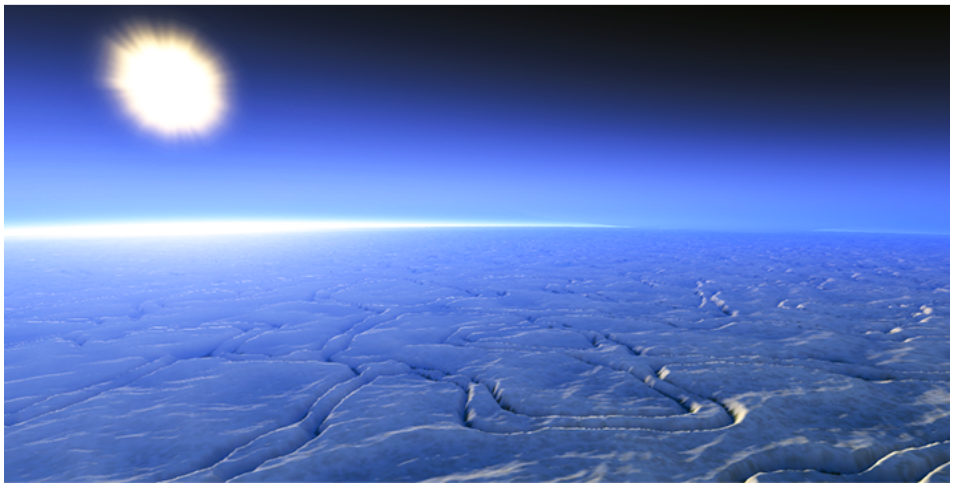
Individual Project MEng

STRUCTURED PROCEDURAL WORLDS WITH DATA INFERENCE



James Webb

Department of Computing

Imperial College London

MEng Computing 2014

Supervised by Abhijeet Ghosh

Abstract

Modern graphics applications require increasingly large and detailed environments. One area this particularly impacts upon is the rendering of terrains. Much work has gone into deriving new and improved level of detail algorithms that allow the rendering of very large terrains at various scales, but rather less into how to generate convincing content to populate them.

In this project we look at three main elements that come together to form parts of a larger system. Firstly we build a novel terrain system that facilitates rule-based construction of terrains with generation occurring on the GPU. These rules also allow the arbitrary mixing of procedural data with purely user generated content. Next we look at creating a graphics back-end to support rendering the generated terrains. Highlights of this section include a new hardware accelerated LOD technique based on CDLOD[41], a texturing system allowing an unlimited number of textures on the terrain surface and a storage system that minimises the number of draw calls sent to the GPU. Finally, we move on to looking at how we may infer procedural rules for the terrain system from user provided data. This includes matching user provided elevation data to fractals and reflectance data to shading and texturing rules. In this section we also look at a way of recolouring the physically based atmospheric scattering lookup tables generated using the technique proposed by Bruneton and Neyret in Precomputed Atmospheric Scattering[6].

In our evaluation, we examine the performance of the terrain system and graphics back-end and the expressiveness of system as a whole in its ability to create a wide range of landscapes. We found that generating GPU rule-based terrains is a highly performant option, offering in many cases order of magnitude increases over a comparable CPU implementation while also giving the user the ability to recreate a wide array of landscape styles. We also demonstrate that our graphics system as a whole offers roughly double the performance over more traditional methods, at the cost of additional memory usage.

Acknowledgements

I would like to thank my supervisor, Abhijeet Ghosh, for his willingness to supervise this project and for his continued support throughout.

Contents

List of Figures

1 Introduction

The rendering of terrains concerns the underlying mesh which determines the height and physical features, the shading and texturing which gives a realistic surface appearance and the placement of world objects such as rocks and trees. The particulars of how to do this has been the subject of research for many years. The applicability in many fields such as film, games and simulation combined with the ever increasing computing power available means there is always a demand to find new and improved ways of creating realistic terrain visuals. Visualisations can now be produced in real time that would have been limited only to time-intensive pre-rendering a few years ago.

Procedural generation concerns the algorithmic generation of what is typically art-based content. This has many applications, in the context of rendering terrains, procedural generation has been applied to all aspects of the task. From generating the physical displacements of the terrain to the models for the flora and fauna that inhabit it.

The overall aim of this project is to explore the creation of a complete system that allows a structured approach to creating procedurally generated terrains. This includes building a system of rules to define the procedural generation and creating a graphics back-end to support its rendering. We also look at how to infer rules for the terrain system from user data. Finally the project looks to deliver a polished user interface for creating these terrains to avoid becoming an entirely academic exercise.

1.1 Motivation

Although a great deal of effort has gone in to how best to render large and detailed terrains, there has been something of a void in how to create and manage the content to fill them. Artist-generated content is generally of the highest quality but is also expensive to produce and limited in scale. It would be entirely unrealistic for even a large team of artists to produce a compelling environment even the size of a small country, much less continents and entire worlds. The largest environments seen in games are usually found in open world games such as the GTA series, the Elder Scrolls series and World of Warcraft[11][43]. These typically span a area between tens and hundreds of square kilometres. Larger terrains can be found where real world data is used such as in flight simulators, although the detail in these maps is often very low and many areas may be completely flat with a low quality texture applied.

There is clear disparity in the size of terrains that can be rendered versus the amount of content that can be produced to populate them. As GPUs become more powerful, the detail that it is possible to render and therefore the amount of artist work required per unit area

increases. To deal with this, there are a number of possible solutions:

- Reduce the size and increase the detail of environments
- Keep environment sizes the same and keep the detail level in line with current technology
- Hire more artists. With budgets for AAA title already very high this seems to be reaching its upper limit.
- Generate some or all assets without direct input from artists - a higher level way of specifying features than modelling displacements and textures on a per vertex/pixel basis.

Although all these options have some validity, only the final one really seems to offer the potential for any innovative solutions. One approach to realising the final option is through the use of procedural generation - generation through the use of algorithms. Procedural generation has been used to create convincing terrains in software such as Bryce[1] and Terragen[40] but these are more focussed on producing photorealistic renders with raytracing than on being the basis for a real-time application. Procedural generation has been put forward as an answer to content generation in real time applications in the past, but there remains no truly convincing implementations. The results mostly lack visual quality and, perhaps more importantly, give the artist little control over the final appearance.

1.2 Objectives

This paper aims to deliver a new way of dealing with the creation and rendering of terrains that are highly detailed, realistic and as minimally constrained by technical limitations as possible. This will be realised through the culmination of various work, looking at:

- How to define vast terrains using a system of rules that empowers artists while reducing the time spent per unit area.
- How to build a graphics back-end that can render the terrains produced.
- How to infer rules for the terrain system from real world data.

We now consider these three components in more detail.

1.2.1 Structured procedural content generation and blending with user content

In this part of the project we propose a way of defining the content which determines the appearance of a terrain using a set of hierarchical procedural rules to be executed on the

GPU. The rules will allow the creation of spherical and planar terrains using the same rules by selections being defined in two dimensions, using either cartesian or polar coordinates (with radius being fixed). The system will also facilitate the seamless blending of procedural content with user generated content using rules which allow the streaming or static insertion of data. Although it will be possible to create terrains entirely procedurally with this system, and every effort will be made to deliver a high quality set of rules and tools to make them as good as possible, it is reasonable to expect that, at least in the near future, user created content will always be superior to procedural content. So the vision for this is that artists will be able to precisely create certain areas of the terrain and then 'fill the gaps' with procedural content that broadly recreates the style environment they are aiming for.
The rough classifications for the types of rules that will be available are:

- Selectors - shapes and other masking rules which define a selection of a certain area of the terrain. For example, a selection mask may be defined by a circle with a certain radius and origin or all terrain above a certain height.
- Modifiers - change a parameter of the terrain, such as displacement or surface type, within the area masked off my selector rules.
- Rule-sets - contain other rules and rule-sets. Rule-sets act as containers for the rules and provide structure to the system. They also have some special properties that provide additional control over the rules they contain.

The creation of this system also involves creating the supporting features in the graphics back-end. For example how to balance the GPU load and prevent pipeline stalls while simultaneously creating content and performing rendering.

1.2.2 Rendering vast terrains at any scale, without compromising artistic expressivity

For a terrain to appear realistic, features at all levels of detail need to be present. From the largest of mountains to grains of sand. With the current generation of GPUs it would be possible to brute-force the rendering of a relatively large terrain, say 20km by 20km with details down to around the meter level (see 2.1.1.1). This does not give a particularly large terrain, certainly not one of planetary size, nor does it give a particularly detailed one if the eye position was very close to the terrain. The key observation in getting around this is the further a camera is from a terrain feature, the less detail is required for that feature. For example, a small rock would be indistinguishable from other soil as the eye position moves beyond a few of meters. When rendering, this distance can be calculated using an error metric. From this observation, we arrive at the canonical way for terrains to be rendered at

a large scale. The terrain is divided into a number of smaller areas which are dynamically loaded in depending on the camera position and proximity. The data is created or steamed in as required and disposed of if no longer required.

The particulars of how to do this has been the subject of much research in the past, a selection of which we look at in the background material (see 2.1.2). In this project we look at how to extend and modify previous work to be in line with current GPU advances. We will also look at other components in the rendering of large scale terrains such as minimising draw calls and the ability to have an unlimited number of textures on the surface.

1.2.3 Inferring procedural rule-sets from photographs or other data-based sources

This is the most open ended part of the project where we look at how to infer a rule-sets for the previously detailed system which can mimic the features seen in pre-existing data, such as satellite imagery. The pipe dream for this part of the project is to be able to take an arbitrary photo, input it to the system and output a set of rules that can mimic all the various components of that terrain. More realistically, given the other elements of the project to be implemented, we will look at mapping elevation and reflectance data to the rules and noise types in the terrain system to provide a basis for the array of future work possible (see 8.1).

1.3 Contributions

The project's contributions can be broadly split into three categories:

- A system for the construction and execution of rule-based procedural terrains on the GPU using DirectX compute shaders (4.1). The construction is based on the selection and modification of different areas of the terrain. It also allows users to augment the generated data with their own data either directly through streaming/static loading of data into certain rules (4.2) or by using the data inference tools (1.3).
- A graphics back-end that will support, without imposing any limitations, the rendering of terrains expressible by the procedural system. The highlights of this system include:
  - A new terrain level of detail system that builds on previous work in CDLOD[41] to bring it in line with the latest graphics hardware and APIs by supporting adaptive hardware tessellation.
  - Support for an unlimited number of textures per terrain patch by introducing a rasteriser stage instead of using the traditional texture splatting approach. This stage uses real time BC1 and BC3 compression on the GPU to minimise the increased memory as much as possible.
  - A storage system for terrain that minimises CPU overhead by reducing draw calls.
  - A terrain patch generation pipeline which prevents stalls on the GPU.
- An initial investigation into how the rule-based system of construction can be used to infer procedural terrains from real data. This includes:
  - How to match generated noise types to real world elevation data
  - Generating rules to match colouring depending on gradients, heights
- As part of the data driven investigation, we also propose a method by which to recolour inscattering tables computed with the Precomputed Atmospheric Scattering paper with user data. This goes some way towards giving more artistic control to such systems, whose parametrisation is often not user-friendly and limited in range.

2 Background

2.1 Terrain

Terrains in real-time applications fall broadly into two categories, planar and spherical. Planar terrains are the most common and used in applications, including the majority of games, where the height and distance travelled by the camera in the scene are limited such that the flatness is not apparent. Spherical terrains are also used, particularly in applications such as flight simulators or Google Earth.

2.1.1 Digital elevation models

Terrains created by artists or obtained through data collection via LiDAR, IfSAR, etc are typically stored in one of two ways. Either as meshes or as heightmaps.

2.1.1.1 Meshes

Perhaps the most simple way to render a terrain is to create a mesh in a 3D package of choice (3ds Max[2] for example) and import it into the rendering engine. This can work fine for small and/or low detail terrains but does not scale well due to level of detail problems. Current GPUs can render around 2 billion polygons per second, which equates to being able to brute-force render terrains around $20km^2$ with regular grid vertices per metre (for a s_x by s_y grid, there are $\frac{1}{3}((vx * 2) * (vy–1) + (vy–2))$ polygons). Many applications require more detail and/or size than this, which leads to the use of level of detail algorithms (see 2.1.2.

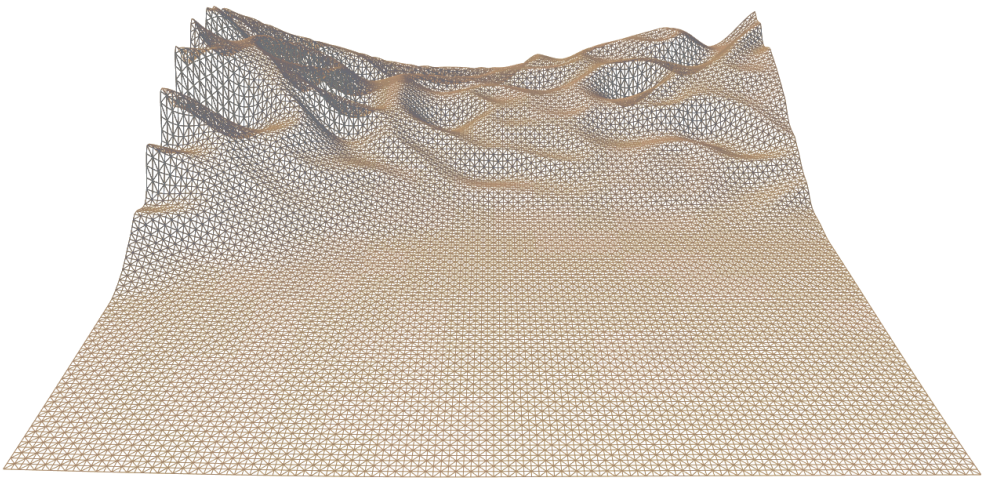


Figure 2.1: A brute force rendered terrain mesh with constant level of detail

2.1.1.2 Heightmaps

In a heightmap, surface displacement information is encoded as a greyscale image where the pixel value represents the height of that part of the terrain. For a planar terrain, height means a displacement in the direction perpendicular to the plane. For spherical terrains, height means a displacement in the direction of the radius through the surface at that point (i.e. the surface normal before any displacement). Actual implementations may use more than one colour channel if the image format being used does not provide sufficient precision as a greyscale image. This was common practice before the availability of higher precision texture types on the GPU, but is no longer necessary since the advent of single channel 16-bit or 32-bit textures being supported.

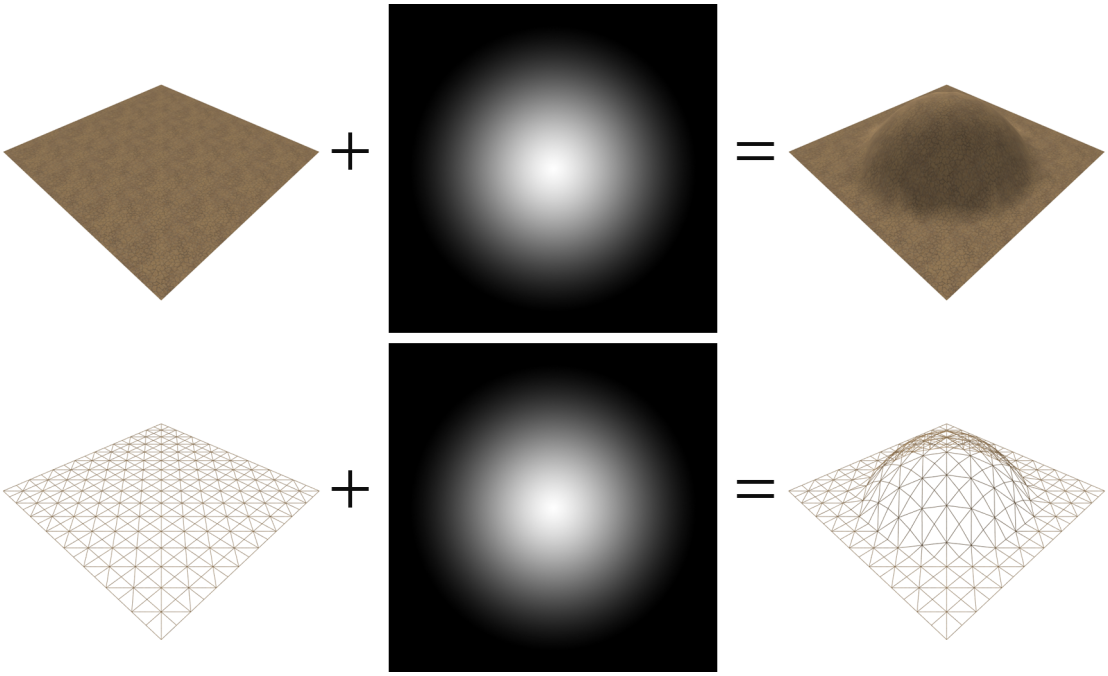


Figure 2.2: Applying a circular heightmap to a planar mesh

Pixel values are stored at regular intervals so the image represents a grid of samples from the terrain it encodes. The height values get mapped to a regular mesh grid with distance between vertices correlating to the spacing between the recorded samples in the heightmap. This is shown in figure 2.2 and schematically in figure 2.4.

There are two notable advantages to using heightmaps. Firstly it lends itself to dynamic level of detail changes (see 2.1.2). Secondly, it reduces storage requirements compared to meshes as only height data needs to be read from disk versus having to read 3D coordinates for each vertex in a mesh. The main limitation with is that they do not allow for overhangs

due to the use of regular grids. They also do not render very steep gradients well as grid polygons can be stretched over a large physical area resulting in a loss of apparent detail and texture stretching as seen in figure 2.3.

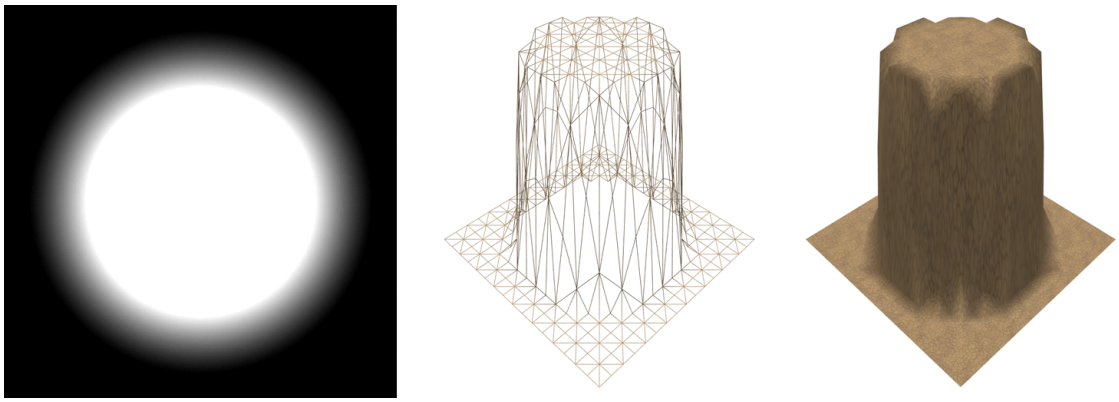


Figure 2.3: Heightmap disadvantages with steep gradients

2.1.1.3 Vector field displacement

Vector field displacement offers at least a partial solution to the limitations on the terrain features that can be encoded by heightmaps alone. It still uses a regular grid of vertices as with heightmaps but with a key difference. Rather than representing the surface as a offset in the direction of the surface normal, each vertex is offset by a 3D vector. This opens up the possibility of creating overhangs and steep cliffs which are essential if the system is to attempt recreating a full repertoire of landscapes.

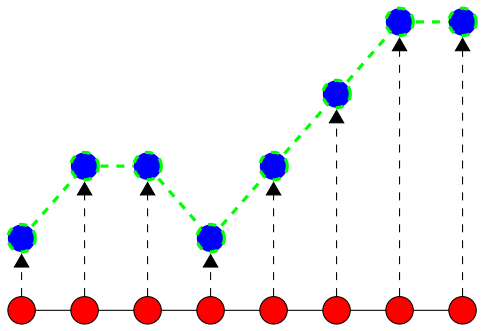


Figure 2.4: Heightmap displacement

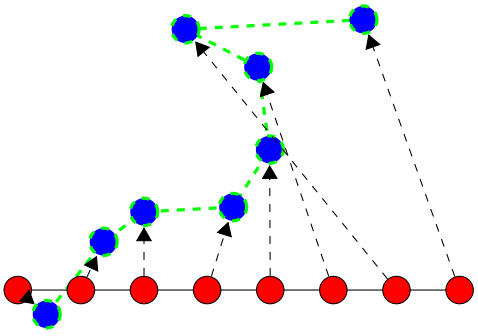


Figure 2.5: Vector field displacement

There are some disadvantages compared to heightmaps however, and some hurdles to overcome.

- It requires more computation to calculate three dimensional vector offsets than height displacements.

- It requires three times more memory to store, assuming no special encoding.
- It does not deal with the texture stretching issue, potentially making it worse as vertex distances can increase further.
- Aside from the rendering of the terrain itself, in an interactive system there will be implications for collision detection, path finding and other subsystems. While these could be worked around, it is unavoidable that the complexity of the terrain and anything that interacts with it is increased.

Overall we feel the benefits in realism gained by VFD should offset the effort of implementing it and the performance impact it will have. It represents what we feel is the best trade-off between the fast and simple heightmaps (see 2.1.1.2) and the GPU-unfriendly voxels (see 2.1.1.4)

2.1.1.4 Voxels

Voxels (volume elements) represent one of the possible future directions of terrain rendering. Voxels can be thought of as pixels with volume, so the terrain is made up from small cubes which allow for any type of geometry to be recreated. This gives even more flexibility than vector field displacement by facilitating caves and deep cliffs with potentially much better precision.

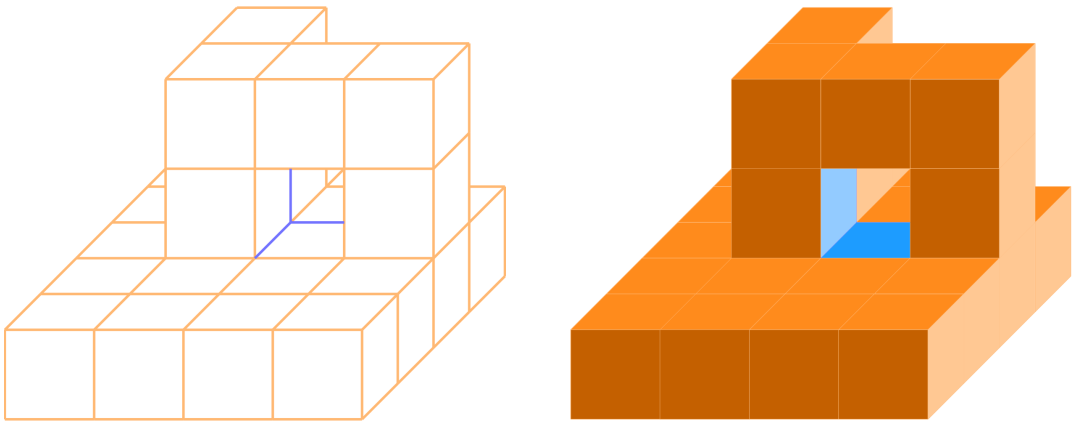


Figure 2.6: Schematic representation of a voxel terrain. The blue shaded area shows a cave. A real voxel terrain would voxel counts orders of magnitude larger to look convincing, along with texturing and so on.

The problem has been and remains that there is no hardware support for voxels at a primitive level. This greatly limits the number of voxels that can be rendered and puts far more load

on the CPU. The only current way of being able to use voxels to produce terrains at a convincing level of detail is to use a isosurface extraction method to smooth larger voxels in order to make a smooth surface. The most common way to do this is using the marching cubes algorithm[22][4].

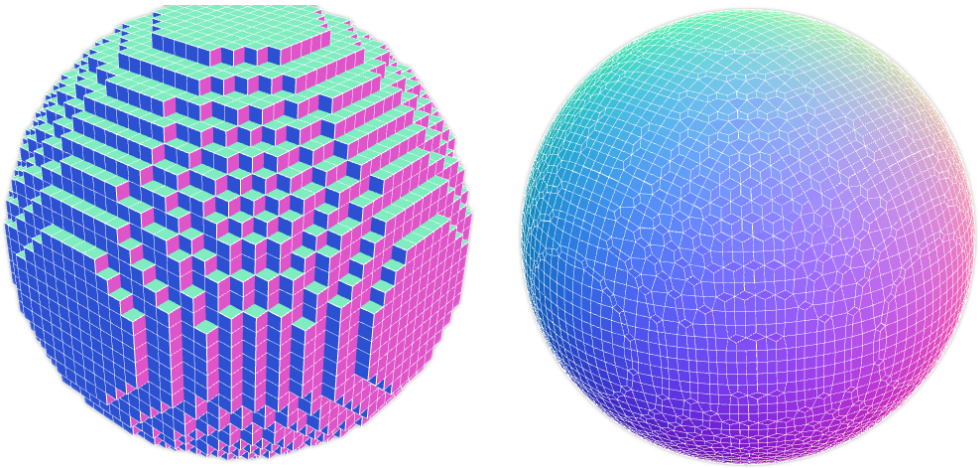


Figure 2.7: Extracting an isosurface from a sphere of voxels. Image courtesy: Mikola Lysenko

Crytek used voxels to a limited extent in their much revered CryEngine in order to accurately model cliffs[10] and caves. The results look very good but due to the limitations in GPU acceleration it is their recommendation that voxels are not used for the other parts of the terrain; although it is possible to do so at a high cost to performance.

For this project, the lack of hardware support and issues with detailed large scale rendering preclude the use of voxels. It would be possible to use them in a limited capacity interacting with the other terrain similar to what was done by Crytek.

2.1.2 Level of detail

With current and foreseeable hardware it is not possible to render very large terrains at full level of detail. The GPU is not capable of rendering the required number of polygons; nor is it capable of storing anywhere near the required amount of data in its memory or streaming it over the PCI-E bus. The only option is to reduce detail. This is normally embodied by focusing detail around the camera position or around areas with complex features or both. For example, in a terrain it requires far less detail to make flats look convincing than it does rocky mountains and so the detail could be focussed mainly around those areas.

There are two types of LOD algorithm, one provides a continuous level of detail (CLOD) and the other provides discreet levels of detail (DLOD, uncommonly). A CLOD algorithm

is always refining the level of detail of the object depending on the LOD metric, usually distance or error based. A DLOD algorithm refines the detail level in fixed steps, a patch of terrain may have its level of detail doubled every time the camera gets two times closer for example.

2.1.3 Terrain patches/chunks

Most terrain level of detail algorithms rely on the concept of patches or chunks in some form so the concept is introduced first. Rather than the algorithm attempting to modify a single vertex/index buffer representing the entire terrain, it is generally easier to subdivide the terrain or combine smaller parts in order to refine or coarsen the level of detail. Figure 2.8 shows a typical refinement of a terrain patch into four smaller patches to give a higher level of detail. The patch sizes used are normally around 64x64 to 128x128. Historically this number was much smaller, such as 8x8 or 16x16 since it can save additional geometry from being rendered. On modern GPUs however, the cost of having more draw calls would far outweigh the cost of drawing slightly less geometry.

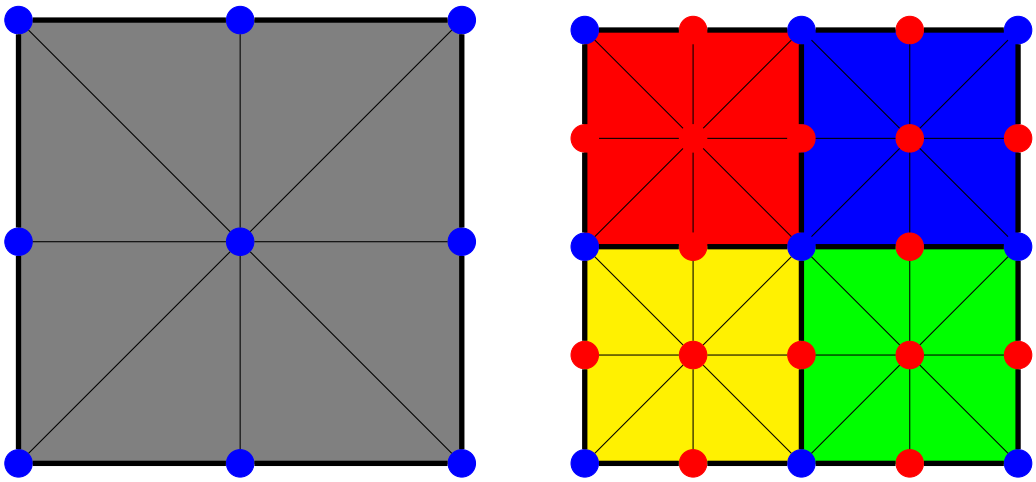


Figure 2.8: Level of detail refinement from one 2x2 patch to four 2x2 patches. Blue points are the same across the two patches, but the higher level of detail contains additional points shown in red.

2.1.4 Errors

By lowering levels of detail, errors in the rendering are introduced since the object being rendered no longer exactly represents the source data. Often an error metric can be calculated which quantifies how large this error may be. This value can be used to determine which level of detail a portion of the terrain should be rendered at, i.e., limit the maximum error rather than just using distance. This can be configured such that the error is reduced beyond

what will have a visible impact on the geometry at the camera's position. This is the ideal scenario for LOD algorithms. Figure 2.9 shows the errors that occur between the two levels of detail in figure 2.8. The common way of calculating a geometric error metric for the patch would be to take the maximum of δ_1 to δ_n.

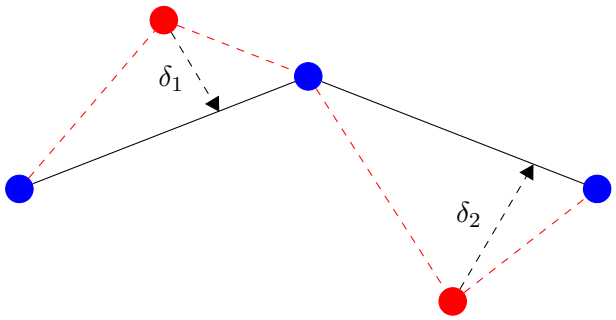


Figure 2.9: Errors from coarsening of detail (from red path to black path)

2.1.5 ROAM

Real-Time Optimally Adapting Meshes or ROAM is an early technique introduced by Duchaineau in 1997[13]. For a long period of time it was one of the most popular methods for rendering large terrains due to its good performance and continuous nature. Unlike most other algorithms where patches are used, the standard ROAM implementation refines an entire terrain. This means it is only suitable for medium sized terrains without some further modification. It works at the triangle level by using triangle binary trees or bintrees (the refinement process for which is shown in 2.10). The essence of the process is that triangles are split and merged depending on an error metric (see 2.1.4). This is achieved using a top-down algorithm that recurses the tree reaching progressively smaller triangles, splitting and merging as necessary depending on the error metric until the entire tree has been traversed.

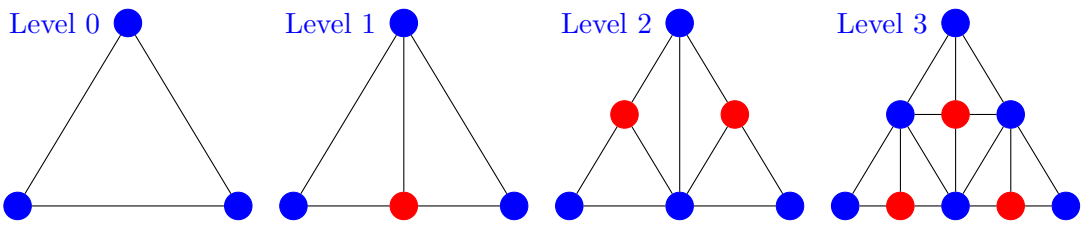


Figure 2.10: Triangle binary tree refinement process

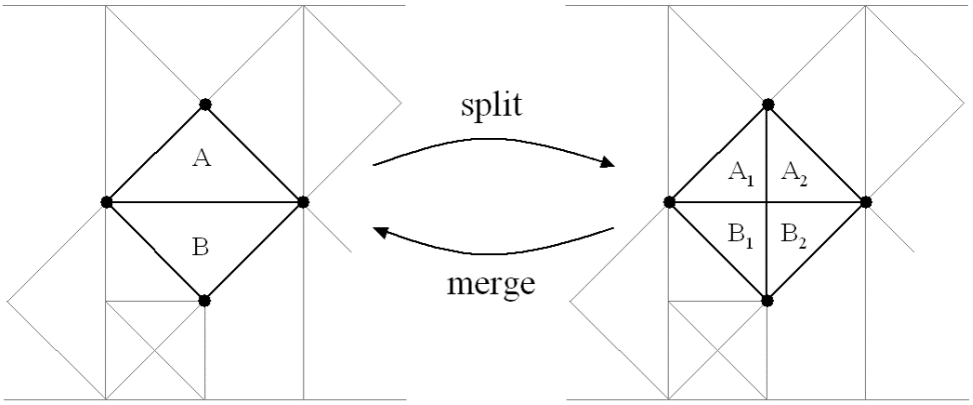


Figure 2.11: Splitting and merging triangles in a triangle binary tree

Two priority queues are used to implement the splitting and merging steps, one queue for splitting and the other for merging. The priority is determined by the error metric so that areas with the most severe errors are served first.

The ROAM algorithm produces triangle strips which can be rendered fast on modern hardware but requires too many draw calls and relies too heavily on the CPU to be of any consideration for modern hardware. ROAM 2.0 is under development and portions are available but as yet it is incomplete and so will not be considered.

2.1.5.1 Geomipmaps

Geometrical mipmaps or geomipmaps was introduced by de Boer in 2000[12]. It is so called because the method handles geometry levels of detail in a way analogous to what is seen in texture mipmaps.

The terrain starts as a single low detail patch and is split recursively into four patches as seen in figure 2.8. This means each refinement of a patch quadruples the amount of detail for that area. The patches are stored in a quadtree that allows for easy traversal and refinement. In this paper, the entire quadtree is populated at start-up other than the actual terrain data itself. This can make loading very large terrains consume a large amount of memory. The splitting of terrain nodes occurs according to an error metric. The initial proposal for this metric is based on the geometric error (see 2.1.4) projected into screen-space to work out the pixel error at a certain distance. This is quite expensive however and the paper goes on to detail some simplifications that can be employed.

Using a quadtree also makes it straightforward to detect when neighbouring patches have

a differing level of detail. This is important or cracks will appear leaving holes in the terrain (a common issue with many similar algorithms). The solution proposed in the paper suggests using different indexing patterns when neighbouring patches differ in level of detail to match up the edges. This works well, but makes it desirable to ensure having at most one level of detail difference between patches to save storing many indexing patterns.

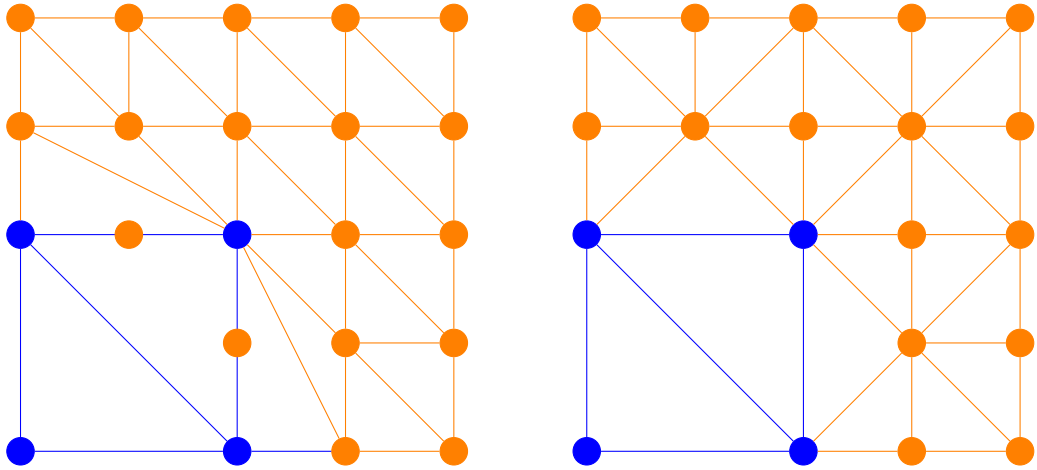


Figure 2.12: Preventing cracks in geomipmaps. An indexing pattern is chosen so that the vertices in the higher detailed patches (coloured orange) are correctly matched up to vertices in the neighbouring less detailed patch (coloured blue). Two alternative patterns are shown.

This is a simple and proven algorithm that could produce suitable results for the project, but the discreet nature of the level of detail changes mean there will be undesirable popping between the levels of detail. Implementing the algorithm entirely as presented would also present problems at the scale desired by this project due to populating the entire quadtree at load time.

2.1.5.2 Chunked Level of Detail

Ulrich introduced the chunked level of detail algorithm in 2002 at SIGGRAPH [45]. It builds on the work presented in geomipmaps (2.1.5.1) by making a number of changes and additions.

The paper first presents a variation on the use of quadtrees for the storage and culling of nodes. In geomipmaps, the entire quadtree is populated on startup with all information other than the terrain data (for example bounding volumes). This presents a storage problem for very large terrains. The method used in chunked LOD instead dynamically creates and destroys quadtree nodes as the camera position changes. This makes supporting out-of-core rendering (that is, using data not stored in main or GPU memory by some form of paging) simple.

The paper goes on to propose an alternative way of calculating the maximum screen-space error for a chunk, ρ. This is simpler than the method seen in geomipmaps as it avoids the need for a screen-space projection by instead looking at the viewport width (the horizontal resolution of the render) and the horizontal FOV or field of view (the angle of the observable view). It defines the error value ρ as:

$$\rho = \frac{\delta}{K} \tag{2.1}$$

where

$$K = \frac{\text{Viewport width}}{2 \cdot tan\frac{\text{Horizontal FOV}}{2}} \tag{2.2}$$

$$\delta = \text{Geometric error of a chunk, calcuated as max } \delta_1 \text{ to } \delta_n \text{ as in 2.1.4} \tag{2.3}$$

Next the paper presents a cheaper and simpler method of dealing with the cracks between terrain patches of differing level of detail. One of the drawbacks of the index pattern switching method presented in geomipmaps is that patches need to be able to determine their neighbouring patch. This normally requires additional traversals of the quadtree or storing of extra information. At the time these papers were published, such a cost was significant but is less material now. The solution involved adding additional geometry to the terrain patches to form "skirts". These skirts make the cracks far less apparent to the viewer, but add additional geometry to render and do not produce pixel-perfect results.

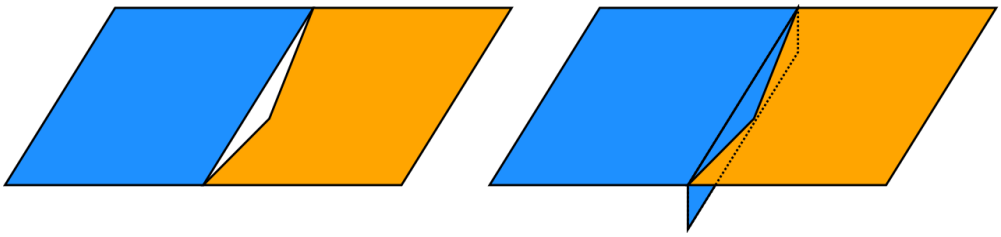


Figure 2.13: Filling cracks between differing LOD patches with skirts

The final main point introduced by the paper is morphing. This is a way of making the change between LOD patches less obvious to the observer. This is done by reducing the detail on a patch smoothly before it reaches the point where it switches out to a lower level of detail. This is done by moving the extra vertices, that is, the red vertices shown in figure 2.8 to be in line with the average of their neighbouring vertices so that the higher level patch effectively becomes the same resolution as the lower one. The paper suggests doing this by having an additional morph parameter in the per-vertex information of the chunk. However it could equally be done by sampling from the lower level of detail patch or by averaging adjacent vertices (if an appropriate format is chosen that supplies adjacency information).

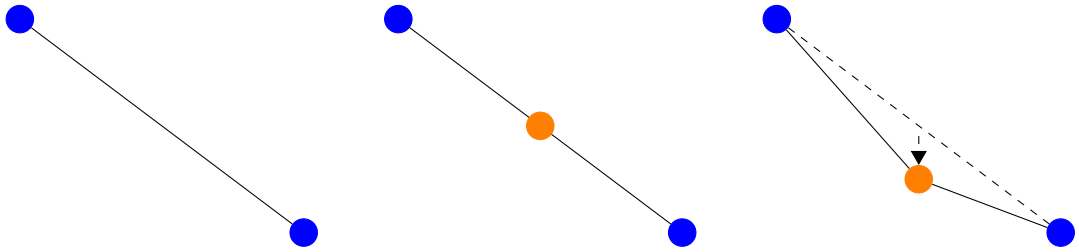


Figure 2.14: Morphing new vertices into place after switching to a higher LOD

The paper gives a simple way to calculate the morph amount based on the error metric:

$$t_{morph} = clamp(\frac{2\rho}{\tau} - 1, 0, 1) \text{ where } \tau \text{ is the threshold of } \rho \text{ before subdividing a node} \quad (2.4)$$

As with geomipmaps, chunked LOD would make a fine solution for the project but is rather unambitious, does not make full use of modern hardware features and still suffers some of the drawbacks of a DLOD algorithm.

2.1.5.3 Geoclipmaps

Geometry clipmaps were introduced by Asirvatham and Hoppe in 2004[23] and a more detailed analysis and explanation was given by Brettell in 2005[5]. The foremost goal of the technique is to shift a larger portion of the work in rendering terrains to the GPU than is seen in previous implementations like geomipmaps.

Unlike geomipmaps and chunked LOD, the level of detail selection in geoclipmaps is based solely on distance. The system works by splitting the terrain into nested rectangular grids, forming rings of each level of detail (other than a square in the centre). The height data to populate these grids is obtained using downsampled textures or buffers representing the terrain at power of two intervals, stored on the GPU. This data is updated as the camera moves for levels where it is not possible to store the entire terrain information at once. Cracks are dealt with by morphing vertices on borders much like in chunked LOD (2.1.5.2).

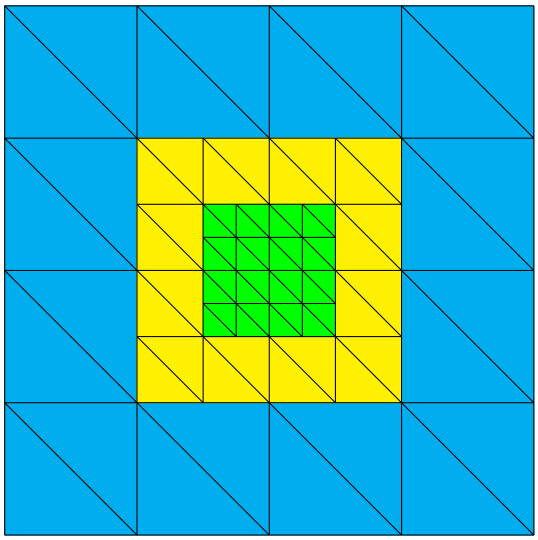


Figure 2.15: Morphing new vertices into place after switching to a higher LOD

One less appealing aspect of the system is that the nested grid structure requires an unnatural configuration of vertex and index buffers as seen in figure 2.17. Three different shapes are used to build up the terrain and the sample implementations make the whole solution seem rather inelegant and messy.
As well as this drawback there are a few other faults in the default implementation, including ignoring height of the camera above the terrain when making LOD selection and sub-optimal distribution of detail due to the underlying geometry being based on rectangles. These faults could be fixed or improved so that geoclipmaps is able to provide an acceptable solution for the project but it lacks overall appeal due to its drawbacks.

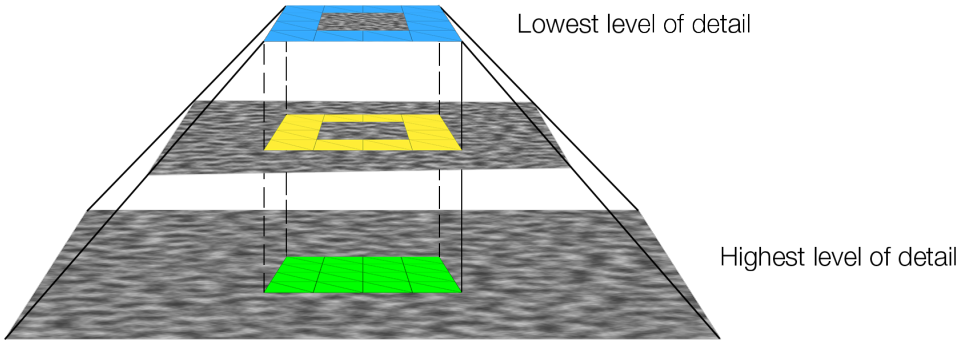


Figure 2.16: Sampling pyramid for the three levels shown in 2.15

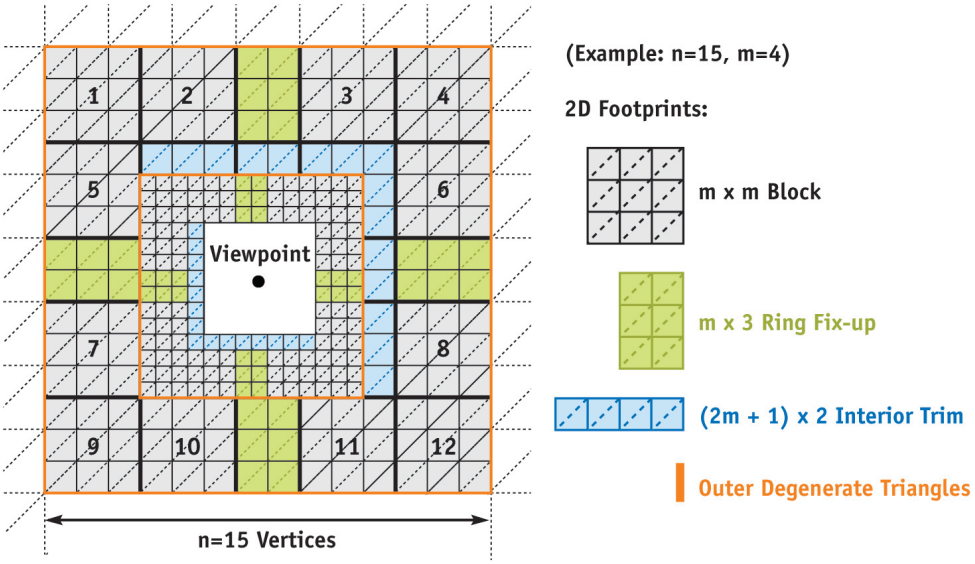


Figure 2.17: Geoclipmap ring partitioning system. Three different shapes are used to build the rings of the nested grid structure.

2.1.5.4 Continuous Distance-Dependent Level of Detail

Continuous Distance-Dependent Level of Detail or CDLOD was proposed by Strugar in 2009[41]. It builds on previous work, particularly that of Ulrich and de Boer. CDLOD uses the same quadtree system but with a key modification allows for a continuous level of detail even though the underlying implementation remains discreet. This removes entirely the need for stitching as in geomipmaps or skirts as in chunked LOD by smoothly transforming a chunk based on the distance of vertices (rather than patches as in previous algorithms) from the camera. This offers the best features from both chunked LOD and ROAM - it is simple, provides a continuous level of detail, maps well to modern hardware (not quite as well as geoclipmaps in some regards), easily supports adding detail and deformation with a few modifications and supports paging out-of-core chunks which is ideal for streaming in data from a separate CPU thread.

The transition between patch LODs works by translating vertices in the terrain vertex shader based on distance. The larger a patch the larger its morph region as shown in 2.18. The vertex blending can be seen in 2.19.

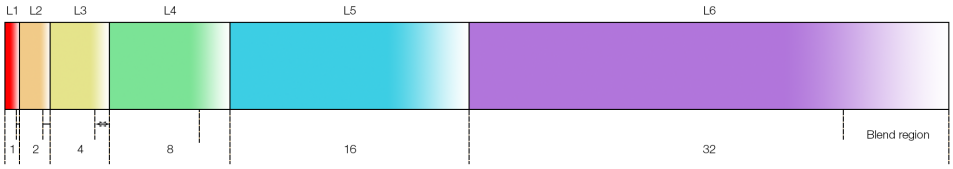


Figure 2.18: Mesh vertex transformations for smooth transitions between terrain patches of different levels of detail

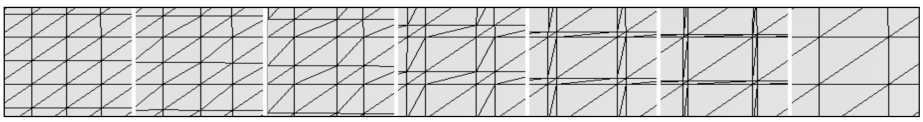


Figure 2.19: CDLOD morphing vertices to a lower level of detail. Image courtesy: Filip Strugar

CDLOD seems an excellent candidate for adaptation to the project due to the benefits previously discussed. There is scope to explore the suggestion in the paper of using hardware tessellation to modernise the algorithm.

