JAMES CLAYTON

1501082@uad.ac.uk

Abstract

An Explanation and Evaluation of a Desert Environment  
 Generated Using Lindenmayer Systems and Improved Perlin Noise.

Procedural Desert

CMP 301 – Procedural Methods

Introduction

The application demonstrates a procedurally generated Desert Landscape, complete with procedural plants, topography, texturing and a post-processing edge detection effect. The application was built using the provided Rastertek DirectX11 framework.

Features

The main feature of the application is procedural cacti, generated using a Lindenmayer System (or L-System for short.) Specifically, the application features a Stochastic Turtle L-System.

The terrain is modelled using Improved Perlin Noise and two smoothing algorithms. This terrain features slope-based texturing.

The terrain can be reset to an unsmoothed state in which it is moulded only by Improved Perlin Noise using the spacebar.

A uniform smoothing can be applied to the terrain using the ‘X’ key, and a “plateau” smoothing algorithm may be applied by pressing the “C” key. The latter only smooths parts of the terrain above a certain height in order to produce a topography inspired by the Grand Canyon.

An Edge Detection post processing feature may be toggled by pressing the ‘P’ key.

Code Structure

The “Basic Shader” class is what could be described as a most basic shader class. It simply renders an object without applying any operations to the vertices or pixels. This was used in order to achieve downscaling and upscaling during post processing and served as a building block for more complex shaders.

The “Improved Noise” class contains a C++ implementation of Ken Perlin’s Improved Perlin noise. (Perlin, 2002) Instances of this class contained the required permutation table and the gradient calculations. A “Sample” function can be used to sample the Improved Noise at a point specified by the function’s parameters, which correspond to the X, Y and Z coordinates of a specific point in the noise.

The “Sky Dome Class” and “Sky Dome Shader Class” are based on the Rastertek Tutorial, and are largely unchanged as the Noise Texture Class was not successfully implemented. As it stands, it merely provides an element of visual polish to the scene and nothing more.

The “Noise Texture Class” was not successfully implemented for the purposes of the project, but in theory would have programmatically created a noise texture using the Improved Noise class. This texture would have then been passed over to the “Sky Dome Shader Class” and panned across the sky dome in order to achieve basic cloud cover.

For generating the L-System, a bespoke “LSystemClass” was made which contains the axiom and the various rules for creating permutations of the axiom. Specifically, this is a simple Stochastic L-System, meaning that a selection of rules are available, and are applied to the axiom based on a prescribed probability.

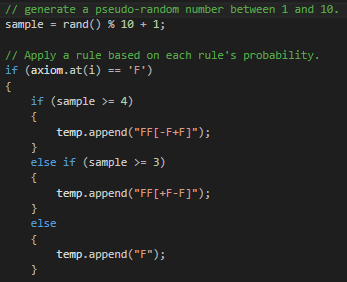
This function included access to the Improved Noise class in order to use the noise for pseudo-random variations in the different plant instances, although this was not successfully implemented. The std::rand() function is used instead for similar results.

Each Cactus is created using the Cactus Class. The model for each branch is loaded into memory, and then multiple instances of the branch model are created. Each instance has a bespoke transform matrix that defines its location relative to the overall cactus. The code for parsing the axiom generated by the L-System into 3D geometry transformations is contained within this class.

The L-System used the following symbols to create the 3D Geometry:

* “F”
  + Move out one segment’s length.
  + Spawn a new segment.
* “+”
  + Rotate by a positive angle in the X-Axis.
* “–“
  + Rotate by a negative angle in the X-Axis.
* “<”
  + Rotate by a positive angle in the Z axis.
* “>”
  + Rotate by a negative angle in the Z axis.
* “[“
  + Begin a new branch.
* “]”
  + End a branch.

