth environment for the user.

While undertaking this research various problems were found with the current methods of achieving much of our

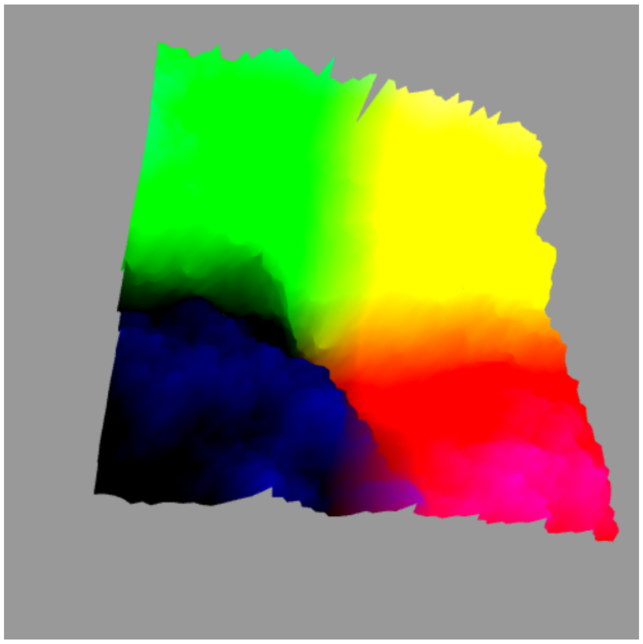

Figure 5: **Terrain Generation 64x64**: Landscape produced using the Diamond Square algorithm in WebGL. Map size is 64x64, here we see that the edges have become smaller, and smoother, helping to create more realistic environments.

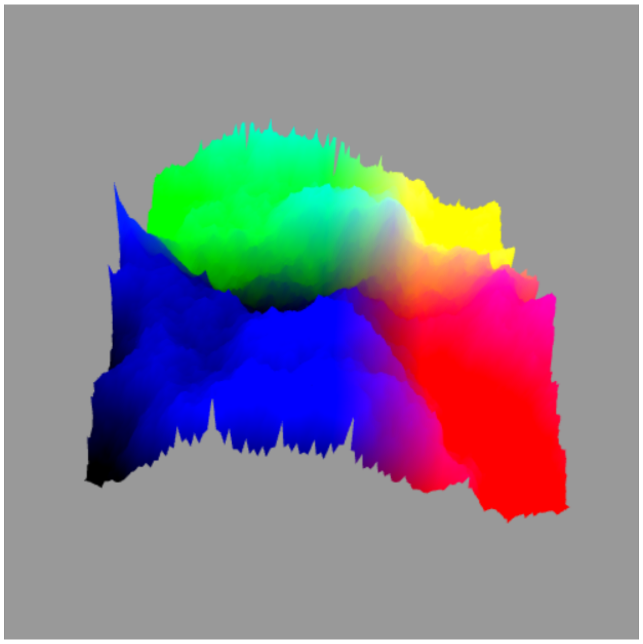

Figure 6: **Terrain Generation 128x128**: Landscape produced using the Diamond Square algorithm in WebGL. Map size is 128x128, another improvement in creating a realistic scene within WebGL with noticeably fewer sharp edges, and a greater range in terrain created.

aim. These issues were comprised of the limit in elements able to be drawn at a single time, the size of a created mesh when loaded from a file, and the methods used to help smooth jagged edges.

In future this set of utilities will be added to, improving the creation of procedurally generated environments, with the addition of trees and water flow. It is hoped that with additional features such as these, creating a fully interactive and high quality environment will become simple and effective. We hope that once these future goals are complete, the use of procedural generation will make a substantial effect on how 3D applications are created and used in the Web.

In summary, procedural generation implemented with JavaScript has been shown to achieve significant reduction in bandwidth requirements and this is particularly useful in web-oriented scene generation applications.

References

[1] Anttonen, M., Salminen, A.: Building 3d webgl applications. Tech. Rep. Report 16, Tamperer University of Technology, Finland, Department of Software Systems (2011)

[2] Bailey, M., Cunningham, S.: Graphics Shaders - Theory and Practice. CRC Press, second edn. (2012), iSBN 978-1-56881-434-6

[3] Cantor, D., Jones, B. (eds.): WebGL Beginner's Guide. PACKT (2012), iSBN 978-1-84969-172-7

[4] Cozzi, P., Riccio, C. (eds.): OpenGL Insights. CRC Press (2012), iSBN 978-1-4398-9376-0