2.1.6 Procedural generation

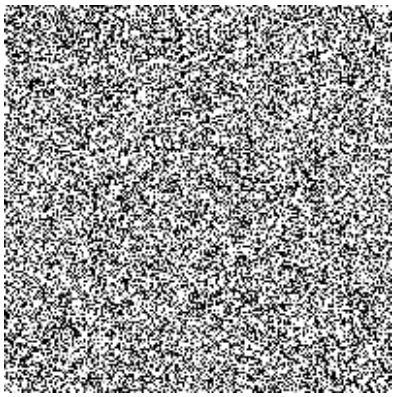
2.1.6.1 Noise generation

Noise is key to many tasks in procedural generation. It involves creating patterned n-dimensional textures using different algorithms that lend themselves to certain applications. For example there are noise algorithms that are particularly suited to generating convincing clouds, cells, foliage and so on.

Noise functions fall under two main categories, coherent noise and non-coherent noise. Coherent noise is smooth and pseudo-random, it therefore exhibits two main features:

1. The result is always the same for the same input parameters, typically an n-dimensional position in the texture being generated along with time if the texture is animated.
2. The output value changes only by a small amount for close input parameters, but randomly for far apart input parameters. This is because of smoothness, if a small change in input parameter can result in a large change in output value then it can not be smooth.

Non-coherent noise need not have either of these two properties. Figure 2.20a shows non-coherent noise generated by software for photography and coherent noise generated with Perlin noise (see 2.20b). It is notable that the non-coherent noise could be made in coherent noise by applying a blur function to smooth the transitions between light and dark areas. This would be a valid way of generating coherent noise but higher performance methods exist.



(a) Non-coherent noise

(b) Coherent noise

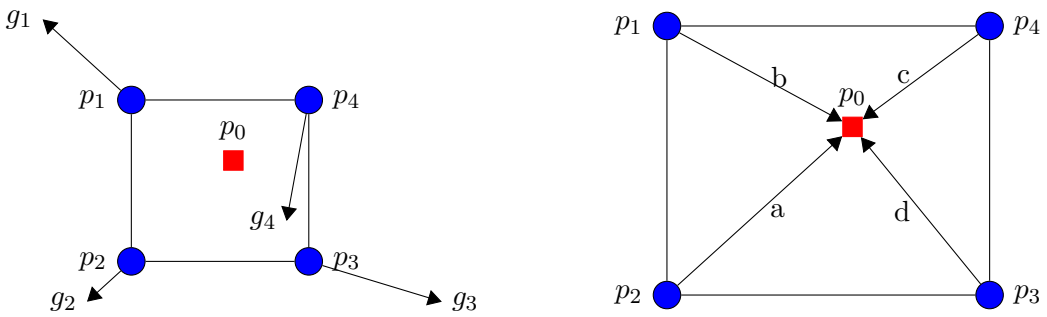
Figure 2.20: Two categories of noise

In rendering terrains and most other procedural generation processes, it is coherent noise that is desired. Rarely in reality are the sharp changes in non-coherent noise wanted and the

pseudo-random nature of coherent noise also makes content generation repeatable given the same set of parameters. If an artist creates a landscape using a particular seed for a noise function, they want the result to look the same way every time.

2.1.6.2 Perlin noise

Perlin noise is a coherent noise generation algorithm ubiquitous across computer graphics, particularly in procedural generation. It was created in 1989 by Ken Perlin[35] and in conjunction with Fractional Brownian motion (see 2.1.6.4) can be used to create convincing textures for entities such as clouds, fire, coronas and others. Sometimes "Perlin noise" is used synonymously for Perlin noise with fBm applied, but this section talks about only the basic interpretation of Perlin noise.



(a) Pseudo-random gradient vectors

(b) Delta vectors

Figure 2.21: Perlin noise calculation steps

Perlin noise can be applied in any number of dimensions, to get an idea of how it works, consider the process when applied in two dimensions. First the input parameters are taken and the four surrounding integer coordinates are calculated. Then, these four coordinates are looked up in a previously generated table of pseudo-random, evenly distributed numbers (this gives coherence in the result versus using a true random number generator). The result is interpreted as four gradients. Next, the vectors from the four surrounding points to the original coordinate are calculated and the dot product is taken with the random gradients found previously. This gives a contribution for each surrounding vertex to the final noise result. This final result is calculated by a weighted average based on the distance of a point from the original coordinate. The distance is first modified however using the S-curve function which exaggerates values close to zero or one, which gives a better result.

An interesting consequence of this method is that the noise function will return zero given integer input parameters. This means that contrary to appearances, there is a regular pattern in the noise at every whole number although it does not appear this way by eye.

Perlin noise is has $O(2^n)$ complexity (where n is the number of dimensions) and is therefore perfectly suitable for noise generation in this project with a lower number of dimensions. However, a better solution exists in Simplex noise (see 2.1.6.3), an improvement on Perlin noise proposed by Ken Perlin in 2001.

2.1.6.3 Simplex noise

Simplex noise is a newer algorithm also created by Ken Perlin[34] that improves upon the original Perlin noise implementation. It exhibits a largely similar visual appearance but has a few key benefits[17] including:

- A lower computational complexity of $O(n^2)$ rather than $O(2^n)$ and a lower constant cost per calculation
- Easy to implement on hardware
- No directional artifacts and well defined and cheap to compute gradient at all locations

These benefits make simplex noise an obvious choice for inclusion in the project over Perlin noise. It would also be reasonable to just support both Perlin noise and simplex noise and give the end-user a choice. Simplex noise should normally be preferred but there may be situations where the fairly nuanced differences in the original implementation would be preferred. For example it exhibits less contrast and has an arguably more natural look.

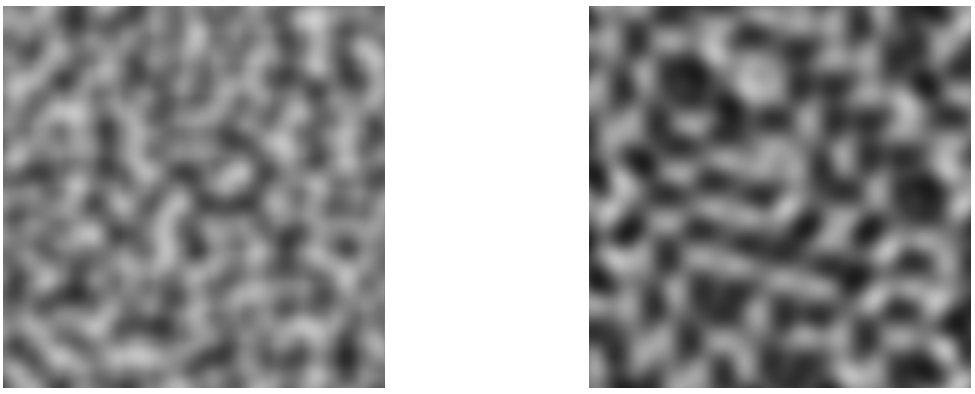


Figure 2.22: Perlin noise (left) versus simplex noise (right)

2.1.6.4 Fractional Brownian motion

Fractal Brownian motion is a common method used to turn the noise generated by coherent noise functions into fractals. It achieves this by combining layers of noise sampled from a coherent noise function at differing frequencies. Typical parameters that can be used in the fBm process are:

- **Frequency** - Determines how compressed the noise is into the area. As frequency increases, smaller features will be added to the texture.
- **Octaves** - Determines the number frequencies used in compositing the noise layers which produce the final fractal texture. The use of the word "octaves" derives from the use of a standard lacunarity value of 2.0 which results in the frequency doubling each octave mimicking the doubling in frequency in musical octaves.
- **Lacunarity** - Determines the gap between successive frequencies between octaves. The standard lacunarity is 2.0 which doubles the frequency every octave.
- **Gain** - Determines how much the amplitude is modified each octave. The amplitude determines how much the current octave values contribute to the final result. It is normally desirable that small details have a lower amplitude. For terrain, consider that a mountain would have high values and the rocks on it would have low values.
- **H (Hurst exponent)** - Determines the fractal dimension, which is a measure of the complexity of the fractal pattern changes with the scale at which it is measured.
- **Offset** - Offset determines multifractality[28]. Multifractality is a term used to define fractals whose dimension is not homogeneous with location.

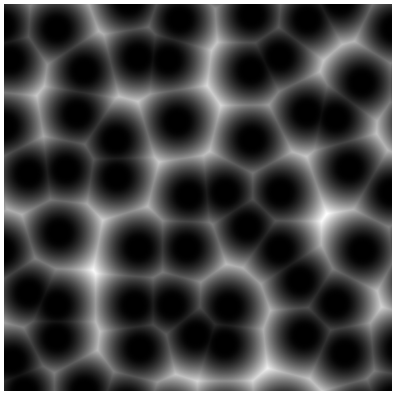
Additional controls could easily be added to the fBm algorithm to give more control over the output. For example gamma, bias, gain, absolute offsets and variations on how octave layers are combined. Figure 2.1 shows a basic fBm algorithm using just some of the possible parameters previously described.

```
double fBm(float x, float y)
{
  double total = 0.0f, frequency = 1.0f, amplitude = 1.0f;

  for (int i = 0; i < octaves; ++i)
  {
    total = noise->getNoise(x * frequency, y * frequency) * amplitude + total);
    frequency *= lacunarity;
    amplitude *= gain;
  }

  return (total + 1.0) * 0.5;
}
```

Listing 2.1: Simple fBm implementation

2.1.6.5 Other coherent noise functions and fractal construction methods

A number of other coherent noise types exist that have common applications in computer graphics. Some examples are hermite noise which can be used to produce similar results to Perlin noise and Worley noise, also known as Voronoi noise produces a cellular pattern. The more of these that can be integrated into the terrain system, the more expressive it will be.

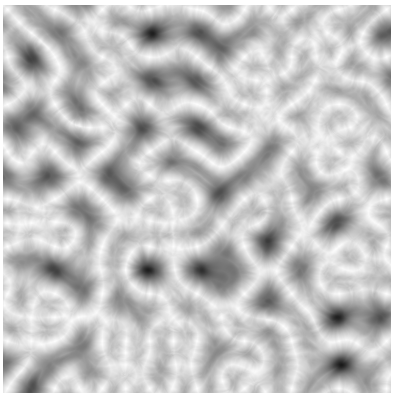


(a) Worley noise

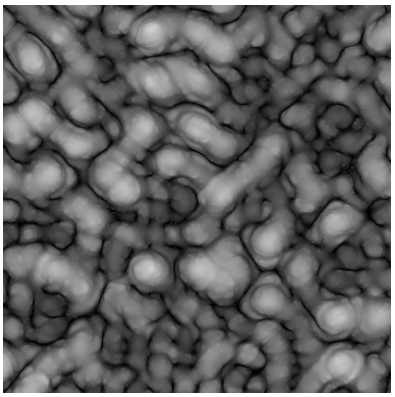
(b) Hermite noise

There are also more esoteric forms of fBm that have been developed for more specific applications. Two excellent examples of this for usage in terrains are the so called Swiss turbulence

and Jordan turbulence by de Carpentier[7], named after the regions of the world which inspired their look. These functions produce more convincing noise types that go some way to emulating the effects of hydraulic erosion on terrain.



(a) Swiss turbulence



(b) Jordan turbulence

The basis of these functions is still the standard fBm procedure, taking parameters for the number of octaves, lacunarity and so on. They differ in also taking into consideration the derivative of the underlying noise type with some additional parameters to produce more complex and asymmetric results. The construction of these types of fBm variations is not an exact science and largely comes down to trial and error of finding techniques and values that produce pleasing results.

2.1.7 Rendering spherical terrains

Rendering spherical terrains using an underlying spherical structure often produces excessively complex and difficult to work with systems. For example, creating LOD algorithms 2.1.2 that work with truly spherical structures (e.g. the spherical ROAM algorithm proposed in "A Real-Time Procedural Universe, Part Two: Rendering Planetary Bodies"[31]) is challenging when compared to the methods used for planar terrains. Instead, a simple way of rendering a planet is to apply a cube to sphere mapping. This can be used both for mapping the generated data to the planet[9] and for creating the planet mesh itself, for example by using a cube made from six planar terrain quadtrees. This has some considerable benefits. Firstly the level of detail algorithms previously discussed can be used with only minimal modifications such as how to handle the detection of the level of detail of an adjacent node from another quadtree. The second advantage is it potentially simplifies the creation of the content itself. Particularly if the system used is able to use the 2D Cartesian coordinates of the planes before the spherical mapping is applied.

The most simple way to map a cube to a sphere would be to normalise the values of a

unit cube then scale it. Doing this does not produce an even distribution of vertices. A better solution is to use the following mapping which does[30], at a slightly higher computational cost:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} x\sqrt{1 - \frac{y^2}{2} - \frac{z^2}{2} + \frac{y^2 z^2}{3}} \\ y\sqrt{1 - \frac{z^2}{2} - \frac{x^2}{2} + \frac{z^2 x^2}{3}} \\ z\sqrt{1 - \frac{x^2}{2} - \frac{y^2}{2} + \frac{x^2 y^2}{3}} \end{bmatrix} \tag{2.5}$$

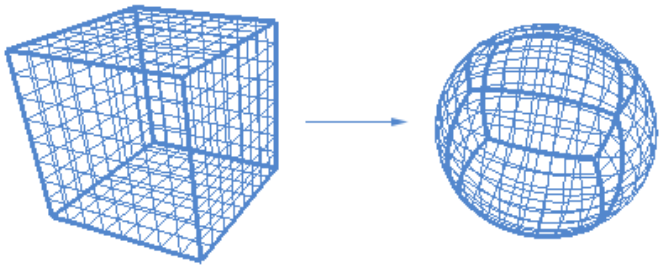


Figure 2.25: Cube to sphere mapping. Image adapted from [30].

2.2 Image analysis

2.2.1 Histogram matching

Histogram matching is a method of comparing and adjusting two or more images using histograms. First the histogram of the reference and comparison images are calculated. From here the matching function, m(i), is computed. This is done by computing the CDF function, F_r and F_c of the reference and comparison histograms. Then, for each intensity or colour level, i_2 is found such that $F_r(i_1) = F_c(i_2)$. This forms the result for m, so $m(i_1) = i_2$.

2.3 Atmospheric scattering

Atmospheric scattering is the modelling of how light scatters in a planet's atmosphere due to its elemental composition - gasses, pollutants and dust for example. Despite the light from the Sun having a fairly even spectral distribution, the Earth's atmosphere generally appears blue due to particles scattering a greater portion of the blue wavelength of light compared to others. At sunset the sky appears red as a result of the angle the Sun makes with the atmosphere causing light to travel a greater distance through it. Travelling this greater distance causes more scattering of the higher frequency blue light, leaving predominately red wavelengths reaching the eye.

In the context of computer graphics, most modern work is derived from the paper by Nishita et al in 1996[29]. It describes the colour of the sky as being a function of the spectral distribution of the sun light, the solar zenith angle, atmospheric conditions, light reflected from the planet's surface, scattering and absorption by atmospheric particles and clouds and view direction. The papers describes accurate mathematical models for the multiple and single scattering of photons in the atmosphere. Single scattering occurs when scattered photons reach the observer after a single step. Multiple scattering occurs when photons are scattered a number of times before reaching the observer. At a high level, these models work by integrating the incident light over the hemisphere which comprises the atmosphere. Even many years after publishing, it is still not possible to perform all of the integrals in the paper in real-time. Most subsequent work is based on finding compromises and optimisations for hardware that make real-time rendering possible. These models typically make a number of simplification steps. Single scattering is a commonly used approximation[36][32] which works well for daytime rendering but gives dark biased results at night. This can be partially compensated for by adding constant values. Simplifications also typically make assumptions about factors like the number of dust particles, fixing them for all locations of the atmosphere.

2.3.1 Precomputed Atmospheric Scattering

The current state of the art in physically based models for real-time usage is the 2008 paper Precomputed Atmospheric Scattering by Bruneton et al [6]. The model accounts for both single and multiple scattering but makes simplifications in considering the particle composition and thickness of the atmosphere to be constant. The physical model used in the paper is fairly ubiquitous across all papers concerning real-time rendering. The model assumes the atmosphere to be a mixture of air molecules and aerosol particles in a layer of increasing density from the top of the atmosphere down to the planetary surface. The air molecules are small and are responsible for the blue appearance of the sky discussed previously, the aerosol

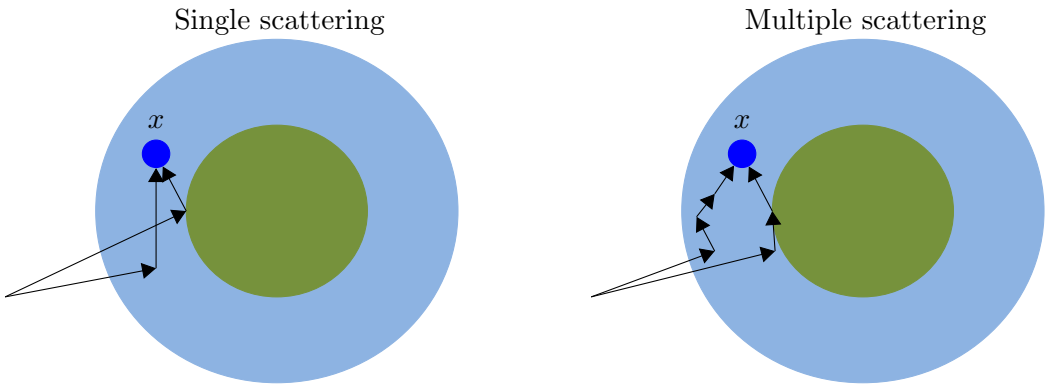


Figure 2.26: Single versus multiple scattering of photons in the atmosphere, reaching an observer at x.

molecules are larger and distribute all light evenly (consider the greyish appearance in highly polluted or foggy areas). The scattering as a result of these two components are modelled by Rayleigh and Mie scattering for molecules and aerosols respectively. The proportion of light scattered θ degrees from its incident direction is given by the product of the scattering coefficient β^s and a phase function P.

2.3.1.1 Rayleigh scattering

Rayleigh scattering, named after physicist Lord Rayleigh, models the scattering of photons due to particles considerably smaller than the wavelength of the light (less than 10%). The scattering coefficient for Rayleigh scattering, $\beta_R^s(h, \lambda)$, is a function of the wavelength of the incoming photon λ, the refractive index of the air n, the air density at surface level N, the height above the surface of the interaction h and the height scaling coefficient H_R. It takes as parameters λ and h. We also give the phase function for Rayleigh scattering (for unpolarised light), $P_R(\mu)$ where μ is the cosine of θ, the angle the scattered ray direction makes with the incident photon ray. This function describes the angular distribution of the intensity of the scattered light. In the case of Rayleigh scattering the distribution is nearly uniform with a small amount of falloff at the perpendicular angles. There is no absorption in air particles so the extinction coefficient for Rayleigh scattering is the same as the scattering coefficient, $\beta_R^e = \beta_R^s$.

$$\beta_R^s(h, \lambda) = 8\pi^3 \frac{(n^2 - 1)^2}{3N\lambda^4} e^{-\frac{h}{H_R}} \tag{2.6}$$

$$P_R(\mu) = \frac{3}{16\pi}(1 + \mu^2) \tag{2.7}$$

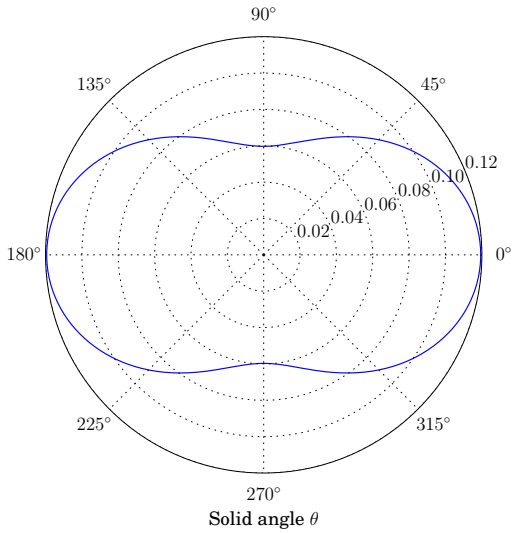


Figure 2.27: Rayleigh phase function.

2.3.1.2 Mie scattering

Mie scattering, named after Gustav Mie, models the scattering of photons due to aerosols with a particle size close to the wavelength of the light ($\pm 90\%$ is considered close). The scattering formula, $\beta_M^s(h)$, is the same as in Rayleigh scattering with the wavelength parameter removed since Mie scattering remains constant for all wavelengths (note the original paper gives the Mie Scattering equation including a λ term, but we omit it since the RGB parametrisation used to find β_M^s has $\lambda_r = \lambda_g = \lambda_b$). We also use a separate height scaling factor, H_M, where $H_M < H_R$ since the heavier particles reside closer to the surface. The phase function for Mie Scattering differs considerably because the scattering is predominately forward, that is, light is scattered mostly in direction of its incidence. The parameter $g \in [0, 1]$ determines the degree of this forwardness, as can be seen in figure 2.32.

In Rayleigh scattering, the extinction coefficient was equal to the scattering coefficient since no absorption occurs. It does occur in the larger aerosol particles however, so we have an additional term β_M^a for this absorption. So the extinction coefficient for Mie Scattering is $\beta_M^e = \beta_M^s + \beta_M^a$.

$$\beta_M^s(h) = \beta_M^s(0, \lambda) e^{-\frac{h}{H_M}} \tag{2.8}$$

$$P_M^s(\mu, g) = \frac{3}{8\pi} \frac{(1 - g^2)(1 + \mu^2)}{(2 + g^2)(1 + g^2 - 2g\mu)} \tag{2.9}$$

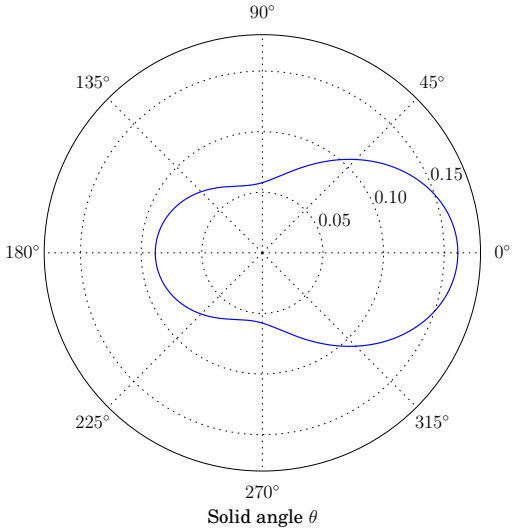


Figure 2.28: $\theta = 0.1$

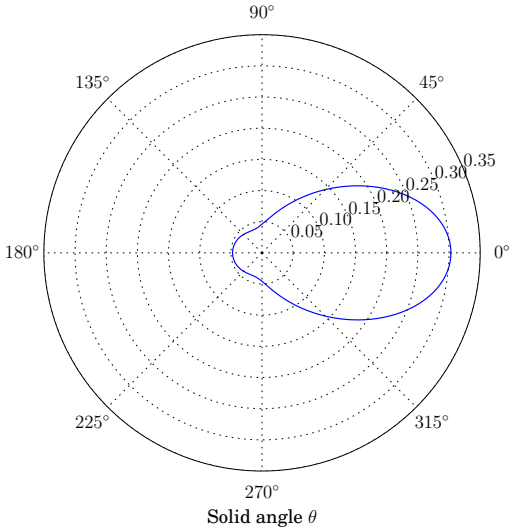


Figure 2.29: $\theta = 0.3$

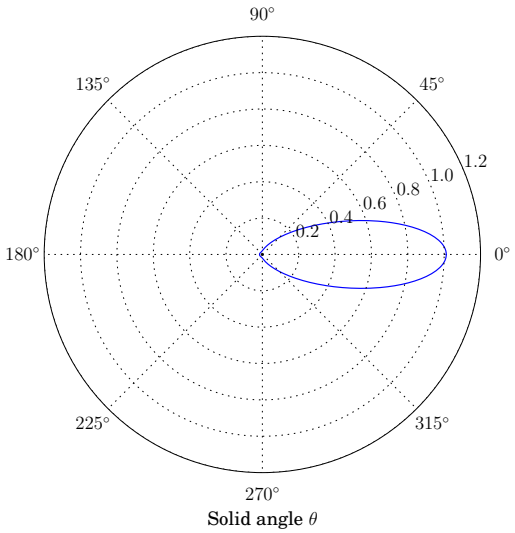


Figure 2.30: $\theta = 0.6$

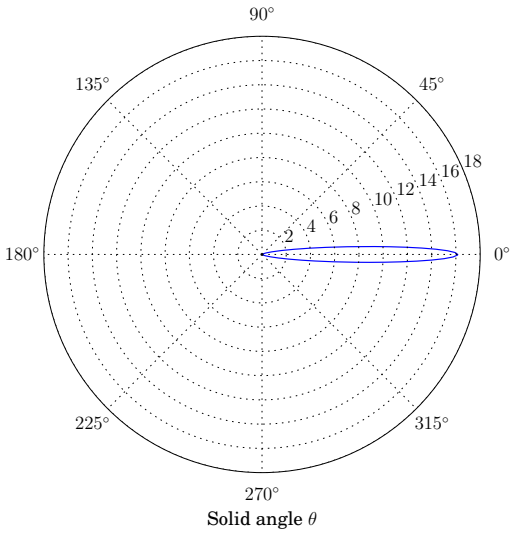


Figure 2.31: $\theta = 0.8$

Figure 2.32: Mie phase function for a variety of θ values showing increasingly forward scattering as $\theta \to 1$. Note that when $g = 0$, the result is the same as the Rayleigh phase function.

2.3.1.3 Rendering equation

We describe the rendering equation for precomputed atmospheric scattering as part of our implementation discussion to avoid unnecessary separation/duplication of material. Please see 6.2.

2.4 Related work

Creating rule-based procedural terrains remains a largely unexplored area and to the best of our knowledge, our effort to create a GPU-based system for real-time generation is the first of its kind. There are two notable commercial products which use a rule-based system, World Machine and Star Wars Galaxies. These are both closed source and have limited details available on their internal workings but we use them as inspiration in creating the rules for our system and uncover how some of their components may have been created.

2.4.1 World Machine

World Machine[37] is a mature tool, developed by Stephen Schmitt since around 2005 specifically for generating procedural terrains and then exporting them for use in other software. For example they may want to use the output directly in a game or a simulation or they may want to alter the results further in image editor or 3D software. World Machine generates terrain on the CPU (using multi-threading in the Pro version). It has a similar system to what we will create whereby terrains are essentially defined by making masking off certain areas then making alterations to them. To the user it presents a node based system, as seen in figure 2.33 with a left-to-right flow. The creation process works by the following process:

1. The user places down a generation device (node), such as Perlin noise.
2. The output from that device can be fed into another device's input that modifies/filters the result. There are many different such filters offered such as erosion, terracing, blurring, splitting and inversion.
3. When the user has built up the node system and is happy with the result, they connect the output of the last device in the sequence to an output node. This allows saving the resulting heightmaps, splat maps and so on to image files for use in other software.

2.4.2 Star Wars Galaxies

Star Wars Galaxies was a massively multiplayer online role-playering game (MMO/MMORPG) developed by Sony Online Entertainment and was perhaps the first piece of software to implement a significantly rule-based procedural generation system. The game was developed from around 2000 up until its release in 2003 and then operated until 2011 when its subscriber base was no longer economically viable. As with World Machine, it is a closed source commercial product and details on the terrain system are fairly scarce. However, from archived developer interviews[19][47] we can infer some details of the system. One article lists the following features comprising the system: terrain textures and rules, alpha blend patterns for the textures,

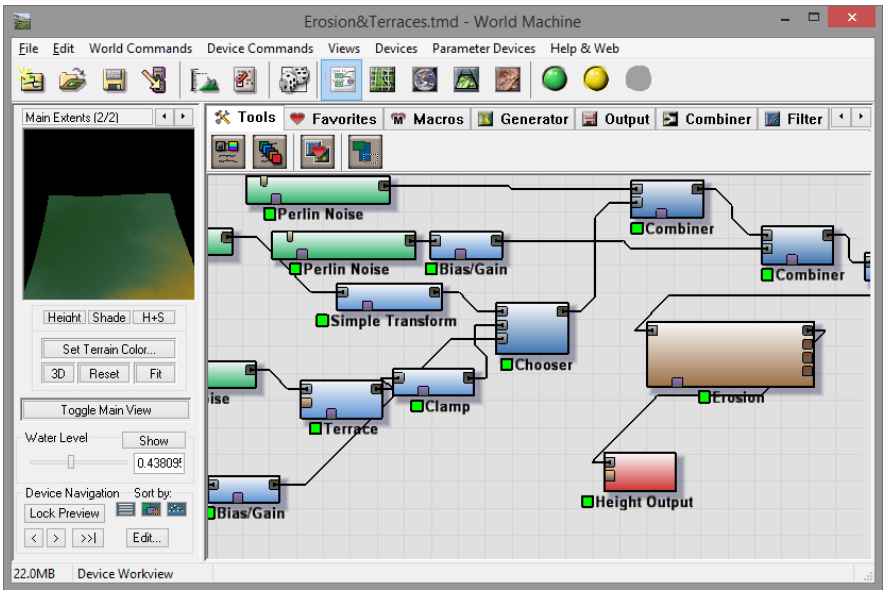


Figure 2.33: The World Machine user interface.

plants models and textures, sprites for the radial grass, sun colours for all times of day, fog colours for all times of day, level of detail for all the objects and water textures and placement.

As with World Machine, the rules offered broadly concern the masking and modification of regions of terrain. The key differences we notice from World Machine, is that the terrain system was used in real-time for generation and offered rules for the features that populate the terrain as well as the heights and texturing. Figure 2.34 shows a screenshot of the terrain generated. While it looks very simple now, it was highly regarded for its realism at the time of its release.

Figure 2.34: Star Wars Galaxies

3 Tools and implementation overview

In the background chapter (2), we gave information mainly related to the rendering of terrains. For this chapter and the remaining implementation chapters we assume the reader has a good working knowledge of the basics of real time rendering such as texturing techniques, shading techniques and linear algebra. This is an advanced graphics project and including all necessary background information is not within our scope. We also avoid giving unnecessary implementation details and code snippets instead focussing on the key ideas and components of our implementation.

This section details the tools used and takes an overall look at the implementation structure before we move on to look at the details starting in 4.1.

3.1 Toolset

3.1.1 C++

C++ is the de-facto choice for most graphics programming and this project is no exception. Unlike other popular object oriented languages like C# and Java, C++ compiles to native assembly code and allows lower level access to the computer's hardware allowing, for example, direct memory access, inline assembly and the use of SSE instructions. This native compilation and access to low level features means C++ when used appropriately will almost always outperform implementations in managed languages. Another key advantage is the ease of use of other graphics and high performance libraries. Such libraries also tend to be written C++ and harnessing them from managed languages, while possible, often defeats part of the purpose of such languages in the first place.

3.1.2 DirectX 11.2

DirectX is a set of APIs developed by Microsoft for use primarily in graphics and video applications. The project uses the latest version, DirectX 11.2, found in Windows 8.1 and Windows Server 2012 R2. We will make use of three of the APIs contained:

- Direct3D - an API for the high performance rendering on 3D graphics.
- DirectCompute - an API that allows the use of CUDA compatible GPUs for more general tasks via compute shaders (see 3.1.2.1.9). We use DirectCompute extensively for the GPU generation of terrain (see 4.1).
- DirectX Graphics Infrastructure (DXGI) - provides a mapping between Direct3D and the graphics kernel in Windows, which then interfaces with the graphics driver and

finally the GPU itself. The project uses DXGI throughout as it is heavily integrated in Direct3D and provides enumerations for texture types, buffer types and so on.

The alternative to using DirectX would have been to use OpenGL with either CUDA or OpenCL replacing DirectCompute for GPGPU tasks. This would have made the application fully cross-platform but DirectX provides a cleaner interface allowing more focus on the key parts of the project.
The following subsections detail some the DirectX features that will be mentioned numerous times throughout the project and serve as additional background reading. Readers familiar with DirectX should skip to 3.1.3.

3.1.2.1 High level shader language and the Direct3D graphics pipeline

We give a short introduction to the main components of the DirectX pipeline, how it is programmed using high level shader language or HLSL and the resources available to it.

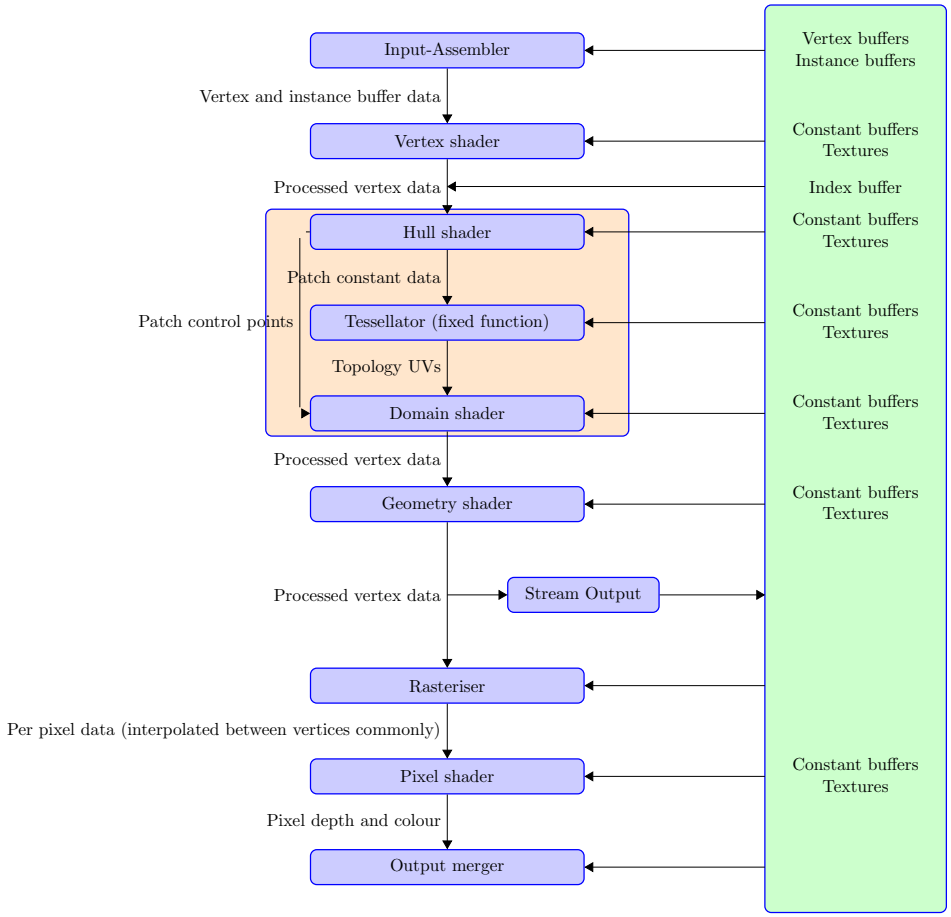


Figure 3.1: The DirectX Pipeline

3.1.2.1.1 Input assembler stage, vertex buffers and instance buffers The first stage of the pipeline is the input assembler. This is configured via calls to the Direct3D API from the main application parametrised with previously constructed vertex and instance buffers. Vertex buffers store the vertices which make up a 3D object, a box for example would have 8 vertices forming each corner. Vertices normally carry positional information but can contain any amount of other information such as colour and texture coordinates. Instance buffers are used for instancing - drawing the same object many times. They provide per-instance information such as transformations, colours and animation offsets. The purpose of instancing is to reduce the number of draw calls and constant buffer updates (constant buffers hold values in shaders which remain the same over at least the duration of a draw call). Each time a draw call is made a CPU overhead is introduced, constant buffer updates are fast but become significant over thousands of updates. Without instancing, to draw a box one hundred times in different positions would require one hundred draw calls each time updating a constant buffer with a matrix transformation. With instancing, we can make a single DrawInstanced call having previously constructed an instance buffer that contains the matrix transformation for each instance.

3.1.2.1.2 Vertex shader stage The vertex shader stage is a fully programmable stage of the pipeline (via HLSL vertex shaders) and performs operations on the input from vertex buffers and instance buffers. Example operations include transforming vertices for the camera's view perspective, calculating texture coordinates and performing additional displacements from texture lookups.

After the vertex shader, the next pipeline can either progress to the hull shader, the geometry shader or the pixel shader depending on requirements.

3.1.2.1.3 Second input assembler stage and index buffers The second input assembler stage occurs after the vertex shader stage if the draw call is using indexing and converts indexed vertices into raw vertices. Indexing is a way of reducing the size of vertex buffers and thus GPU memory usage when the topology of the objects being drawn involves the same vertex multiple times. Figure 3.2 shows a square made from four vertices, v_1 to v_4. To draw this square using a triangle list topology not using index buffers requires a vertex buffer of six vertices: v_1, v_2, v_4, v_2, v_3, v_4. If we use index buffers, the vertex buffer contains v_1, v_2, v_3, v_4 and the index buffer contains 16 or 32 bit integers referencing the offset in the vertex buffers: 0,1,3,1,2,3. For using index buffers to be worthwhile, the combined vertex and index buffer size should be smaller than a the vertex buffer would be if indexing were not used.

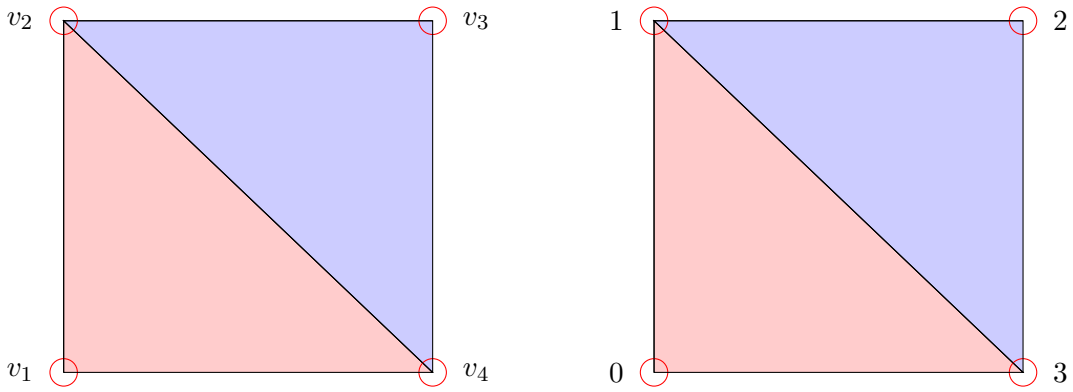


Figure 3.2: Index buffer example diagram

3.1.2.1.4 Hull shader stage The hull shader is a fully programmable stage that converts vertex input from the vertex shader or input assembler second stage into geometry patches and patch constants. These are then input into the domain shader and tessellator stages.

3.1.2.1.5 Tessellator stage The tessellator stage is a fixed function stage that creates a sampling pattern of the chosen topology (lines, triangles or quads) that represents the geometry patch and generates a set of smaller objects (triangles, points or lines) that connect these samples. That is, it performs tessellation on a patch of the specified parameters.

3.1.2.1.6 Domain shader stage The domain shader stage is a fully programmable stage that takes the output of the hull and tessellator stages and converts them back into vertices for input into the geometry or pixel shader stage.

3.1.2.1.7 Geometry shader stage The geometry shader stage is a fully programmable stage which processes vertices in batches according to the selected topology. It can add or remove vertices to the stream making it a very versatile but in some cases a costly stage.

3.1.2.1.8 Pixel shader stage The final stage is the pixel shader stage which is again fully programmable via HLSL. It takes as input a pixel which is generated by interpolating (there is a nointerpolation option for some specific uses) the vertex data for each pixel on the screen. This stage is normally used to sample from high resolution textures as pixels are denser than vertices so sampling before this stage would give blurred results. Lighting calculations and such are also common in the pixel shader.

3.1.2.1.9 Compute shader stage The compute shader is a separate stage to the rest of the pipeline that extends the capabilities of HLSL shaders beyond purely graphics based tasks (many other tasks are actually possible with vertex and pixel shaders but the process is

less natural). Unlike the other stages compute shaders provide direct control over threading on the GPU and features such as shared memory. There is a separate API call for using computer shaders, the dispatch call. We explain more details of the compute shader as we use it for terrain generation in 4.7. The computer shader stage is part of the DirectCompute API rather than the Direct3D API as with the other stages, although it is heavily integrated with Direct3D, allowing the use of Direct3D resources and views. So in practical terms, the distinction is of no consequence.

3.1.2.2 Resources and resource views

Resources in DirectX come in two forms, buffers and textures. The most common uses for buffers have alreay been mentioned as vertex buffers, instance buffers, index buffers and constant buffers. Textures store images and come in a variety of forms including 1-dimension, 2-dimension, 3-dimension, 2-dimensional arrays and 3-dimensional arrays. Buffers and textures are created with certain flags which determine how they may be used. For example, there are flags to determine whether resources can be written and read by the CPU, whether they can be used as a render target in the pixel shader stage and whether they can be written to in the compute shader. To make use of these properties, resource views are used. A ShaderResourceView binds a resource to a shader for read access, such as sampling a texture in the pixel shader stage. A RenderTargetView binds a resource to be written to by the output of a pixel shader. An UnorderedAccessView allows the binding of a resource to a compute shader which can allow the writing and reading of arbitrary components in that view.

3.1.3 Hieroglyph 3

The project uses a modified version of the rendering library Hieroglyph 3 as a light wrapper around parts of DirectX. Features we added include:

- Support for deleting resources.
- Safety for multi-threaded resource creation and deletion.
- DirectX 11.2 support (previously 11.1).
- An expanded texture loader to support additional DDS formats.
- Functionality to interface with Qt to provide an editor GUI on top of the graphics system.
- The DirectXMath SSE optimised maths library.

Hieroglyph 3 is primarily intended for small projects and rapid application development. It initially seemed a good choice to avoid dealing with some parts of DirectX directly. In retrospect, the time spent fixing bugs and adding the necessary features proved costly. As such, we would not recommend it to anyone developing similar scale projects in its current state and would instead favour the use of a more fully fledged engine or raw DirectX.

3.1.4 Qt

Qt is a cross-platform application framework for creating UIs written in and designed for use with C++. It provides a powerful events system, easily extendable and modifiable widgets and highly versatile data models, all of which we harness in creating the tools for using terrain system.

3.2 User interface and overall architecture

This report focusses on the core aspects of the project - the terrain system, graphics back-end and data driven elements. The complete implementation contains many other components. It is unrealistic to attempt covering all of these components in a single report but they would be necessary for any readers wanting to implement a similar system. We give a brief overview of some of these features and a look at how the system is structured at a high level before continuing to talk about our core components, starting in 4.1.

3.2.1 Code organisation and features

Visual Studio was used as an IDE because of the excellent graphics debugging features for DirectX which allow step-by-step tracing of HLSL shaders. This proved invaluable when implementing the rule compute shader functions. The code is organised into three projects, one for the engine, one for the implementation specifics and one for testing features. The engine includes the Hieroglyph 3 source code along with our modifications and additions mentioned in 3.1.3. We give a diagrammatic overview of the larger system in 3.3, the main components of which are:

1. The terrain system. For each rule and palette type there is a CPU side representation which is used in the compute shader generation process and by the UI for creating terrains. Each rule and palette type also provides a serialisation function for I/O. The CPU representation for our terrain system also has a full execution path which we use for benchmark comparisons between the GPU and CPU.

2. Shader permutation generator - provided our skeleton files for derived fBm noise types, generates variations that use different underlying generators such as Perlin noise (2.1.6.2),

Simplex noise (2.1.6.3) and so on.

3. I/O - wrappers around the basic C++ stream I/O classes to aid our serialisation system for loading and saving terrains.

4. UI - the custom widgets, windows and controls we use to build our user interface (3.2.2).

5. INI based settings handler - this is used to set and store graphics, benchmarking and directory information for when the program is executed. Storing commonly changed parameters means avoiding long compile times. We give an example configuration in A.1.

6. Graphics components

   (a) CPU-side frustum culling system

   (b) Vertex structures and information providers

   (c) Spacebox entity

   (d) Local star entity

   (e) Supporting components of the terrain system including the specific components we talk about

      i. GPU patch generator (4.7)

      ii. Multithreaded CPU patch generator (used for the CPU benchmarking in our evaluation, see 7)

      iii. Singlethreaded CPU patch generator

      iv. DXT compressor (5.3)

      v. Patch rasteriser (5.2)

      vi. Texture cache (5.2)

      vii. Terrain patch cache - manages the creation and destruction requests for the quadtree as the camera moves.

      viii. Noise preview generator (used for the noise palette UI)

      ix. Texture preview generator (used for the material and flora texture palette UI)

      x. Functions to generate index and vertex buffers for the various terrain patch types we use for CDLOD and our modification HWACDLOD

      xi. Quadtree storage system for terrain patches. The quadtree stores the location and other metadata for patches but rendering data is stored in the texture cache, which each node stores an index to when populated.

      xii. Terrain configuration structures

(f) Image classes for performing the operations used in the data driven components of the project (6).

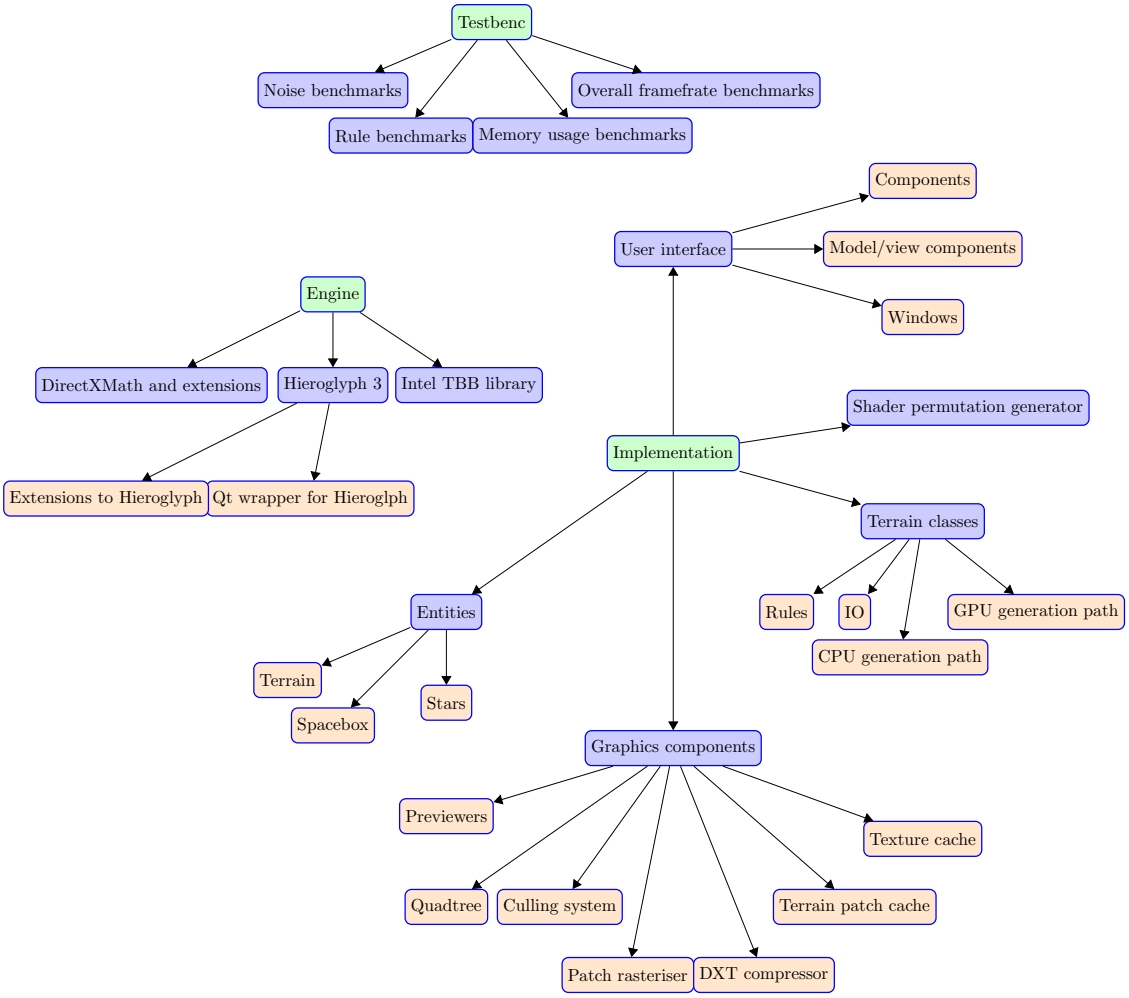


Figure 3.3: Partial overview for the structure of the complete system

3.2.2 User interface

Considerable effort was made in the development to make the program a usable piece of software rather than just a technical demo. To this end we implemented an extensive user interface that can be used to adjust all the implemented properties of the terrain system (4.1) and control the data driven (6) aspect of the project. Having this full, easy to use control over the terrain makes testing the system immeasurably easier than just hardcoding examples. The main UI is aesthetically simple but requires considerable work in synchronising modifications with the rendering of the terrain, supporting editing of all rule types, modification of rule

order, deletion/insertion of new rules and so on. The UI also provides standard features such as saving and restoring files, copy and paste (for rule and palette items) and so on.