Several rules were tested to varying results (as shown in the Reflection.) The final ruleset to generate the scene featured in the Results section is as follows:



A stack of matrices is used to implement the branching behaviour; each segment’s rotations and transforms are applied first, and then multiplied by the transform of the previous segment to position it correctly at the tip of the previous segment. When a branch is begun, the current “parent” transform is pushed onto the stack, and then the process of adding on segments continues. Then, once a branch ends, the next segment to be created is spawned relative to the matrix that was recorded on the stack. By sensible pushing and popping this matrix stack, complex branching structures can be created with relative ease.

The “Convolution Shader Class” is used in the Application’s post processing. It is built upon the Basic Shader, with the key addition of an additional constant buffer containing the Convolution Kernel as a 4X4 matrix that is used to pass the Kernel into the pixel shader for image manipulation.

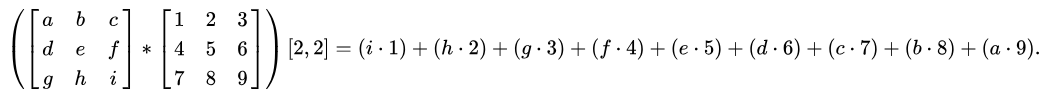
1. In the corresponding vertex shader, the 9 surrounding texels for a given texture are generated, calculated and passed into the pixel shader for manipulations. This process can be illustrated using the following diagram;

A picture containing indoor, object

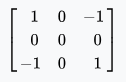
Description generated with high confidence

…although it should be noted that only the 9 texels in the red and orange bands were needed for the purposes of edge detection.

1. In the pixel shader, each texel and its neighbours are manipulated using values from the Convolution Kernel using the relatively straightforward maths outlined below:



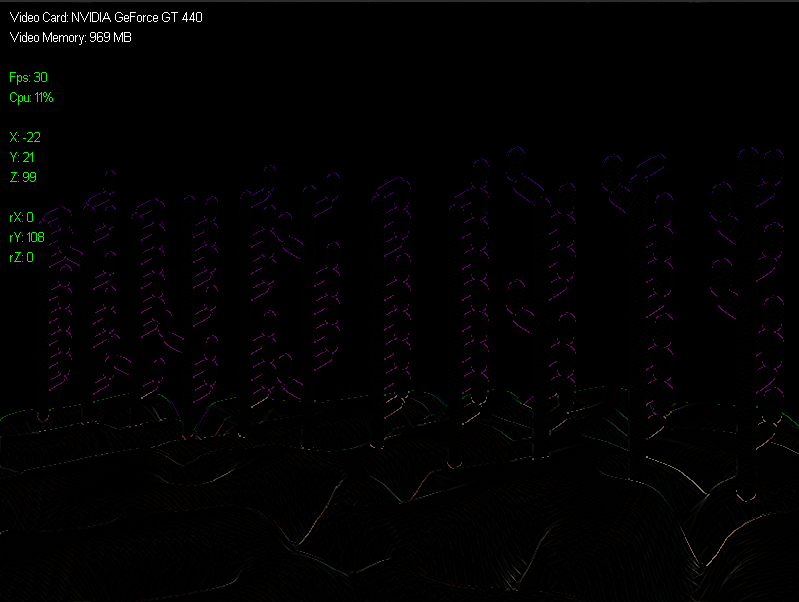
Simple edge detection was implemented in the Application, using the following Kernel;



[Both images taken from - <https://en.wikipedia.org/wiki/Kernel_(image_processing)> (Accessed 20/04/2018)]

Results





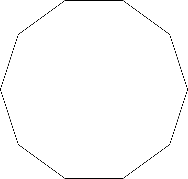
Critical Evaluation

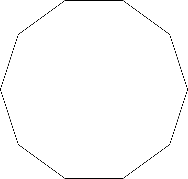
# Efficiency of Implementation

Performance wise, the Application’s biggest issue is that the transformation matrix for every sub-model within the cactus is passed in via its vertex input buffer. As a result, the amount of data being passed per vertex was more than double that of the standard position, normal and texture coordinates.

A more efficient alternative to this would have been to generate a series of point meshes rather than whole models. It would then theoretically be possible to use the Geometry Shader stage of the DirectX pipeline to generate a cylindrical model algorithmically.