[5] Emilien, A., Bernhardt, A., Peytavie, A., Cant, M.P., Galin, E.: Procedural generation of villages on arbitrary terrains. Vis. Comput. 28, 809–818 (18 April 2012)

[6] Hearn, D., Baker, M.P.: Computer Graphics with OpenGL. No. ISBN 0-13-015390-7, Pearson Prentice Hall, third edition edn. (2004)

[7] Hendrikx, M., Meijer, S., Velden, J.V.D., Iosup, A.: Procedural content generation for games: A survey. ACM Trans. on Multimedia Computing, Communications and Applications 9(1), 1–22 (February 2013)

[8] Huisman, P.: Procedural content generation with use of a domain-specific language - Nature's recursive nature and other natural phenomena. Master's thesis, Centrum Wiskunde and Informatica, Universiteit van Amsterdam, Netherlands (15 August 2012)

[9] Ilcik, M., Wimmer, M.: Challenges and ideas in procedural modeling of interiors. In: Proc. Eurographics Workshop on Urban Data Modelling and Visualisation. pp. 29–30 (2013)

[10] Khaled, R., Nelson, M.J., Barr, P.: Design metaphors for procedural content generation in games. In: Proc. ACM CHI'13. Paris, France (27 April 2013)

[11] Kim, J., Kim, D., Cho, H.: Procedural modeling of trees based on convolution sums of divisor functions

for real-time virtual ecosystems. Computer Animation and Virtual Worlds 24, 237–246 (2013)

[12] Leist, A., Playne, D.P., Hawick, K.A.: Exploiting Graphical Processing Units for Data-Parallel Scientific Applications. Concurrency and Computation: Practice and Experience 21(18), 2400–2437 (25 December 2009), CSTN-065

[13] Longay, S., Runions, A., Boudon, F., Prusinkiewicz, P.: Treesketch: Interactive procedural modeling of trees on a tablet. In: Proc. Eurographics Symp. on Sketch-Based Interfaces and Modeling (2012)

[14] Lopes, R., Hill, K., Jayapalan, L., Bidarra, R.: Mobile adaptive procedural content generation (2013), delft University of technology, Netherlands

[15] Mandelbrot, B.B.: The Fractal Geometry of Nature. W.H. Freeman (1982)

[16] McMullen, T.H., Hawick, K.A.: Webgl for platform independent graphics. Tech. Rep. CSTN-185, Computer Science, Massey University, Auckland, New Zealand (October 2012), in 8th IIMS Postgraduate Conference

[17] McMullen, T.H., Hawick, K.A.: Improving platform independent graphical performance by compressing information transfer using json. In: Proc. 12th Int. Conf. on Semantic Web and Web Services (SWW'13). p. SWW4052. No. CSTN-174, WorldComp, Las Vegas, USA (22-25 July 2013)

[18] McMullen, T.H., Hawick, K.A., Preez, V.D., Pearce, B.: Graphics on web platforms for complex systems modelling and simulation. In: Proc. International Conference on Computer Graphics and Virtual Reality (CGVR'12). pp. 83–89. WorldComp, Las Vegas, USA (16-19 July 2012), cSTN-157

[19] Muehl, W., Novak, J.: Game Development Essentials - Game Simulation Development. Delmar (2008), iSBn 978-1-4180-6439-6

[20] Novak, J.: Game Development Essentials - An Introduction. Delmar, 3rd edn. (2012)

[21] Pirk, S., Stava, O., Kratt, J., Said, M.A.M., Neubert, B., Mech, R., Benes, B., Deussen, O.: Plastic trees: interactive self-adapting botanical tree models. ACM Trans. Graph. 31(4), 50:1–10 (Jul 2012)

[22] Powell, T.A., Schneider, F.: JavaScript: the complete reference. McGraw-Hill (2012), iSBN 9780071741200

[23] Prusinkiewicz, P., Lindenmayer, A.: The Algorithmic Beauty of Plants. No. ISBN 978-0387972978, Springer (1990)

[24] Saunders, K.D., Novak, J.: Game Development Essentials - Game Interface Design. Delmar, 2nd edn. (2013), iSBN 978-1-111-64288-4

[25] Vanegas, C.A., Kelly, T., Weber, B., Halatsch, J., Aliaga, D.G., Muller, P.: Procedural generation of parcels in urban modeling. Eurographics 31, 681–690 (2012)

[26] Wright, R.S., Haemel, N., Sellers, G., Lipchak, B.: OpenGL Superbible. No. ISBN 978-0-321-71261-5, Pearson, fifth edn. (2011)