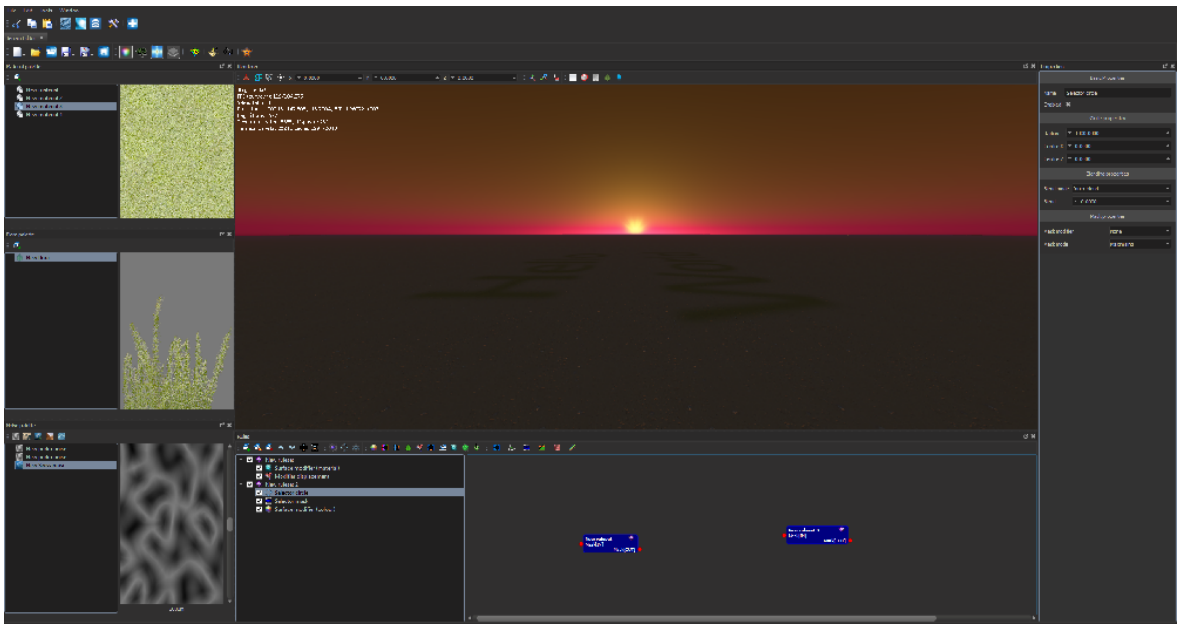


Figure 3.4: The main user interface. On the left are the material, flora and noise palettes (4.3). Along the bottom the rule-set (4.2) editor. On the right is the property editor. Central is the renderer.

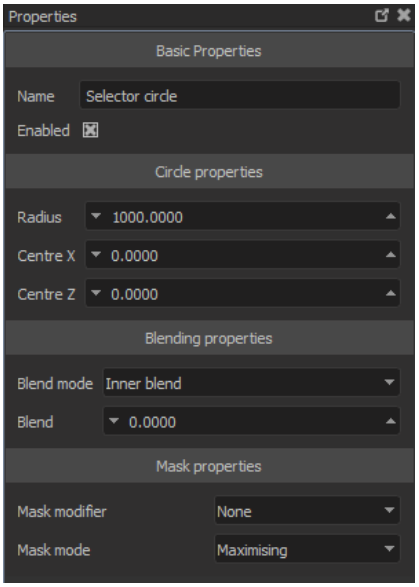


Figure 3.5: Properties window of the user interface displaying the properties for a circle mask selector (see 4.5.2)

4 Terrain system

4.1 Overview

In this part of the project we create a terrain generation system that can, in real time, generate procedural worlds with a high level of detail while also providing a coarse to fine level of artistic control.

Creating high quality procedural data will require the use of combinations of complex noise types as the input to fBm (see 2.1.6.4) and variations on fBm such as Swiss turbulence (see 2.1.6.5). One of the overarching goals of the project is for the system not to limit the applicability of the technology developed to any one use. As such it is necessary that this noise generation happens at a very high speed to facilitate applications where the camera position can move fast and unpredictably, such as flight simulators. To this end it was decided early on that the generation of noise should take place on the GPU due to its affinity for highly parallel tasks. This left the decision of whether to make the generation process run entirely on the GPU or mix between GPU and CPU generation. Using the GPU results in a higher shader complexity over just computing noise and imposes some restrictions on the kind of tasks that can be performed being a less general purpose processor than the CPU. However mixing between GPU and CPU generation may require multiple stages of execution. This is due to the current limitations on unordered access views (3.1.2.2) in compute shaders (four maximum as of DirectX 11.2, see 3.1.2.1.9) and render targets (3.1.2.2) in conventional shaders (eight maxiumum as of DirectX 11.2). As a result of this, generation of patches that require more than this number of noise results would be split into multiple stages increasing CPU overhead significantly and introducing either generation delays, synchronisation stalls or slowed rendering. Such cases are very likely to occur, especially on very large or even planet scale terrains where low level of detail patches can span vast areas each making use of differing generation techniques. With these points in mind, we opted for full GPU generation looking to minimise delay and generally explore the potential of such a system. Having made this decision from the offset, we looked to minimise the need for executing code that is not GPU-friendly by design.

Generating noise alone is not enough to produce a convincing output that can mimic realistic terrains and neither does it empower the user to guide the results. When considering the core components of how an artist would go about creating a terrain, we concluded that the process is essentially one of selecting areas and modifying them. In traditional brush-based systems for painting terrains, the task of selection happens implicitly at the same time as modification. In a system of procedural rules however, we may wish to, for example, to select areas with steep gradients and apply a rocky material or select areas above a certain

height and apply a snowy material. This idea gives the basis for the two main rule types in the system, which we call selectors and modifiers. Selectors create a mask over regions of the terrain and modifiers then alter its properties. A group of such rules forms a rule-set. Rule-sets provide the basis by which terrains can be constructed hierarchically by allowing nesting. A nested rule-set is bounded by the mask of its parent which allows the user to avoid duplicating rules in rule-sets that share a common boundary.

4.2 Rule-set design and implementation

Rule-sets are the base element used to construct terrains and give structure to the execution algorithm (see 4.7). They can contain any combination of other rule-sets, selectors and modifiers. They also define the base mask value which the contained selectors operate on. The default mask can be set by the user but has a default value of one. However, the default value and values contributed by contained masking rules are bounded by the parent rule-set's mask (root rule-sets are unbounded and the default mask value spans the entire terrain surface). Finally, rule-sets contain the ability to apply operators to the combined result of contained selectors. The operations offered are the same as those that can be applied to individual selectors (see 4.5).

4.3 Palettes

Palettes are the means by which users can import and create data in the system. We have three types of palettes in the system for materials, flora textures and noise. A screenshot of these palettes in the UI can be seen in 3.2.2.

4.3.1 Materials

The materials palette stores materials which comprise of various texture types that can be used for high frequency texturing of the terrain via the surface modifier rule (4.6.4). Our material system supports five texture types, not all of these have to be used but the user should supply at least a diffuse texture. In addition, zero or more of textures can be supplied for a normal map, specular map, emissive map and height map. The use of this many textures is supported by our rasterisation system (5.2). Diffuse textures provide the basic colour information. Normal textures allow the simulation of a higher resolution surface by modifying the surface normal and thus lighting results in the pixel shader (3.1.2.1.8). Height textures allow actual displacement of the surface (at lower resolution than normal textures due to being applied in the vertex shader, see 3.1.2.1.2). Emissive textures emit light rather than

just reflecting it which is supported by our HDR rendering and specular textures represent how shiny a surface is.

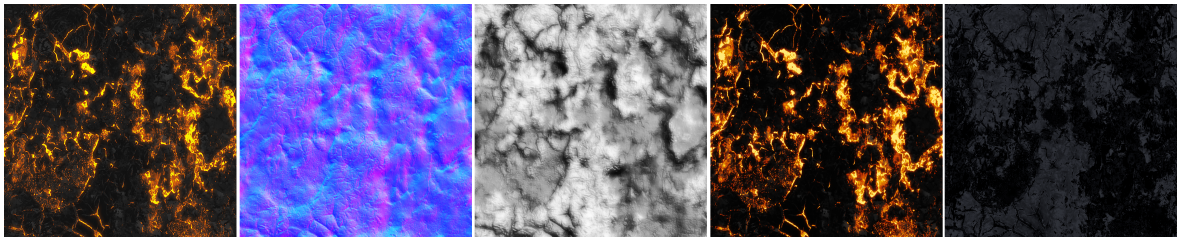


Figure 4.1: Examples of the texture types we support for a lava texture. From left to right: diffuse, normal, height, emissive and specular

4.3.2 Noise

The noise palette is used to define instances of the procedural noise types we have implemented - standard fBm, Swiss turbulence, Jordan turbulence, IQ turbulence and erosive turbulence. We do not cover the details of their implementation in this report since we follow the original algorithms besides one crucial addition which we take this opportunity to make note of. The standard fBm and derivative implementations do not guarantee a result in the [0,1] interval. So we calculate a normalising value and multiply the fBm result by that value. We noted in our background (2.1.6.4) that fBm works by for each octave, adding the noise result multiplied by a gain value. So to normalise a fBm result, we calculate the maximum possibel value. That is, the result obtained if every noise result returned one. We then invert this result to get a normalising multiplier.

The noise palette can also stores links to external noise inputs like heightmaps and vector field displacement maps. These palette items are referenced by various rules, for example the noise mask selector (4.5.11) will refer to an item in the noise palette to define its application. The details of this are explained in the individual rule explanations to follow.

4.3.3 Flora textures and billboarding

Flora textures reference a single diffuse texture file and can be used in the placement of billboarded flora. Our implementation of flora uses a fairly standard geometry shader technique where the edges of the terrain heightfield are extruded to form 2D rectanges. Rendering these rectangles with flora textures using transparency and alpha-to-coverage enabled then gives a quite convincing flora effect. This technique is used in almost all current real-time applications other than some more recent efforts to individually render blades of grass.

4.4 Rule parameters and modes of operation

We briefly introduce two points of terminology used before looking at the implemented rules individually. Each rule has zero or more parameters and one or more modes of operation. A parameter is an input parameter to the HLSL implementation of the rule. A mode of operation is a permutation of the function which is selected at rule compile-time in the shader creation and compilation stage of the generation pipeline (see 4.7). To the end user, there is no distinction as both parameters and modes of operation appear as options when configuring a rule using the property editor in the GUI.

4.5 Selector design and implementation

Selectors provide the means by which users can select areas of the terrain for further modification. Providing a range of highly customisable selector types ensures the rule-sets are powerful enough to allow users to create the kind of environments they desire.

The selector system works by creating 32-bit, single channel floating point image masks with values in [0,1]. A value of 1.0 means that the area is subject to the full effects of a modifier rule (4.6). A value of 0.0 means that the area will not be influenced by any modifiers. Values in-between cause modifiers to linearly interpolate between their results and the existing value of the property being adjusted. All selectors have the option to be inverted and adjusted using a mask modifier (see 4.5.12). These modifiers apply functions to the mask image which preserve the [0,1] range such as raising the result to a power.

Additionally, selectors have two modes of operation, minimising and maximising. In the maximising mode, the higher of its value and the current mask value is taken when processed. In the minimising mode the lower of its value and the mask value is taken, essentially filtering the existing result.

4.5.1 Constant mask

The constant mask returns a constant value c, which is parameterised by the user in [0,1]. It is most useful when combined with other selectors to act as a minimum or maximum value depending on the mode of operation.

4.5.2 Circular mask

A circular mask has two modes of operation, outer blend and inner blend. It takes as input parameters a value for radius, centre and blend amount. In the inner blend mode, the blend amount describes what fraction of the circle is blended to 0.0. With a blend amount of 0.0, the entire circle is filled to 1.0. With a blend amount of 1.0 only the pixel exactly on the centre has a value of 1.0 and the value is blended out to 0.0 as distance from the centre increases. In the outer mode, the blend amount describes what multiple of the radius the circle is extended by to blend to 0.0 from the initial radius. This is shown in 4.2.

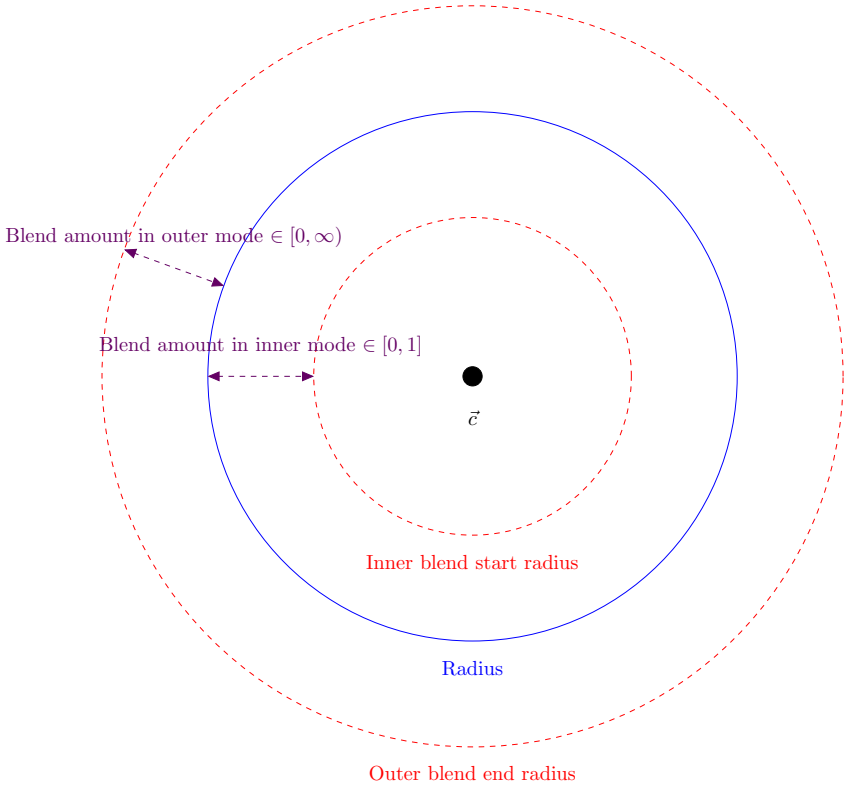


Figure 4.2: Outer and inner blend modes for circular masks. Inner blend preserves the input radius and blends towards the centre. Outer blend extends the input radius by blending out from it.

The intensity i of a pixel at $\vec{p}$, for a circlular mask with radius r, centre $\vec{c}$ and blend amount β is calculated by the following algorithm in the inner blend mode:

1. Calculate the blend radius, $\beta_r = (1.0 - \beta) \times r$, the radius at which blending begins.
2. Find the two values marked x and y in 4.3. Y is the difference between the blend radius and outer radius, $r - r_\beta$ and x is the distance between $\vec{p}$ and the inner radius $|\vec{c} - \vec{p}| - r_\beta$.
3. Dividing x by y gives the inverse intensity, so the intensity is $1.0 - \frac{x}{y}$ since when $x = y$, $\frac{x}{y} = 1.0$, but this occurs on at the outer radius where intensity should be 0.0.
4. Finally we must saturate the result (saturate is a common function in computer graphics that clamps a value in [0,1] and is an intrinsic function in HLSL). This ensures the result is valid when $\vec{p}$ does not lie within the blend region. First consider $\vec{p}$ outside the radius of the circle. Then $|\vec{c} - \vec{p}| - r_\beta > r - r_\beta$ and so $1.0 - \frac{x}{y} < 0.0$ and so gets clamped to 0. Next if p is within the blend region, $|\vec{c} - \vec{p}| - r_\beta$ is negative and so $\frac{x}{y}$ is negative so $1.0 - \frac{x}{y} > 1$ and so gets clamped to 1 as required.

So, the intensity is equal to:

$$i = \sqrt{saturate(1.0 - (|\vec{c} - \vec{p}| - r_\beta)/(r - r_\beta))} \qquad \text{where } r_\beta = (1.0 - \beta) \times r \tag{4.1}$$

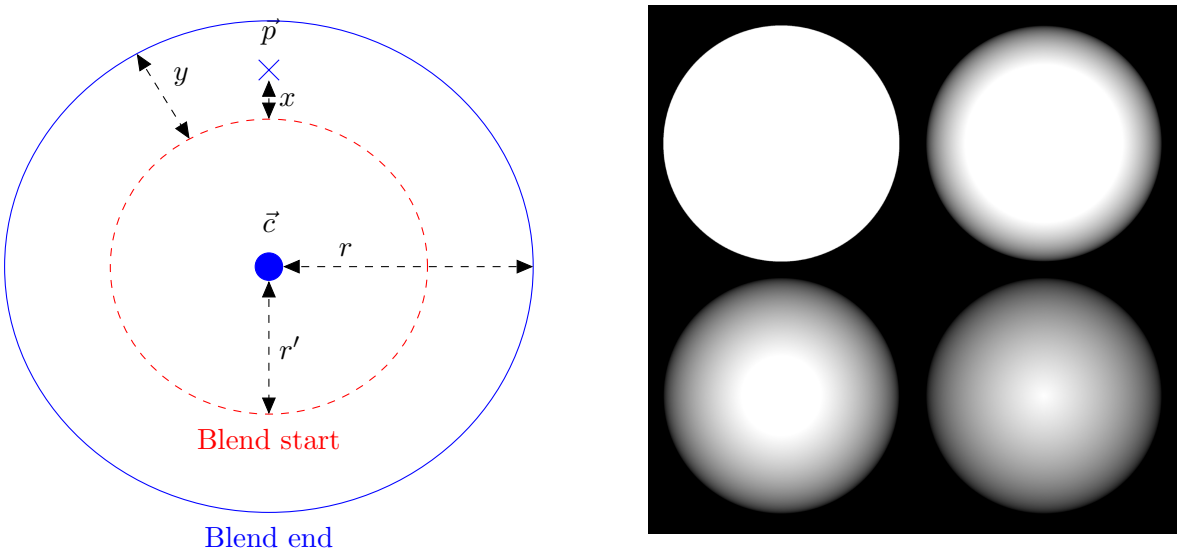


Figure 4.3: Circle mask diagram (inner mode) and an example showing circular masks with blend values of 0.0, 0.33, 0.66 and 1.0

The algorithm requires some minimal modifications to work in the outer blend mode. The blend distance is now a multiple of the radius added to the original outer radius. Since the blend radius is now larger than the original radius, the inversion is no longer required as the blend starts at the original outer radius rather than ending at it.

$$i = \sqrt{saturate(r_\beta - (|\vec{c} - \vec{p}|)/(r_\beta - r))} \qquad \text{where } r_\beta = (1.0 + \beta) \times r \qquad (4.2)$$

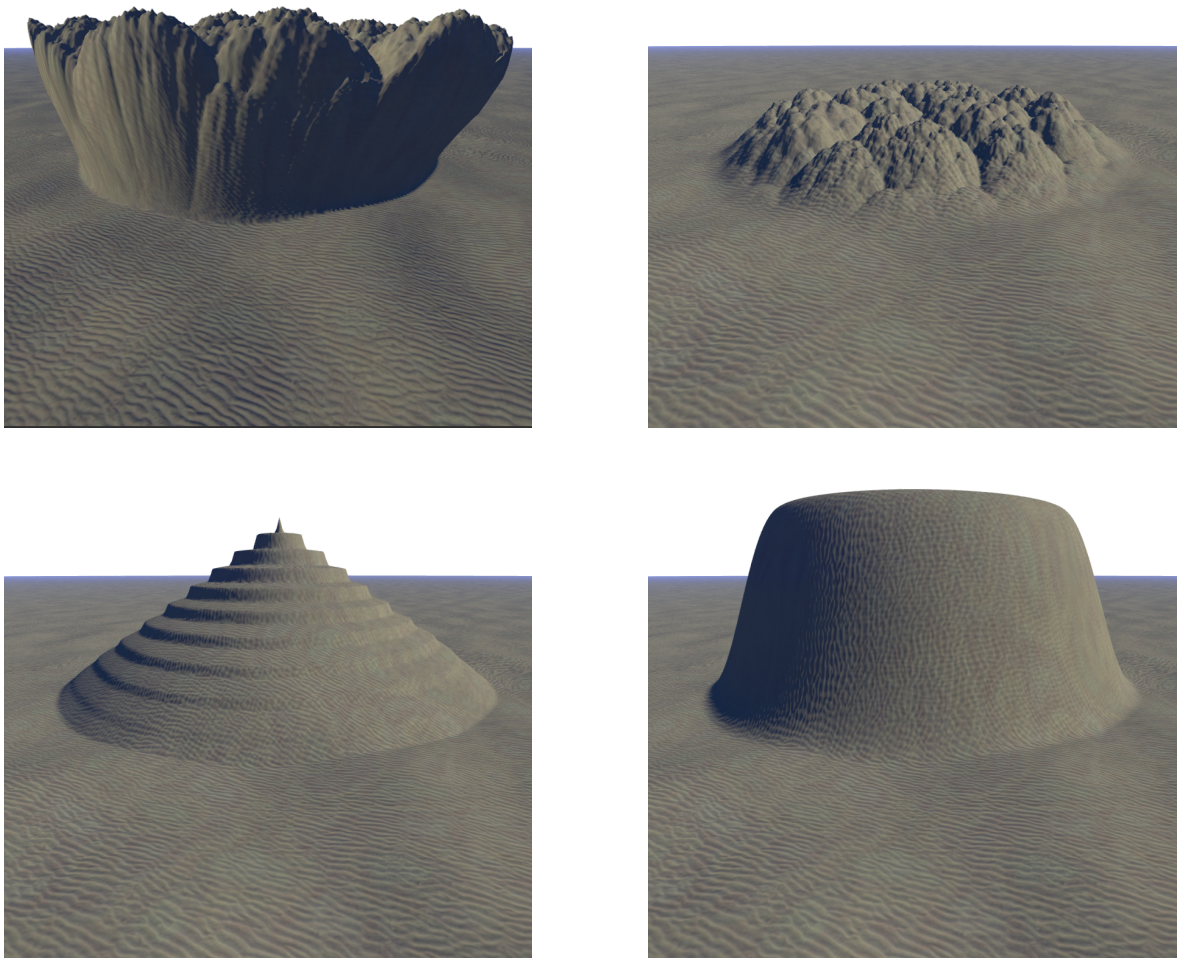


Figure 4.4: A selection of circular mask examples with various modifiers (4.6), mask modifiers (4.5.12) and parameters applied.

4.5.3 Rectangular mask

A rectangular mask takes as input a value for the minimum vertex, maximum vertex and blend amount. As in the circlular mask, there are two modes of operation for blending where the edges of the rectangle are blended outwards or inwards. The difference is that the rectangle has two potentially different distances to an edge so we consider the fraction to be of the distance to the minimum edge. The algorithm to calculate intensity i at a point $\vec{p}$, for a rectangular mask with blend amount β, minimum $\vec{v_0}$ and maximum $\vec{v_1}$ in the inner mode is:

1. Test if $\vec{p}$ lies within the rectangle, if not $i = 0$
2. Calculate the blend distance, $\beta_d = \dfrac{\beta * min(\vec{v_{1_x}} - \vec{v_{0_x}}, \vec{v_{1_y}} - \vec{v_{0_x}})}{2}$, the distance from the closest edge to the centre where blending occurs. This is marked x in 4.5.
3. Find the minumum distance d from $\vec{p}$ to an edge of the rectangle. This is simply the minimum component of $\vec{p} - \vec{v_0}$ and $\vec{v_1} - \vec{p}$.
4. The intensity is equal to the value of the distance as a fraction of the blend distance $\dfrac{d}{\beta_d}$. This does not account for the case when $d > \beta_d$ however. There are two solutions here, either saturate the value or let $d = min(d, \beta_d)$. We tested that the point actually lies in the rectangle to begin with so there is no negative d case to consider.

The outer blend mode requires only that the blend distance be subtracted from the minimum vertex and added to the maximum vertex before the initial check to test whether the point lies in the rectangle. It is then sufficient that the rest of the algorithm remains the same.

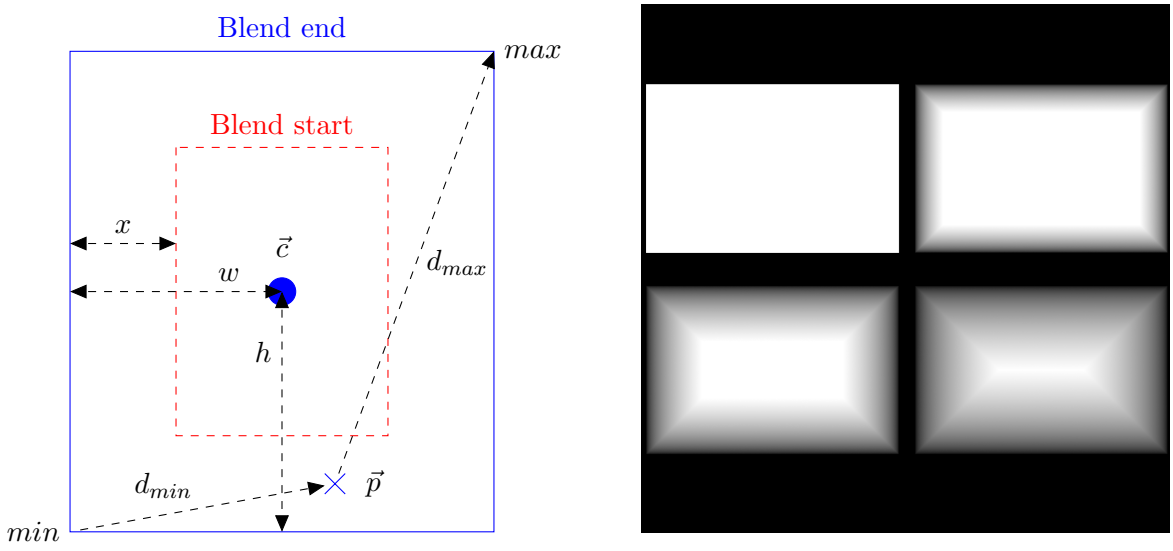


Figure 4.5: Rectangle mask diagram (inner mode) and an example showing rectangular masks with blend values of 0.0, 0.33, 0.66 and 1.0

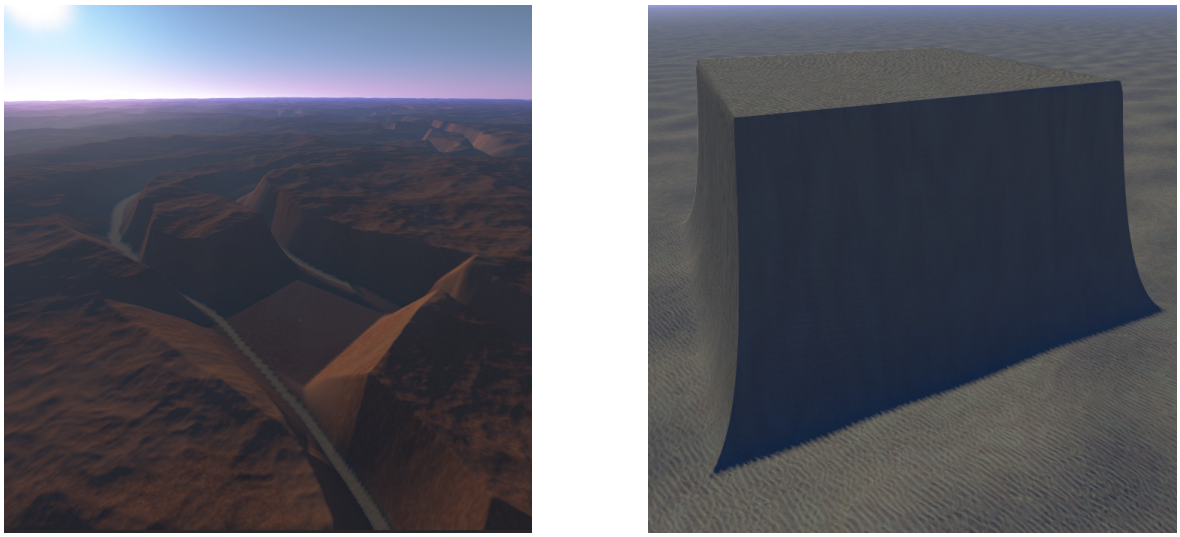


Figure 4.6: A rectangular mask used to flatten and extrude a region.

4.5.4 Polygonal mask

The polygonal mask takes as input a list of vertices, a bounding rectangle around those vertices and a blend amount. Similar to the circle and rectangle masks it allows the user to define a non-complex polygon with intensity blended out towards its edges. The process of calculating the intensity follows from those discussed previously but requires considerably more computational effort.

The first challenge we look at is how to find the blend distance. With no immediately apparent centre, one option would be to let user input an absolute distance rather than a normalised value $\beta \in [0, 1]$ as in the previous selectors (in inner mode). This would lead to guesswork however and is less intuitive. Instead we determine the blend distance for a polygon such that a user can input a value in the unit interval by first using the algorithm proposed by Martinez [24] for finding the inscribed circle in an irregular polygon . This algorithm returns the point inside a polygon which is furthest from any border or edge. From here, we find the closest distance from that point to any vertex or edge by testing in turn against each vertex and projecting onto each edge. The method used for finding the projection onto an edge is the same as used in calculating the intensity as discussed in the text to follow.

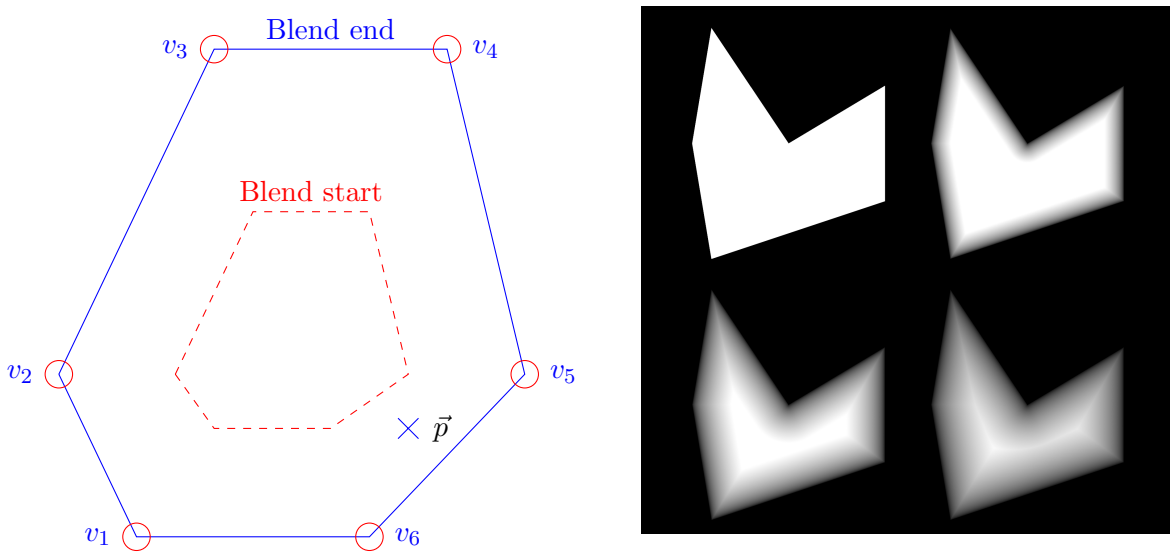


Figure 4.7: Polygon mask diagram and example showing rectangular masks with blend values of 0.0, 0.33, 0.66 and 1.0

The algorithm to calculate intensity i at a point $\vec{p}$, for a polygonal mask with blend amount β, vertices $V = \{\vec{v}_0, \ldots, \vec{v}_n\}$ with a bounding rectangle x_0, x_1, y_0, y_1 is:

1. Test whether $\vec{p}$ is within the rectangle defined by x_0, x_1, y_0, y_1. If not, $\vec{p}$ is definitely not in the polygon itself so return 0. Otherwise continue to step 2.

2. Test whether $\vec{p}$ is inside the polygon by using one of the standard methods. The implementation uses the crossings algorithm where a ray is projected in any direction from the test point and the number of times the ray crosses an edge of the polygon is counted. If the number of crossings is odd, the point lies in the polygon and if the number of crossings is even, the point lies outside the polygon. If $\vec{p}$ is determined to be outside, return 0, otherwise proceed to calculate the blended intensity value. If $\vec{p}$ is inside and a 0 blend distance is specified we could optionally just return 1.0 here.

3. Find the minimum distance from the test point to any vertex in the polygon or point on the edge.

   Finding the minimum distance to any vertex is calculated as $\forall v_i \in V, d_{min} = min(\vec{p} - v_i, d_{min})$, where d_{min} is initially a large value.

   The process for finding the distance from $\vec{p}$ to the edge between $\vec{v}_{i-1}$ and $\vec{v}_i$ is shown in figure 4.8. The vector $\vec{x}$ from $\vec{v}_{i-1}$ to $\vec{p}$ is projected onto the vector from v_{i-1} to $\vec{v}_i$. We call this vector $\vec{y}$. $\vec{y}$ ends at the point on the edge closest to $\vec{p}$, so $|\vec{y} - \vec{x}|$ gives the distance between the point and the edge. It may be the case that the projection of $\vec{p}$ onto $\vec{v}_{i-1} - \vec{v}_i$ does not lie on $\vec{v}_{i-1} - \vec{v}_i$. In this case the result is discarded. Mathematically, this gives:

$$\vec{edge} = \vec{v}_i - \vec{v}_{i-1} \tag{4.3}$$
$$\vec{x} = \vec{p} - \vec{v}_{i-1} \tag{4.4}$$
$$proj_{\vec{x}}\vec{edge} = (\vec{edge} \cdot \vec{x})/(\vec{edge} \cdot \vec{edge}) \tag{4.5}$$
$$\vec{y} = \vec{x} - \vec{edge} \times proj_{\vec{x}}\vec{edge} \tag{4.6}$$
$$\vec{b} = \vec{x} - \vec{y} \tag{4.7}$$
$$d_{min} = min(\vec{v}, d_{min}) \tag{4.8}$$

4. Return the minimum of the blend distance and minimum distance as a percentage of the blend distance

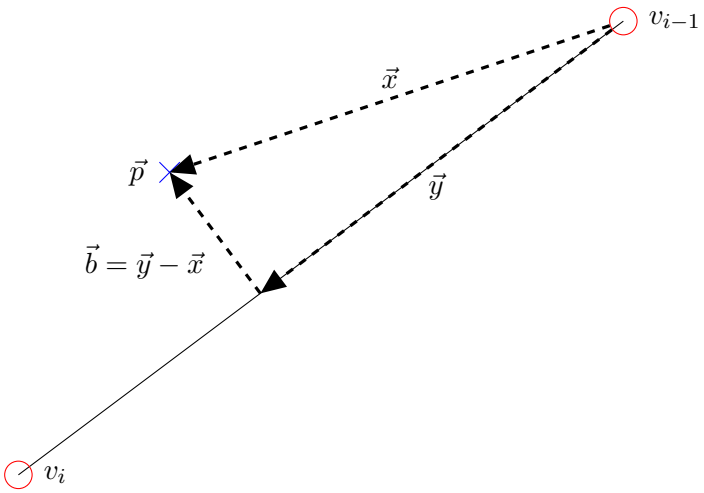


Figure 4.8: Polygon mask detail

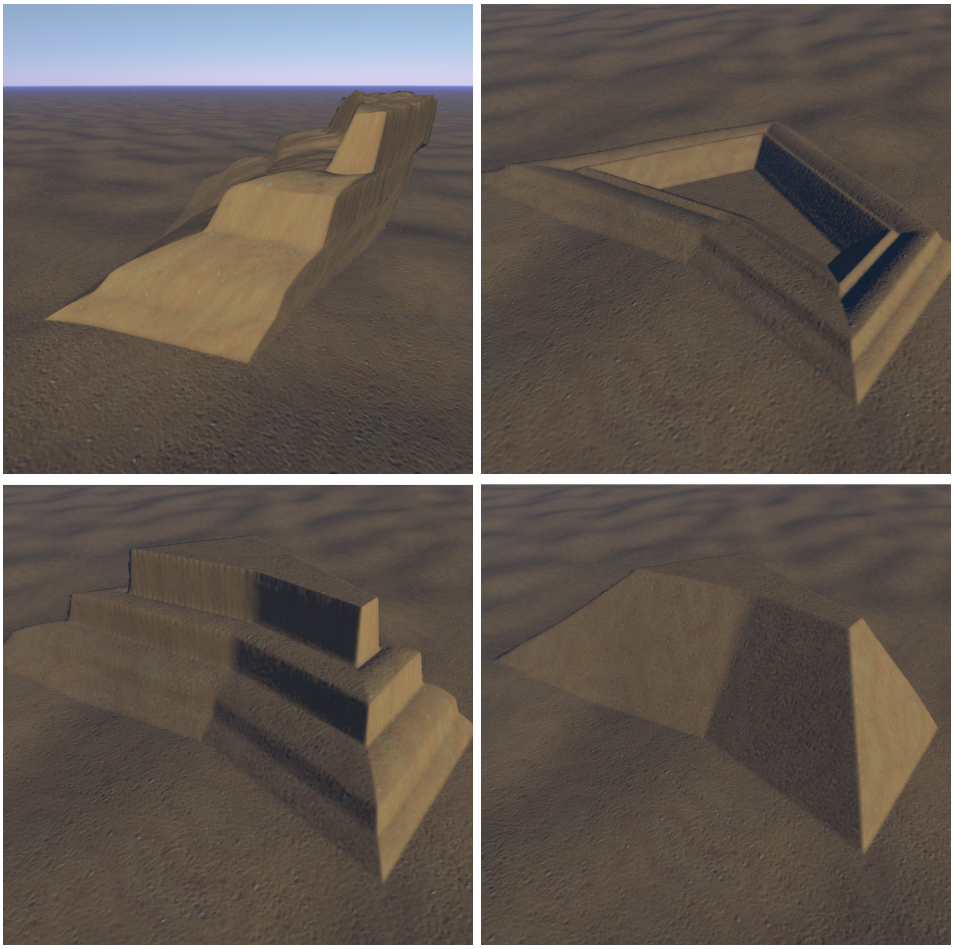


Figure 4.9: A selection of polygonal mask examples with various modifiers.

4.5.5 Feature selector overview

The subsequent selectors create masks based on the existing features of the terrain, namely the height, gradient and direction. We often see features in real terrains that correlate to certain physical properties of it so this needs to be able to be replicated in the system. Some examples include:

1. Snow can be seen on mountain peaks above a certain height depending on the time of year.
2. Steep slopes tend to be more rocky and have less vegetation.
3. Directional features are less common and not so obvious but we include the feature for completeness and to inspire novel designs by artists.

Figure 4.10: Mountains in Berchtesgaden, Germany

4.5.6 Height selectors

The height selector takes a minimum (h_{min}) and maximum (h_{max}) height value, test height value h and a blend amount (b) as parameters and creates a mask over regions of terrain that fall within the given range. We calculate the blend distance $d_b = \frac{b}{2}(h_{max} - h_{min})$, this is the distance from the maximum and minimum value where blending occurs. Please note that this section covers the dual-blending mode of operation for the height selector only for clarity, we summarise other modes in 4.5.9.

Calculating the mask value is straightforward. There are four cases to consider depending on where h lies:

1. If h is less than h_{min} or greater than h_{max} then h lies outside of the height range so return 0.0.
2. If h is greater than $h_{min} + d_b$ or less than $h_{max} - d_b$ then h lies inside the range and not in a blend region so return 1.0.
3. If $h > h_{min}$ and $h < h_{min} + d_b$ then return the fraction $\frac{h - h_{min}}{d_b}$ so that the closer h is to h_{min}, the lower the blend result.
4. If $h < h_{max}$ and $h > h_{max} - d_b$ then return the fraction $\frac{h_{max} - h}{d_b}$ so that the closer h is to h_{max}, the lower the blend result.

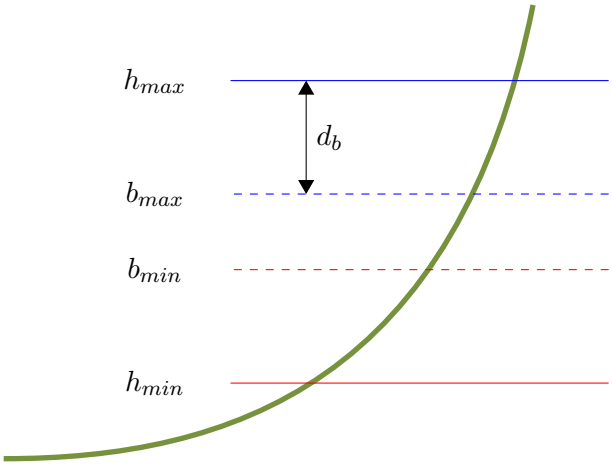


Figure 4.11: Diagram and variables for height selector

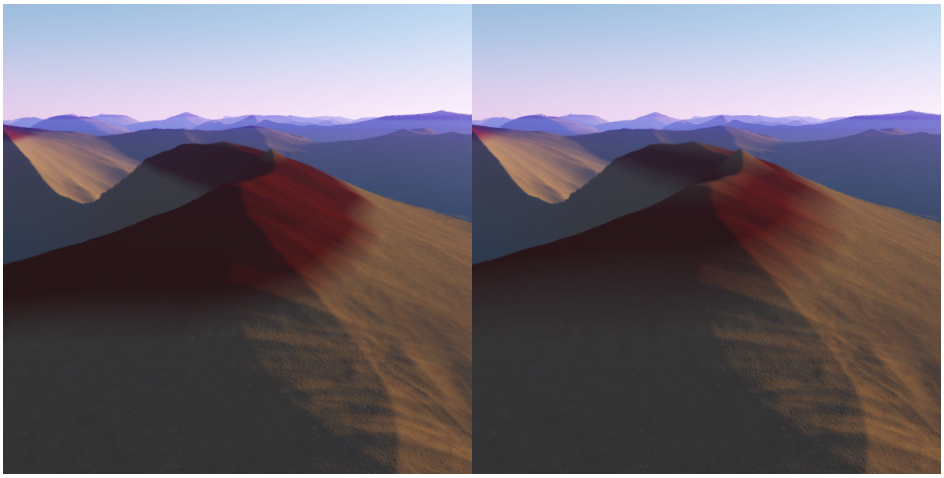


Figure 4.12: Height selector used to colour a band at the top of a elevated area of terrain with different blends amounts visible.

4.5.7 Gradient selector

The gradient selector works almost identically to the height selector only rather than specifying a height range, it is parametrised by an angle range. The parameters are the minimum angle $n_{y_{min}}$, maximum angle $n_{y_{max}}$, test value n_y and blend amount b. The n_y naming comes from the gradient of the terrain at any point being represented in the y-component of the normal vector. Prior to the execution of gradient selectors and other rules that require normal vector information, a memory synchronisation call and normal calculation step are added explicitly to avoid calling them when it is not necessary. In the GUI, the angle is shown in degrees to users for familiarity but all storage and calculation uses radians. Please note that this section covers the dual-blending mode of operation for the gradient selector only for clarity, we summarise other modes in 4.5.9.

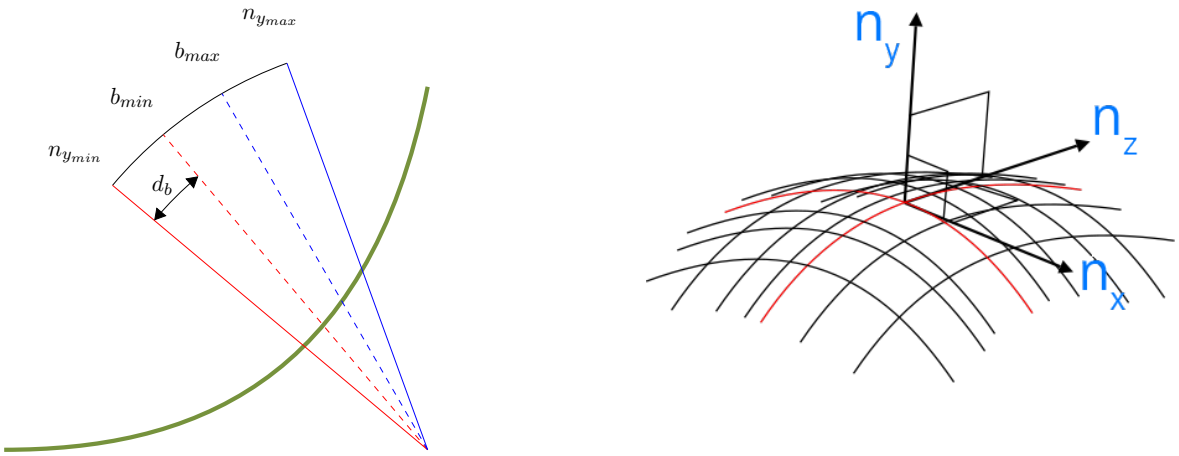


Figure 4.13: Diagram and variables for gradient selector (left) and normal vector components of the terrain surface (right).

The blend angle is calculated as $\theta_b = \frac{b}{2}(n_{y_{max}} - n_{y_{min}})$. Calculating the mask value comes down again to four possible cases depending on where n_y lies:

1. If n_y is less than $n_{y_{min}}$ or greater than $n_{y_{max}}$ then return 0.0.
2. If n_y is greater than $n_{y_{min}} + \theta_b$ or less than $n_{y_{max}} - \theta_b$ then n_y lies inside the range and not in a blend region so return 1.0.
3. If $n_y > n_{y_{min}}$ and $n_y < n_{y_{min}} + \theta_b$ then return the fraction $\frac{n_y - n_{y_{min}}}{\theta_b}$ so that the closer n_y is to $n_{y_{min}}$, the lower the blend result.
4. If $n_y < n_{y_{max}}$ and $n_y > n_{y_{max}} - \theta_b$ then return the fraction $\frac{n_{y_{max}} - n_y}{\theta_b}$ so that the closer n_y is to $n_{y_{max}}$, the lower the blend result.

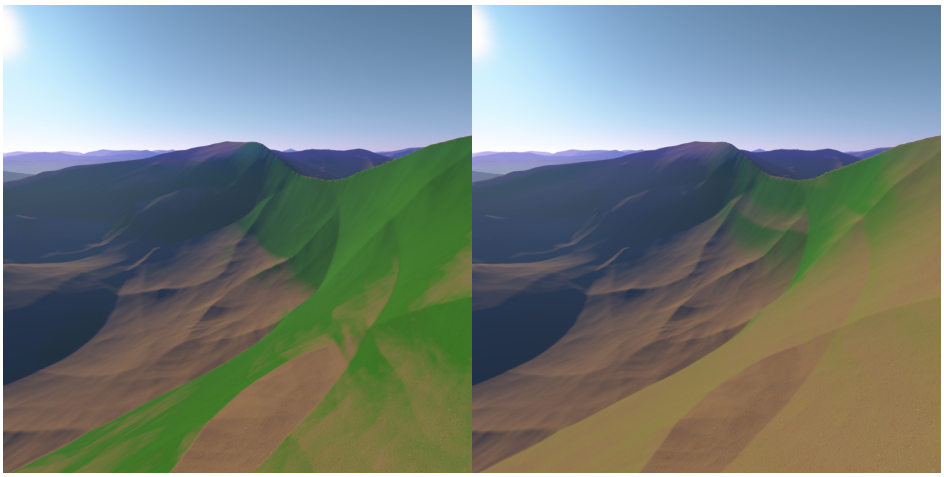


Figure 4.14: The gradient selector used to select the steeper parts of a terrain with low (left) and high (right) blend amounts.