In theory, if each point mesh has its own transformation and rotation, then it would be possible to mathematically extrapolate that into a plane, and then “stitch” a cylinder from one theoretical plane to the next as illustrated below. If successful, this would ensure that no joints were visible between branches, and could be used to make the branches get progressively smaller as their distance from the “trunk” of the plant increased.





It may be noted that applying post processing to the scene reduces the application’s frame rate significantly. Testing indicated that the bottleneck lies with the approach to downscaling the texture, as the performance drop exists so long as the multiple renders take place regardless of the application of the actual post processing. A possible solution to this would be do downscale the initial render texture of the scene rather than by rendering 3 separate orthographic meshes in sequence.

# Code Structure

Inherited from the Rastertek Framework, there’s clearly a lot of redundant code, particularly the functions that each shader class has in common (SetShaderParameters, for one.) Re-structuring the code making greater use of inheritance to reduce this redundancy would likely make for much smaller files and a cleaner interface for creating objects.

Furthermore, there are several functions that take parameters by value, rather than by reference. Again, SetShaderParameters is the worst offender, taking at least 3 4X4 matrices as parameters. While this may not be a major performance bottleneck, it’s still worth bearing in mind as it is bad practice.

On a similar note, there are a total of four defined matrices in the Cactus Class that correspond to:

1. A positive rotation in the X-Axis.
2. A negative rotation in the X-Axis.
3. A positive rotation in the Z-Axis.
4. A negative rotation in the Z-Axis.

Instead of this, it may have proved prudent to use the “D3DXMatrixRotationYawPitchRoll” function to encapsulate all of the rotations in a single matrix. This would then have had the benefit of making it easier to apply subtle variations in the angles by which each segment is rotated. Essentially, that would have meant defining a float between 0 and 2π during the appropriate section of L-System parsing and then using that as the rotation value.

# Procedural Content

The terrain generation is fairly simplistic, and not especially realistic. The application as a whole would have benefited from a more sophisticated terrain modelling technique, such as mid-point displacement, or perhaps a combination of techniques.

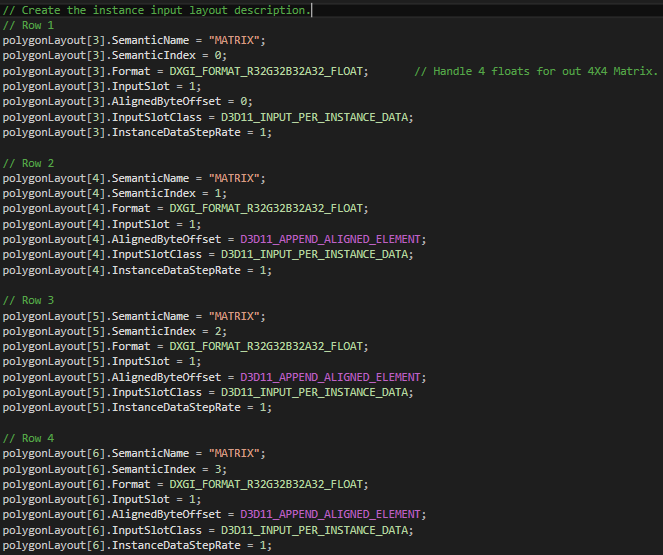
As was mentioned previously, procedurally generating the geometry of each branch segment would have made the resulting cacti far more believable and aesthetically pleasing. This addition would also lend itself quite nicely to generating multiple species of cacti, or other plant species entirely. Were the L-System class refactored to serve as a parent class, and different species of plant creating using bespoke child classes, then multiple species of plant may have been created in much less time than the initial cacti, as much of the code that had to be developed could have been repurposed.

Similarly, it may have been possible to use a variation of the incomplete “Noise Texture Class” to generate a texture based on a 2D L-System. This texture could then have been applied as a texture to a number of small quads to produce a number of smaller plants scattered throughout the scene.

As mentioned earlier, the project would have featured procedural clouds in the sky above the scene, but this could not be successfully implemented in time.

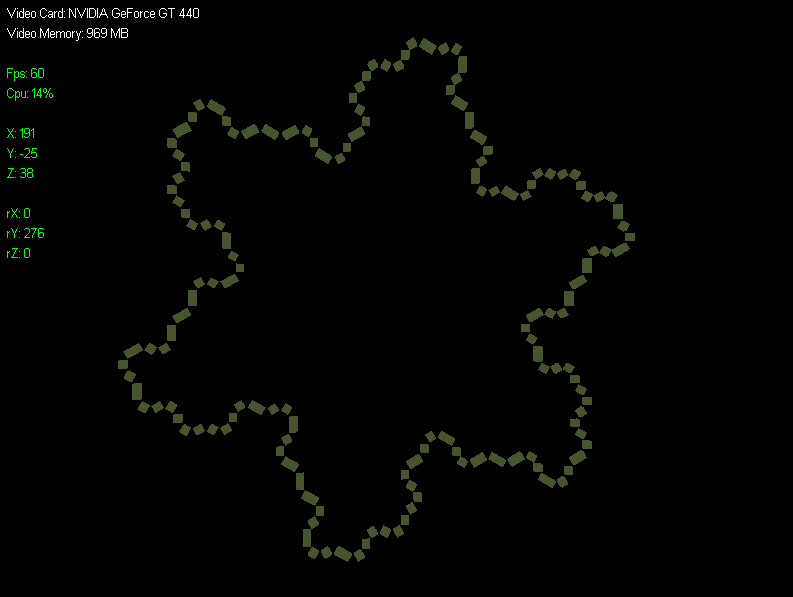
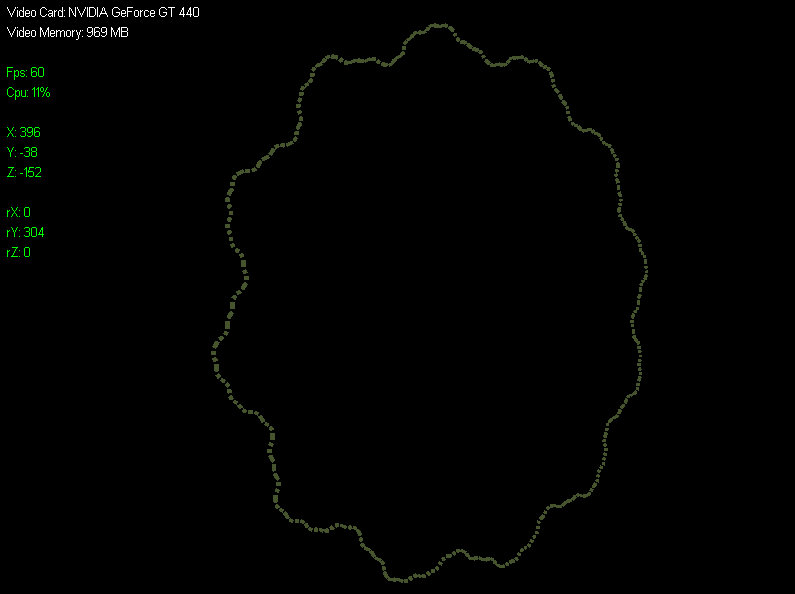
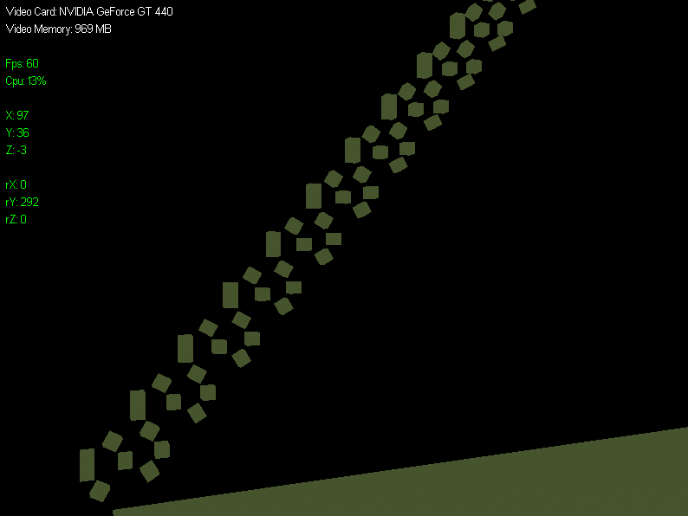
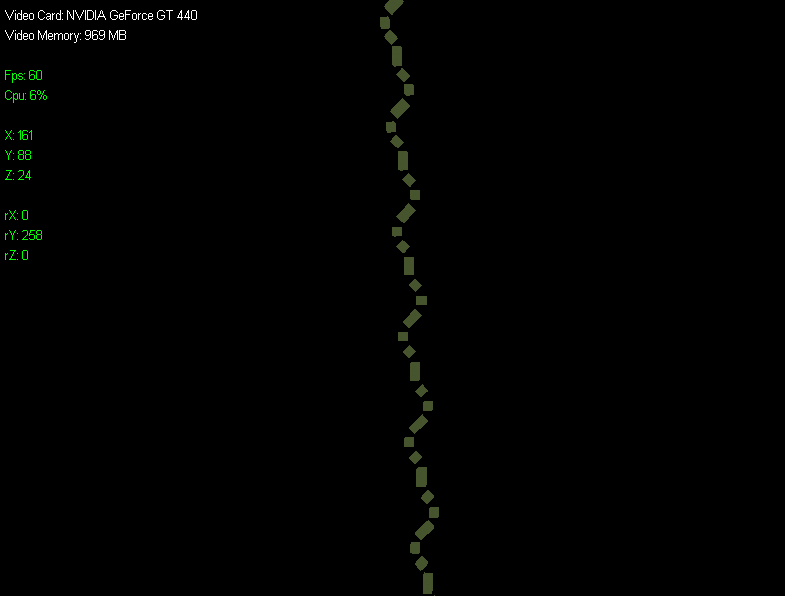
Reflection