4.5.8 Direction selector

The direction selector is a function of the x and z-components of the normal direction, which is calculated prior to the selector execution as in the gradient selector. We calculate that arctangent of the two components to give a single angle for comparison against the user provided range, $n_xz = arctan(n_x, n_z)$. There is one additional factor to consider in the direction selector, that is the treatment for the undefined result of arctangent(0,0). We let the user decide either to return 0.0 or 1.0 in this case with an boolean additional parameter, u. Please note that this section covers the dual-blending mode of operation for the direction selector only for clarity, we summarise other modes in 4.5.9.

The blend angle is calculated as $\theta_b = \frac{b}{2}(n_{xz_{max}} - n_{xz_{min}})$ and the test angle calculated as $n_{xz} = arctan(nx, nz)$. Calculating the mask value is similar again:

1. If $n_x = 0 \wedge n_y = 0$, return 1.0 if u else return 0.0.
2. If n_{xz} is less than $n_{xz_{min}}$ or greater than $n_{xz_{max}}$ then return 0.0.
3. If n_{xz} is greater than $n_{xz_{min}} + \theta_b$ or less than $n_{xz_{max}} - \theta_b$ then n_{xz} lies inside the range and not in a blend region so return 1.0.
4. If $n_{xz} > n_{xz_{min}}$ and $n_{xz} < n_{xz_{min}} + \theta_b$ then return the fraction $\frac{n_{xz} - n_{xz_{min}}}{\theta_b}$ so that the closer n_{xz} is to $n_{xz_{min}}$, the lower the blend result.
5. If $n_{xz} < n_{xz_{max}}$ and $n_{xz} > n_{xz_{max}} - \theta_b$ then return the fraction $\frac{n_{xz_{max}} - n_{xz}}{\theta_b}$ so that the closer n_{xz} is to $n_{xz_{max}}$, the lower the blend result.

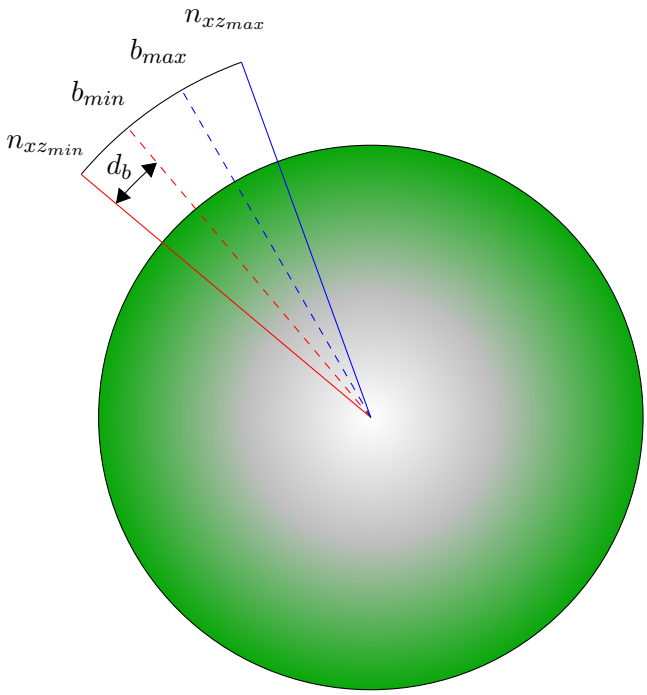


Figure 4.15: Diagram and variables for direction selector

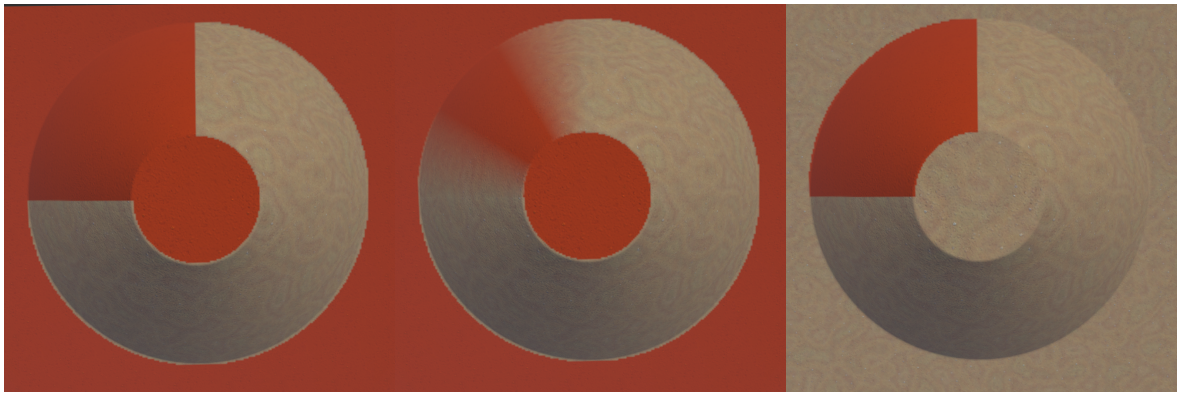


Figure 4.16: The direction selector with low (left) and high (middle) blend values and different treatments for non-defined angles (left and middle versus right).

4.5.9 Additional modes of operation for height, gradient and direction selectors

In the previous three sections we discussed the dual-blend mode of operation for the height, gradient and direction selectors. This mode is best for blending in new features that affect displacement as it is rarely wanted to have any sharpness. However, for blending in surface properties like colours and materials it is often desirable to blend only from a single direction to produce a linear gradient. For this we modify the algorithm slightly. We give the details on how to alter the algorithm for the height selector only as the changes parallel those made to the gradient and direction selectors.

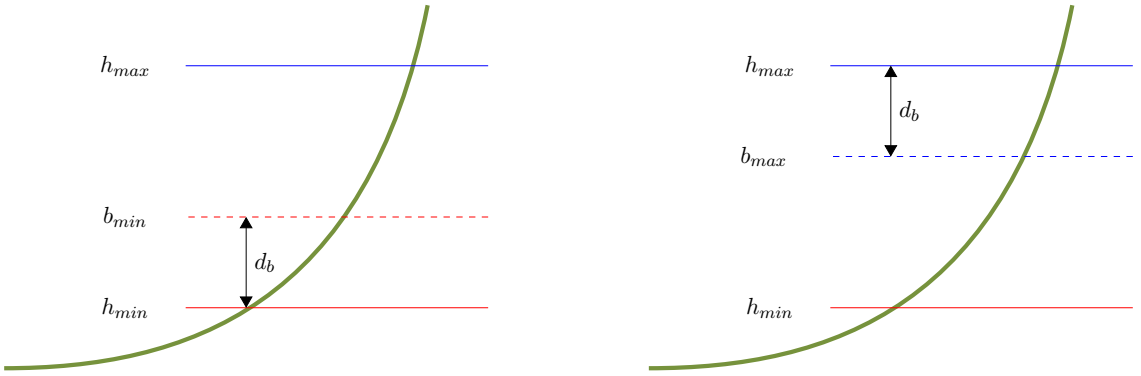


Figure 4.17: Diagram and variables for gradient selector (left) and normal vector components of the terrain surface (right).

Now there are only three cases in the algorithm, the height either lies within the blend region, in the fully masked region or outside the range. d_b now needs to span up to the entire ranger, so $d_b = b(h_{max} - h_{min})$. So to blend at the minimum:

1. If h is less than h_{min} or greater than h_{max} then return 0.0.
2. If $h > h_{min}$ and $h < h_{min} + d_b$ then return the fraction $\dfrac{h - h_{min}}{d_b}$ so that the closer h is to h_{min}, the lower the blend result.
3. Otherwise return 1.0.

Or, to blend at the maximum:

1. If h is less than h_{min} or greater than h_{max} then return 0.0.
2. If $h < h_{max}$ and $h > h_{max} - d_b$ then return the fraction $\dfrac{h_{max} - h}{d_b}$ so that the closer h is to h_{max}, the lower the blend result.
3. Otherwise return 1.0.

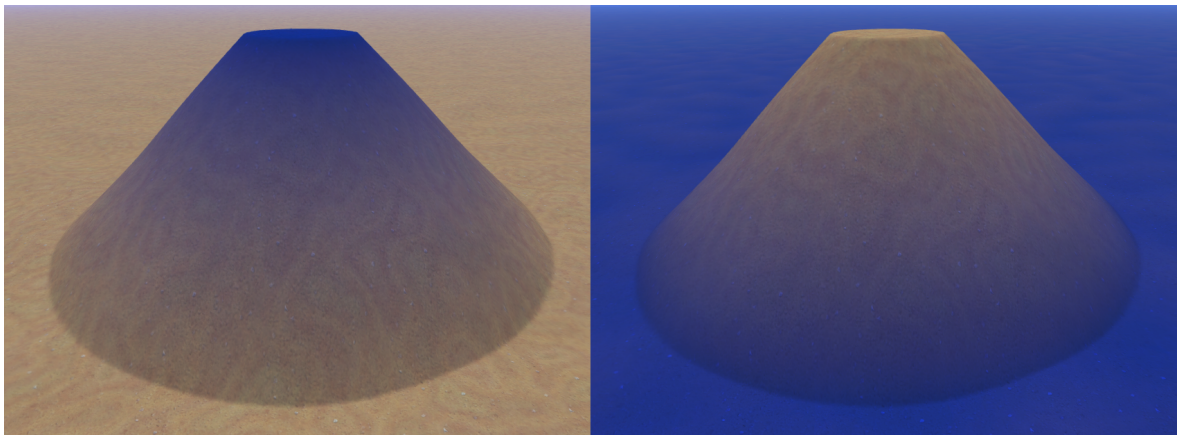


Figure 4.18: Height selector with single blending from the maximum (right) and the minimum (left).

4.5.10 Band selectors

The band selector is essentially syntactic sugar that allows users to more naturally select multiple height, direction or gradient ranges. Using the user interface the user selects any number of min-max pairs for the selected band type. The blend amount is fixed for all bands in the list, but this implementation could easily be altered so that each band can take its own blend amount should such a feature be deemed useful. As with the entire system, extensibility is always in mind. The current HLSL implementation is equivalent to having multiple maximising single selectors.

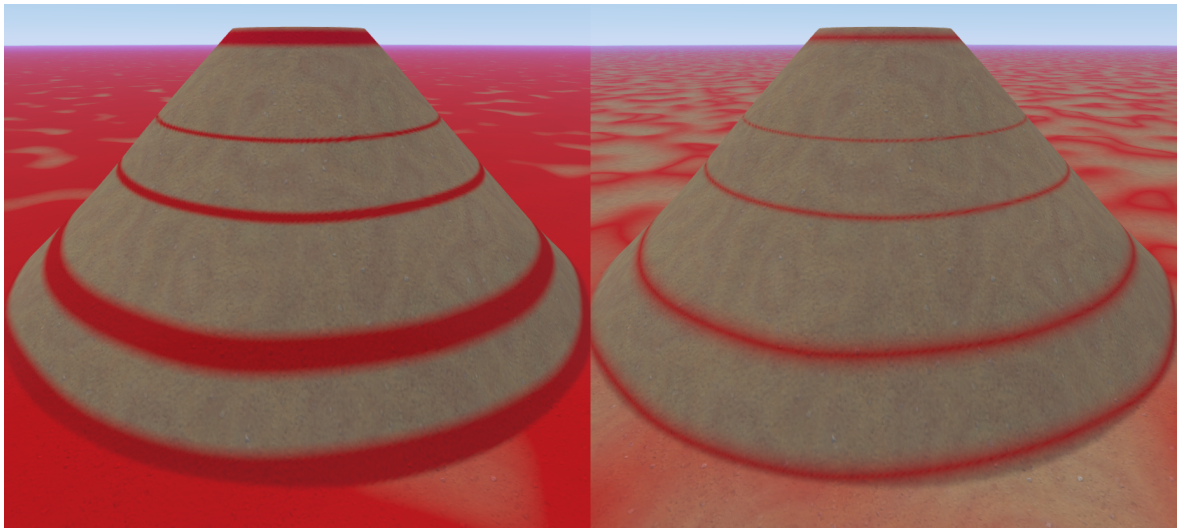


Figure 4.19: Banded height selector with low (left) and high (right) blending.

4.5.11 Noise mask

The noise mask lets users make selections based on the value of fractal noise generators or steamed user input masks. It works in the same way and has the same modes of operation as the height, gradient and direction selectors but has one additional parameter. This extra parameter enables or disables the multiplying of the mask result by the original noise value. This allows the masking to give almost any desired selection. With the multiplication enabled, the mask could be the original noise value or a blended version between the minimum and maximum. With the multiplication disabled, a band of the noise could be selected with a maximum or blended mask value. This type of mask proves particularly useful for creating features like rivers.

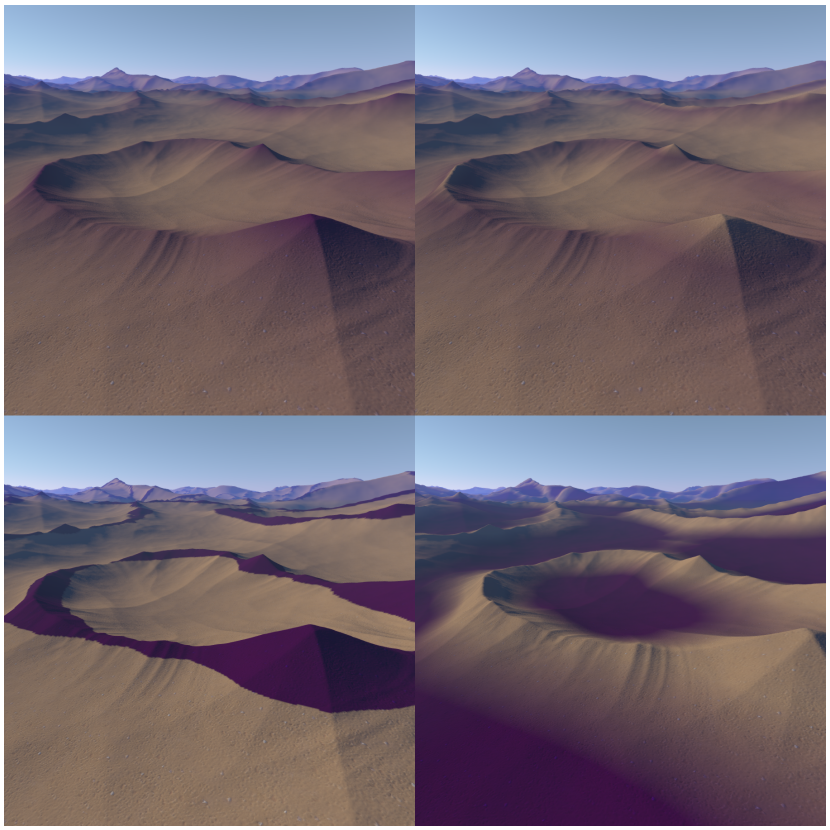


Figure 4.20: Variety of example noise masks with various blend amounts and modes of operation showing the range of selections that can be made.

We give the algorithm for the mask including the multiplication only due to the similarities with the previous selectors. Let i_{min} and i_{max} be the minimum and maximum intensity values, i be the test intensity value from fractal noise or user mask input, b be the blend amount. The blend distance is again either $d_b = b(i_{max} - i_{min})$ or $d_b = \frac{b}{2}(i_{max} - i_{min})$ depending on the mode of operation.

1. If i is less than i_{min} or greater than i_{max} then i lies outside of the height range so return 0.0.

2. If i is greater than $i_{min} + d_b$ or less than $i_{max} - d_b$ then i lies inside the range and not in a blend region so return i.

3. If $i > i_{min}$ and $i < i_{min} + d_b$ then return the fraction $i\dfrac{i - i_{min}}{d_b}$ so that the closer i is to i_{min}, the lower the blend result.

4. If $i < i_{max}$ and $i > i_{max} - d_b$ then return the fraction $i\dfrac{i_{max} - i}{d_b}$ so that the closer i is to i_{max}, the lower the blend result.

4.5.12 Mask modifiers

Mask modifiers are simple functions which alter the result of a mask or the result of multiple masks when applied from a rule-set. They make the selector system more expressive by increasing the diversity of masks that can be produced. The requirement for these functions is that the output range must be no larger than [0,1] for any input. The structure of the system means that end users could create their own functions if desired, but the currently implemented types are:

1. Square - $i_{out} = i_{in}^2$

2. Cubic - $i_{out} = i_{in}^3$

3. Quartic - $i_{out} = i_{in}^4$

4. Square root - $i_{out} = \sqrt{i_{in}}$

5. Smootherstep - $i_{out} = i_{in}^3(i_{in}(6i_{in} - 16) + 15)$

6. Smoothstep - $i_{out} = i_{in}^2(3 - 2i_{in})$

7. Cosine smooth - $i_{out} = (1.0 - cos(\pi i_{in})) * 0.5$

8. Terracing - terracing is implemented as explained in 4.6.2, only the linear interpolation parameter is removed.

9. Bias - the bias function was first described by Perlin and Hoffert [35]. It is a power curve defined over [0,1] that takes a single input parameter $t \in [0, 1]$. When t=0.5 the output is linear, otherwise it shifts the middle region of the inputs down for values less than or up for values greater than 0.5.

10. Gain - the gain function was also proposed in Hypertexture [35]. It combines two bias curves to produce an 's' shape function which centres more or less output values around the mid-range, again depending on the value of a single parameter $t \in [0, 1]$. As with the bias function, the response is linear for $t = 0.5$.

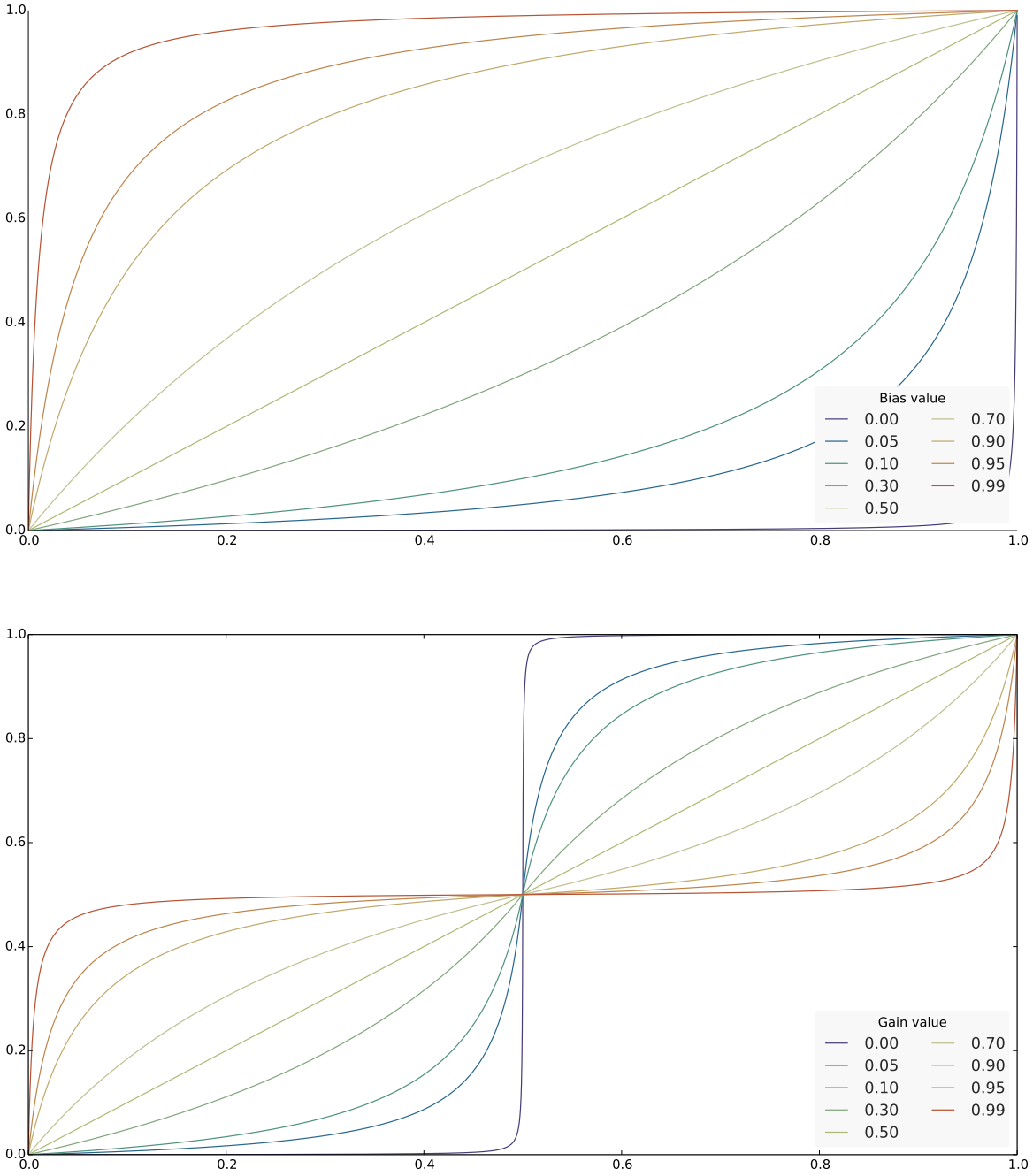


Figure 4.21: Gain and bias functions

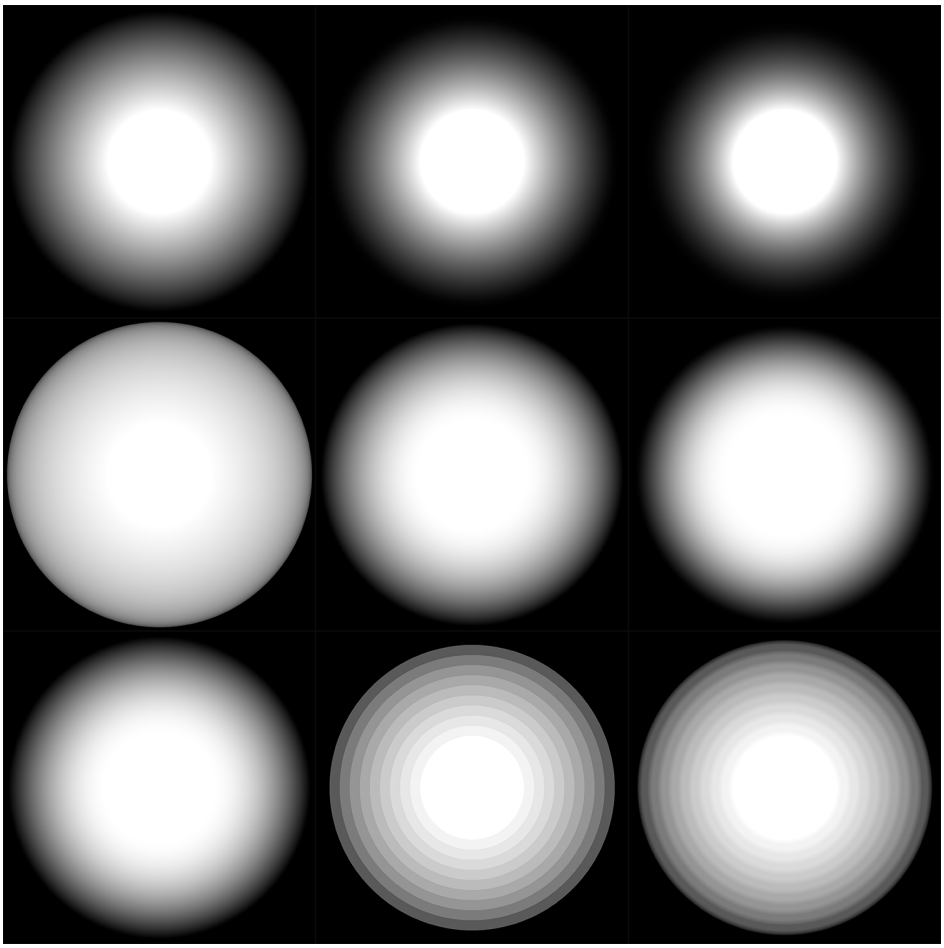


Figure 4.22: Mask modifers applied to a 60% blended circle mask (see 4.5.2). From left to right, top to bottom: square, cubic, quartic, square root, smoothstep, smootherstep, cosine smooth, terracing (no smoothing), terracing (50% smoothing).

4.6 Modifier design and implementation

The current terrain implementation provides five per vertex properties. These properties are altered, either directly or indirectly, by the modifier rules over regions defined by the masks created by selectors. These properties could easily be expanded in the future to add additional properties and rules to the system, some suggestions are included in 8.1. The properties are:

1. Displacement - how much a vertex should be displaced from its default location on the planetary surface.

2. Surface normal - the gradient of the surface, during rendering this can optionally be generated in real time but the value is also needed during generation by the gradient

(4.5.7) and direction (4.5.8) modifiers.

3. Low frequency texture - a per-vertex diffuse colour which is multiplied by the high frequency texture during rendering to create a more natural variation than is possible with a single texture alone.

4. High frequency texture - an integer identifier to be used by the rasteriser stage when generating the surface reflectance maps.

5. Flora type/density - an integer identifier to a texture array offset to be used when rendering flora using the geometry shader in the rendering stage.

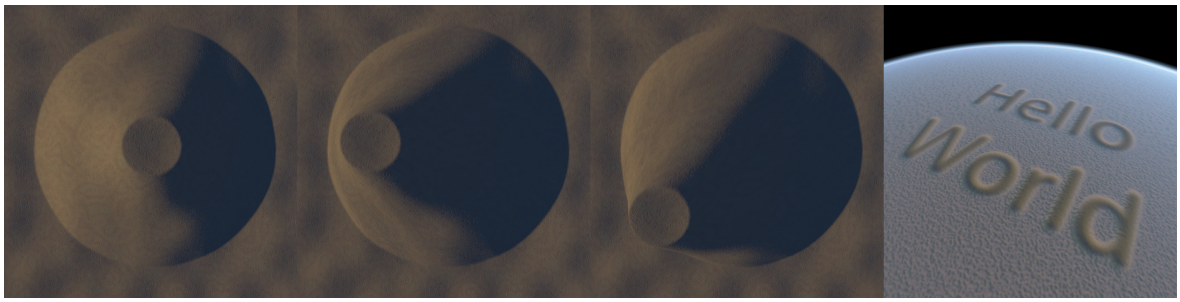


Figure 4.23: A variety of displacement vectors applied to a circular mask 4.5.2 and user provided mask used to displace a spherical terrain.

4.6.1 Displacement modifier

The displacement modifier alters the value of the the displacement property of the terrain. It has two modes of operation, absolute and relative and parameters for the displacement amount $\vec{d}$, the current value $\vec{v}$ (passed by reference as an inout HLSL parameter), the mask value m. In the absolute mode the result is the linear interpolation between the current value and new value by the mask amount, $\vec{v} = (1.0 - m) * \vec{v} + \vec{d} * m$ or $\vec{v} = lerp(\vec{v}, \vec{d}, m)$. In the relative mode, the new value is added to the current value $\vec{p} = \vec{p} + m\vec{d}$. We can conceivably add other modes of operation such as a subtractive relative mode but the value of such modes would be questionable.

The input value for the displacement amount depends on how the modifier is configured. There are three options given to the user as to how to provide displacement information, by a constant amount, by a procedurally generated amount and by an amount defined by streamed data. This input value is calculated in the compute shader before being passed into the modifier function.

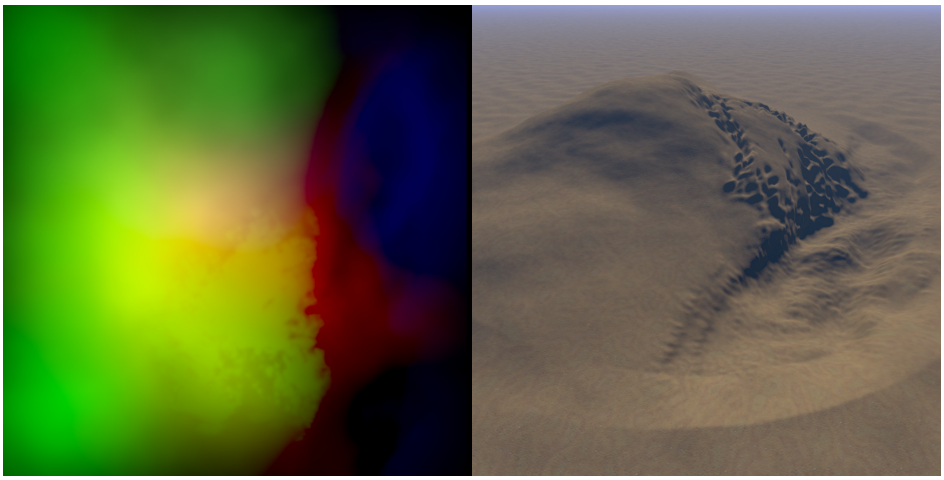


Figure 4.24: Example of vector field displacement map usage with displacement modifier

4.6.1.1 Displacement by a fixed amount

The user provides a constant value for $\vec{d}$. An example is shown on the left of figure 4.23.

4.6.1.2 Displacement by procedural data

The user provides a reference to one of the fractal noise generation functions in the noise palette. This is then sampled at the current position in the compute shader before being passed to the displacement modifier.

4.6.1.3 Displacement by external user data (heightmaps and vector field displacement maps)

The user provides a reference to one of the user defined maps in the noise palette as a parameter. During execution, the data is either statically loaded or streamed into a texture resource (3.1.2.2) as required depending on the size of the source image. Bilinear sampling is used in the shader to get the value of the texture at the current position when processing the rule. When using static data, the normalised 2D texture coordinate $\hat{uv}$ for performing sampling is taken to be $((\vec{p}_x - \vec{b}_{x_{min}})/(\vec{b}_{x_{max}} - \vec{b}_{x_{min}}), (\vec{p}_y - \vec{b}_{y_{min}})/(\vec{b}_{y_{max}} - \vec{b}_{y_{min}}))$, where p is the current position and b is the bounding box around the current rule-set. This is shown in figure 4.25. The bounding box is calculated and compiled in the shader when it is generated. If the rule-set is not bounded, then for planar terrains the mask is applied over the whole terrain with each corner of the terrain correlating to one corner of the source image. In spherical terrains, the image is treated as a latitude/longitude image and sampled by conversion of polar coordinates to Cartesian coordinates to perform the sampling. This can be particularly useful in applications where a low-resolution source image is available for a planet surface and the user wants to add detail with procedural data. When data is streamed, it is loaded into a texture of precisely the resolution of the patch being generated, in which case the value of $\hat{uv}$ is calculated just as the current normalised offset in the patch.

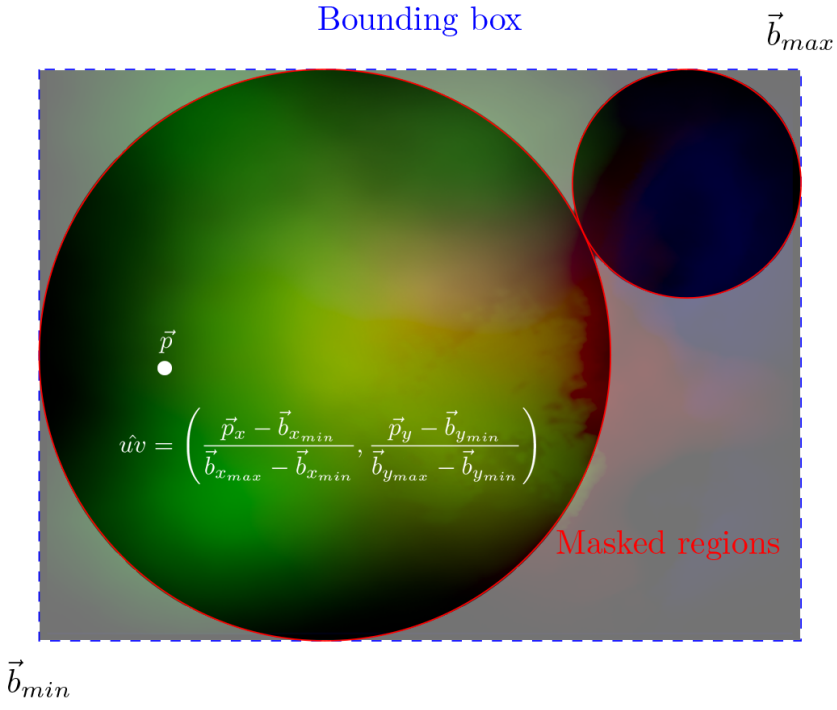


Figure 4.25: Example of vector field displacement map usage with source image (left) and result (right).

4.6.2 Terracing modifier simple

Terracing is the shaping of a terrain so that it has a stepped gradient rather than a smooth one. This is often seen in agriculture, particularly in the rice fields of China where human created terraces can span many kilometres.

Figure 4.26: Longji Terraced Rice Fields in China. Image courtesy: Dav Wong

The effect can also be seen in parts of naturally formed terrain such as on mountains, although it is far more subtle and less uniform. To recreate this range of effects we want the terracing modifier to have parameters for the height of the terrace platforms and their sharpness. The non-uniformity of natural terracing can be recreated using the selector rules available, for example by using a noise function limited to steep slopes.

The most basic form of an algorithm to implement terracing would simply be to round all values to nearest multiple of the terrace height producing a stair-like effect, $h_out = round(h_{in}/h_t) * h_t$, where h_t is the terracing height. To make the result more realistic and the modifier more expressive, we need to consider a sharpness value, $s \in [0, 1]$, and how far each value is from the nearest platform.

To find how far a value is from the nearest platform the initial thought would be to use the modulo function, however alone it is not sufficient due to the HLSL implementation using truncated division in the function's internals. This means that using $fmod(h_{in}, h_t)$ would return the signed remainder when h_{in} is divided by h_t. That is, $r = h_{in} - h_t * trunc(\frac{h_{in}}{h_t})$. Instead we want to use floored division, as promoted by Knuth[18], so that $r = h_{in} - h_t * \left\lfloor \frac{h_{in}}{h_t} \right\rfloor$. This gives the desired result of finding how far the value is to the nearest platform below.

Truncated and floored division are shown in 4.27.

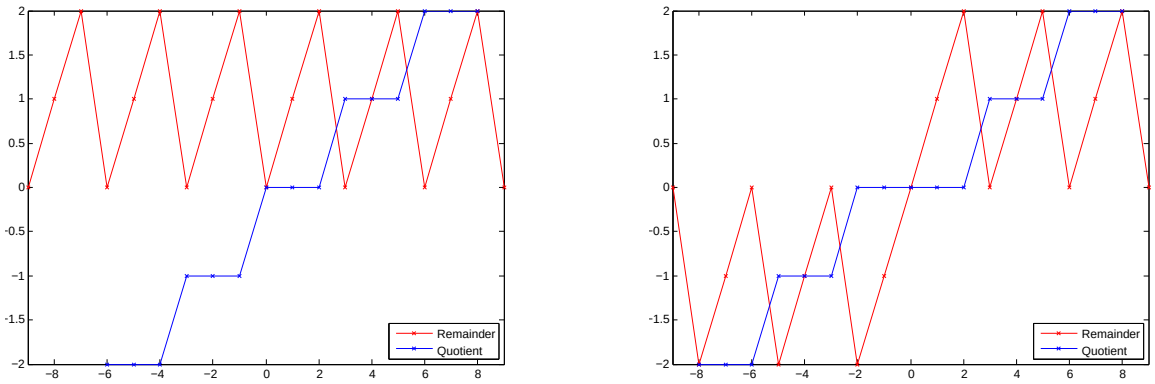


Figure 4.27: Behaviour of default HLSL fmod function (left) and desired behaviour using floored division (right)

We now consider how to use this value to soften the regions the between terracing platforms. We want the staircase effect when the sharpness is set to 1.0 and the original elevation when then sharpness is set to 0.0. This can be realised by finding a sharpness threshold, $\theta_s = s * t_h + r$. If $h_{in} < \theta_s$, then return the terracing height below $h_{out} = h_{in} - r$. Otherwise, we need to calculate a blend b of t_h based on the height above θ_s as a fraction of the maximum height above $theta_s$, that is, the height of the platform above. We add this blend amount to the height of the lower platform as the result. There is one more part to the equation however, as we need to be able to account for partial application of the modifier due to masks with values in (0,1). So the final result is to linearly interpolate between the input value and the calculated value. The complete process algorithmically is as follows (note this is given in the most readable form rather than the minimal form, the HLSL implementation uses clamps to remove branching and removes redundant statements):

$$r = h_{in} - h_t * \left\lfloor \frac{h_{in}}{h_t} \right\rfloor \tag{4.9}$$

$$h_b = h_{in} - r \tag{4.10}$$

$$h_a = h_b - h_t \tag{4.11}$$

$$\theta_s = h_t * s + h_b \tag{4.12}$$

$$b = \begin{cases} 0 & \text{if } h_{in} <= \theta_s \\ \dfrac{h_t(h_{in} - \theta_s)}{h_a - \theta_s} & \text{otherwise.} \end{cases} \tag{4.13}$$

$$h_{out} = lerp(h_{in}, h_b + b, m) \qquad \text{where m is the mask value} \tag{4.14}$$

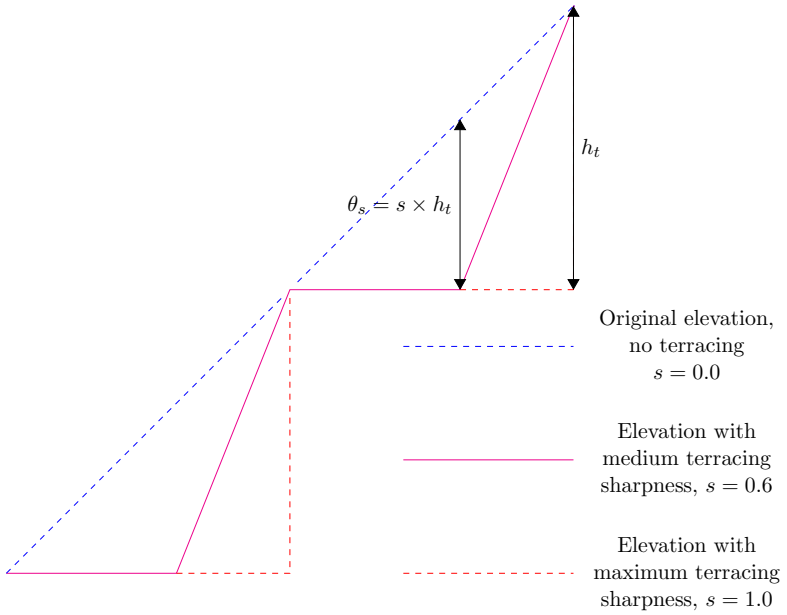


Figure 4.28: Diagram of terracing process

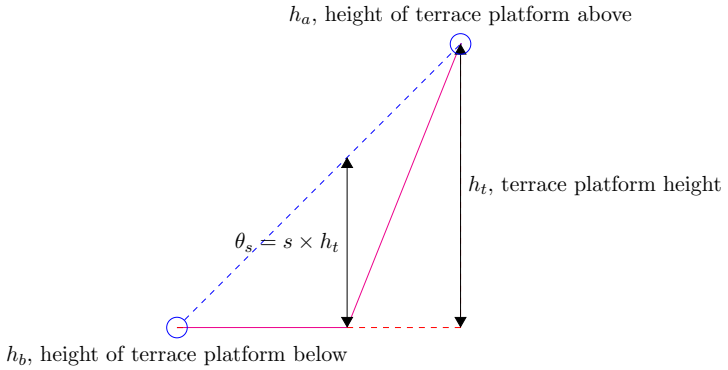


Figure 4.29: Variable visualisation for terracing algorithm

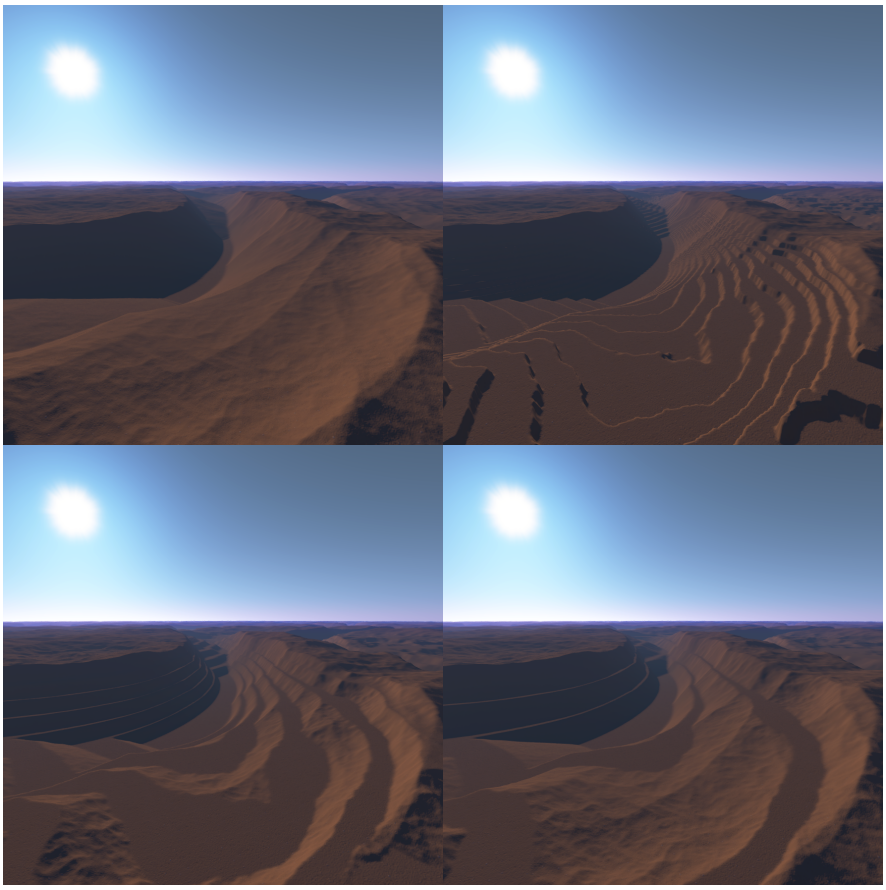


Figure 4.30: Example use of the terracing modifier. No terracing (top left), small stepped terracing with low blend (top right), high stepped terracing with high blend (bottom left and right)

4.6.3 Terracing modifier advanced

The previously discussed terracing algorithm works well but the flattened areas are perfectly flat as can be seen in 4.26. This is suitable for man-made terracing in agriculture but we need something smoother to create even more subtle effects than the sharpness control allows. The idea for the advanced terracing modifier is to have a new parameter, flatness relief factor $f \in [0, 1]$, which creates a more gradual rise to the platform edges.

The algorithm works mainly the same way, but in the step to calculate b, instead of setting the value to either zero or the platform edge height we set it to the maximum of the relief height and the platform edge height and clamp the result. The relief height is calculated as the fraction of the platform height made by the remainder multiplied by the platform height scaled by the flatness relief factor. So the algorithm becomes:

$$r = h_{in} - h_t * \left\lfloor \frac{h_{in}}{h_t} \right\rfloor \tag{4.15}$$

$$h_b = h_{in} - r \tag{4.16}$$

$$\theta_s = h_t * s + h_b \tag{4.17}$$

$$b = max((h_t * f) * (r/h_t), \frac{h_t(h_{in} - \theta_s)}{h_t * (1.0 - s)}) \tag{4.18}$$

$$h_{out} = lerp(h_{in}, h_b + b, m) \qquad \text{where m is the mask value} \tag{4.19}$$

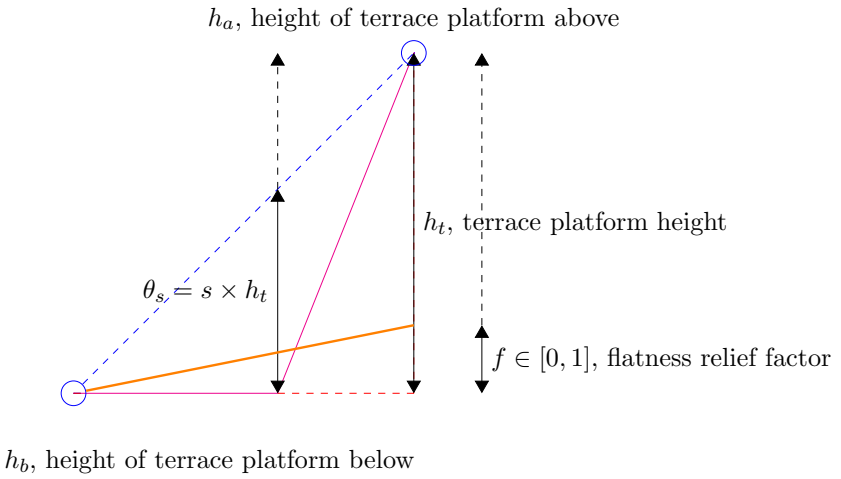


Figure 4.31: Diagram and variables for terracing with flatness relief

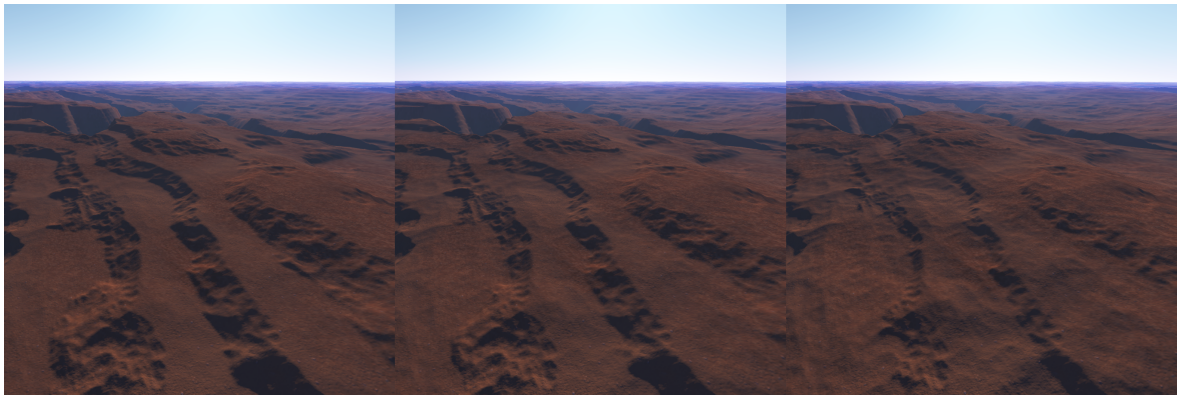


Figure 4.32: Terracing with flatness relief enabled with increasing values from left to right.

4.6.4 Surface modifier

The surface modifier changes one of the three surface properties depending on mode of operation - low frequency texture, high frequency material and billboarded flora texture.

4.6.4.1 Low frequency texture

In this mode of operation, the user can apply a constant colouring or use a ramp file to recolour the low frequency texture property of the terrain. When using a constant value, the mask determines the intensity of the new colour. For example, pure red (1.0,0.0,0.0) would be scaled between (0.0,0.0,0.0) and (1.0,0.0,0.0) depending on m. When using a ramp source file, the mask determines the texture coordinate for sampling the file loaded as a DirectX texture resource. This loading is done at compile time, since the ramp files are only 1D they are not of concern for memory usage and remain in GPU memory for the duration of the terrain being active. The UV coordinates to sample from the 1D ramp texture are $\hat{uv} = (0, m)$, assuming a vertical ramp texture is provided.