A good deal of time had to be spent researching DirectX and similar APIs for various techniques and features that would allow passing a bespoke matrix for each segment of the plant. Using a constant buffer was immediately ruled out, as all of the segments are generated using the same Shader Handler, and thus each need their own matrix. Instead, the vertex input buffer was expanded to accommodate a matrix as pictured below:



While this was inspired by OpenGL’s more extensive use of matrix stacks to manipulate geometry, then final implementation is what’s referred to as “Hierarchical Modelling,” a common basis for skeletal structures used in 3D computer animation and rigging. It is worth mentioning that DirectX once featured a specialised Matrix Stack class, but this was deprecated in DirectX9 (reference.) and the essential functionality was ultimately achieved using the stack implementation from the C++ Standard Library for the purposes of the Application.

Perhaps the most surprising lesson learned from the project was the extensibility of Lindenmayer Systems. The technique itself has humble origins describing the growth of simple microscopic *Anabaena catenula (*Prusinkiewicz, P. and Lindemeir, A. 2004,) and even more sophisticated variations have relatively easy to understand theory at their core. However, even a single implementation of the technique can be remarkably flexible; below are some of the results generated during testing the basic Turtle L-System (before branching was implemented) using example rulesets. (Heijltjes, W. 2008)



The last two images are actually generated using the exact same rules, and have only had the angle by which instance is rotated changed from 30° to 60° and the number of instances adjusted.

Implementing the branching behaviour proved to be tricky to get working properly, largely because an unfortunate rule set can make it difficult to determine if the branching is working or not. A series of simple branching structure with rotations limited to the X-Axis were created to prove that branching had been successfully implemented.

A picture containing device

Description generated with very high confidence

Pictured – A simple two branch structure generated from the axiom [F+F+F+F+F+F+F][F-F-F-F-FF-F] without any iteration.



This was then expanded with axiom iteration to produce a more complex structure based on the previous simplified version.

A close up of a logo

Description generated with very high confidence

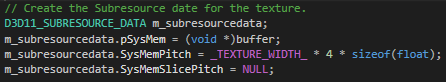
This was then expanded further to prove that it was possible to make branches begin on other branches.



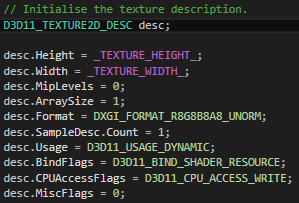
At this point, the cubes were swapped out for cylinders for aesthetic reasons, and the system was expanded to include rotations In the Z-Axis. The cylinder models were created in Autodesk Maya 2017, and then converted to .txt using the converter provided by Rastertek (Rastertex, 2016)

Using Improved Perlin Noise to mould the terrain proved surprisingly easy and effective, although using Noise to calculate probabilities for the stochastic element of the L-Systems proved more difficult, and was not successfully implemented in time. It would also have been worthwhile to spend more time experimenting with the terrain manipulation, but the plants were the focus of the project so this element was not developed as far as it could have been.

Research was carried out into generating texture resources programmatically, although this feature was not successfully implemented. In theory, this would have been done by creating an array of values from the Improved Noise that was the same size as the texture being generated. Then, sub-resource data would have been created using this array.

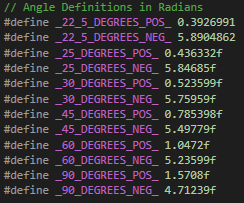


A 2D texture would then have been created to eventually store this sub-resource data.



Then, a shader resource view would have been used to transfer this data over to the GPU. This feature was not successfully implemented, so the validity of this approach is unconfirmed.

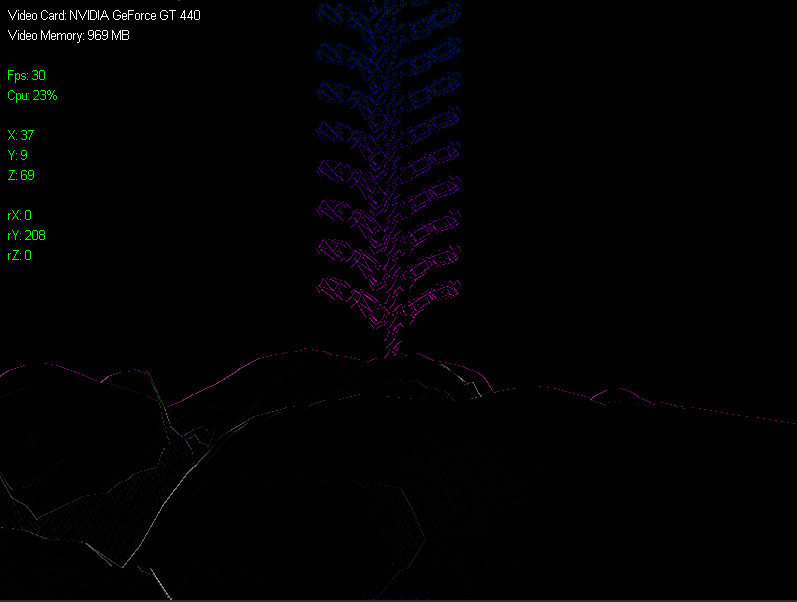
When performing matrix rotations, the rotation function that the framework supplied operates in radians, not degrees, so the project includes definitions for some common angles for readability’s sake. Using definitions instead of variables to do this achieves effectively the same results, but without the additional memory overhead.



Before developing the post processing too far, it was found helpful to start by rendering a smaller test window before replacing the entire scene with the post-processed image.

In this case, it revealed early on that the post processing was non-functional not because of the convolution, but because of the matrices being used.

Post processing via convolution shader was surprisingly easy, and the results of the edge detection kernel used exceeded expectations. What started as a basic rendering shader handler (namely the Basic Shader Class featured in the application) was easily extended to produce more sophisticated post processing effects using the convolution kernel, as the technique is surprisingly flexible.