The final step, having calculated the colour value for the constant or ramp source, is to alter the existing parameter value for colour. The surface modifier has three options for performing this. Either to linearly interpolate the colour with the new colour, add the new colour to the old one and scale it so that no component exceeds 1.0, or to take the average colour. An example of the results obtained by these options is shown in 4.33. Precisely, we calculate these results as follows for a mask value m, existing colour $\vec{c}$ and new colour $\vec{c'}$:

- Linearly interpolated: $\vec{c} = lerp(\vec{c}, \vec{c'}, m)$
- Additive colour: let $\vec{t} = \vec{c} + (m\vec{c'})$, $\vec{c} = \frac{t}{max(t_r, t_g, t_b)}$
- Average colour: $\vec{c} = lerp(\frac{\vec{c}+\vec{c'}}{2}, \vec{c}, m)$

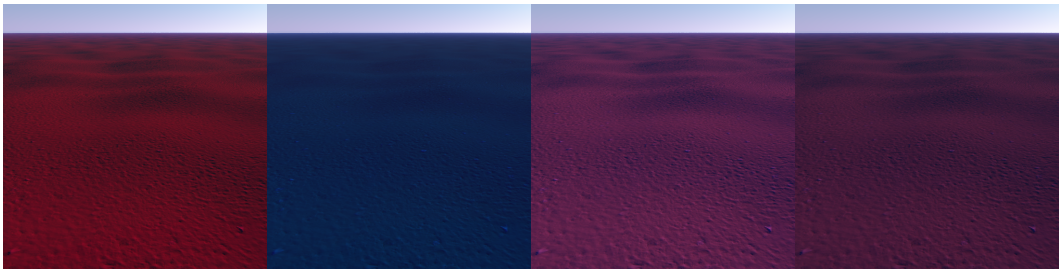


Figure 4.33: The original red terrain is modified by adding a blue surface modifier. In the replace mode, the terrain becomes blue. In the additive mode, the terrain becomes bright purple. In the average mode, the terrain becomes a darker purple.

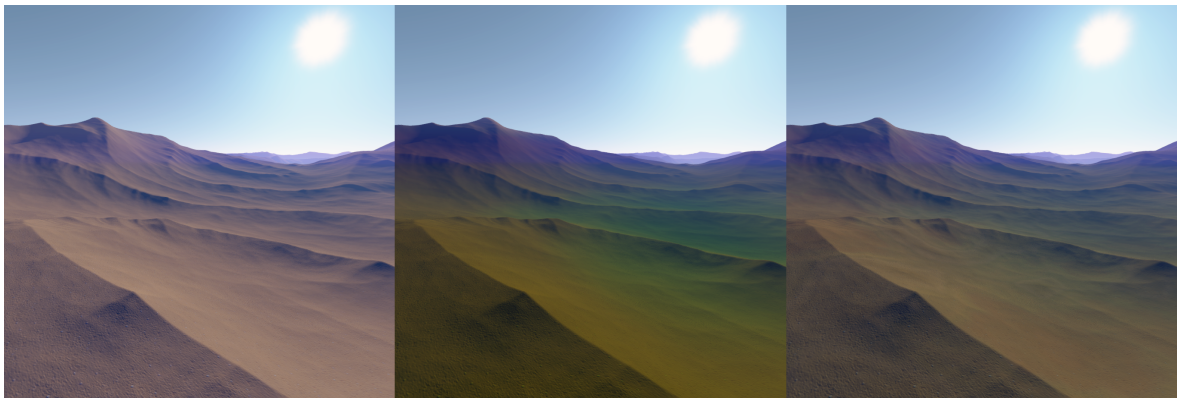


Figure 4.34: Progression of applying low frequency texturing to vastly improve the visual quality of a simple terrain. The left image shows the starting result, the middle image shows the result after adding a ramp to a height selector (4.5.6) mask result, the final image shows the result after adding another colour ramp based on a noise mask parametrised with a fractal using the additive mode.

4.6.4.2 High frequency material

Although the current texturing solution (for details on the texturing system see 5.2) uses blend maps, they are not explicitly calculated during the main generation process. Instead we store a material ID at each position and process this into a blend map later on. This makes the modifier algorithm for changing high frequency materials very simple. The user parametrises the modifier with a threshold value θ and a material from the material palette with ID t. If the current mask value m is less than θ, then the current ID $v = t$, else v is unchanged. We implemented the system this way due to a distaste of the way textures are often blended in other solutions such as in figure 4.35, with smooth gradients between completely different textures. We feel this looks entirely unrealistic and harder edges are more desirable. An example of our texturing results can be seen in 4.36.

Figure 4.35: Traditional texture splatting with a change in textures over a long distance - not realistic.

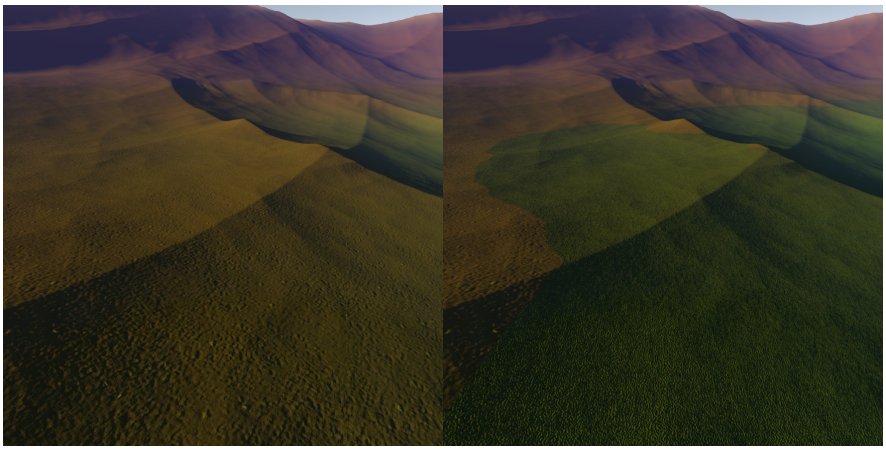


Figure 4.36: Texturing results using our system. A grass material has been used to replace the sand material using a height selector mask. We feel this looks better than the traditional splatting technique. Due to our texturing system some natural blurring occurs in the distance but distinction between materials is kept close up. Having billboarded flora helps improve the result further by partially obscuring the transition - see figure 4.37

4.6.4.3 Billboarded flora texture

The flora billboarding mode works as per the high frequency material option. The user parametrises the modifier with a threshold value θ and a flora texture from the flora palette with ID t. If the current mask value m is less than θ, then the current ID $v = t$, else v is unchanged.

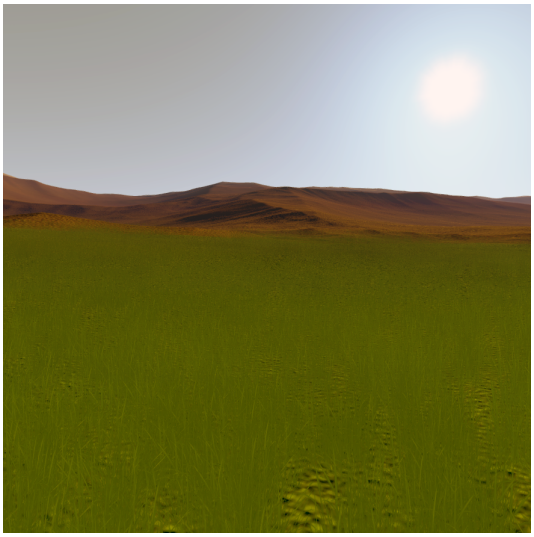


Figure 4.37: Additional surface modifier added to the rule-set that generated the scene in 4.36. The modifier adds a grass billboard flora to the same region as the grass material was applied.

4.7 Shader compilation, structure and workings

In this section we explain the compilation process while at the same time introducing how the generation is designed for execution on a compute shader. We do not give every detail of this process and look to provide more of a summary.

The compilation algorithm works by recursively stepping through the CPU side implementation of the system. The CPU implementation stores the data and structure of the terrain and also provides a full implementation of the execution process which we use for comparison with the GPU implementation in our evaluation (7).

The process operates by compiling all rules for a terrain at once. We discuss some other possibilities in our future work suggestions (8.1) including on the fly compilation but this proved sufficient for our current needs. One of the particularly nice benefits of compiling our rules on the GPU is that we get optimisation by the compiler on the actual parametrisations. A typical CPU implementation would not give such a benefit. The process is as follows:

1. The main GPU generation class calls the shader compilation function of the terrain. It passes as parameters the desired thread size and dispatch size for the compute shaders (more detail shortly).

2. The template file for rule shaders is loaded into a string. We include the source for this template in listing 4.3.

3. For each top level rule-set in the terrain, we recursively get the local and global variable declarations required for the rules contained in the rule-set and its children. We also get information about the external resource requirements of the rules. For example a noise selector backed by a heightmap source specifies either having a streamed or static 2D texture resource. Low frequency surface information supplied by a colour ramp is another example, in this case a static 1D texture is specified. This information is stored so that the parameters can be initialised and then set before each execution of the shader.

4. The template file has a marker for local and global variable declarations, these markers are replaced by the actual definitions found in the previous step.

5. We recurse through the rule-sets again, this time generating the actual rule code for the shader. Each rule has an interface that provides HLSL code for this and the previous stage for getting declarations. The process for the code generation is:

(a) Print the mask variable for this rule, set it to the default value configured in the rule-set.

(b) Print the output for the selectors in the maximising mode, adjusting the result of the current rule-set's mask variable. For example,

```
1     mask_1 = max(mask_1, maskModifySmooth(selectorCircle(pos.xy,
      float2(0, 0), 2500, 1.00)));
```

(c) Print the output for the selectors in the minimising mode, adjusting the result of the current rule-set's mask variable.

(d) Apply rule-set level mask modifiers

(e) Bound the rule by the parent mask, if one exists, by taking the minimum of the two masks:

```
1     mask_1 = min(mask_0, mask_1);
```

(f) Print modifier rules. Modifiers do not return results instead taking the values to be modified as inout parameters. For example, the terracing modifier operates on the heights store:

```
1       modifierTerracing(heights[GroupIndex].y, 10, 0.5, mask_1);
```

(g) Repeat for all child rules, passing the mask variable parameter of this rule as the bounding mask variable.

6. The rule content marker is replaced with the generated code.

7. We are done. Compile the rule with DirectX and store the result ready for execution.

We mentioned in the compilation process the thread size and dispatch size for compute shaders. Compute shaders uniquely give direct control over how threading is performed on the GPU. Figure 4.38 illustrates the threading system. We are limited to 32x32 total threads but our patch size will typically be larger than this. So multiple groups of the chosen thread count will be sent. The number of these groups is the dispatch size. And the thread size multiplied by the patch size will give the total number of threads executed in the shader for a single dispatch call.

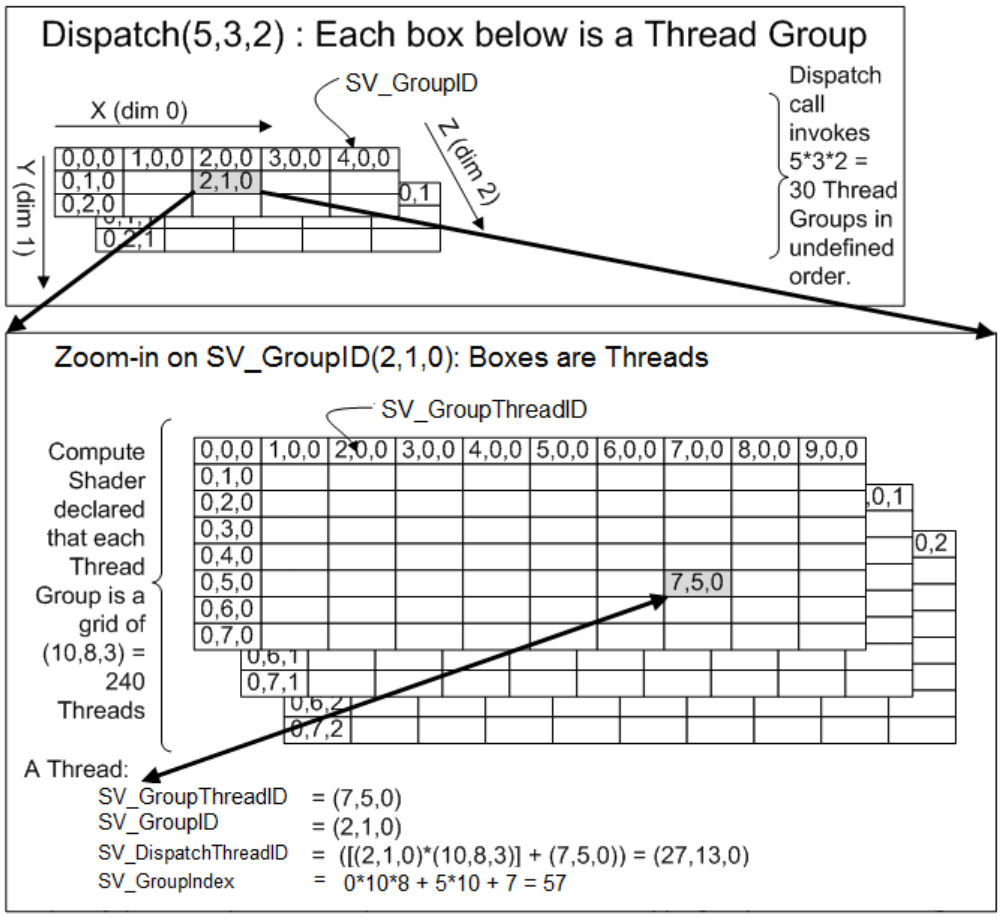


Figure 4.38: Diagram showing threading system for dispatch calls. Image courtesy Microsoft.

We want to calculate patches that are padded by two vertices either side so we have some adjacency information for adaptive tessellation (used in 5.1) and normal calculations. For our typical patch size of 64 this means padding to 69 vertices per patch. With 69x69 vertices the best thread size we can use is 23x23, using a dispatch size of 9 to cover all vertices in the complete patch. We illustrate these ideas in figure 4.39. However, this splitting introduces another problem which is that our GPU generation requires normal calculations for certain modifiers to be executed. To get around this we introduce additional padding in the shader

for each *group* of threads. This can be seen in the groupedshared memory array *heights* in listing 4.3. Memory declared as groupedshared allows for synchronisation barriers across groups, but not across dispatches. We call the synchronisation function GroupMemoryBarrierWithGroupSync() each time before performing a normal calculation so that all heights in the array are at the same value. This gives us correct normal values even when subdividing the patch.

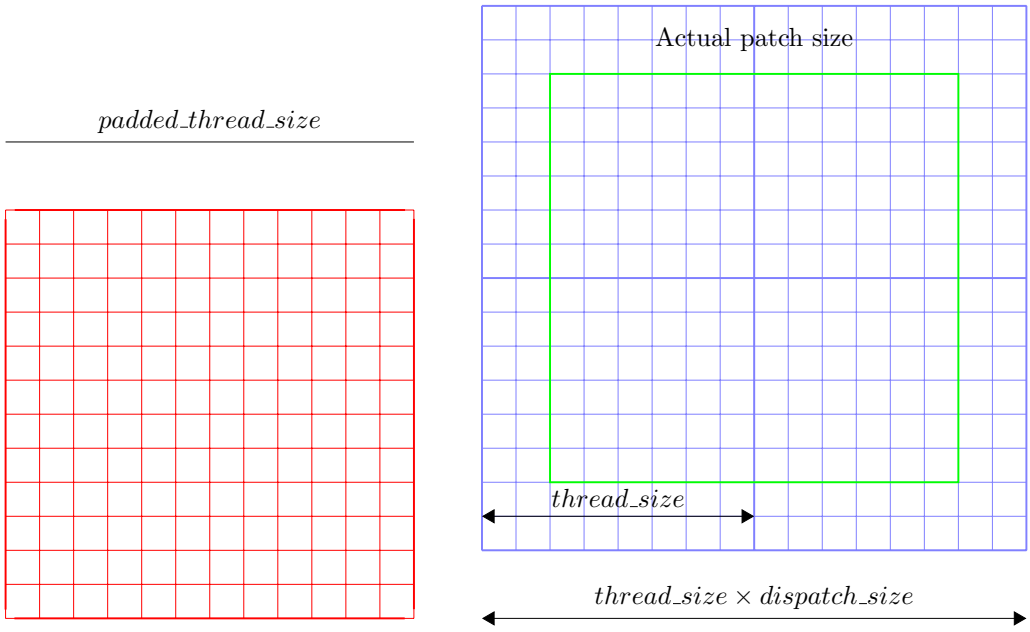


Figure 4.39: Thread size and dispatch size for patch compute executions

```
#include "FBM.h"
#include "ComputeHeader.h"
#include "Rules.h"

#define thread_size_padded (thread_size + 4)

// shared memory for heights, so we can synchronise for normal calculations
groupshared float3 heights[thread_size_padded * thread_size_padded];

SamplerState LinearSampler : register( s0 );

RULE_RESOURCES_MARKER

[numthreads(thread_size_padded, thread_size_padded, 1)]
void main( uint3 GroupID : SV_GroupID, uint3 DispatchThreadID :
        SV_DispatchThreadID, uint3 GroupThreadID : SV_GroupThreadID, uint
        GroupIndex : SV_GroupIndex )
```

```
{
  TerrainData data = (TerrainData)0;

  // get position in patch
  uint2 posPatch = (GroupThreadID + (GroupID * uint3(thread_size, thread_size,
       1))).xy - uint2(2, 2);

  // get world coordinates
  float2 pos = float2(posPatch.x * PatchInfo.z, posPatch.y * PatchInfo.z) +
       PatchInfo.xy;
  uint offset = posPatch.x + posPatch.y * (dispatch_size * thread_size);

  // load the current values (no need to check range)
  heights[GroupIndex] = InputBuffer[offset].Height;
  data.Influence = InputBuffer[offset].Influence;
  data.Normal = InputBuffer[offset].Normal;
  data.Diffuse = InputBuffer[offset].Diffuse;
  data.TextureID = InputBuffer[offset].TextureID;

  float2 polyVertices[MAX_POL_SIZE];
  RULE_DECLARATION_MARKER
  RULE_CONTENT_MARKER

  // recalculate normals
  GroupMemoryBarrierWithGroupSync();
  data.Normal = calculateNormal(GroupThreadID.x, GroupThreadID.y);

  // set output buffer values if we're on a valid thread not for padding
  if ( ( GroupThreadID.x > 1 ) && ( GroupThreadID.x < thread_size_padded - 2 )
       && ( GroupThreadID.y > 1 ) && ( GroupThreadID.y < thread_size_padded - 2
        ) )
  {
    OutputBuffer[offset].Height = heights[GroupIndex];
    OutputBuffer[offset].Influence = data.Influence;
    OutputBuffer[offset].Normal = data.Normal;
    OutputBuffer[offset].Diffuse = data.Diffuse;
    OutputBuffer[offset].TextureID = data.TextureID;
  }
}
```

Listing 4.1: Rule-set computer shader template

4.7.1 Rule optimisations

We spent a considerable amount of time trying to find the best implementations possible for our rules. We used the same performance measurement techniques as in our evaluation (7.1.1.1) when comparing implementations. We also used an HLSL disassembler to look at the number of instructions being generated - less instructions did not always correlate to better performance but it is interesting to see how implementation decisions can impact the final assembly code. Typically this level of scrutiny is not given to HLSL shaders, but since we will be getting many millions of executions of our rules, we wanted them performing as well as possible.

One optimisation we often found beneficial was removing branching operations. Branching used to be considered very bad for shaders because both paths would always be executed, now it can be beneficial in some cases since the branching is dynamic so avoiding large blocks of code is a performance win. However, avoiding small branches can still help and we were often able to replace an if/else statement by instead clamping a result or using bit twiddling. For example in the polygon mask selector, replacing the double nested if statements in the parity check loop gave around a 50% performance increase which was quite unexpected.

```
1  bool parity = false;
2  float2 last = vertices[polyCount - 1];
3
4  for (int i = 0; i < polyCount; ++i)
5  {
6    if ((vertices[i].y <= y && y < last.y) || (last.y <= y && y < vertices[i].y))
7    {
8      if ( (y - vertices[i].y) * (last.x - vertices[i].x) / (last.y - vertices[i
         ].y) + vertices[i].x > (double)x )
9      {
10       parity = !parity;
11     }
12   }
13   last = vertices[i];
14 }
15
16 if (!parity)
17 {
18   return 0.0;
19 }
```

Listing 4.2: Original parity check implementation

```
uint parity = 1;
for (i = 0; i < polyCount; ++i)
{
  float2 current = vertices[i];
  int test = ((current.y <= y && y < last.y) || (last.y <= y && y < current.y))
       ;
  int test2 = (y - current.y) * (last.x - current.x) / (last.y - current.y) +
       current.x > x;
  parity += (test2 * test);
  last = current;
}

if (parity % 2)
{
  return 0.0;
}
```

Listing 4.3: Improved parity check implementation giving around 50% increased performance by removing branching

4.8 Serialisation

Since one of the aims for the project was to deliver a fully working editing environment for terrains rather than just a hardcoded demonstration, we needed a system for serialisation. A binary format was chosen for saving as there was no need to make it easily readable by users. The structure is straightforward:

1. Terrain header - stores a terrain's name, size, whether it is planar or spherical and the atmospheric scattering properties including data driven parameters (6.2). The header also stores the number of each entries each palette (4.3) contains and the number of top level rule-sets so they can be read next.

2. Noise palette - each palette has a serialisation and loading function to save and restore its elements. The various noise types are identified by unique identifiers which determine how subsequent data is read.

3. Material palette - stores the list of textures referenced by a material

4. Flora palette - stores the diffuse texture reference

5. Rule-sets - rule-sets and rules also have serialisation and loading functions. Since rule content is dynamic, we store the rule type as a unique identifier and store the number

of rules and child rule-sets in the record header for rule-sets. So the flow would be as follows: read a rule-set header, read an identifier to determine the type of the first rule, pass the file to the loader for that rule type, read the next rule identifier and so on until all rules are read.

5 Graphics back-end

In this chapter we look at the supporting technology employed to facilitate efficient and uninhibited rendering of the terrains generated by our system. We cover the most interesting topics only - many other components make up the entire graphics back-end as we talked discussed in 3.2.

5.1 HWCDLOD and HWACDLOD

In this topic we present our modifications to the CDLOD algorithm to support hardware tessellation and adaptive hardware tessellation. We call these two schemes HWCDLOD and HWACDLOD respectively. We follow the early parts of original paper[41] for implementing node selection and calculation of morph regions, adjusted slightly for our system as necessary. This gets us to the point where we have a list of quadtree nodes ready to render using our system for reduced draw calls (5.4). We also have the means to calculate by how much each region of a patch should be blended towards the appearance of the patch of a lower level of detail. In the original CDLOD implementation, this blending works by adjusting the position of vertices in the patch via calculations in the vertex shader. This is done by moving odd vertices (defined as (i, j) where i or j is odd), towards even vertices as the morph factor increases from zero to one.

This system works well enough, but we thought there was room for improvement given the latest hardware (the original paper was from 2010). The idea is to use the hardware tessellator to morph between patches. The approach to this is the opposite of CDLOD - rather than simulating the removal of detail by moving vertices, we instead actually add detail at the low morph regions. To achieve this we use quad control patches with fractional even partitioning and adjust the tessellation factor between 2 on the fully morphed vertices and 4 on the untouched vertices. We note that by using this scheme, we get a crack free tessellation pattern as in CDLOD but by default we are introducing tessellation by a factor of 4 and thus increasing overall detail by a factor of 16 - this is not what we want. So with hardware tessellation enabled, we reduce the detail of the index pattern generation by a factor of 16 to compensate. This gives us the same overall detail as with the original CDLOD implementation. The benefit of our system, is that it actually reduces the amount of geometry rendered. We explore how much of a benefit this actually gives in our evaluation (7).

```
HS_CONSTANT_DATA_OUTPUT MorphConstantHS( InputPatch<HS_CONTROL_POINT_IN_OUT, 4>
        ip, uint PatchID : SV_PrimitiveID )
{
    HS_CONSTANT_DATA_OUTPUT output;

  // calculate average morph factor for edges
  output.Edges[0] = 4.0f - 2.0 * ((ip[0].Morph + ip[1].Morph) * 0.5);
  output.Edges[1] = 4.0f - 2.0 * ((ip[0].Morph + ip[3].Morph) * 0.5);
  output.Edges[2] = 4.0f - 2.0 * ((ip[3].Morph + ip[2].Morph) * 0.5);
  output.Edges[3] = 4.0f - 2.0 * ((ip[1].Morph + ip[2].Morph) * 0.5);

  output.Inside[0] = ( output.Edges[0] + output.Edges[2] ) / 2.0f;
  output.Inside[1] = ( output.Edges[1] + output.Edges[3] ) / 2.0f;

    return output;
}
[domain("quad")]
[partitioning("fractional_even")]
[outputcontrolpoints(4)]
[outputtopology("triangle_cw")]
[patchconstantfunc("MorphConstantHS")]
HS_CONTROL_POINT_IN_OUT HS( InputPatch<HS_CONTROL_POINT_IN_OUT, 4> ip, uint i :
        SV_OutputControlPointID, uint PatchID : SV_PrimitiveID )
{
  ...
}
```

Listing 5.1: Tessellator stage function for HWCDLOD

Listing 5.1 shows the code for the HWCDLOD tessellator stage and partial code for the hull shader. The tessellator code looks at the average morph factor along the edge of each quad then sets the blend factors for the patch accordingly. Figure 5.1 shows examples of the HWCDLOD scheme in action with increasing base patch size.

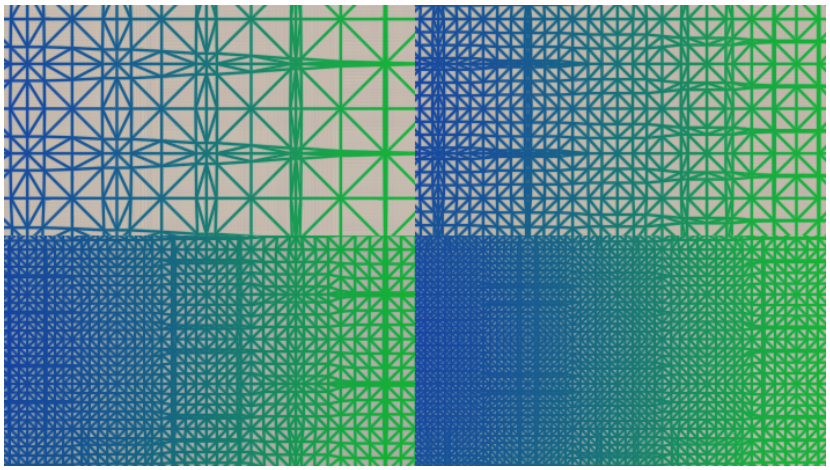


Figure 5.1: Wireframe view of the tessellation pattern with HWCDLOD

The scheme in listing 5.1 has an interesting property. Any multiple producing an integer result of the 4/2 scheme will also produce a crack free tessellation - this is a perfect basis for performing adaptive tessellation. In adaptive tessellation, the patch level of detail is varied according to some factor. As a proof of concept for this system, we implemented a distance based adaptive scheme that we call HWACDLOD. It increases detail in a patch as distance between vertices increases. This gives more detail on steep slopes and reduces unnecessary geometry on flat ground. In our implementation we allow up to a factor of 16 for tessellation level. To compensate for this, we further reduce the indexing pattern detail by another factor of 16. We show an example of the tessellation pattern using this scheme in 5.2.

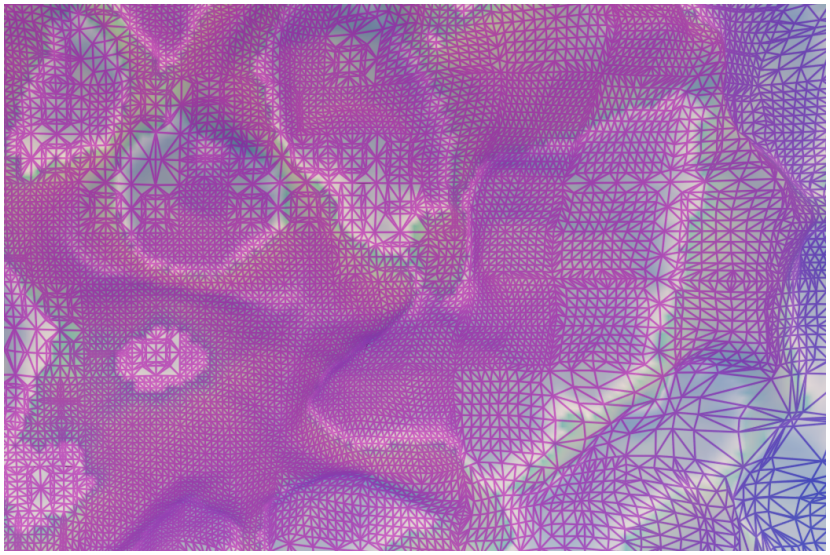


Figure 5.2: Wireframe view of the tessellation pattern with HWACDLOD

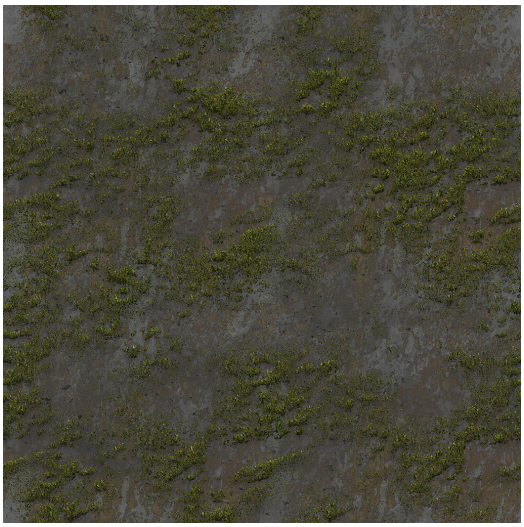
5.2 Unlimited texturing via patch rasterisation

The standard way to apply texturing on the surface of terrains is to use a technique called texture splatting. In this technique, 2D textures are generated such that each channel represents the level of influence for a particular texture in that area. An attempt is sometimes made to reduce the number of splat textures required to represent the desired number of texture blends, by dividing each channel such that a certain range in that channel represents a certain texture instead of the entire channel. This means reduced precision in return for less sampling and memory usage. In many applications, this technique can work quite well. It is particularly suited to terrains that are either small or have a low number of total textures and so requires only, say, a single 4-channel splat texture. An example of the process can be seen in 5.3.
The downfall of splatting is that it requires a per-pixel sample of each splat map and then a per-pixel sample of each texture type for the number of blends represented by the splatting maps. With our terrain system, it would additionally be desirable to sample from the blend maps of the parent patch to smoothly blend in new details as we morph between patch LODs.

The overall goal of the graphics back-end created in the project is to prevent any artificial restrictions on the expressiveness of the terrain rule system. For each material, our current texture system has a potentially filled slot for diffuse, specular, emissive, height and normal textures. With, for example, two blend textures for splatting encoding the influence of eight textures, this would mean $8 \times 5 \times 2 = 80$ texture samples in the pixel shader for texturing alone for 2D texturing. For triplanar texturing this is increased again by three times to 240 samples per pixel. Although a modern GPU may be able to render this in real time, it would be far too expensive just for rendering a terrain. In addition to this, eight blends may not be enough when considering a low LOD patch covering a large portion of a planet which could easily contain more than eight textures.

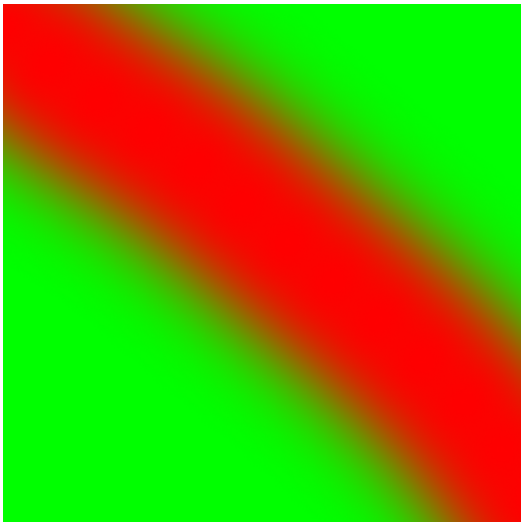
One common approach to help improve the performance of texture splatting is to use shader permutations so that each patch uses a shader that only samples the number of textures it requires. So for example patches that only use two textures would use a shader that only samples from two textures and so on. This would still not be good enough for our system and would prohibit the use of our draw call reduction technique (5.4) by requiring many different shader switches to render different patches.

In coming to a solution, the key observation in the process is that for any patch, the sampling always produces the same result (if we ignore differing sampling levels due to mipmapping). This means the result can be precomputed. By adding in a rasterisation stage to the gen-



(a) Source texture A

(b) Source texture B



(c) Blend map

(d) Result

Figure 5.3: Two source textures (a) and (b) are blended by using texture splatting (c) to produce the result in (d). In (c), the red channel represents the contribution of texture (b) and the green channel the contribution of texture (a).

eration pipeline, for each texture type we support we can pre-bake the result onto a texture using a blend map and then dispose of that map. Since it is a one-time operation, using expensive sampling techniques such as triplanar mapping with many textures is acceptable. We can also use shader permutations in the rasteriser itself so that each baking operation is as inexpensive as possible. This solves the problem, but has two downsides:

- Triplanar mapping will still work to improve the problem of texture stretching, but resolution will be compressed in those areas and will be visibily lower quality if the camera is close enough.
- The GPU memory requirement is vastly increased as every baked texture map type needs to be considerably larger than the blend maps it replaces.

We have ignored the first problem for this project but we do briefly suggest an option for how it could be solved. One option would be to store a UV mapping for each patch that evenly distributes the baked texture across the surface rather than mapping in a regular grid as we do in the current implementation. This would work, probably quite effectively, but it does add yet another memory cost for storing the unique UVs per patch. For the second problem, the obvious choice to reduce video memory usage is to use compression. GPUs natively support a variety of compressed texture formats. The biggest issue is compressing the texture in real time. We look at how to do this in 5.3.

The shader code to perform the rasterisation is straightfoward but quite long due to all the texture sampling involved so we omit including it in full. The process involved is:

1. Bind texture array inputs for the texture types used (we store these in a texture array for simplicity, it saves having permutations for number of inputs or having more input slots than necessary).
2. Bind the blend texture sources.
3. Bind the render target texture. Rendering is done either to a temporary texture if using BCn compression or directly to the target texture array if not.
4. Draw a 2D quad that covers screenspace, sampling from coordinates configured to match that of the target patch using splatting.
5. We are either done if not using BCn, if we are using BCn then the result is compressed before then being copied to the array for its texture type.

Figure 5.4 shows a partial view of two of the final texture array targets for rasterised diffuse and normal textures.

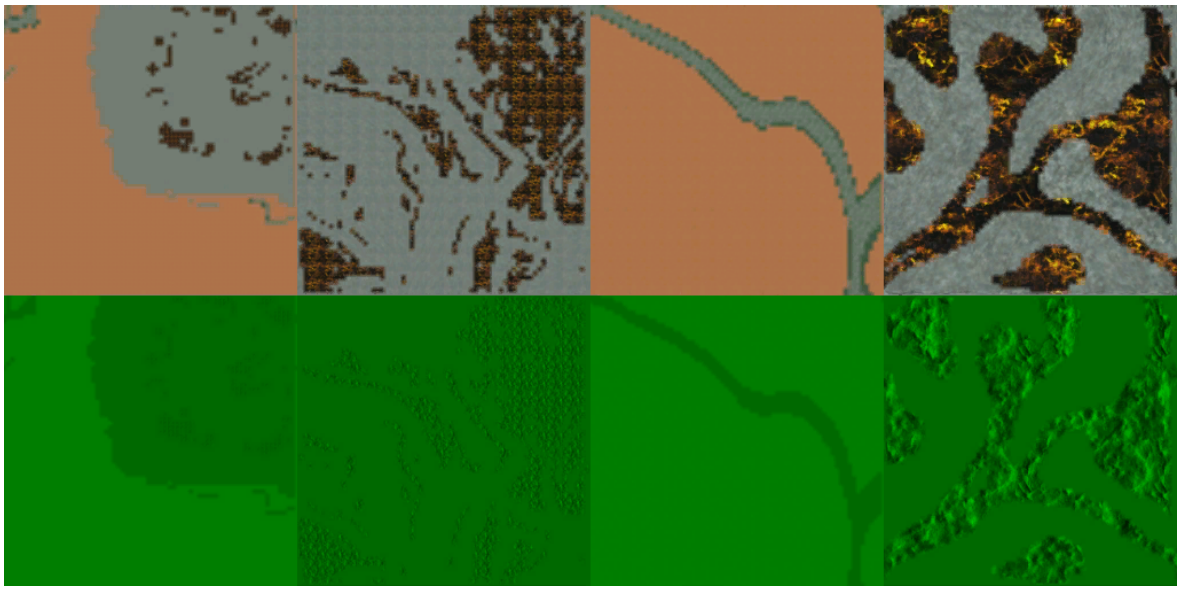


Figure 5.4: Debug view of patches in BC1/BC3 texture arrays (5.4) for rasterised diffuse and normal textures. Note the normal texture is still in its swizzled state (5.3) so appears only green.

5.3 Real-time BCn compression using compute shaders

There are two main candidates for texture compression, BC1 and BC3, also known as DXT1 and DXT5. Using these compression methods gives the following benefits:

- Reduced memory usage. For an R8G8B8 texture BC1 will reduce storage requirements by a factor of 6 and by a factor of 8 for R8G8B8A8. BC3 stores extra information for an alpha channel, and reduces R8G8B8 memory usage by a factor of 3 and R8G8B8A8 by a factor of 4.
- Textures remain compressed in L1 cache, giving better cache performance and so better overall performance
- BC1 performs better than BC3 due to compressing into a smaller space
- BC3 in its swizzled form is suitable for normal maps

We also looked into the more recent BC7 which gives higher quality results, but after some initial tests, despite using compute shaders the implementation was far too slow to be of any consideration for use in the rasteriser stage (one or two compressions per second when we have thousands of patches to be processed). For BC1 and BC3 however, work has been done to derive methods that allow for real-time compression. Much of this work uses the CPU, often with vector operations to maximise performance. There have also been some GPU-based

solutions proposed and implemented but these are surprisingly few in number considering the ubiquity of the compression formats across real-time graphics. We base our solution off the only two references we found which performance generation in the pixel shader rather than a compute shader. Our BC1 compression is based on [44] and our BC5 compression on [46].

BC1 works by compressing 16 pixels into 8 bytes. It has a mode for alpha transparency but this is not of interest to us. In the mode we want to use, the first 32 bytes are used to store two colours in an R5G6B5 format, in our implementation we take the minimum and maximum pixel values for the current 4x4 block. From this, two other colours are implicitly generated when the texture is used in sampling with values of $\frac{2}{3}c_0 + \frac{1}{3}c_1$ and $\frac{1}{3}c_0 + \frac{2}{3}c_1$. The remaining 32 bits of the DXT1 are used to provide 2-bit indices for the 16 original pixels into these four colours. Using an R16G16B16A16_UINT render target, we can easily create a compute shader to perform this function and then copy the result to a BC1 texture (this is permitted by DirectX).

BC3 does exactly the same as BC1, but additionally stores alpha values and indices to those values for each of the 16 pixels in the extra 8-bytes used by the format. We can exploit this extra channel to get higher precision normal maps. Normal maps, as the name implies, are normalised. So the third channel can be restored if only two are stored. The highest precision channels in a BC3 texture are the alpha texture and the green texture. So if we store the red and green channels (the most varying channels in normal maps, the blue value tends to be high and less variable), we can store at a better precision than storing all three channels because our interpolation and min/max value storage is based on only a single channel in each case. This gives far better results for normal compression than BC1 and is commonly called swizzled DXT5 or BC3.

The BC1 compression function shader is fairly short so we include it here as an example in listing 5.2. The BC3 version can be found in our source code.

```
1 SamplerState            LinearSampler       : register( s0 );
2
3 Texture2D<float4>         InputTexture      : register( t0 );
4 RWTexture2D<uint4>          DXTResult       : register( u0 );
5
6 [numthreads(thread_size, thread_size, 1)]
7 void main( uint3 GroupID : SV_GroupID, uint3 DispatchThreadID :
        SV_DispatchThreadID, uint3 GroupThreadID : SV_GroupThreadID, uint
        GroupIndex : SV_GroupIndex )
8 {
9   float2 texel_size = (1.0f / texture_size);
```

```
10  float2 uv = (DispatchThreadID.xy / float2(texture_size * 0.25, texture_size *
        0.25));
11
12  uint i = 0;
13  float3 samples[16];
14  float3 min_color = float3(1.0, 1.0, 1.0);
15  float3 max_color = float3(0.0, 0.0, 0.0);
16
17  for (i = 0; i < 4; i++) {
18    for (uint j = 0; j < 4; j++) {
19      samples[i*4+j] = InputTexture.SampleLevel(LinearSampler, uv + (float2(j,
          i) * texel_size) + float2(0.5 / texture_size, 0.5 / texture_size), 0).
          rgb;
20      min_color = min(min_color, samples[i*4+j]);
21      max_color = max(max_color, samples[i*4+j]);
22    }
23  }
24
25  uint3 color_0 = min_color*255;
26  color_0 = color_0 / uint3(8, 4, 8);
27  uint color_0_565 = dot(color_0, float3(2048, 32, 1));
28  uint3 color_1 = max_color*255;
29  color_1 = color_1 / uint3(8, 4, 8);
30  uint color_1_565 = dot(color_1, float3(2048, 32, 1));
31
32  float3 endpoints[2];
33  if(color_0_565 == color_1_565)
34  {
35    uint4 dxt_block;
36    dxt_block.r = color_0_565+1;
37    dxt_block.g = color_0_565;
38    dxt_block.b = dxt_block.a = 21845; // hard code to 01
39    DXTResult[DispatchThreadID.xy] = dxt_block;
40    return;
41  }
42  else
43  {
44    endpoints[0] = max(min_color, max_color);
45    endpoints[1] = min(min_color, max_color);
46  }
47
48  float3 color_line = endpoints[1] - endpoints[0];
49  float color_line_len = length(color_line);
50  color_line = normalize(color_line);
51
52  uint2 indices = 0;
```

```
53   for(i=0; i<8; i++)
54   {
55     uint index = 0;
56     float i_val = dot(samples[i] - endpoints[0], color_line) / color_line_len;
57     float3 select = i_val.xxx > float3(1.0/6.0, 1.0/2.0, 5.0/6.0);
58     index = dot(select, float3(2, 1, -2));
59     indices.x += index << i*2;
60   }
61
62   for(i=0; i<8; i++)
63   {
64     uint index = 0;
65     float i_val = dot(samples[i+8] - endpoints[0], color_line) / color_line_len
         ;
66     float3 select = i_val.xxx > float3(1.0/6.0, 1.0/2.0, 5.0/6.0);
67     index = dot(select, float3(2, 1, -2));
68     indices.y += index << i*2;
69   }
70
71   DXTResult[DispatchThreadID.xy] = uint4(max(color_0_565, color_1_565), min(
         color_0_565, color_1_565), indices);
72 }
```

Listing 5.2: BC1 compression on the computer shader

5.4 Terrain storage

Our solution for storing terrain tiles was developed with the dual goal of reducing draw calls and avoiding performing dynamic GPU memory allocations when patches are generated. Performing many draw calls increases overhead on the graphics driver and CPU and thus reduces performance. Performing dynamic allocation can also have a negative impact on performance. Our solution is achieved using texture arrays to store each type of terrain patch data - heights, normals (optionally as they can be computed as required on the GPU), low frequency texturing and rasterised high frequency textures. With this method, the entire terrain could conceivably be rendered with a single call, or in the case of spherical terrains six calls for each quadtree component of the sphere. This is complicated however by the five index buffer permutations we use in the LOD algorithm, the current limitation on texture array dimensions of 2048 slices and the necessity for each patch to also have access to the data if its parent patch. So the results we actually achieve are slightly worse, if we only use a single texture array in the cache as we do for all our evaluation tests, then 5 draw calls are needed. With two texture arrays in the cache, this increases to a maximum of 10 to support all combinations of parent/child indices.

5.5 Avoiding pipeline stalls

Avoiding pipeline stalls was simple as a result of the excellent performance we saw in our terrain generation stage. We simply follow the API recommendations to ensure at least a 3-frame buffer period after making a dispatch call to using its results. We do this for each stage of our pipeline - generation, rasterisation and compression. This causes a 9-frame delay in the generation process which does not prove noticeable.

6 Data-driven elements

In this part of the project we look at how the terrain system can be harnessed in order to create terrains similar, but not identical to user data. We only really touch the surface of the potential for such a system and suggest extensions which could form the basis for future projects or PhDs in 8.1). These extensions generally involve more complex computer vision techniques that would examine features and work from arbitrary view angles.

In the last section, we also propose a system for performing a recolouring of the tables and results of the atmospheric scattering technique proposed in Precomputed Atmospheric Scattering[6].

6.0.1 Overview of the data driven tool

The user interface for the tool shown in 6.2 serves as a good overview for its features at a glance. A summary of the process is as follows:

1. The user inputs a greyscale aerial elevation and RGB reflectance source and provides values for the height, width and displacement scale for these images so the system has a starting point for inferring feature frequencies.

2. The input elevation is matched to the output of the fractal generators available in the system using histogram based matching for elevation and (optionally) gradient. This can be configured in a number of ways. The user can select a particular noise type they think best matches the input or they can use the default setting and test against all noise types for the best statistical match. They also choose whether or not surface normals should be calculated for the noise results and source image and factored into the comparison. By default this comparison is enabled as it helps, sometimes quite significantly, in getting a more usable match over just comparing elevation. The result is improved since the gradient tells us more about the relative distribution of the heights in the scene. It introduces an additional linear time computation per test, but execution speed is not a concern. The user can choose to adjust the frequency analysis parameters (6.0.2.1) and fractal testing parameters for the minimum, maximum and step size values for the octave count, lacunarity and gain.

3. Next the system matches the colouration in the reflectance image to the height and gradient of the elevation source. This detects the low frequency changes in colour that often occur over a landscape. For example steep gradients are often more rocky and colours change from grasses in lowlands to snow on mountain peaks. With the results for the colour variation with height and gradient we create 1D ramp textures which can

be used with height (4.5.6) and gradient (4.5.7) selector masks respectively to apply colouration over the terrain.

4. In an optional step, we compute an average colour for the diffuse element of all the materials in the open terrain file's material palette. This value can then be used to match materials to the ramp previously generated.

5. The final step is to actually generate procedural rules and noise palette items for our system based on the results of the analysis. The user can choose whether to generate rules or just to add the noise match to the palette.

6.0.2 Histogram based matching of elevation data to procedural noise

Note that for all calculations involving colour for the rest of this topic we assume a normalised floating point value. For elevation data we would expect the source file format to be either 16 or 32-bit floating point but for reflectance input an 8-bit RGB format would be more typical. We make no assumptions about this input type however and use the same texture loader as for the rest of project and so can support a very wide variety of inputs such as DDS, PNG, JPG, TIF, TGA and BMP, then convert as required.

The scheme for matching the source elevation to noise types is straightforward. We create a list of possible parametrisations and noise types for testing. Some of these parameters are fixed in the current implementation and some are configurable by the user as described in 6.0.1. One exception we make is to estimate the frequencies f_x and f_y of the predominant features in the terrain which we describe in 6.0.2.1. Once this list is created, for the source image we create a histogram for a given number of bins k (we arbitrarily use $k = 100$) for either just the height E or both the height and calculated normal values N. When creating histograms for coloured images, we bin each colour channel separately rather than calculating an intensity value:

$$f_R(p) = \begin{cases} 1 & p \in R \\ 0 & \text{otherwise} \end{cases}$$

$$E_j = \sum_{i=0}^{k-1} f_{(l_j,u_k]}(p_i)$$

$$N_{j_{\{c\in r,g,b\}}} = \sum_{i=0}^{k-1} f_{(l_j,u_j]}(p_{i_c})$$

Next for each noise parameterisation, we calculate the noise values and gradient (if being

used) and compare the histograms of these results to that of the source image. This is done by adding up the absolute value of the differences between pixel values for heights and intensity values for normals.

$$d_E = \sum_{i=0}^{k-1} \left| E_i - {E'}_i \right|$$

$$d_N = \sum_{i=0}^{k-1} \sum_{\{c \in r,g,b\}} \left| {N_i}_c - {N'}_{i_c} \right|$$

We base our final score for a noise result based on a 50/50 weighting of the height and normal difference values, $\frac{1}{3}(d_{N_r} + d_{N_g} + d_{N_b}) + d_E$ keeping track of the current best result.

6.0.2.1 Estimating frequency

We estimate the frequency of the dominant features in the terrain for use in finding an initial parametrisation for the frequency value of the noise generators used in the matching process. The minimum and maximum values for all pixels in the elevation image are calculated, E_{min} and E_{max}, then both vertically and horizontally we look at the number of times the terrain crosses within a threshold ϵ of those values (we use $\epsilon = 0.2$ by default but it is user configurable). This value is halved since the threshold will be crossed twice per feature, as can be seen in figure 6.1.