References/Bibliography

L-Systems:

Heijltjes, W (2008) *Turtle Graphics and L-Systems.* Available at: http://www.inf.ed.ac.uk/teaching/courses/inf1/fp/2008/tutorials/Tutorial7.pdf (accessed 03/04/2018)

Prusinkiewicz, P. and Lindemeir, A. *The Algorithmic Beauty of Plants* (2004) Available at: <http://algorithmicbotany.org/papers/abop/> (accessed 23/03/2018)

Smoleňová, K. *L-Systems*. Available at: <http://www.sccg.sk/~smolenova/elearning/ks_fmfiuk06.pdf> (Accessed 02/04/2018)

Noise:

## Perlin, K. (2002) *Improved Noise reference implementation*. Available at: <https://mrl.nyu.edu/~perlin/noise/> (Accessed 02/02/2018)

Post Processing:

Ahn, S.H. (2005) *Example of 2D Convolution*. Available at: <http://www.songho.ca/dsp/convolution/convolution2d_example.html> (accessed 02/04/2018)

Convolution Kernel images taken from - <https://en.wikipedia.org/wiki/Kernel_(image_processing)> (Accessed 20/04/2018)

Textures:

Surfacecurve (2014) *Grass Texture, Seamless 2D.* Available at: grass-textureseamless-2d (Accessed 17/02/2018)

Etory (2011) *Seamless Rock Texture 2.* Available at: https://opengameart.org/content/seamless-rock-texture-2 (Accessed 17/02/2018)

Tiziana (2009) *Orange-brown concrete wall 512px*. Available at: <https://opengameart.org/content/orange-brown-concrete-wall-512px> Accessed (17/02/2018)

Tiziana (2009) *Beige concrete wall 512px*. Available at: <https://opengameart.org/content/beige-concrete-wall-512px> (Accessed 17/02/2018)

Passing Matrices:

*Direct3D11 and SharpDX - How to pass a model instance's world matrix as an input to a vertex shader.* Available at: <https://gamedev.stackexchange.com/questions/64298/direct3d11-and-sharpdx-how-to-pass-a-model-instances-world-matrix-as-an-input> (Accessed 30/03/2018)

*SharpDX - How to set my vertex shader to take an instance's world matrix as input.* Available at: <https://www.gamedev.net/forums/topic/649603-sharpdx-how-to-set-my-vertex-shader-to-take-an-instances-world-matrix-as-input/> (Accessed 30/03/2018)

Rastertek Tutorials:

Rastertex (2016) *Tutorial 7: 3D Model Rendering.* Available at: <http://www.rastertek.com/dx11tut07.html> (Accessed: 09/03/2018)

Rastertex (2016) *Tutorial 8: Loading Maya 2011 Models.* Available at: http://www.rastertek.com/dx11tut08.html (Accessed: 09/03/2018)

Rastertex (2016) *Tutorial 10: Sky Domes*. Available at: <http://www.rastertek.com/tertut10.html> (Accessed 27/04/2018)

Rastertex (2016) *Tutorial 14: Slope Based Texturing*. Available at: <http://www.rastertek.com/tertut14.html> (Accessed 24/02/2018)

Rastertex (2016) *Tutorial 22: Render to Texture.* Available at: <http://www.rastertek.com/dx11tut22.html> (Accessed 21/04/2018)

Rastertex (2016) *Tutorial 36: Blur.* Available at: <http://www.rastertek.com/dx11tut36.html> (Accessed 21/04/2018)

Rastertex (2016) *Tutorial 37: Instancing*. Available at: <http://www.rastertek.com/dx11tut37.html>

(Accessed 09/03/2018)

Rastertex (2016) *Tutorial 3: Terrain Texturing.* Available at: <http://www.rastertek.com/dx11ter03.html> (Accessed 24/02/2018)

GitHub Repository:

https://github.com/JamesCLC/ProceduraMethods