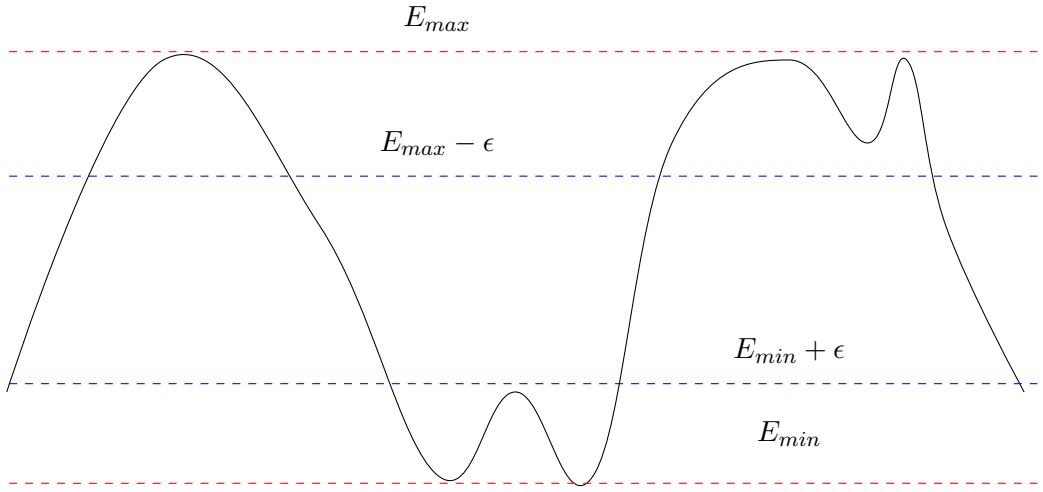


Figure 6.1: Frequency estimation

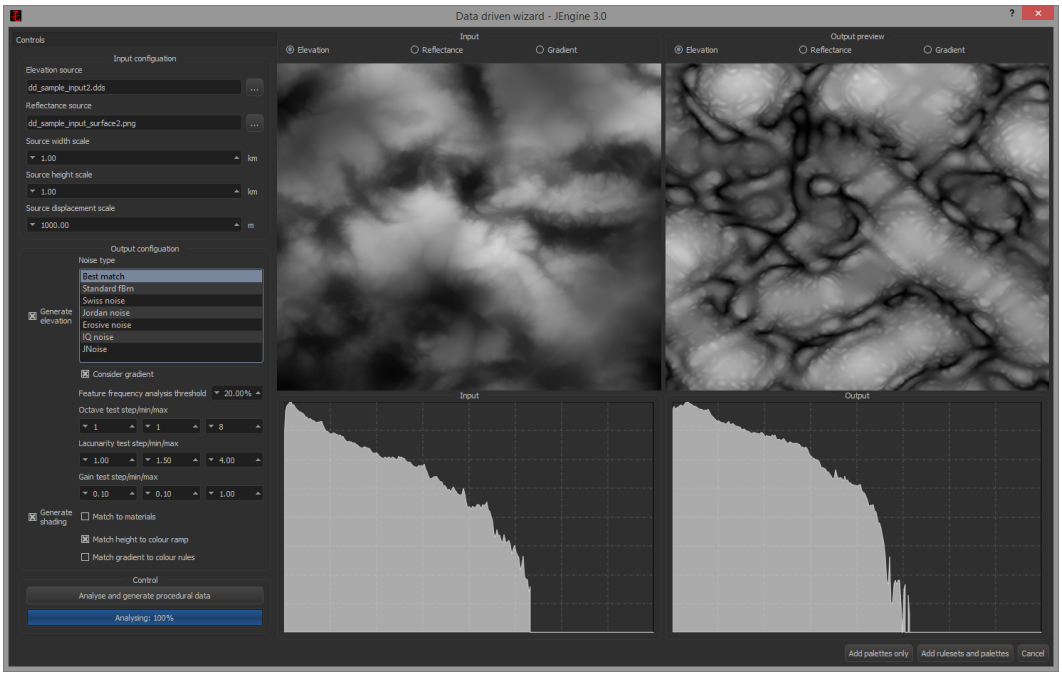


Figure 6.2: UI showing a greyscale histogram for a generated heightmap.

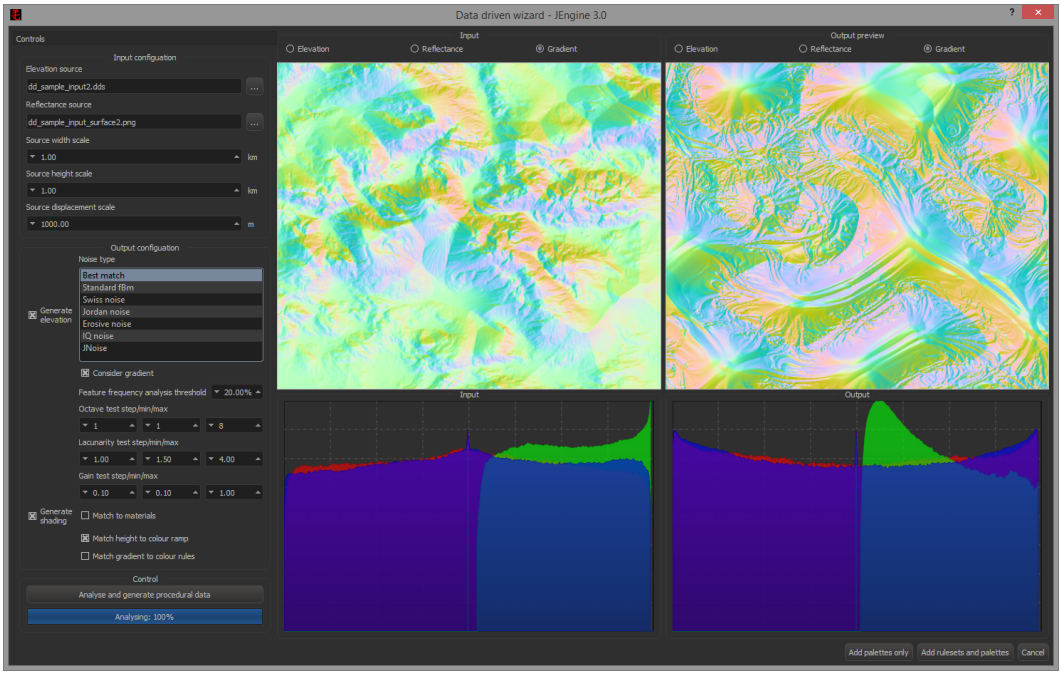


Figure 6.3: UI showing an RGB histogram for the surface normals of a generated heightmap and the source image.

6.0.3 Matching source reflectance data to elevation, gradient and materials

To match elevation data to reflectance data, we create a pool of size n for storing reflectance texture results at each pixel range in the height image divided into n value bands. The use of bands helps minimise anomalies that could occur over smaller ranges. The results are then averaged for each element of the pool to produce a colour gradient over the height range. It will often be the case that the heights in the source elevation image do not cover the full range, and so we get black pixels at the extremities. In the noise result however, heights may fall outside the range of the source so we adjust the gradient result by copying the highest valid result to all black pixels above and the lowest valid result to all black pixels below. It could also be the case that ranges in the middle of the gradient are missed. This should happen rarely when using a sensible pool size but we do handle the case by linearly interpolating between the closest valid pixels on either side of the black range(s). Figure 6.4 shows an example of these generated ramps in use for recolouring the original elevation image.

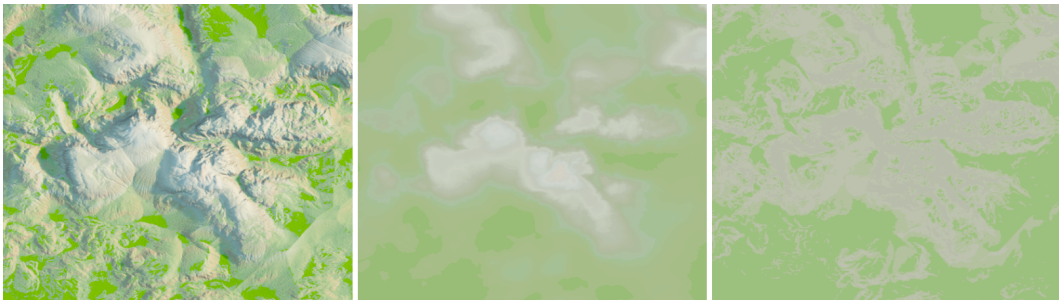


Figure 6.4: Results from recolouring the original elevation map from the ramps generated from the height (middle) and gradient (right) analysis

The final option we offer users for reflectance matching is to generate average colour values for all the diffuse textures in the materials palette and then match these to the previously generated ramps when creating rules in the final step. We give an example of this averaging in figure 6.5.

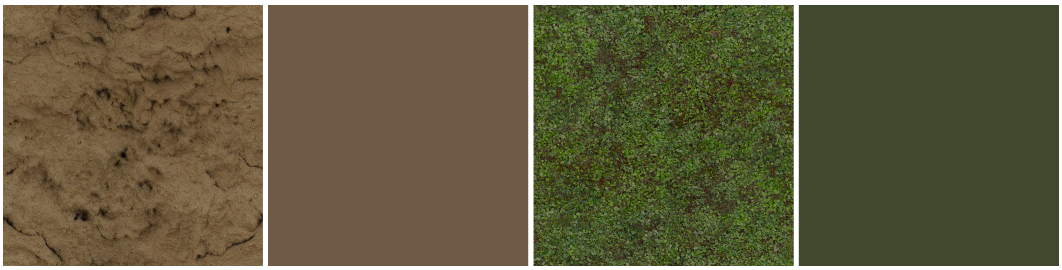


Figure 6.5: Average colour computation for a sand and grass texture

6.1 Inferring procedural rules from user data

At this point, we have determined the best matching fractal for the available noise types and parametrisation settings. We have also found a colour ramp that encodes how the terrain colour changes at a low frequency according to height and optionally also gradient. We have finally found the average colours of the available materials in the open terrain file. What remains is to create procedural rules for our system and add them to the terrain. The rule insertion progress is as follows:

1. The best matching fractal is added to noise palette directly, named after the source image it is derived from.

2. A rule-set is created, bound by a rectangular mask (4.5.3) of the dimensions specified for height and width scale centred around the user's current position.

3. To the rule-set is added a noise mask (4.5.11) for the newly added fractal and a displacement modifier (4.6.1) to add the height value specified by the user initially for scale.

4. Next low frequency surface modifiers (4.6.4) are added for the height and gradient based shading. We add a height and gradient selector (4.5.6 and 4.5.7) with single direction blending (4.5.9) and then apply the respective ramp textures over the mask.

5. Finally we add child rule-sets for changing the high frequency surface texture based on the average values calculated at a fixed bands over the height colour ramp. In the current implementation four such rules are added.

After this process it is likely the user will want to edit the rules to adjust the masked region, disable rules that they do not like the result of and so on.

6.2 Atmospheric scattering recolouring

In this part of the project, we take a step back from looking purely at terrains and consider how they and the surrounding atmosphere can be lit using a data-driven extension of the atmospheric scattering solution proposed by Bruneton[6] (for background refer to 2.3).

We can describe the radiance of light reaching a point $\vec{x}$ from a point $\vec{x_0}$ in direction $\vec{v}$ with the star in direction $\vec{s}$ using the rendering equation for participating media. $\vec{x_0}$ is either a point on the terrain surface, an object in the sky or the limit of the atmosphere where the density of the particles is zero. This point where the direction vector v meets the atmosphere from $\vec{x}$ is referred to as $\vec{x_s}$. We now define three values that make up the rendering equation:

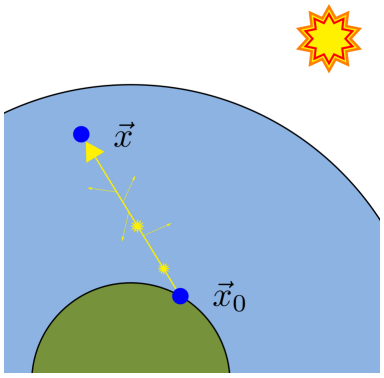


Figure 6.6: $T(\vec{x} \leftrightarrow \vec{x_0})$

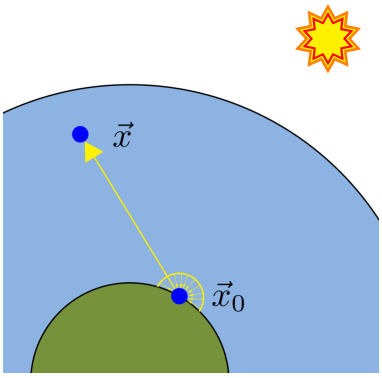


Figure 6.7: $I(\vec{x_0}, \vec{s})$

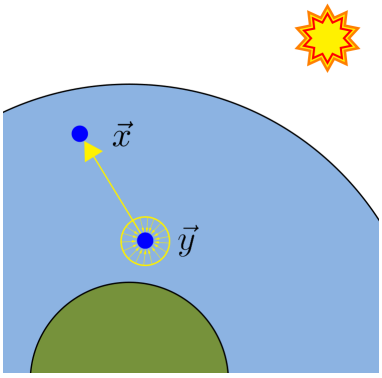


Figure 6.8: $J(\vec{y}, \vec{v}, \vec{s})$

- The transmittance $T(\vec{x} \leftrightarrow \vec{x_0})$, is the fraction of photons that travel from $\vec{x_0}$ to $\vec{x}$ unaffected by the medium. We calculate this by integrating over the path considering the Rayleigh (2.3.1.1) and Mie (2.3.1.2) extinction coefficients so as to remove photons that are lost due to out-scattering and absorption.

- The radiance $I(\vec{x_0}, \vec{s})$, is the radiance reflected at x_0. This is generally the reflectance of the planetary surface but could also be objects in the sky if they existed. Note that if the direction does not cause an intersection with the terrain or any object in the sky then $I(\vec{x_0}, s)$ is 0. Its calculation is given here just as a standard diffuse model but in our actual rendering implementation we use a few more factors. α is the surface reflectance and $\vec{n}(\vec{x_0})$ is the surface normal at x_0 and the calculation performs an integral over the hemisphere above $\vec{x_0}$ for the incident irradiance at x_0.

- The radiance $J(\vec{y}, \vec{v}, \vec{s})$, is the radiance of light scattered at a position $\vec{y}$ along $\vec{v}$ towards $\vec{x}$. It is an integral over the entire sphere around $\vec{y}$ since light can enter from any direc-

tion. It factors in the sun direction and the phase function and scattering coefficients for both Rayleigh and Mie scattering.

$$T(\vec{x} \leftrightarrow \vec{x_0}) = exp\left(\int_{\vec{x}}^{\vec{x_0}} \sum_{i \in \{R,M\}} \beta_i^e(\vec{y})dy\right) \tag{6.1}$$

$$I(\vec{x}, \vec{s}) = \frac{\alpha(\vec{x_0})}{\pi} \int_0^{2\pi} L(\vec{x_0}, \vec{\omega}, \vec{s})\vec{\omega} \cdot \vec{n}(\vec{x_0})d\vec{\omega} \tag{6.2}$$

$$J(\vec{y}, \vec{v}, \vec{s}) = \int_0^{4\pi} \sum_{i \in \{R,M\}} \beta_i^s P_i(\vec{v} \cdot \vec{\omega})L(\vec{y}, \vec{\omega}, \vec{s})d\vec{\omega} \tag{6.3}$$

We use these terms to formulate the final rendering equations for calculating the radiance at $\vec{x}$, $L(\vec{x}, \vec{v}, \vec{s})$.

$$L(\vec{x}, \vec{v}, \vec{s}) = L_0 + R[L] + S[L])(\vec{x}, \vec{v}, \vec{s}) \tag{6.4}$$

$$L_0(\vec{x}, \vec{v}, \vec{s}) = T(\vec{x}, \vec{x_0})L_{sun} \tag{6.5}$$

$$R[L](\vec{x}, \vec{v}, \vec{s}) = T(\vec{x}, \vec{x_0})I(\vec{x_0}, \vec{s}) \tag{6.6}$$

$$S[L](\vec{x}, \vec{v}, \vec{s}) = \int_{\vec{x}}^{\vec{x_0}} T(\vec{x}, \vec{y})J(\vec{y}, \vec{v}, \vec{s})dy \tag{6.7}$$

L_0 is the direct sunlight arriving at $\vec{x}$ and is 0 in the cases where $\vec{s} \neq \vec{v}$ or where the sun is otherwise occluded by the terrain itself. $R[L]$ is the light reflected at x_0 attenuated by the transmittance value to account for absorption and outscattering along the path. $S[L]$ is the inscattered light towards $\vec{x}$, calculated by integrating for all points $\vec{y}$ over the path from $\vec{x_0}$ to $\vec{x}$, the inscattering at $\vec{y}$ attenuated again by the transmittance. The direct light term is straightforward to calculate in real-time, but the inscattered and reflected light are what prove difficult. This is particularly true when considering multiple scattering which will produce nested integrals the more steps are performed. Using multiple scattering it is not currently possible to produce real-time results with this formulation.

Bruneton's paper[6] focusses on how to precalculate as large a portion of L as possible. We implemented the algorithm proposed as per the paper so avoid reiterating the details of its contents here as a full understanding is not required to appreciate our recolouring step. However, the idea of the paper is to precompute three textures for transmittance, irradiance and inscattering under the assumption that the terrain is perfectly spherical. This assumption means that any point and viewing angle in the atmosphere can be described in terms of the height above the surface and the view zenith angle. This is key since it means precompu-

tations can be stored in reasonable amount of memory. The transmittance texture provides a lookup for $T(\vec{x} \leftrightarrow \vec{x_0})$ by computing a 2D table of values based on the height above the surface and view angle. The irradiance texture provides a 2D lookup for the irradiance value used in $I(\vec{x}, \vec{s})$ that takes multiple scattering into account, again parameterised by the height above the surface and the view zenith angle. The inscattering table provides a 4D lookup for the inscattering along infinite paths through the atmosphere, parametrised by the height above the surface, the view zenith, sun zenith and view sun angle. This is used in calculating $J(\vec{y}, \vec{v}, \vec{s})$.

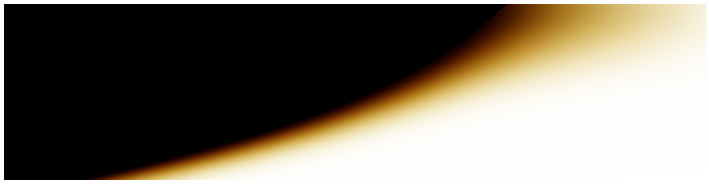


Figure 6.9: The transmittance texture. The x-axis is the view zenith and the y-axis the height above the surface.

Figure 6.10: Example irradiance table. The x-axis is the view zenith and the y-axis the height above the surface.

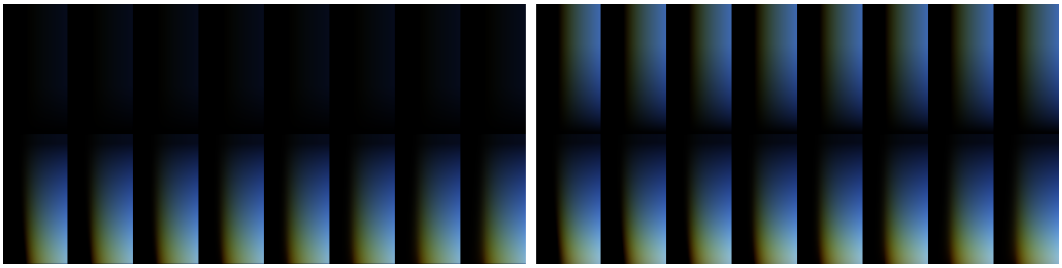


Figure 6.11: Example of two slices of the 4D inscattering texture. Note that DirectX can not provide 4D textures, so a 3D texture is used where the view zenith, sun zenith and view sun angle are stored in a 2D texture slice whereby the x-axis actually stores two dimensions of data (note the divisions). The multiple slices of these three parameters that make up the 3D texture are for different heights above the surface.

Our recolouring step is quite simple, it lets a user draw a gradient which is used to recolour the inscattering table and adjust some of the rendering functions. The gradient they provide should store the desired variation with view zenith angle around the time of sunset or sunrise so that both Rayleigh and Mie scattering factors are evident. Figure 6.12 shows an example configuration and result where we have a desired sky that is orange and reddens around sunset.

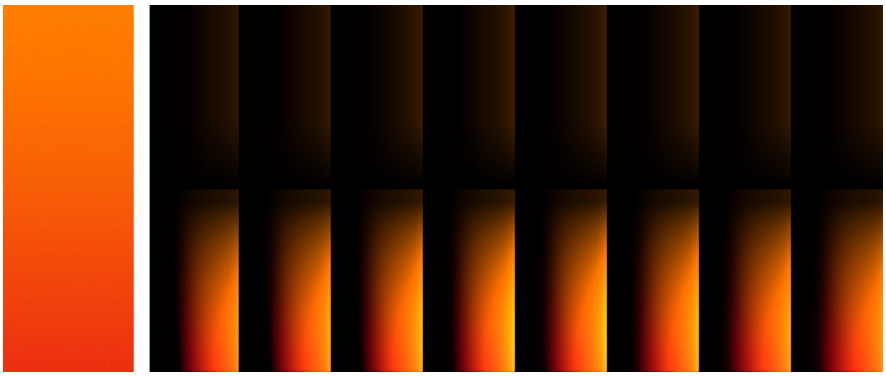


Figure 6.12: Example recolouring

The first question to answer in order to attempt a recolouring is looking at what determines the final colour of the original values in the inscattering table. Looking at the formula for inscattered light 6.7, the colour components come from the use of the transmittance value and the evaluation of the integral J in 6.3 since it contains the scattering coefficients for Rayleigh and Mie scattering. In 6.3, only the Rayleigh component will contribute colour as β_M^s equal across its components. The transmittance value however depends on the extinction coefficients for Rayleigh and Mie scattering, this will result in the absorption and out-scattering of some wavelengths at different rates and is what gives rise to the variation in colour over the atmosphere. This can be seen in the example transmittance texture in figure 6.9, where the red/orange wavelengths take longer before being fully absorbed than green and blue wavelengths. The inscattering table example in figure 6.11 also highlights this - the resultant colour is a combination of the blue Rayleigh scattering coefficient with varying transmittance values. The change predominately occurs as the view zenith angle changes (y-axis), i.e. the sky appears redder around the horizon.

To perform recolouring we initially calculate the standard tables with default parameters (we use the parameters for Earth). Doing this completely absolves the end user from having to be concerned with the physical properties of the model when trying to achieve their desired result. The RGB component of the inscattering table is then recoloured by sampling the gradient texture G and replacing the existing colour S with a sample from the gradient based

on how close current value is to the Rayleigh scattering coefficient value. So S is alterated at given (u, v) coordinates by:

$$S'(u,v) = |S(u,v)| \cdot G(0.0, \frac{S(u,v)}{|S(u,v)|} - \frac{\beta_R^s}{|\beta_R^s|}) \tag{6.8}$$

In a variation of this, we can also allow the user to provide a two dimensional gradient. This can be used to provide different shadings at different altitudes by adding an additional parameter to the sampling w, the normalised slice coordinate. In our experiments, we did not find this ability particularly useful but leave it as an option for the end user.

$$S'(u,v) = |S(u,v)| \cdot G(w, \frac{S(u,v)}{|S(u,v)|} - \frac{\beta_R^s}{|\beta_R^s|}) \tag{6.9}$$

There is one more step required to make this system fully compatible with the system proposed in the paper. There are two render-time functions that depend on the Rayleigh scattering coefficient. Since our recolouring the original coefficient is no longer valid and using it would result in visible discontinuities. Therefore when it is used in calculating the restored Mie scattering term and value for analyitic tranmittance (see [6]), we replace the original value for β_R^s with $|\beta_R^s| * G(0, 1.0)$. That is, the recolour value at the highest view zenith angle. If a 2D gradient is used, then this is altered to $|\beta_R^s| * G(w, 1.0)$, where w is now the height above the surface as a fraction of the atmosphere height. This gives the colour at the highest view zenith angle for the appropriate slice.

7 Evaluation

7.1 Terrain system

7.1.1 Patch generation performance

We evaluated the performance of the patch generation system using a separate testing suite that references the shader generation and execution portion of the main program to compile and then benchmark a given test terrain. Using this suite we ran a wide variety of tests, looking at the performance of individual rules, rule-sets of various complexities, noise types and rule compilation times.

To give more context to the results, we ported the entire terrain generation system minus a few noise types to C++ to perform a comparison with CPU evaluation speed. Considerable effort was made to make the CPU implementation high performing. To this end, it runs on multiple threads using (n-1) available CPU cores to simulate one core being used for rendering. It also uses SIMD instructions (via the DirectXMath library) for vector operations. Despite these efforts, it should be kept in mind that implementing the CPU version of the terrain system was not a core focus of the project and could invariably be optimised further. The GPU implementation was done with high execution speed in mind so achieving at least a significant overall improvement over the CPU implementation is key to determining the success of the system from a performance standpoint. Generation on the CPU is less complex and potentially more flexible, particularly if more dynamic rule types were to be implemented. So we depend our recommendation to readers who may be interested in implementing a similar system on these results.

Our GPU tests were performed on an NVIDIA GeForce GTX 680, a reasonably modern and high-end GPU at the time of writing. The CPU tests were performed on a 6-core Intel 3930K (and so rule tests and so on utilised 5 of those cores). The system requires full support for Shader Model 5.0 due to its usage of compute shader features, so performance and compatibility with older systems has not been a concern for the project. We also had a brief opportunity to run the system on a laptop with a NVIDIA GeForce GTX 770M. There was not sufficient time to run the full test suite. However, we informally found that performance scaled as would be expected. The overall performance was roughly 50-70% of that of the desktop GPU and we encountered no issues with patch generation speed.

7.1.1.1 Measuring performance

We time the execution of GPU rules using the DirectX query system and a high resolution timer. Using queries allows us to synchronise the GPU with the CPU so that timer results are accurate. We show an example snippet for this in listing 7.1.

```
D3D11_QUERY_DESC Query;
Query.Query = D3D11_QUERY_EVENT;
Query.MiscFlags = 0;
ID3D11Query *pEventQuery;
m_pD3DDevice->CreateQuery( &Query, &pEventQuery );

NanoTimer t;

// spin to ensure all previous GPU requests have been processed
m_pContext->End(Query);
while (m_pContext->GetData(Query, NULL, 0, 0) != S_OK);

// start timing
t.start();
// make the dispatch call for the test rule
m_pContext->Dispatch(*m_pShaderCompute[test], DispatchSize, DispatchSize, 1,
        pParamManager);

// spin to synchronise with the completion of the dispatch call
m_pContext->End(Query);
while (m_pContext->GetData(Query, NULL, 0, 0) != S_OK);

// stop timing
t.stop();
```

Listing 7.1: Timing GPU dispatch calls

The NanoTimer class is a wrapper around calls to the Windows API function QueryPerformanceFrequency which provides the high resolution timing at microsecond precision. We use this class as well in timing CPU performance.

7.1.1.2 Individual rule performance

For testing individual rule performance we created a set of tests that cover almost our entire set of rules and ran them on both the GPU and CPU. Figure 7.1 shows an example of the test patterns output by the system when executing various rules. The exclusions are as follows:

1. Constant masks are omitted (4.5.1) due to triviality.

2. For brevity, we avoid testing all modes of operation that involve slightly modified blending functions, such as those for height, gradient and direction selectors. This is because there is negligible computational difference between them.

3. We test the noise mask (4.5.11) only with fixed user data. When using a fractal source, preliminary tests showed that the mask computation contribution becomes insignificant compared to the computation of the fractal itself. We evaluate fractal generation performance separately (7.1.1.4) and in rules when testing entire rule-set performance for a more overall look at performance (7.1.1.3).

4. The streaming mode of operation for noise masks is heavily CPU/IO bound. We do not test this mode of the rule since it has not been our focus to write a highly efficient streaming system. As far as GPU-implementation is concerned, streaming mode is almost identical besides slightly different UVs and requires the same number of texture look-ups.

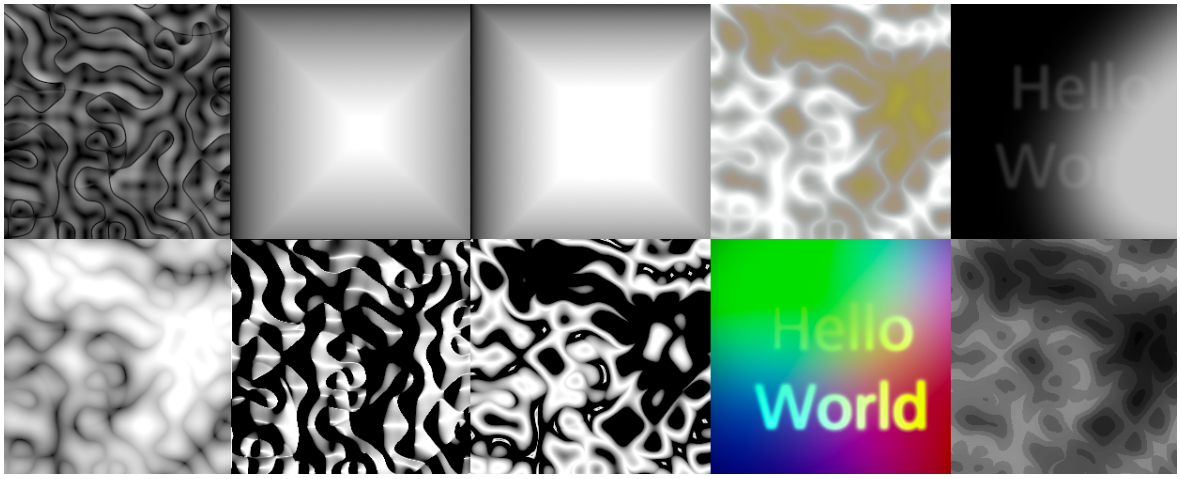
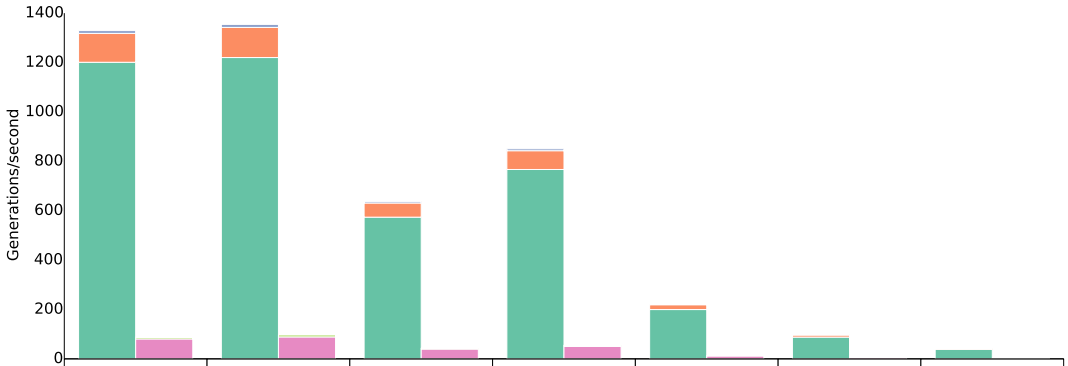


Figure 7.1: An example of the test patterns generated by the testing process.

For each rule type, we look at the scaling with number of executions and patch size to see how consistent the results are. Note that we always ensure that rules are parameterised such that they get a low number of inputs whereby a short execution path can be taken. For example there is often a short path if an input location falls outside of a selector region. This means our results are approximately worst case results - testing parametrisations that can give short exits regularly would give trivial results.

Figure 7.2 shows the test results for the shape-based mask selectors. The circle selector performs best due to its simple geometry. The rectangle selector is slightly slower (the inner and outer modes are not parametrised to produce exactly the same result hence the noticeable difference between them). Finally the polygon selectors are considerably slower due to their higher complexity. The vertex count roughly doubles between the small, medium and large tests and we see a proportional change in generation speed. As the majority of work in the polygon selector is linear with respect to the vertex count, this is what we would expect. We also see linear scaling with patch size and number of executions. Generation times are increased by around 16% and 10% as patch size changes from 64x64 to 256x256 and number of executions are increased by a factor of ten respectively.

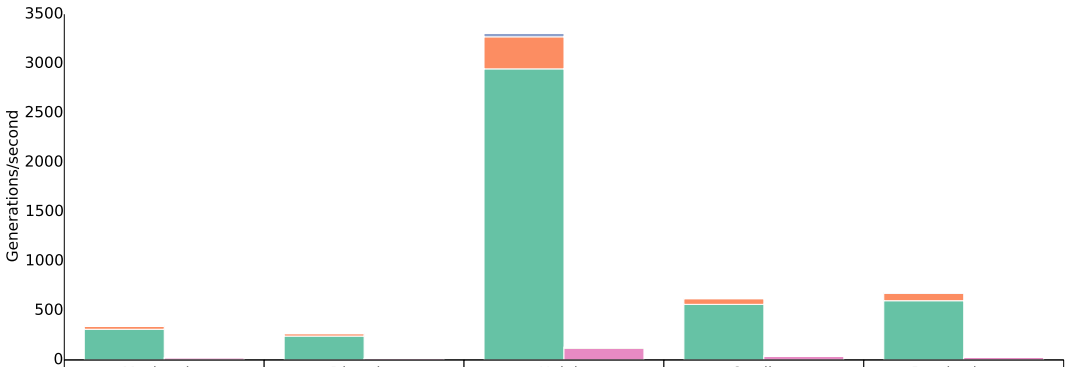


	Mask circle (outer)	Mask circle (inner)	Mask rectangle (outer)	Mask rectangle (inner)	Mask polygon (small)	Mask polygon (medium)	Mask polygon (large)
100 executions (64x64 patch)	1201.05	1220.18	573.80	766.90	199.65	88.10	38.27
1000 executions (64x64 patch)	117.01	122.03	57.34	75.98	19.52	8.77	3.79
10000 executions (64x64 patch)	11.26	12.18	5.72	7.60	1.99	0.88	0.38
100 executions (256x256 patch)	80.71	88.84	39.61	50.06	11.97	5.56	2.52
1000 executions (256x256 patch)	6.64	8.72	3.91	4.96	1.26	0.55	0.25
10000 executions (256x256 patch)	0.81	0.88	0.39	0.50	0.13	0.06	0.03

Figure 7.2: CPU selector test results for shape based selectors

Figure 7.3 shows the results for the remaining selector rules. The results again all seem in line with what we would expect in terms of linear scaling with patch size and number of generations. The height selector performs best and the band selector using the height selector performs roughly four times slower as expected. For each execution of the gradient and direction selector we are computing surface normals - we keep this consistent above both the GPU and CPU tests. Without this calculation enabled, the performance is more in line with the height selector due to similar operations. The direction selector is slightly slower since it requires more calculations and trigonometric operations. The mask noise using a static heightmap texture source is also a number of times slower than the height selector. Although

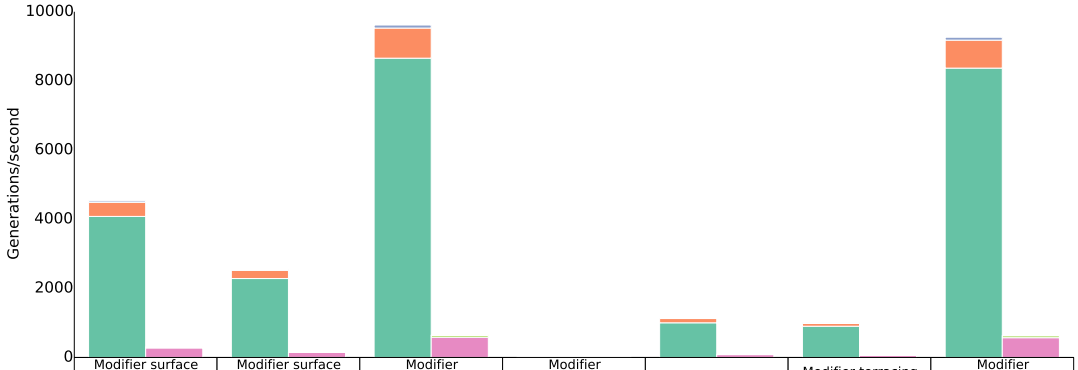
the blending operation is comparable, the sampling of the source texture is relatively expensive on the CPU have no native methods for performing interpolated sampling. We anticipate far better performance on the GPU with its hardware support for texture operations.



	Mask noise (static user data)	Direction selector	Height selector	Gradient selector	Band selector height (4 bands)
100 executions (64x64 patch)	310.79	241.51	2946.57	562.91	599.33
1000 executions (64x64 patch)	31.13	24.06	324.44	56.79	74.98
10000 executions (64x64 patch)	3.14	2.42	33.83	5.73	8.23
100 executions (256x256 patch)	21.71	16.10	118.91	37.62	25.80
1000 executions (256x256 patch)	2.18	1.60	11.26	3.68	2.69
10000 executions (256x256 patch)	0.22	0.16	1.07	0.35	0.32

Figure 7.3: CPU selector test results for remaining selectors

Figure 7.4 shows the results for the modifier rules. The high frequency surface modifier and displacement modifier in the constant mode of operation perform best only requiring one or two operations per call. The low frequency surface modifier is slower due to how the result is combined with the existing result. Using a ramp texture only slows the result slightly; unlike in the displacement modifier and noise mask we only have a linear interpolation to perform with the 1D source texture which is quite inexpensive. The terracing modifiers seem to perform in line with their complexity relative to the more simple modifiers.



	Modifier surface low frequency (constant)	Modifier surface low frequency (static ramp)	Modifier displacement (constant)	Modifier displacement (static user noise)	Modifier terracing	Modifier terracing (advanced)	Modifier surface high frequency
100 executions (64x64 patch)	4079.81	2282.79	8656.89	11.77	1002.23	903.53	8369.41
1000 executions (64x64 patch)	408.09	236.09	871.19	1.16	123.59	85.61	809.50
10000 executions (64x64 patch)	40.92	23.91	87.40	0.12	12.39	9.04	83.42
100 executions (256x256 patch)	274.92	145.17	586.02	8.77	81.97	61.87	571.32
1000 executions (256x256 patch)	27.60	14.54	58.70	0.87	8.35	6.18	57.62
10000 executions (256x256 patch)	2.76	1.45	5.88	0.08	0.85	0.62	5.80

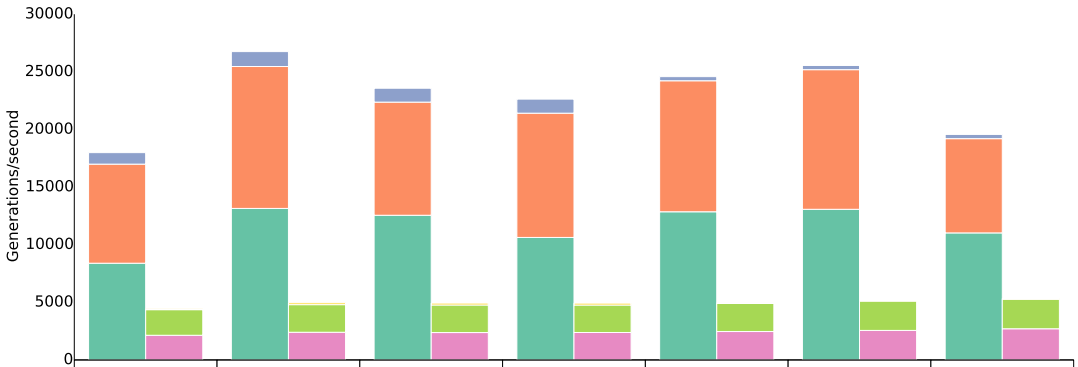
Figure 7.4: CPU modifier test results

We now move onto the results for the GPU implementation of the rules. We had anticipated a significant increase but found the degree of this increase very surprising in some cases. It was common to see a performance increase between 10-100x over the CPU implementation. While we do not claim the CPU implementation to be perfectly optimised, it does run on multiple cores and use SIMD instructions where possible to emulate the HLSL code. With this in mind, we would expect it to be reasonably performant compared to naive solutions.

The results in figured 7.5 show GPU performed more evenly across the shape mask tests. There are two main reasons behind this. The massively parallel architecture of the GPU means that it is quite typical to see performance scaling rather different than what would be expected on the CPU. Our GPU rule implementation also has the advantage of rules being compiled with their values in place. This means the compile can make optimisations based on actual parametrisations of rule. In comparison the C++ compiler produces code which is then called with the various parameters.

One result that seems out of place is the circular mask in the outer mode of operation being the worst performer. There was no apparent reason for this so we disassembled the two generated shaders to inspect the HLSL assembly. We include these in the appendix at 2 and 3 as an example of the code generated. We found that the two shaders were identical

other than two different instructions at lines 16 and 17. It seems rather unlikely that this would cause the difference but since the results were consistently repeatable it is the only explanation we can provide.

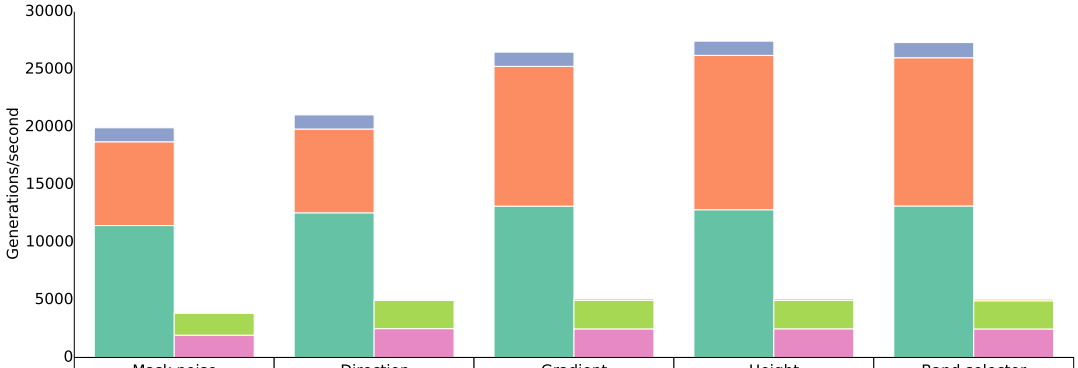


	Mask circle (outer)	Mask circle (inner)	Mask rectangle (outer)	Mask rectangle (inner)	Mask polygon (small)	Mask polygon (medium)	Mask polygon (large)
100 executions (64x64 patch)	8393.68	13148.51	12554.78	10629.72	12856.94	13072.60	11023.02
1000 executions (64x64 patch)	8597.58	12313.38	9817.60	10783.02	11369.09	12115.75	8178.42
10000 executions (64x64 patch)	997.04	1293.51	1205.60	1220.36	357.25	363.83	367.37
100 executions (256x256 patch)	2140.45	2401.81	2375.88	2381.53	2465.30	2562.11	2704.93
1000 executions (256x256 patch)	2221.58	2395.94	2393.09	2384.02	2437.06	2544.11	2564.24
10000 executions (256x256 patch)	82.10	155.96	160.99	160.85	40.30	49.99	51.72

Figure 7.5: GPU selector test results for shape based selectors

The results for the remaining selector types in figure 7.6 were as expected overall. The height selector was the best performing and the band selector almost as fast unlike the scaling we saw in the CPU. The gradient and direction selectors are again slightly slower due to the normal calculation being performed in these tests. The massively parallel architecture of the GPU makes it much more suited to doing per-pixel style operations like normal calculations than the GPU so the comparitively smaller difference with the height selector was anticipated. The noise mask is also far more comparable to the height selector due to the fast hardware support for texture operations on the GPU.

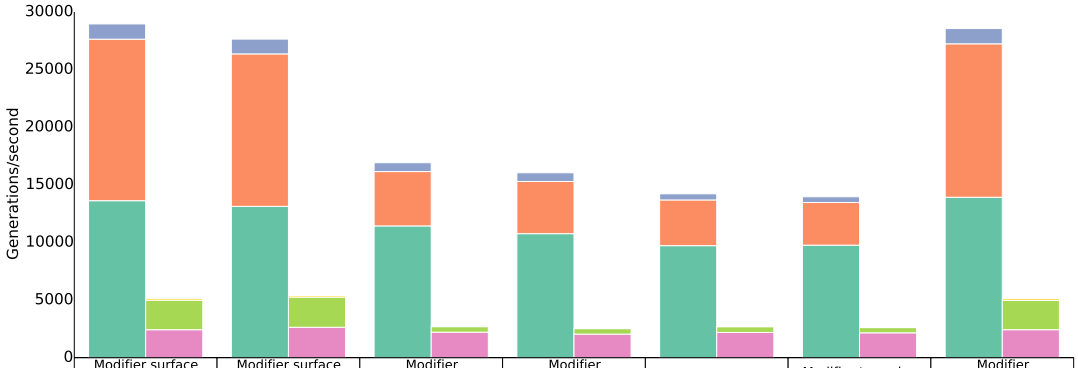
We do notice a few slight anomalies in the performance again. In particular the result for 10,000 executions for 256x256 patches and the seemingly negligible impact the move from 100x256x256 to 1000x256x256 had on performance. We again checked the assembly to try and uncover a reason for this but it offered no explanation. The compilation results are identical aside from a change in loop range. We can therefore only put the reasoning down to the underlying architecture of the GPU which we are unable to investigate.



	Mask noise (static user data)	Direction selector	Gradient selector	Height selector	Band selector height (4 bands)
100 executions (64x64 patch)	11445.29	12549.74	13130.28	12817.92	13146.28
1000 executions (64x64 patch)	7267.93	7273.34	12129.39	13395.93	12861.18
10000 executions (64x64 patch)	1215.19	1224.59	1234.33	1235.26	1315.22
100 executions (256x256 patch)	1933.86	2498.69	2477.85	2483.50	2471.28
1000 executions (256x256 patch)	1901.69	2442.45	2483.30	2469.18	2444.63
10000 executions (256x256 patch)	93.01	101.43	160.97	160.24	156.70

Figure 7.6: GPU selector test results for remaining selectors

The modifier results shown in 7.7 again follow what we would expect, with big gains in selectors that use texture operations and have more instructions.



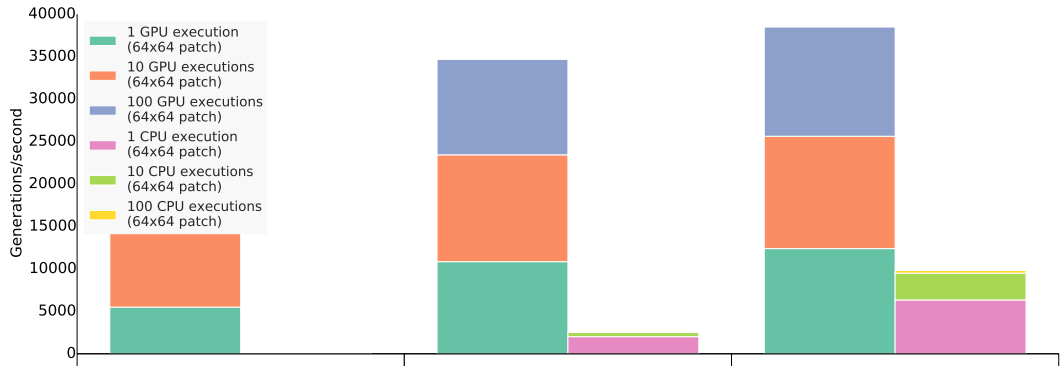
	Modifier surface low frequency (constant)	Modifier surface low frequency (static ramp)	Modifier displacement (constant)	Modifier displacement (static user noise)	Modifier terracing	Modifier terracing (advanced)	Modifier surface high frequency
100 executions (64x64 patch)	13619.53	13128.62	11431.06	10758.53	9717.16	9760.26	13919.81
1000 executions (64x64 patch)	14016.58	13228.65	4714.98	4528.79	3966.72	3703.08	13313.19
10000 executions (64x64 patch)	1322.28	1284.57	766.35	753.67	530.58	495.56	1322.98
100 executions (256x256 patch)	2423.21	2636.76	2209.44	2042.83	2188.62	2154.81	2424.35
1000 executions (256x256 patch)	2565.04	2618.20	479.56	475.33	496.02	452.77	2542.17
10000 executions (256x256 patch)	158.35	133.78	52.41	52.35	54.58	49.30	155.97

Figure 7.7: GPU modifier test results

Overall we feel the performance gains from the GPU implementation were compelling. We do need to take caution in interpreting these results though. The biggest gains over the CPU implementation were seen in tests with larger patch sizes and execution counts. In reality, we are never going to execute a rule thousands of times per shader. As shader complexity reduces, the overhead of the dispatch call itself becomes far more significant than the particulars of what is executed. So in more realistic cases hundred times gains are not likely to be seen. We did not include tests for only single executions as the results can fluctuate widely - benchmarking GPU performance is less absolute than CPU performance.

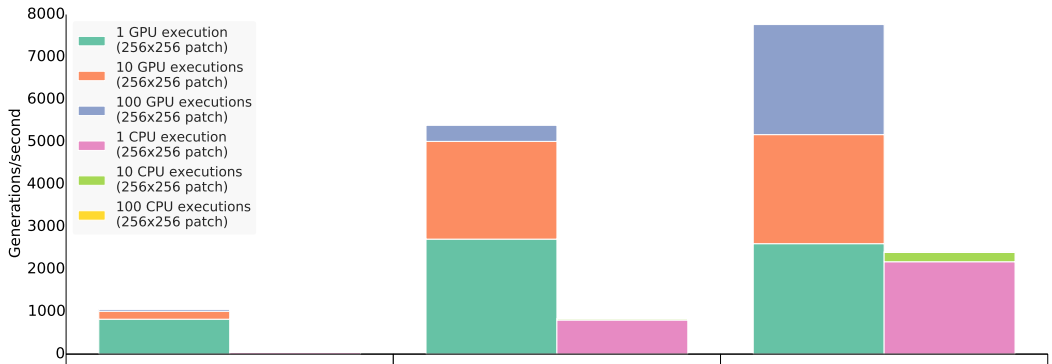
7.1.1.3 Ruleset performance

In 7.1.1.2 we looked at the performance of individual rules and found a massive performance benefit from using the GPU over the CPU. However, we also noted that these results should not be taken at face value due to their quite theoretical nature. In this section, we look at the performance of some actual rule-sets we created while testing terrains. The smallest rule-set we test has around ten varied rules and the largest is around the top end of what we expect to be used with hundreds of rules. This makes considerably more complex shaders which we can benchmark more reliably even for a single execution. Figures 7.8 and 7.9 shows the results of our tests. Note that our multithreading solution for generating patches on the CPU uses one thread per execution, so the results for a single patch are not multithreaded.



	Large ruleset	Average ruleset	Simple rule
1 GPU execution (64x64 patch)	5499.01	10850.06	12389.55
10 GPU executions (64x64 patch)	10850.06	12586.15	13236.51
100 GPU executions (64x64 patch)	926.24	11241.51	12860.66
1 CPU execution (64x64 patch)	135.07	2017.52	6339.03
10 CPU executions (64x64 patch)	65.98	486.62	3160.54
100 CPU executions (64x64 patch)	6.90	104.74	321.02

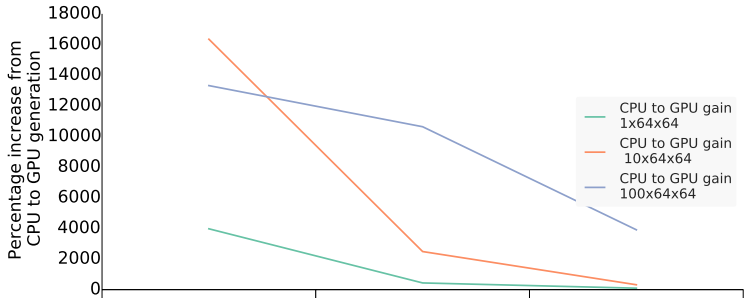
Figure 7.8: Rule-set performance on the GPU and CPU for 64x64 patches



	Large ruleset	Average ruleset	Simple rule
1 GPU execution (256x256 patch)	822.48	2708.94	2601.16
10 GPU executions (256x256 patch)	187.72	2302.47	2569.30
100 GPU executions (256x256 patch)	42.42	377.88	2592.70
1 CPU execution (256x256 patch)	46.18	802.84	2175.67
10 CPU executions (256x256 patch)	4.80	28.89	217.86
100 CPU executions (256x256 patch)	0.49	6.98	22.18

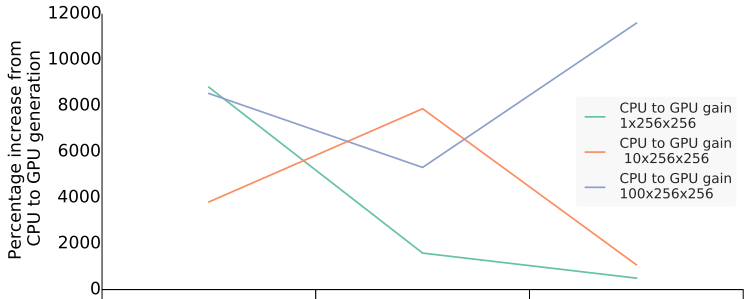
Figure 7.9: Rule-set performance on the GPU and CPU for 256x256 patches

These results are more representative of what we would expect to see in real-world usage of the system. In figures 7.10 and 7.11 we show the percentage gains from using the GPU over the CPU. In the most realistic cases - single executions - we see that there is a far smaller gain than there was in the single rule tests. However, the results are still pleasing. To see approximately a 2x increase even on the most simple rule-sets was beyond our expectation. It had seemed quite possible that the cost of executing the shader would outweigh the natural performance benefits over the CPU for simple rule-sets. As rule complexity increases, we see increasing benefits coming from the parallel nature of the GPU. There is a significant increase too between the 64x64 and 256x256 patch sizes.



	Large ruleset	Average ruleset	Simple rule
CPU to GPU gain 1x64x64	3971%	438%	95%
CPU to GPU gain 10x64x64	16345%	2486%	319%
CPU to GPU gain 100x64x64	13318%	10633%	3906%

Figure 7.10: Rule-set percentage gains from CPU to GPU performance for 64x64 patches



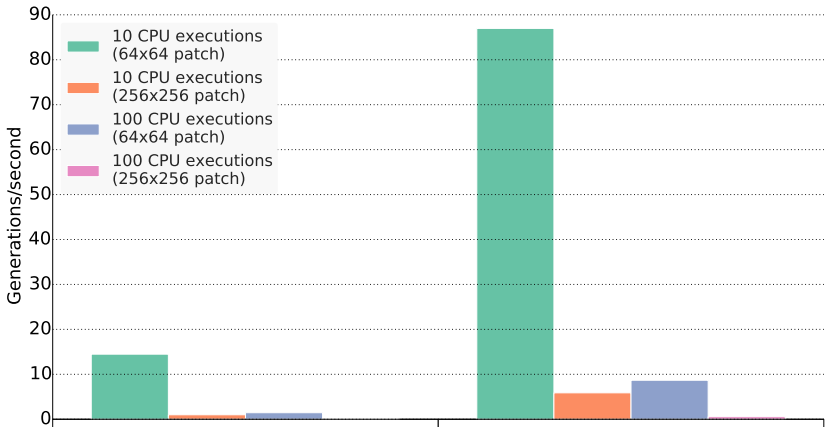
	Large ruleset	Average ruleset	Simple rule
CPU to GPU gain 1x256x256	8805%	1587%	498%
CPU to GPU gain 10x256x256	3813%	7869%	1079%
CPU to GPU gain 100x256x256	8531%	5315%	11591%

Figure 7.11: Rule-set percentage gains from CPU to GPU performance for 256x256 patches

7.1.1.4 Noise performance

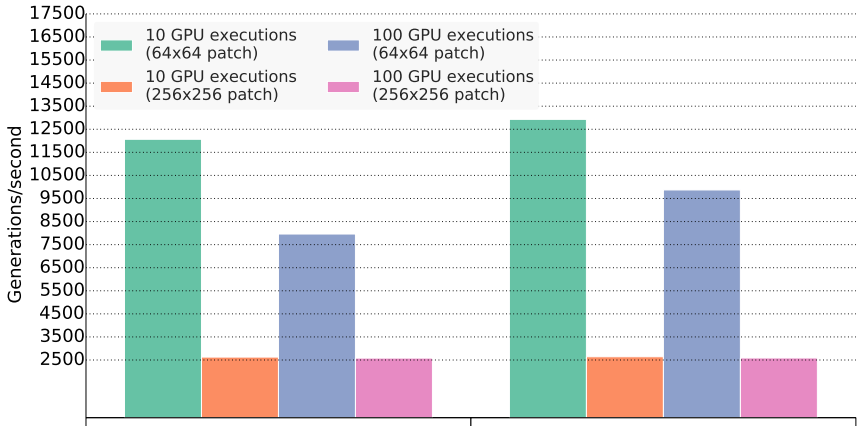
We tested the performance of all permutations of the fBm-inspired noise generators we implemented. We tested using six octaves for each of them - a fairly high and conservative value for testing. We omit including graphs for the full results here as they are quite repetitive. The full raw results are available in the appendix at 4 and 5. What we found was that all the noise types perform very well. For the 64x64 patch size (which is what we have used most commonly in testing), many thousands of noise results can be generated per second. The results are still very good even for a 256x256 patch and real time generation using this size is easily be viable.

As with the terrain rules, we were particularly interested to also see how the system performs compared with a CPU implementation. Porting our entire noise generation code from the GPU would be too time consuming, so we instead used an existing library libnoise[3] which can generate Perlin and Voronoi noise. The library is not multithreaded by default but we made our usage of it multithreaded to try and get as close to the GPU performance as possible. We hard coded a compute shader to match our CPU setup for comparison.



	Voronoi noise	Perlin noise
10 CPU executions (64x64 patch)	14.48	86.97
10 CPU executions (256x256 patch)	0.99	5.87
100 CPU executions (64x64 patch)	1.45	8.67
100 CPU executions (256x256 patch)	0.10	0.57

Figure 7.12: Noise performance on the CPU using libnoise[3]



	Voronoi noise	Perlin noise
10 GPU executions (64x64 patch)	12064.77	12928.75
10 GPU executions (256x256 patch)	2626.03	2640.70
100 GPU executions (64x64 patch)	7959.50	9869.06
100 GPU executions (256x256 patch)	2582.91	2587.87

Figure 7.13: Noise performance on the GPU

There was really no comparison between the two results. For Voronoi noise the GPU was around a thousand times faster. A large speed-up was expected due to the GPU architecture being designed exactly for these kinds of task, but this result does seem excessive. We strongly anticipate that noise could be generated on the CPU considerably faster than what LibNoise achieves even after our adding of multithreading support. We are however confident that even the most optimal CPU solution would still be a number of times slower than on the GPU given these results.

7.1.2 Expressiveness and quality of terrain rules

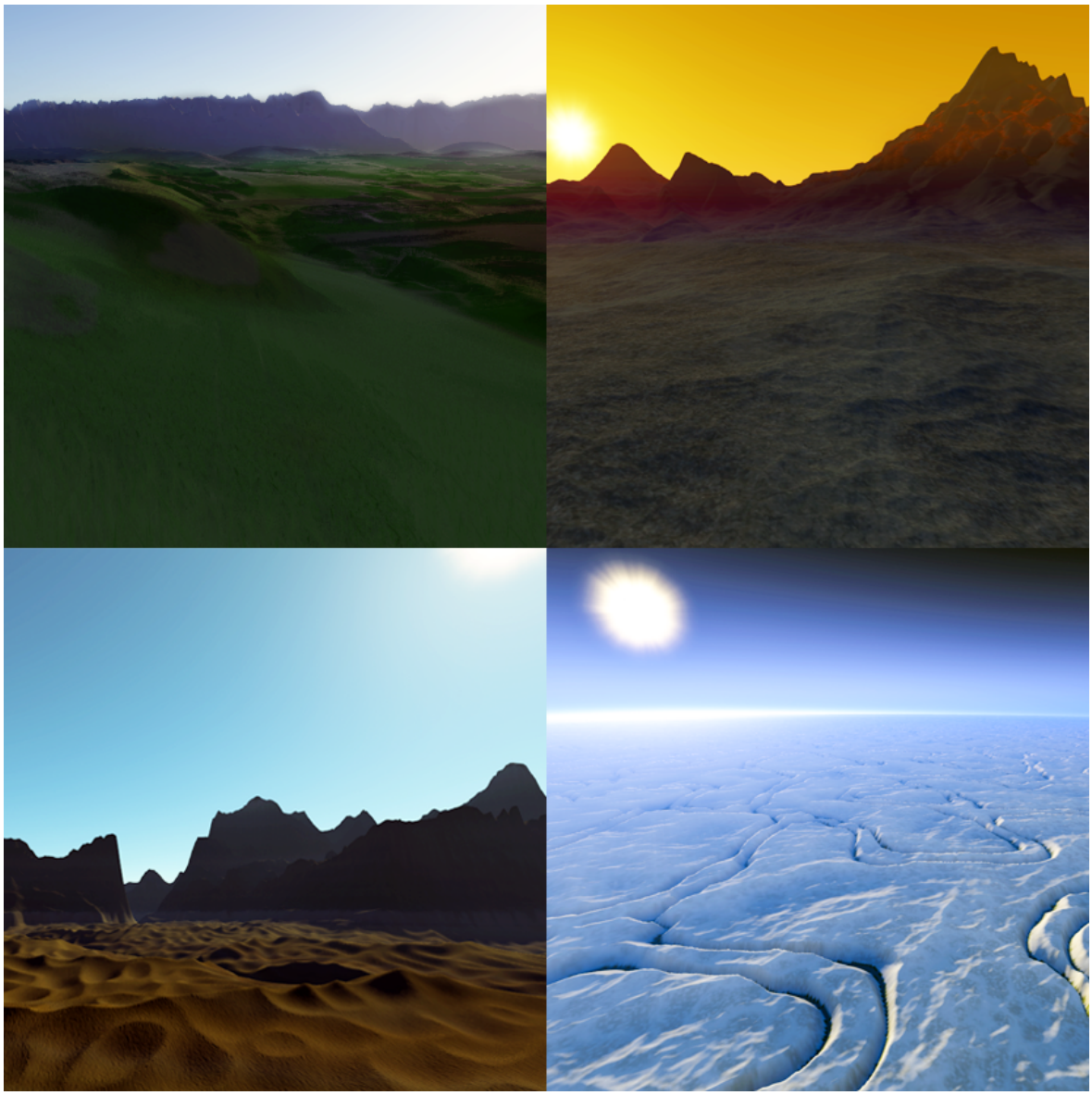


Figure 7.14

Assessing the quality of the visuals we generate is very subjective. It is also important to remember that our system currently has no support for adding models for trees, water and other world objects which greatly limits the overall realism of results. To try and get some idea of how well the system could replicate certain terrain systems, we tasked four users with each creating an archtypical sci-fi world - ice, volcanic, desert and arable. We provided the restriction of not using direct mask input - the system can obviously replicate any terrain if the user just draws it by hand. We also provided a set of suitable materials for them to use.

Figure 7.14 shows the results. Some users got on better than others adapting to the user interface in the short amount of time they spent with the system but we thought the results looked quite impressive considering the terrains are empty of anything beyond billboarded flora. For the arable land, the user managed to create some interesting colour variations and get an overall lush and vibrant look using dense flora, bright materials and low frequency texturing with fractal variation. The volcanic world looks dark and rocky and is helped by a fiery atmosphere created with our data driven system (6.2). Using the height selector (4.5.6), mask selector with procedural noise (4.5.11) and a surface modifier (4.6.4) they were even able to apply some fairly convincing looking lava over a volcano they modelled by displacing fractal results. We were not convinced by the mountains created in the desert world but thought the sand dunes looked really rather realistic. On the ice world the user made great use of the noise selector to clamp the range of a fractal and create interesting channels where rivers may once have run. We particularly liked how this world looked from a high altitude view with the atmosphere gradient they created again using our data driven system.

7.1.3 Ease of use of UI

After creating the terrains in 7.1.2, we asked the users how they found the operation of our user interface. They all managed to use it without issue (admittedly a biased result from them all being engineering students) but the main feature they all wanted was to be able to create rules using a graphical method rather than manually entering settings. For example, they said it would be better if they could just draw a polygon on the terrain rather than working out the coordinates and entering them. We fully agree that this would be an excellent addition in terms of usability of the editor. It is easy to envisage extensions where users can fill mask regions with a paint-bucket like tool for surface modifiers. At its core, the procedural rule system is to creating terrains what vector based drawing is to creating images, so similar kinds of interactive editing tools could be produced. Our testers also had some trouble getting fractals to look exactly how they wanted. We have the data driven tool that can be used but the system would benefit from a templates system with a set of exemplar fractals users can add covering a range of commonly wanted features.

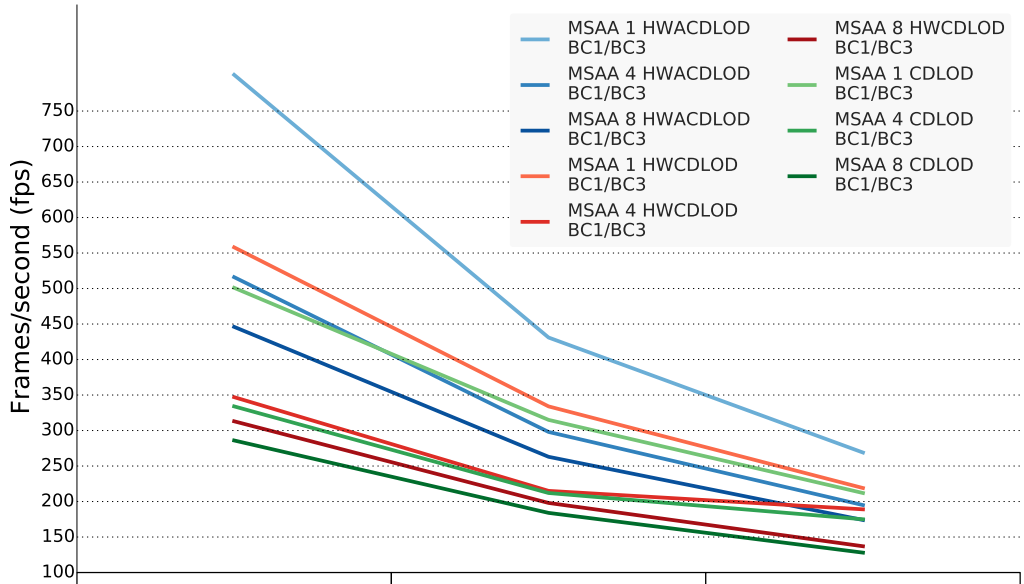
7.2 Graphics engine

In evaluating the graphics engine, our concern is whether or not the techniques in our system can provide a substantial performance and/or quality increase over the techniques they look to supersede.

7.2.1 Framerates

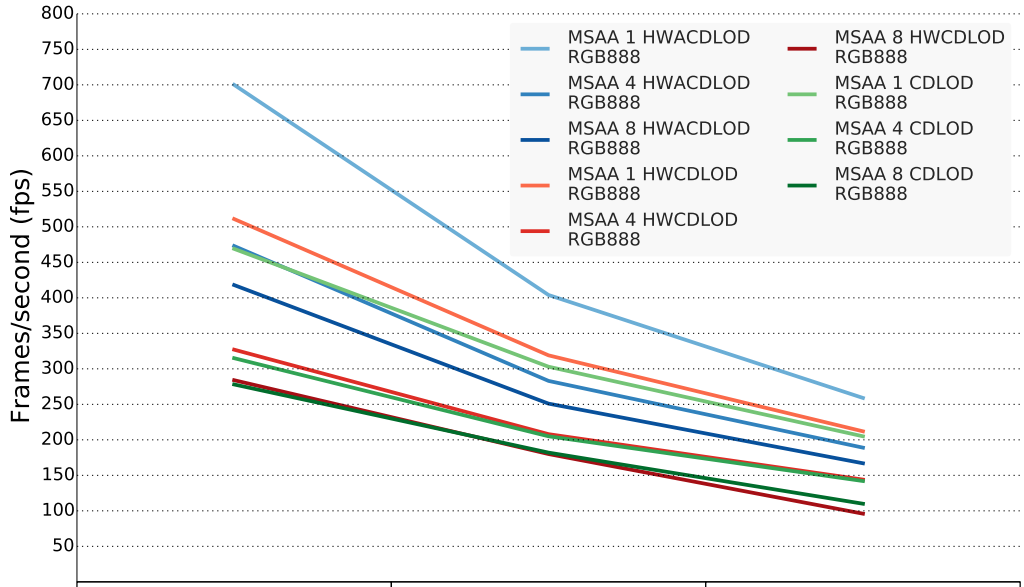
In this first section we look at the overall framerates achieved in the complete system. We look at how performance scales with different resolutions, multisampling levels, different LOD techniques (CDLOD vs our improvements) and different texturing options (rasterisation enabled/disable and texture compression enabled/disabled). The resolutions we test at are 720p, 1080p and 1440p. The multisampling levels used are 1, 4 and 8. Figures 7.15 through 7.17 show the results.

Most well developed games aim to achieve a framerate of 60 frames per second or FPS in order to provide a smooth experience to the user. Although with the outdated hardware found in consoles we can often see developers aiming instead for 30 FPS. When looking at the raw performance numbers, we want to see a considerably higher framerate than this since our rendering only covers the atmosphere and terrain. A real-world usage of our system would likely have many other objects to draw so there needs to be considerable margin.



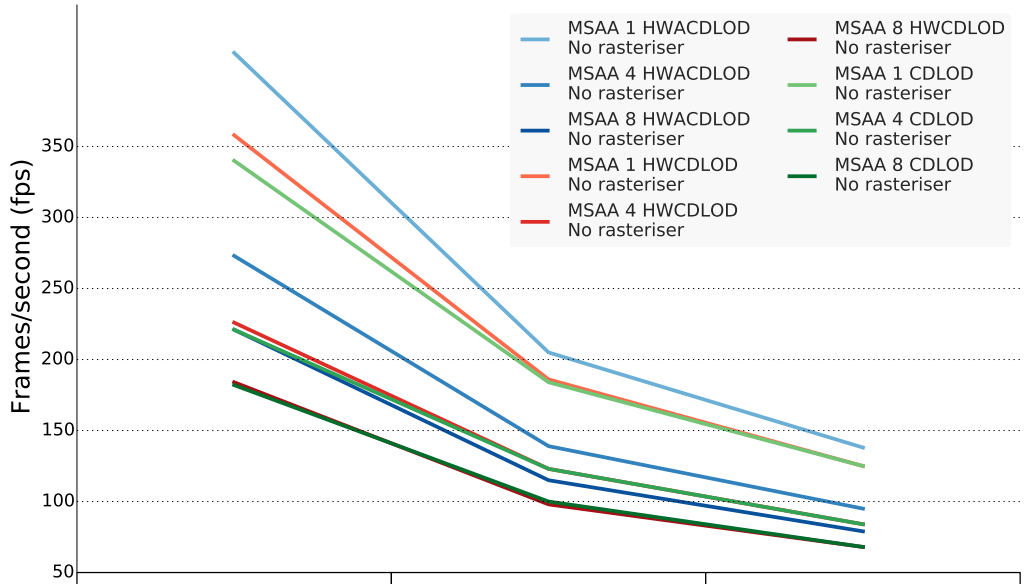
	1280x720	1920x1080	2560x1440
MSAA 1 HWACDLOD BC1/BC3	801.00	431.00	269.00
MSAA 4 HWACDLOD BC1/BC3	516.00	298.00	195.00
MSAA 8 HWACDLOD BC1/BC3	446.00	263.00	174.00
MSAA 1 HWCDLOD BC1/BC3	558.00	334.00	219.00
MSAA 4 HWCDLOD BC1/BC3	347.00	215.00	189.00
MSAA 8 HWCDLOD BC1/BC3	313.00	198.00	137.00
MSAA 1 CDLOD BC1/BC3	501.00	315.00	212.00
MSAA 4 CDLOD BC1/BC3	334.00	212.00	175.00
MSAA 8 CDLOD BC1/BC3	286.00	184.00	128.00

Figure 7.15: Performance across LOD options and multisampling modes with BC1/BC3 compression in the rasteriser stage



	1280x720	1920x1080	2560x1440
MSAA 1 HWACDLOD RGB888	700.00	404.00	259.00
MSAA 4 HWACDLOD RGB888	473.00	283.00	189.00
MSAA 8 HWACDLOD RGB888	418.00	251.00	167.00
MSAA 1 HWCDLOD RGB888	511.00	319.00	212.00
MSAA 4 HWCDLOD RGB888	327.00	208.00	144.00
MSAA 8 HWCDLOD RGB888	284.00	180.00	96.00
MSAA 1 CDLOD RGB888	469.00	303.00	205.00
MSAA 4 CDLOD RGB888	315.00	205.00	142.00
MSAA 8 CDLOD RGB888	278.00	182.00	110.00

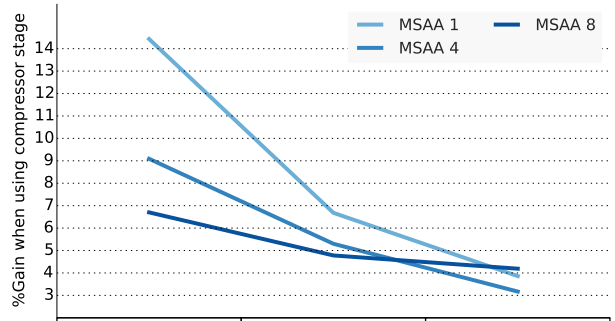
Figure 7.16: Performance across LOD options and multisampling modes



	1280x720	1920x1080	2560x1440
MSAA 1 HWACDLOD No rasteriser	416.00	205.00	138.00
MSAA 4 HWACDLOD No rasteriser	273.00	139.00	95.00
MSAA 8 HWACDLOD No rasteriser	221.00	115.00	79.00
MSAA 1 HWCDLOD No rasteriser	358.00	186.00	125.00
MSAA 4 HWCDLOD No rasteriser	226.00	123.00	84.00
MSAA 8 HWCDLOD No rasteriser	184.00	98.00	68.00
MSAA 1 CDLOD No rasteriser	340.00	184.00	125.00
MSAA 4 CDLOD No rasteriser	221.00	123.00	84.00
MSAA 8 CDLOD No rasteriser	182.00	100.00	68.00

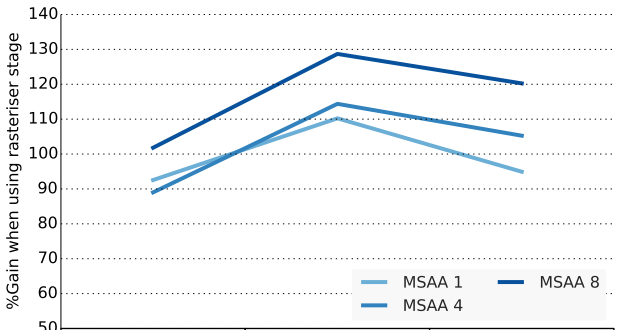
Figure 7.17: Performance across LOD options and multisampling modes with the rasteriser stage disabled (giving 64 samples per pixel for texturing rather than 5)

It is apparent that the system performs well across the board. In particular we pay attention to the results using both the texture compression and rasteriser stages. 1080p is currently by far the popular resolution used for PC games and we see framerates from around 250 to 400 FPS depending on the multisampling level. This is well above 60 FPS and leaves plenty of room for other rendering elements to be added to the system. What is also promising about these results is that they include the use of the relatively expensive atmospheric scattering implementation. A real world use of atmospheric scattering could be with a deferred rendering system whereby, the lighting due to scattering could be applied as a post process once per pixel. Using this method the expensive of using atmospheric scattering should be no worse than what we present here however complex other objects may be. Figures 7.18 and 7.19 show a summary of the percentage gains by using the rasteriser and compressor stages in HWACDLOD.



	1280x720	1920x1080	2560x1440
MSAA 1	14.43%	6.68%	3.86%
MSAA 4	9.09%	5.30%	3.17%
MSAA 8	6.70%	4.78%	4.19%

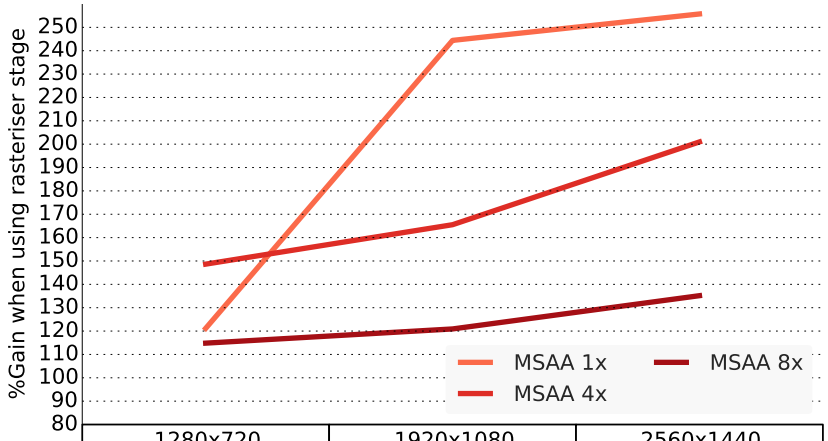
Figure 7.18: Percentage gains by using the rasteriser stage with DXT over no compression.



	1280x720	1920x1080	2560x1440
MSAA 1	92.55%	110.24%	94.93%
MSAA 4	89.01%	114.39%	105.26%
MSAA 8	101.81%	128.70%	120.25%

Figure 7.19: Performance using the rasteriser stage without compression over no rasterisation.

We notice in figure 7.19 a performance decrease when moving from 1080p to 1440p. Also in figure 7.18, the benefits on using DXT compression seem to dimish as the number of samples increases. This is less surprising however as resolution and multisampling increases other factors besides the compression become more dominant in determining performance. However the first result did seem genuinely counter intuitive as the reduced per-pixel cost should provide increasingly higher benefits as resolution and multisampling level increase, especially with so many additional samples being made without the rasteriser. Our immediate suspicion was that this was being caused by the relatively expensive per-pixel atmospheric scattering calculations so we wanted to confirm this by running the tests again this time only rendering the terrain with basic texturing and lighting (T&L). Our suspicions were confirmed and figure 7.20 shows that we do indeed get increasingly higher gains as resolution and multisampling level increase when rendering only the terrain with basic lightning. We do not include the raw results but performance roughly doubles across all results without using the scattering part of the shader.

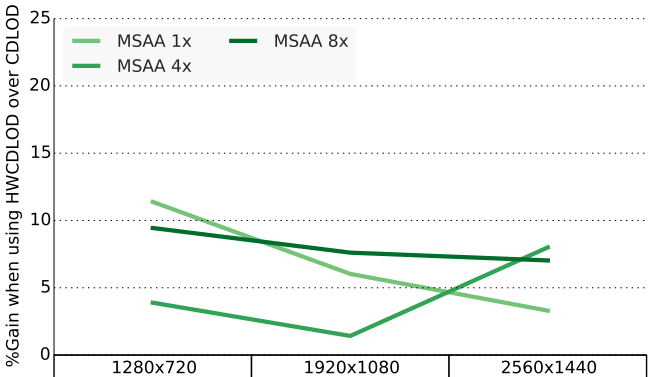


	1280x720	1920x1080	2560x1440
MSAA 1x	121.16%	244.42%	255.73%
MSAA 4x	148.65%	165.53%	200.96%
MSAA 8x	114.83%	120.91%	135.15%

Figure 7.20: Performance gains using by using the rasteriser stage with only the terrain being rendered using simple T&L

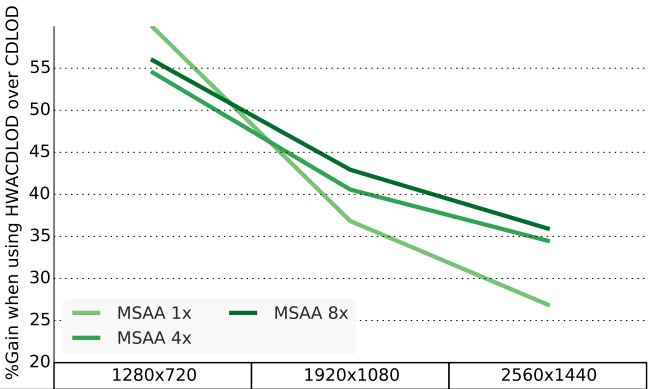
Finally we look at the performance gain percentages between the original CDLOD algorithm and our hardware tessellated versions. We see in figure 7.21 that just switching to hardware tessellation over the standard vertex shader tessellation CDLOD gives a slight performance gain. This happens because with the hardware tessellation, patches with regions blended inbetween LODs actually have reduced geometry. In the original CDLOD implementation patches have their geometry morphed but it still contains the same number of triangles. The adaptive HWACDLOD implementation gives around a 60% increase at 720p down to a 30%

increase at 1440p. We feel this is a good gain for the slight drop in visual quality that resulted with our adaptive settings. The gain trails off as resolution increases because the pixel shader stage becomes more relevant to the overall performance. Reducing the terrain geometry by adaptive tessellation reduces the geometry before reaching the pixel shader stage but we ultimately have to shade approximately the same number of pixels.



	1280x720	1920x1080	2560x1440
MSAA 1x	11.4%	6.0%	3.3%
MSAA 4x	3.9%	1.4%	8.0%
MSAA 8x	9.4%	7.6%	7.0%

Figure 7.21: Performance gains using HWCDLOD over CDLOD



	1280x720	1920x1080	2560x1440
MSAA 1x	59.9%	36.8%	26.9%
MSAA 4x	54.5%	40.6%	34.5%
MSAA 8x	55.9%	42.9%	35.9%

Figure 7.22: Performance gains using HWACDLOD over CDLOD

7.2.1.1 HWACDLOD vs HWCDLOD vs CDLOD index pattern benefits

As well as the performance increases we found from using hardware tessellation previously, there are some subtle visual improvements. In the original CDLOD implementation, all triangles are oriented in the same direction. This means there is a worst case scenario for visual artifacts when the terrain has hard lines running orthogonal to that direction (see figure 7.23). The result of this is a stepping effect and harsh results for the calculated normals along the line. While CDLOD can be adapted to use an alternating indexing pattern by using a slightly more complex calculation in the vertex shader, hardware tessellation provides an even better pattern for free. The advantage of this different pattern can be seen in figure 7.24. While we can still see some normal artifacts along the hard edge, the rate of occurrence is halved. On the other hand, we also remove the best case scenario when the terrain runs parallel to the direction of triangle alignment. On the whole though, we feel it is better to improve the worst case as it is quite obvious when it occurs.

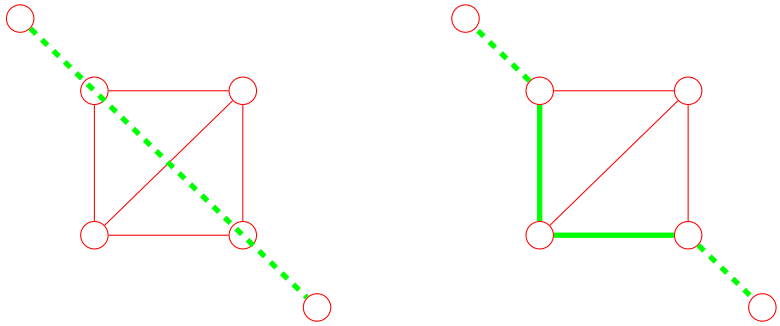


Figure 7.23: Worst case scenario for terrain shape versus indexing pattern. The ideal terrain shape (left) gets altered to the stagged effect (right) of the indexing pattern.

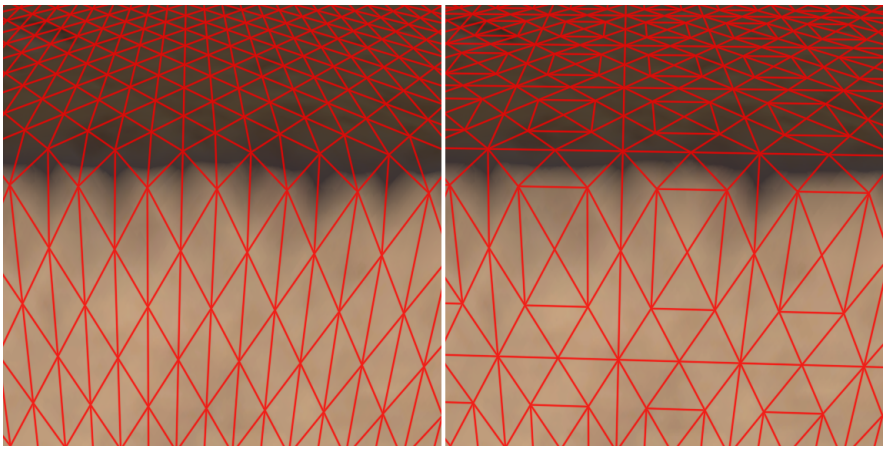


Figure 7.24: Worst case for terrain shape in HWACDLOD (right) gives less artifacts than in CDLOD (left)

7.2.1.2 HWACDLOD vs HWCDLOD vs CDLOD visual quality

We discussed in 7.2.1.1 the subtle advantage the different tessellation pattern produced by hardware tessellation can give over the standard CDLOD implementation. Now we compare the actual visual results. Figure 7.25 shows a comparison between the three techniques. We notice hardly any difference between CDLOD and HWCDLOD, under close inspection top of the volcano appears slightly smoother with HWCDLOD. Using HWACDLOD there is a noticeable reduction in quality, most visible on the mound in the centre of the screen at the base of the volcano (readers may want to view the electronic version to see this). However, the difference is small for what can be up to a 50% performance increase (7.2.1). The parameters for the adaptive tessellation could obviously be adjusted to give more or less detail as well. This will give better or worse visuals while decreasing or increasing the framerate.

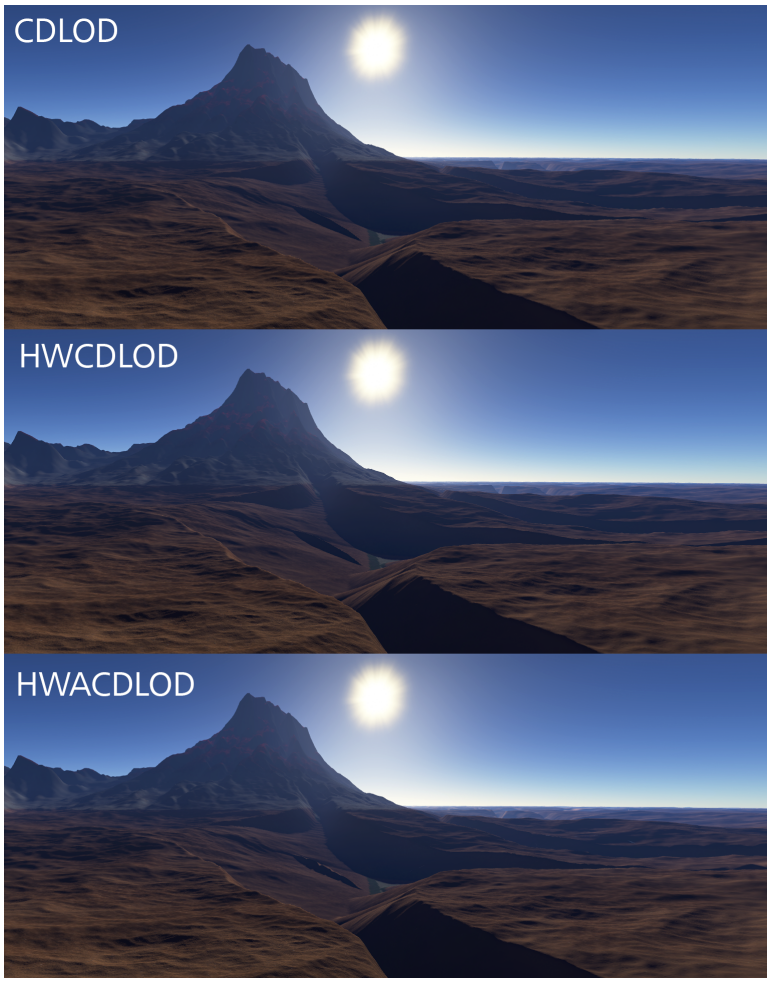
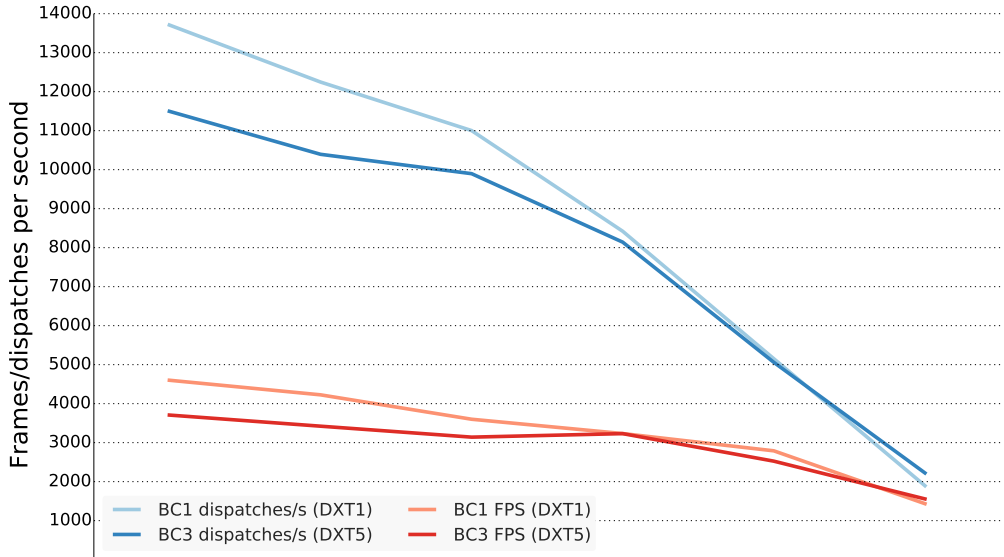


Figure 7.25: Comparison between visuals for LOD implementations

7.2.1.3 GPU BCn compression

Performance

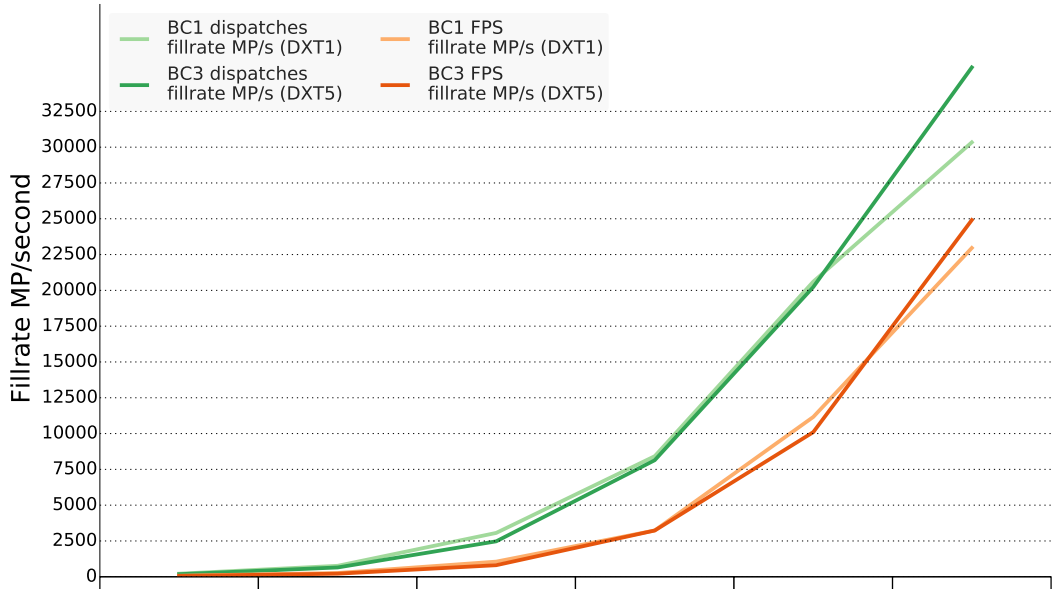
We perform two tests concerning our compute shader implementation of BCn compression (5.3). The first test concerns the theoretical fill-rate of our compute shaders for BC1 and BC3 by using DirectX queries (as described in 7.1.1.1) to time the shader execution time. The second test concerns the real-world fill-rate of the shaders by stressing our actual compressor class implementation and thereby including any additional GPU and CPU overheads introduced by it. Figure 7.26 shows the results for these tests and figure 7.27 shows the calculated fillrates from these values.



	128x128	256x256	512x512	1024x1024	2048x2048	4096x4096
BC1 dispatches/s (DXT1)	13709.00	12247.00	11002.00	8420.00	5152.00	1896.00
BC3 dispatches/s (DXT5)	11496.00	10394.00	9897.00	8141.00	5059.00	2223.00
BC1 FPS (DXT1)	4600.00	4226.00	3600.00	3233.00	2790.00	1435.00
BC3 FPS (DXT5)	3707.00	3422.00	3141.00	3231.00	2525.00	1557.00

Figure 7.26: BCn compute shader performance showing theoretical dispatches/second and actual performance in the compressor stage

We see very high performance in both the more contrived dispatch rate and realistic frame rate measures. Even at 4096x4096 the shader executes in a time negligible compared to the calculation of complex rules and so fits seamlessly into our system with regards to performance. In figure 7.27 we convert the raw frames and dispatches per second into fillrate values for megapixels calculated per second.



	128x128	256x256	512x512	1024x1024	2048x2048	4096x4096
BC1 dispatches fillrate MP/s (DXT1)	214.20	765.44	3061.75	8420.00	20608.00	30336.00
BC3 dispatches fillrate MP/s (DXT5)	179.62	649.62	2474.25	8141.00	20236.00	35568.00
BC1 FPS fillrate MP/s (DXT1)	71.88	264.12	1056.50	3233.00	11160.00	22960.00
BC3 FPS fillrate MP/s (DXT5)	57.92	213.88	807.75	3231.00	10100.00	24912.00

Figure 7.27: BCn compute shader fillrates in megapixels/second for dispatches/second and actual performance in the compressor stage

Quality of results

We now look briefly at the quality of results obtained for the BCn compression methods. Figure 7.28 shows the source diffuse and normal textures we used.

Figure 7.28: Source diffuse and normal textures

Figure 7.29 shows a close-up of the resulting compression of the source diffuse texture using BC1 compression. There is a visible loss of quality this close but at normal distances the result looks fine and for most textures indistinguishable from the source.

Figure 7.29: Source close-up (left) and BC1 compressed image (right)

Figure 7.30 shows the results for compressing the normal map. Using BC1 produces an unacceptable quality of result. The block compression of BC1 is very visible and coloured squares are apparent. BC1 is faster for rendering but the lighting looks blocky if this quality of map is used. Using the swizzled BC3 compression, the result is much more usable. We get a slightly different result to the original but in use it appears smooth and is barely distinguishable from the original. This occurs for the reasons discussed in the compressor implementation (5.3).

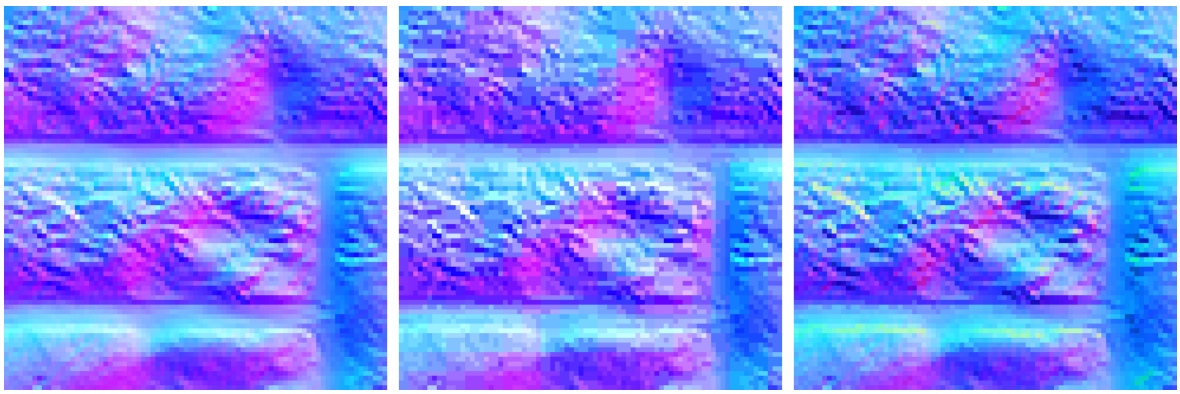
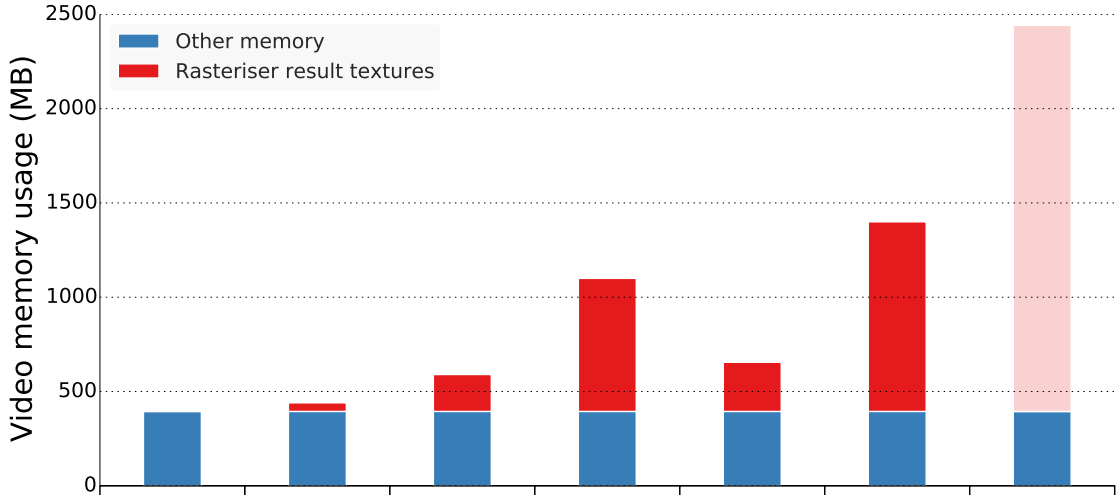


Figure 7.30: Normal map uncompressed (right), compressed with BC1 (middle) and compressed with swizzled BC3 (right)

7.2.2 Memory consumption with and without rasteriser stage

We measured the actual memory consumption by the program with different rasteriser patch sizes and different compression options to ensure it conforms to the reduction we would expect and that there are no hidden costs. We detail BC1 and BC3 compression in 5.3, but in summary for our test setup we should get a 5x memory decrease since BC1 reduces our usage by a factor of 6 and BC3 by a factor of 4 and we have equal amounts of each. Looking at just the additional memory used over the base amount in 7.31, we can see that this is more or less exactly what we see. We were not able to run the program with 512x512 R8G8B8 patches as it exceeded the 2GB memory on our GPU. If the system were actually to be used, the best idea would be to adjust the rasterisation patch size depending on the available video memory.

The base amount of memory used before the rasteriser stage is quite high at around 400MB - more than would be wanted in real world use. This is partly due to pre-allocating memory for all patches in the texture cache. Using sparse textures (DirectX tiled resources) may be a good choice for helping this slightly. We did want to implement this but lack of full support for this new feature on our GPU made it unappealing for now. The memory used is also high since we open up all components of the terrains material at load time for use by the rasteriser. Loading these textures as required could reduce overall memory consumption but might impact performance slightly.



	Rasteriser distabled	128x128 BC1/3	256x256 BC1/3	512x512 BC1/3	128x128 RGB888	256x256 RGB888	512x512 RGB888
Memory usage (MB)	394	440	590	1100	655	1400	>2000

Figure 7.31: Memory usage across various configurations

7.2.3 Draw calls, pipeline utilisation and load balancing

To ensure our draw call system works and to get an overall look at our pipeline usage, we used the NVIDIA Nsight tool. The tool has a wide number of features but among them is the ability to profile graphics applications and see where the majority of the GPU time was spent. We would hope to see a low number of draw calls with a low amount of time spent on them with the majority of time being spent on the rendering itself. Rendering occurs when the present function is called on the IDXGISwapChain instance, so we look for calls to this function. Figure 7.32 shows a trace for traversing a simple terrain and figure 7.33 for traversing a complex terrain. We set a constant speed of 1000m/s which requires a high amount of generations per second for the system to keep up with the camera, putting it the generation pipeline under stress.

Name	Count	Capture Time %	Total Time (µs)	Min (µs)	Avg (µs)	Max (µs)
FinishCommandList	12	0.00	390.712	23.229	32.559	80.747
Map	116142	1.44	677,695.296	0.388	5.835	73,040.001
UpdateSubresource	480	0.08	37,562.325	6.944	78.254	2,600.254
Unmap	116142	0.38	179,669.960	0.369	1.546	721.586
Dispatch	105518	0.19	90,448.948	0.524	0.857	234.514
CopySubresourceRegion	52739	4.35	2,051,112.757	5.457	38.891	18,918.222
CopyResource	17615	1.86	876,054.269	4.577	49.733	36,064.857
Draw	35539	0.12	57,053.666	0.437	1.605	18.984
ClearRenderTargetView	6419	0.02	10,892.795	0.855	1.696	20.990
ClearDepthStencilView	6419	0.01	5,661.266	0.614	0.881	349.196
DrawIndexed	6419	0.03	13,854.558	0.820	2.158	18.561
Present	6418	49.07	23,126,452.862	215.926	3,603.373	27,899.246
DrawIndexedInstanced	31832	0.09	42,316.418	0.411	1.329	21.551

Figure 7.32: API call summary while traversing a simple terrain

Name	Count	Capture Time %	Total Time (µs)	Min (µs)	Avg (µs)	Max (µs)
FinishCommandList	12	0.00	400.641	22.661	33.386	79.148
Map	122012	1.16	912,603.051	0.368	7.479	102,948.628
UpdateSubresource	660	0.07	51,550.842	6.873	78.107	2,743.988
Unmap	122012	0.36	282,428.307	0.369	2.314	720.705
Dispatch	469102	0.45	354,431.423	0.494	0.755	339.301
CopySubresourceRegion	59263	4.51	3,530,650.623	5.022	59.575	34,018.731
CopyResource	19787	0.50	390,865.185	4.744	19.753	49,054.742
Draw	39891	0.09	68,123.949	0.391	1.707	602.093
ClearRenderTargetView	5989	0.01	9,858.018	0.938	1.646	18.852
ClearDepthStencilView	5989	0.01	5,365.043	0.606	0.895	88.553
DrawIndexed	5989	0.02	13,614.747	0.917	2.273	95.169
Present	5988	52.91	41,457,147.948	239.496	6,923.371	47,382.451
DrawIndexedInstanced	29955	0.05	42,246.135	0.402	1.410	104.429

Figure 7.33: API call summary while traversing a complex terrain

We get the results we would expect here. A low number of draw calls as a result of our

instancing system (5.4), the vast majority of the time being spent on rendering via Present() calls and an increase in time spent on dispatch calls between the simple and more complex terrain configurations. One thing we do note here is a high number of present calls despite a low time spent on them. An interesting future experiement may be to try and compress the generation of multiple patches into a single call. The practicalities of this may be difficult to realise though since such a shader could not be generated on the fly.

7.3 Data driven elements

7.3.1 Quality of noise matches to elevation data

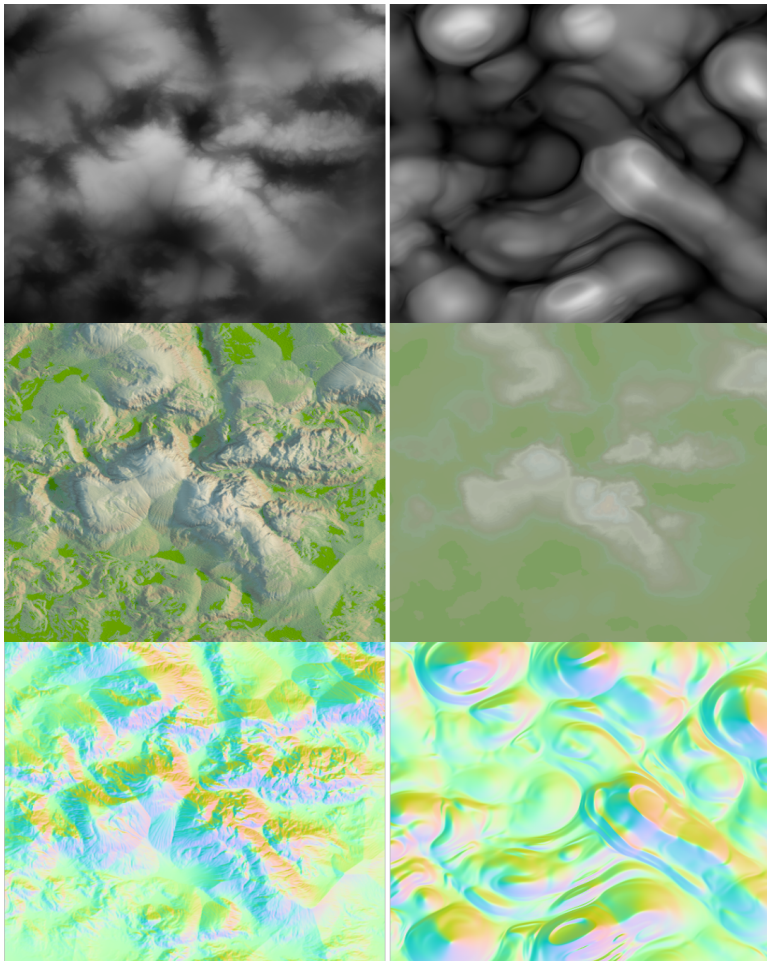


Figure 7.34: Example generated matches for noise result, reflectance result and gradient result

In this section we look at the data driven rule generation to see how closely the output can match the source data. In our main test shown in figure 7.34 we have a very strong statistical

match according to the histogram display on the UI (we showed these histograms in figure 6.2 and figure 6.3). The noise result looks close in terms of frequency matching and the 1D texture ramp shown expanded in figure 7.35 looks a good match to the source reflectance. The normal map is also a strong statistical match and looks reasonable to the eye as well.

Figure 7.35: Colour ramp generated by the tool for the example in figure 7.34

Unfortunately, when we add these rules to the terrain as shown in figure 7.36 the results are not very compelling. The noise result with the strongest statistical match has a very high octave count that does not translate well to the scale of the terrain. We can tone down this octave count to improve the result slightly but really it still does not resemble the original terrain that closely.

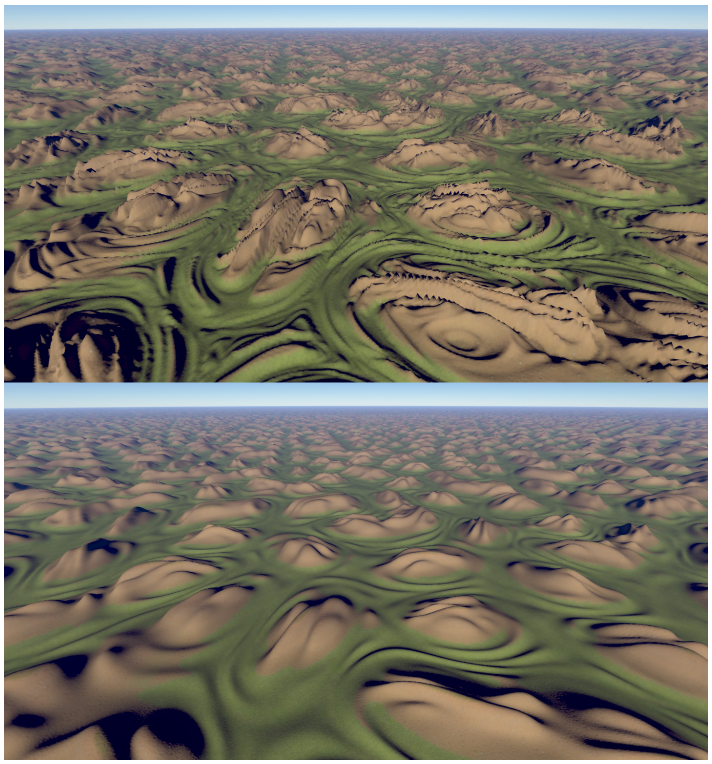


Figure 7.36: Data driven rules added to the terrain. Original (top) and adjusted (bottom)

We ran the tests again as shown in figure 7.37 and found the same results. The noise and gradient results were a slightly worse statistical match with the frequency and reflectance matching still working very well. The reflectance ramp gave results that look almost identical to the original. However, the inserted rules again do not really resemble the original terrain.

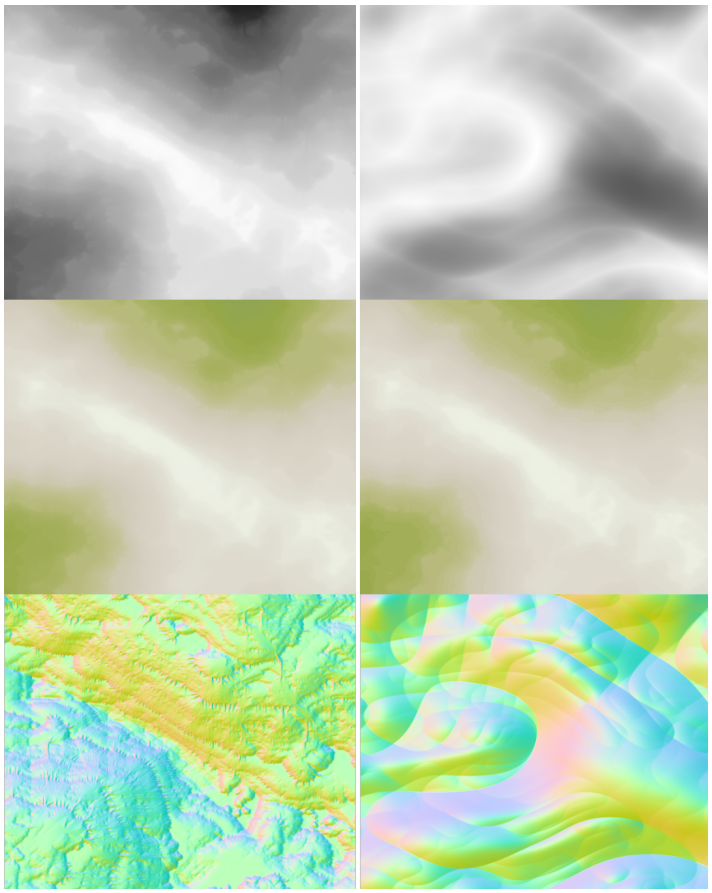


Figure 7.37: Second example generated matches for noise result, reflectance result and gradient result

We tested a variety of other sources and found similar results. The quality of the statistical match ultimately depends on the ability of the implemented noise generators to create a match. On the whole, we think the results for the data driven tool form a good basis for future work which for which we make some suggestions in 8.1. For now though, the tool would not be ready for real world use. It really needs something more than frequency matching combined with histogram matching to make for usable results. Of course, one other easy way we could get closer matches would be to add more noise types. A more intelligent system that combines multiple fractal results is likely to work better and have greater potential though.

7.3.2 Data driven recolouring of precomputed atmospheric scattering tables

We were particular pleased with the results of the atmospheric scattering recolouring (6.2). For the kind of inputs we would expect users to employ, we got precisely the results we would have hoped for. This does not mean the system is flawless though. It works best when the input gradient is dual colour or almost dual colour with some slight variation. When more hard changes in colour are introduced the quality of the result diminishes due to in part to precision problems with the tables. An example of this is shown in the final image of our samples from figures 7.38 through 7.40 where we have a gradient from blue to yellow back to blue. The result is still usable, but it is possible to completely break the system by providing entirely unrealistic inputs. We show this in figure 7.41 where we input a gradient covering the whole spectrum.

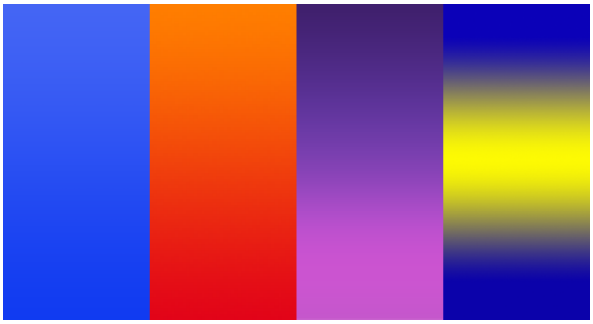


Figure 7.38: Input gradients for recolouring

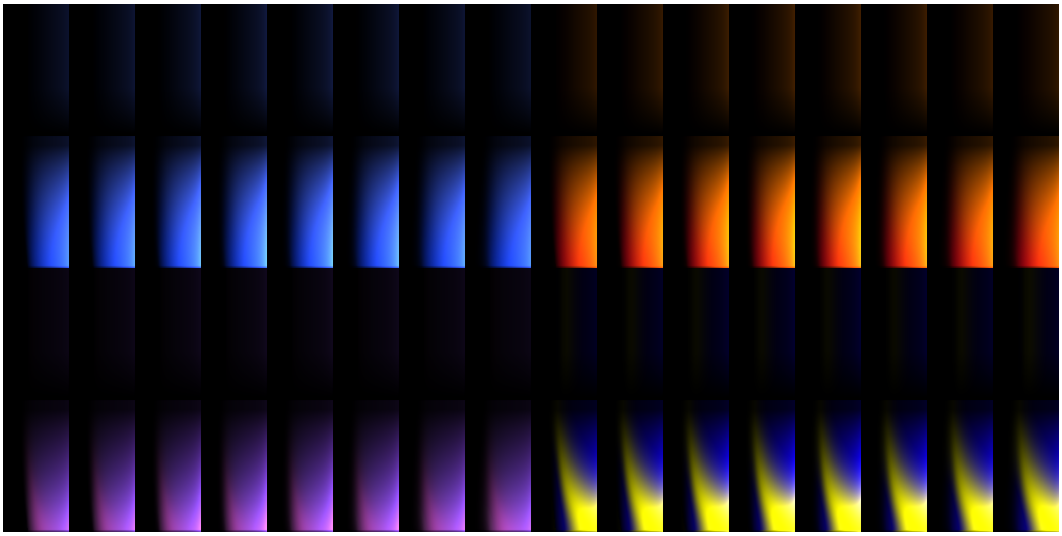


Figure 7.39: Resulting inscattering tables from recolouring

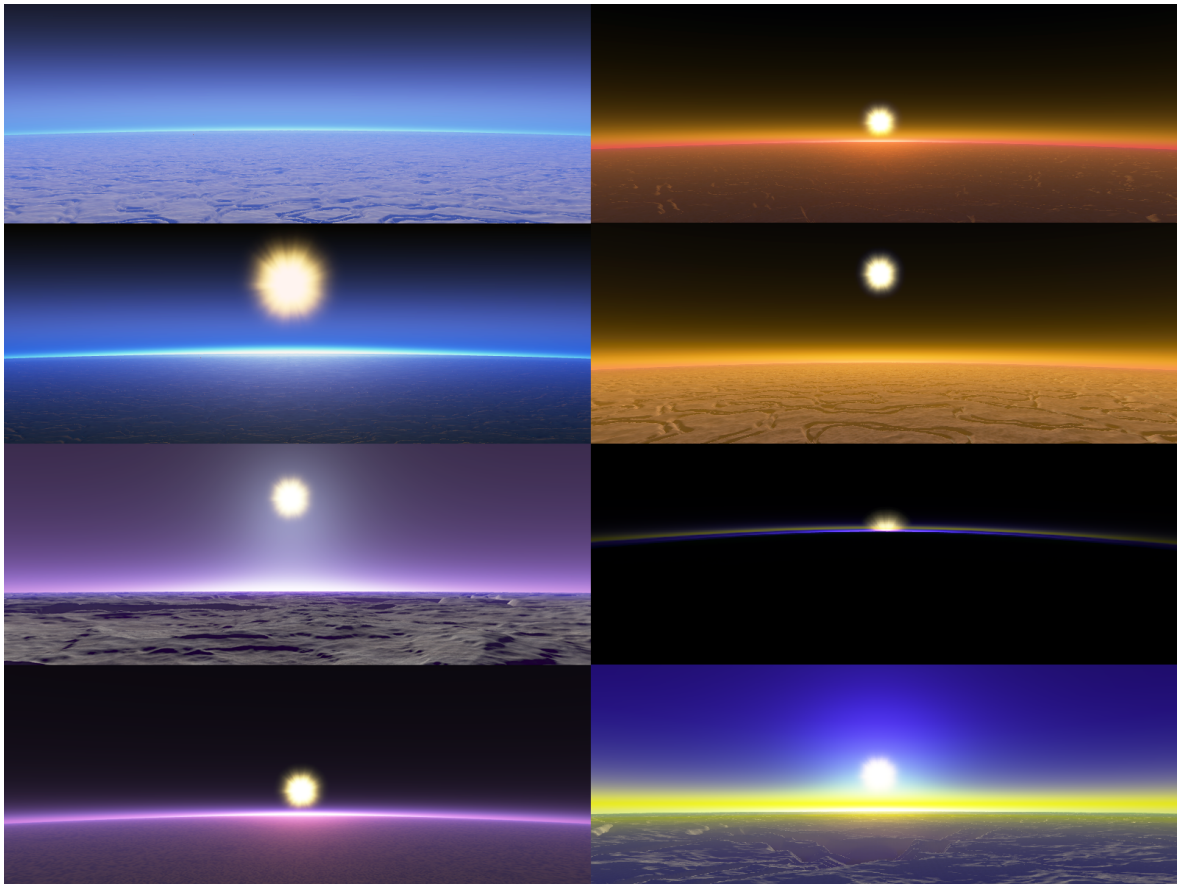


Figure 7.40: Final atmosphere renders with recoloured tables and functions

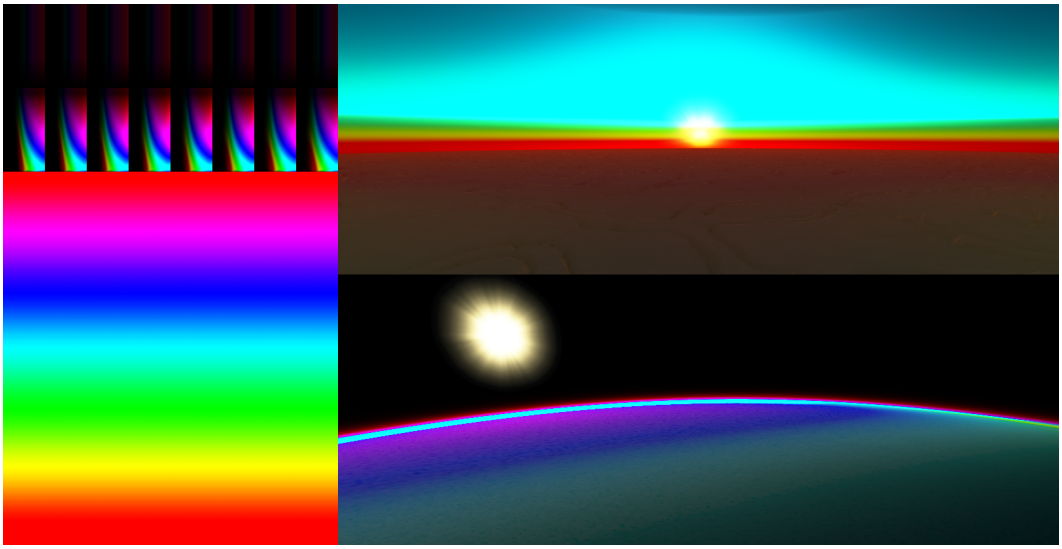


Figure 7.41: Breaking the recolouring system with exotic input gradients

8 Conclusions

We set out to create a rule-based procedural terrain system that generates on the GPU which both performs well and offers a high degree of expressiveness in its ability to recreate a wide range of landscapes. Our main goal along with this was to create a graphics back-end that supports high performance and high quality rendering of the generated terrains. We also wanted to take a look at how data based techniques could help drive the terrain system, and how atmospheric scattering algorithms might be artistically parametrised using user provided data. Finally, we wanted the system to have a working UI that let users design terrains with ease.

We implemented a wide variety of procedural rules that were able to capture a range of different landscape styles when tested by our users. We also provided the option though the rule system to statically load or stream in directly created user content via the noise selector and displacement modifier rules. This means that even when the base system may not be expressive enough to represent a particular terrain, the user can simple go ahead and create exactly what they want and apply it over a certain area using the mask system.

The terrain system also proved highly performant. In real terms, we saw roughly a 5x to 10x increase over the CPU implementation we built for evaluation purposes. In theoretical terms the performance was considerably faster even than this - power which could potentially be exploited in the future through further optimisation.

Our graphics back-end also proved successful, with our implementation providing roughly a 2x increase over the techniques it looked to improve upon. This was not entirely without consequence, our solution provides a slightly lower quality render when using its fastest settings due to the use of compression and lower geometry detail from the HWACDLOD algorithm we developed. It also has higher video memory consumption, but this did allow us to support an unlimited number of textures on the planetary surface through the rasterisation and compression stages.

The user interface was praised by the testers, but it also has plenty of room for improvement. Most of all users wanted to see it become more interactive with 3D editing tools more akin to what is seen in vector graphics software.

Finally the data driven aspect of the project highlighted the potential such a system could have had it been a project on its own. Our work is a first step in what could prove a very useful technique as the graphics requirements of modern applications exceeds what is achievable through purely art-based efforts. We were particularly pleased with the artistic atmospheric

scattering parametrisations, and feel it could be used in real world projects as is to give realistic looking lighting effects, derived from real physics but with more user control than previous solutions offered.

On the whole, we are happy with how the project developed. Our major issue was trying to manage the overall scope - indeed the implementation ended up far too large to conceivably talk about all of it in this report and there was constant opportunity and temptation to deviate from the original plan as new ideas presented themselves. As a result of this, it is perhaps fair to say that no one element of the project was developed to its absolute fullest, but at the same time it was always the goal to produce a whole system rather than an unusable technical demo.

8.1 Future work

8.1.1 Extension of rules, noise types and palettes

The currently implemented rules proved to be quite powerful and expressive but there is room to build on it further with the main structure now in place. Such extensions may include:

- The most obvious omission from the current system in terms of producing a realistic terrain is the absence of objects such as rocks and trees. Adding in rules that generate information for the placement of such items would be as simple as adding the parameter to the compute shader buffer properties and creating new rules to modify its value. The challenge in implementing this would be how to take this generated information and efficiently render potentially huge numbers of objects.
- Another omission that currently limits the realism of the renders is a lack of water bodies. For this we could add new modifier rules and terrain properties. One simple embodiment of this idea would be to add the water height as an extra property and create a modifier rule to adjust its value. The treatment of this value after generation would be simply to render water at the calculated height, by for example, reusing the base terrain patch system with different shaders and displacements. The problem with this is that there could be discontinuities without careful guidance from the user. A example of a more complex method may be to assign areas wetness values, and either by a technique on the GPU or after download onto the CPU calculate where bodies of water would naturally form given the geometry of the terrain. Such ideas may be very difficult to implement given that only a small portion of the entire terrain is generated at a given time.

- An idea I would have particularly liked to investigate in this project given more time is implementing hydraulic erosion to simulate the wearing of terrain over time by weather systems. As with the possible advanced water body generation techniques discussed previously, the main problem with trying to achieve this is that the entire terrain is not generated at any one time to simulate the carrying of particles from one area of the terrain to another. There may be ways to work around this such as generating out to a wide enough area that no particles further than that area could conceivably reach the area being calculated for, but this may be hard to realise.

8.1.2 Further optimisation of rule generation

One downside of the current system is that highly complex rule-sets take a considerable amount of time to compile. We added DirectX 11.2 support to the engine in the hope of employing a new feature which allows the creation of HLSL libraries. It was our hope that this feature could help drastically reduce our compile times by pregenerating assembly code for rules and noise generation and then linking to that code when compiling rule instances. It turned out that the current implementation for generating HLSL libraries does not support all the features our terrain rules require. As such, we abandoned this idea soon after initial attempts at making a rule library showed it would not be fully realisable. If Microsoft can add more features to the library compilation stage, it could help cut compile times for the system considerably. At the moment, for terrains with a large number of complex rules the only option for real world usage would be to precompile the shaders and store them in another format to save the compilation time. This is a trivial extension to implement and would still produce very small terrain files versus a DEM file.

8.1.3 Extension of data inference

Although the data-driven element of the project proved to be successful as something of a proof of concept, there is vast scope for extension that could easily form the subject of a separate project entirely or possibly even a PhD. The ultimate vision for the system is to be able to input an arbitrary photograph and produce rules which create a similar looking, but not exact copy of the terrain. This might be achievable with sufficiently advanced computer vision techniques, some components of which may include:

- For arbitrary photograph, determining the approximate depth of a terrain and analysing the features to create close matching rules. This could be made more complex than finding the closest matching single noise type as in the current implementation. It could look at stacking multiple noise types and give more emphasis to trying to match features rather than just statistically.

- Picking out and recognising objects such as trees, rocks and bodies of water. Along with the suggested extensions to the rules to include modifiers for water, object placement etc, the system could try to replicate such objects.
- Performing texture synthesis to generate new textures for entry into the material palette rather than just performing matching.

Appendix A

A.1

```
[Directories]
dataDirectory=D:/dev/jengine3/Data/
shaderDirectory=Shaders/
compiledShaderDirectory=CompiledShaders/
textureDirectory=Textures/
scriptDirectory=Scripts/
binaryDirectory=Binary/

[Graphics]
terrainGenerator=TG_GPU
numberOfGeneratorThreads=4
useSuperThreading=0
terrainSize=4194304
terrainType=planar
tileSize=64
lodBias=6.0
cdlodBlendStart=0.625
cdlodBlendEnd=0.775
tessellationEnabled=true
useAdaptiveTessellation=false
preloadThreshold=1.1
lruGracePeriod=500
lruTargetSize=2000
calculateTerrainNormals=true
frameInterval=0
multisampleQuality=0
multisampleLevel=4
vsyncEnabled=false
useTriplanarTexturing=true

prebakeSize=512
prebakeUseCompression=true
prebakeCompressionType=BC1BCN

pasUseDataDriven=true
pasHR=4.0
pasBetaRR=5.8e-3
pasBetaRG=1.15e-2
pasBetaRB=3.31e-2
pasHM=1.2
pasBetaMScaR=4e-3
pasBetaMScaG=4e-3
pasBetaMScaB=4e-3
```

```
pasMieG=0.8

showDebugInfo=true
showAtmosphere=true

eyeXStart=1000
eyeYStart=1000
eyeZStart=0
lookXStart=-1000
lookYStart=0
lookZStart=0

[Benchmarks]
benchmarkW=1920
benchmarkH=1080
noiseTests=true
ruleTests=true
compareToCPU=false
repeats=1
tests=100
```

Listing 1: INI file for program settings

```
dcl_globalFlags refactoringAllowed
dcl_constantbuffer cb0[1], immediateIndexed
dcl_resource_structured t0, 60
dcl_uav_structured u0, 60
dcl_input vThreadGroupID.xy
dcl_input vThreadIDInGroup.xy
dcl_temps 5
dcl_thread_group 20, 20, 1
ishl r0.xy, vThreadGroupID.xyxx, l(4, 4, 0, 0)
iadd r0.xy, vThreadIDInGroup.xyxx, r0.xyxx
iadd r0.xy, r0.xyxx, l(-2, -2, 0, 0)
utof r0.zw, r0.xxxy
mad r0.zw, r0.zzzw, cb0[0].zzzz, cb0[0].xxxy
dp2 r0.z, -r0.zwzz, -r0.zwzz
sqrt r0.z, r0.z
mad r0.z, -r0.z, l(0.000707), l(1.000000)
max r0.z, r0.z, l(0.000000)
mov r0.w, l(0)
mov r1.x, l(0)
loop
  uge r1.y, r1.x, l(10)
  breakc_nz r1.y
  iadd r1.x, r1.x, l(1)
```

```
24   mov r0.w, r0.z
25 endloop
26 mov r1.xyz, r0.wwww
27 ult r0.zw, l(0, 0, 1, 1), vThreadIDInGroup.xxxy
28 ult r2.xy, vThreadIDInGroup.xyxx, l(18, 18, 0, 0)
29 and r0.z, r0.z, r2.x
30 and r0.z, r0.w, r0.z
31 and r0.z, r2.y, r0.z
32 if_nz r0.z
33   ishl r0.y, r0.y, l(8)
34   iadd r0.x, r0.x, r0.y
35   ld_structured_indexable(structured_buffer, stride=60)(mixed,mixed,mixed,mixed
         ) r2.xyzw, r0.x, l(12), t0.yzwx
36   ld_structured_indexable(structured_buffer, stride=60)(mixed,mixed,mixed,mixed
         ) r3.xyzw, r0.x, l(28), t0.yzwx
37   ld_structured_indexable(structured_buffer, stride=60)(mixed,mixed,mixed,mixed
         ) r4.xyzw, r0.x, l(44), t0.xyzw
38   mov r1.w, r2.w
39   store_structured u0.xyzw, r0.x, l(0), r1.xyzw
40   mov r2.w, r3.w
41   store_structured u0.xyzw, r0.x, l(16), r2.xyzw
42   mov r3.w, r4.x
43   store_structured u0.xyzw, r0.x, l(32), r3.xyzw
44   store_structured u0.xyz, r0.x, l(48), r4.yzwy
45 endif
46 ret
```

Listing 2: Circle selector compute shader disassembly (inner mode)

```
1  dcl_globalFlags refactoringAllowed
2  dcl_constantbuffer cb0[1], immediateIndexed
3  dcl_resource_structured t0, 60
4  dcl_uav_structured u0, 60
5  dcl_input vThreadGroupID.xy
6  dcl_input vThreadIDInGroup.xy
7  dcl_temps 5
8  dcl_thread_group 20, 20, 1
9  ishl r0.xy, vThreadGroupID.xyxx, l(4, 4, 0, 0)
10 iadd r0.xy, vThreadIDInGroup.xyxx, r0.xyxx
11 iadd r0.xy, r0.xyxx, l(-2, -2, 0, 0)
12 utof r0.zw, r0.xxxy
13 mad r0.zw, r0.zzzw, cb0[0].zzzz, cb0[0].xxxy
14 dp2 r0.z, -r0.zwzz, -r0.zwzz
15 sqrt r0.z, r0.z
16 add r0.z, -r0.z, l(1060.500000)
17 mul_sat r0.z, r0.z, l(0.001414)
```

```
mov r0.w, l(0)
mov r1.x, l(0)
loop
  uge r1.y, r1.x, l(10)
  breakc_nz r1.y
  iadd r1.x, r1.x, l(1)
  mov r0.w, r0.z
endloop
mov r1.xyz, r0.wwww
ult r0.zw, l(0, 0, 1, 1), vThreadIDInGroup.xxxy
ult r2.xy, vThreadIDInGroup.xyxx, l(18, 18, 0, 0)
and r0.z, r0.z, r2.x
and r0.z, r0.w, r0.z
and r0.z, r2.y, r0.z
if_nz r0.z
  ishl r0.y, r0.y, l(8)
  iadd r0.x, r0.x, r0.y
  ld_structured_indexable(structured_buffer, stride=60)(mixed,mixed,mixed,mixed
       ) r2.xyzw, r0.x, l(12), t0.yzwx
  ld_structured_indexable(structured_buffer, stride=60)(mixed,mixed,mixed,mixed
       ) r3.xyzw, r0.x, l(28), t0.yzwx
  ld_structured_indexable(structured_buffer, stride=60)(mixed,mixed,mixed,mixed
       ) r4.xyzw, r0.x, l(44), t0.xyzw
  mov r1.w, r2.w
  store_structured u0.xyzw, r0.x, l(0), r1.xyzw
  mov r2.w, r3.w
  store_structured u0.xyzw, r0.x, l(16), r2.xyzw
  mov r3.w, r4.x
  store_structured u0.xyzw, r0.x, l(32), r3.xyzw
  store_structured u0.xyz, r0.x, l(48), r4.yzwy
endif
ret
```

Listing 3: Circle selector compute shader disassembly (outer mode)

```
**** 64*64 ****
ErosiveCellular: 4773.097082
ErosiveHermite: 10838.750395
ErosivePerlin: 10713.165475
ErosiveSimplex: 5826.459444
ErosiveValue: 12460.179159
FBM_Billowy: 12394.455231
FBM_Linear: 13283.761780
FBM_Plateau: 13205.736494
FBM_Ridged: 12503.050141
IQCellular: 5498.421171
```

```
IQHermite: 5848.594028
IQPerlin: 5824.287135
IQSimplex: 6380.957260
IQValue: 7694.354956
JordanCellular: 5704.576300
JordanHermite: 6138.553782
JordanPerlin: 6597.713070
JordanSimplex: 6556.601374
JordanValue: 8332.588330
SwissCellular: 5378.430633
SwissHermite: 10962.704384
SwissPerlin: 10993.941245
SwissSimplex: 5946.526606
SwissValue: 11657.481309
CPUCmp_Cellular: 12064.771168
CPUCmp_Perlin: 12928.750554

**** 256*256 ****
ErosiveCellular: 631.983245
ErosiveHermite: 2569.613873
ErosivePerlin: 2602.286574
ErosiveSimplex: 759.376546
ErosiveValue: 2483.125916
FBM_Billowy: 2536.678455
FBM_Linear: 2535.938183
FBM_Plateau: 2498.112106
FBM_Ridged: 2540.721049
IQCellular: 641.047358
IQHermite: 719.169865
IQPerlin: 743.148049
IQSimplex: 786.087535
IQValue: 1176.633415
JordanCellular: 698.889026
JordanHermite: 805.840540
JordanPerlin: 835.837501
JordanSimplex: 859.884385
JordanValue: 1303.145230
SwissCellular: 637.308501
SwissHermite: 2573.103996
SwissPerlin: 2590.187094
SwissSimplex: 754.091755
SwissValue: 2517.166892
CPUCmp_Cellular: 2626.033234
CPUCmp_Perlin: 2640.700886
```

Listing 4: Raw noise performance results for 10 executions

```
**** 64*64 ****
ErosiveCellular: 957.682550
ErosiveHermite: 10207.198913
ErosivePerlin: 10120.920551
ErosiveSimplex: 1161.968808
ErosiveValue: 12149.197728
FBM_Billowy: 11962.707947
FBM_Linear: 12149.668138
FBM_Plateau: 12724.504734
FBM_Ridged: 12470.129029
IQCellular: 990.246584
IQHermite: 1102.512650
IQPerlin: 1139.866703
IQSimplex: 1212.092228
IQValue: 1760.502346
JordanCellular: 1082.722602
JordanHermite: 1262.661748
JordanPerlin: 1333.460615
JordanSimplex: 1356.696798
JordanValue: 2077.014693
SwissCellular: 982.840503
SwissHermite: 11078.120648
SwissPerlin: 10219.882587
SwissSimplex: 1154.951829
SwissValue: 11773.869185
CPUCmp_Cellular: 7959.497609
CPUCmp_Perlin: 9869.064682

**** 256*256 ****
ErosiveCellular: 71.808612
ErosiveHermite: 2467.597162
ErosivePerlin: 2473.593210
ErosiveSimplex: 84.104080
ErosiveValue: 2532.301339
FBM_Billowy: 2488.861310
FBM_Linear: 2489.475500
FBM_Plateau: 2484.981576
FBM_Ridged: 2487.433397
IQCellular: 71.506015
IQHermite: 79.267638
IQPerlin: 82.864043
IQSimplex: 88.277478
IQValue: 135.445405
JordanCellular: 80.247484
JordanHermite: 92.455188
```

```
46 JordanPerlin: 96.217555
47 JordanSimplex: 98.734554
48 JordanValue: 161.723621
49 SwissCellular: 71.880094
50 SwissHermite: 2460.641670
51 SwissPerlin: 2512.188217
52 SwissSimplex: 83.906401
53 SwissValue: 2526.772585
54 CPUCmp_Cellular: 2582.908411
55 CPUCmp_Perlin: 2587.870265
```

Listing 5: Raw noise performance results for 100 executions

References

[1] Daz 3D. *Bryce 3D*. URL: www.daz3d.com/products/bryce.

[2] Autodesk. *3ds Max*. 2014 2014. URL: www.autodesk.co.uk/products/autodesk-3ds-max/overview%E2%80%8E.

[3] Jason Bevins. 2003-2007. URL: http://libnoise.sourceforge.net/.

[4] Paul Bourke. "Polygonising a scalar field". In: *a target=" _blank" href='http://local. wasp. uwa. edu. au/~ pbourke/geometry/polygonise'¿ http://local. wasp. uwa. edu. au/~ pbourke/geometry/polygonise¡/a* (1994). URL: http://paulbourke.net/geometry/polygonise/.

[5] Nick Brettell. "Terrain rendering using geometry clipmaps". In: *Master's thesis, Cosc, Canterbury* (2005).

[6] Eric Bruneton and Fabrice Neyret. "Precomputed atmospheric scattering". In: *Computer Graphics Forum*. Vol. 27. 4. Wiley Online Library. 2008, pp. 1079–1086. URL: http://hal.inria.fr/docs/00/28/87/58/PDF/article.pdf.

[7] Giliam de Carpentier. *Scape: 3. Procedural extensions*. Jan. 2012. URL: http://www.decarpentier.nl/scape-procedural-extensions.

[8] Rüdiger Westermann Christian Dick Jens Krüger. "GPU-Aware Hybrid Terrain Rendering". In: (2010). URL: http://wwwcg.in.tum.de/fileadmin/user_upload/Lehrstuehle/Lehrstuhl_XV/Research/Publications/2010/CGVCVIP2010Terrain.pdf.

[9] Kate Compton et al. "Creating spherical worlds". In: *Proceedings of the International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'07)*. 2007, p. 82. URL: https://www.cs.cmu.edu/~ajw/s2007/0251-SphericalWorlds.pdf.

[10] Crytek. *Creating Voxel Objects*. URL: http://freesdk.crydev.net/display/SDKDOC2/Creating+Voxel+Objects.

[11] Stephen Daly. *The Size of The Elder Scrolls Online's World Calculated*. Aug. 2013. URL: http://www.gameranx.com/updates/id/16676/article/the-size-of-the-elder-scrolls-online-s-world-calculated/.

[12] Willem H De Boer. "Fast terrain rendering using geometrical mipmapping". In: (2000).

[13] Mark Duchaineau et al. "ROAMing terrain: real-time optimally adapting meshes". In: *Proceedings of the 8th Conference on Visualization'97*. IEEE Computer Society Press. 1997, pp. 81–88. URL: http://kucg.korea.ac.kr/new/course/2004/CSCE458/paper/roam.pdf.

[14] Ryan Geiss. "Generating complex procedural terrains using the GPU". In: *GPU Gems* 3 (2007), pp. 7–37. URL: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch01.html.

[15] Sarah FF Gibson. "Constrained elastic surface nets: Generating smooth surfaces from binary segmented data". In: *Medical Image Computing and Computer-Assisted Intervention—MICCAI'98*. Springer, 1998, pp. 888–898. URL: http://www.cs.tufts.edu/~frisken/surfaceNets.pdf.

[16] Jason Gregory. *Game Engine Architecture*. 2009.

[17] Stefan Gustavson. "Simplex noise demystified". In: *Linköping University, Schweden* (2005). URL: https://encrypted.google.com/url?sa=t&rct=j&q=simplex%20noise%20demystified&source=web&cd=1&cad=rja&ved=0CCMQFjAA&url=http://www.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf&ei=UP_qUsf9CcfNhAfW8oHQDw&usg=AFQjCNEVzOM03haFrTgLrjJp-jPkQyTOKA&sig2=EZBdr43Y1A0MTpOhqr1Ziw&bvm=bv.60444564,d.ZWU.

[18] Donald E. Knuth. "Fundamental Algorithms". In: Addison-Wesley, 1972.

[19] Clayton Kroh. *Planet Design: The Shaping of Corellia*. URL: http://web.archive.org/web/20011201170650/starwarsgalaxies.station.sony.com/team/articles/corellia_1.jsp.

[20] Peter Lindstrom and Valerio Pascucci. "Visualization of large terrains made easy". In: (2001), pp. 363–574. URL: http://computation.llnl.gov/casc/SOAR/SOAR.html.

[21] Yotam Livny, Zvi Kogan, and Jihad El-Sana. "Seamless patches for GPU-based terrain rendering". In: *The Visual Computer* 25.3 (2009), pp. 197–208. URL: http://wscg.zcu.cz/wscg2007/Papers_2007/full/C43-full.pdf.

[22] William E Lorensen and Harvey E Cline. "Marching cubes: A high resolution 3D surface construction algorithm". In: *ACM Siggraph Computer Graphics*. Vol. 21. 4. ACM. 1987, pp. 163–169.

[23] Frank Losasso and Hugues Hoppe. "Geometry clipmaps: terrain rendering using nested regular grids". In: *ACM Transactions on Graphics (TOG)* 23.3 (2004), pp. 769–776.

[24] Oscar Martinez. "An Efficient Algorithm to Calculate the Center of the Biggest Inscribed Circle in an Irregular Polygon". In: *arXiv preprint arXiv:1212.3193* (2012).

[25] Paul Martz. "Generating random fractal terrain". In: *Game Programmer* (1997). URL: http://www.gameprogrammer.com/fractal.html.

[26] Colt McAnlis. *HALO WARS: The Terrain of Next-Gen*. GDC Vault. 2009. URL: http://www.gdcvault.com/play/1277/HALO-WARS-The-Terrain-of.

[27] Scott Meyers. *Effective C++*. 2005.

[28] F Kenton Musgrave et al. *Texturing and modeling: a procedural approach*. Academic Press Professional, Inc., 1994.

[29] Tomoyuki Nishita et al. "Display method of the sky color taking into account multiple scattering". In: *Pacific Graphics*. Vol. 96. 1996, pp. 117–132.

[30] Philip Nowell. *Mapping a Cube to a Sphere*. 2005. URL: http://mathproofs.blogspot.co.uk/2005/07/mapping-cube-to-sphere.html.

[31] Sean O'Neil. *A Real-Time Procedural Universe, Part One: Generating Planetary Bodies*. 2001-2006. URL: http://www.gamasutra.com/features/20010302/oneil_01.htm.

[32] Sean O'Neil. "Accurate atmospheric scattering". In: *GPU Gems* 2 (2005), pp. 253–268.

[33] Renato Pajarola. "Large scale terrain visualization using the restricted quadtree triangulation". In: *Visualization'98. Proceedings*. IEEE. 1998, pp. 19–26. URL: ftp://ftp.inf.ethz.ch/doc/papers/ti/grpw/Vis98.pdf.

[34] Ken Perlin. "Noise Hardware". In: *Real-Time Shading SIGGRAPH Course Notes* (2001). URL: http://www.csee.umbc.edu/~olano/s2002c36/ch02.pdf.

[35] Ken Perlin and Eric M Hoffert. "Hypertexture". In: 23.3 (1989), pp. 253–262. URL: http://www.cs.jhu.edu/~subodh/458/p253-perlin.pdf.

[36] Arcot J Preetham, Peter Shirley, and Brian Smits. "A practical analytic model for daylight". In: *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*. ACM Press/Addison-Wesley Publishing Co. 1999, pp. 91–100.

[37] Stephen Schmitt. 2005-2013. URL: http://www.world-machine.com/.

[38] Jens Schneider, Tobias Boldte, and Rüdiger Westermann. "Real-time editing, synthesis, and rendering of infinite landscapes on GPUs". In: 2006 (2006), pp. 145–152. URL: http://wwwcg.in.tum.de/fileadmin/user_upload/Lehrstuehle/Lehrstuhl_XV/Research/Publications/2006/vmv06.pdf.

[39] John Snyder and Don Mitchell. "Sampling-efficient mapping of spherical images". In: *Polar* 19 (2001), pp. –29.

[40] Planetside Software. *Terragen - photorealistic scenery rendering software*. URL: planetside.co.uk.

[41] Filip Strugar. "Continuous distance-dependent level of detail for rendering heightmaps". In: *Journal of graphics, GPU, and game tools* 14.4 (2009), pp. 57–74. URL: http://www.vertexasylum.com/downloads/cdlod/cdlod_latest.pdf.

[42] Herb Sutter. *Exceptional C++*. 2000.

[43] Paul Tassi. *A Size Comparison of Massive Open World Video Game Maps*. May 2010. URL: http://unrealitymag.com/index.php/2010/05/07/a-size-comparison-of-massive-open-world-video-game-maps/.

[44] Jason Tranchida. "Texture Compression in Real-Time Using the GPU". In:

[45] Thatcher Ulrich. "Rendering massive terrains using chunked level of detail control". In: *SIGGRAPH Course Notes* 3.5 (2002). URL: http://tulrich.com/geekstuff/sig-notes.pdf.

[46] JMP van Waveren. "Real-time DXT compression". In: *May 20th* 2006 (2006).

[47] Computer Gaming World. June 2002. URL: http://pcg.wikidot.com/pcg-games:star-wars-galaxies.