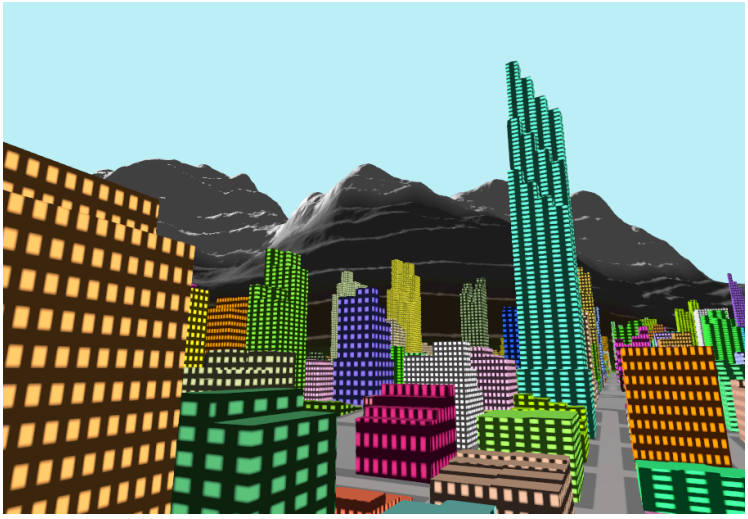
Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2013



Project Title:	**Real Time Procedurally-Generated Graphics using GPGPUs**
Student:	**N. Patel**
CID:	**00594301**
Course:	**EIE4**
Project Supervisor:	**Dr David Thomas**
Second Marker:	**Professor Peter Cheung**

Abstract

This project looks into the research and development of a GPGPU based method for procedural world generation, with a particular emphasis on real time creation, rendering and exploration. Through use of a highly parallel GPGPU algorithm for terrain creation and object placement, and a level of detail system where objects nearest to the camera are created and rendered at the highest detail and distant objects at the lowest, a world with millions of unique buildings and complex terrain can be created in under a second, and rendered at 30 frames per second on low end hardware. As the user moves through the world, objects and terrain are seamlessly generated in real time by the GPU, allowing for a near infinite world to be explored with absolutely no loading times. Extensions to the algorithm are also discussed, including methods for improving memory usage and object diversity.

Acknowledgements

I would like to thank Dr David Thomas for agreeing to supervise this project, and for giving lots of useful advice along the way. I would also like to thank my parents for funding my degree, and just about everything else I have ever done.

Contents

List of Figures

List of Tables

1 Introduction

Recent advances in the architecture of graphics processing units (GPUs) have allowed them to be used for more than just graphics processing. Complex programs that could previously only be run on traditional central processing units (CPUs) can now be run on the GPU through use of general purpose computing on graphical processing unit (GPGPU) APIs and, if the structure of the program can be expressed in a specific parallel way, a massive speed-up can be realised.

While the technology is still young, practically all new consumer laptop and desktop devices come with a GPGPU capable GPU, and even some mobile devices such as tablets have hardware support. Total adoption is still relatively low, however, so few developers for consumer applications are designing software that specifically relies on the GPU for anything other than tradtitional graphics processing, as the risk that the software will simply not be able to run on the platform is too great.

The recent announcement of the next-generation consoles (Xbox One and PlayStation 4) has revealed that their hardware will have a low power CPU core, a very high power GPU core, and an architecture that is speculated to be have heterogeneous Uniform Memory Access (hUMA), allowing for a GPU to operate on data held by the CPU without the overhead of copying it between independent memory spaces. Such an architecture is perfect for taking full advantage of the capabilities of GPGPUs, and we can expect that developers will start creating games that use the GPU for more than just graphics, with features that would have previously been impossible.

One potential use of GPGPUs in the realm of real time graphics is procedural generation, where a game world can be created algorithmically at run time rather than sculpted by hand.

1.1 Project Overview

The aim of this project is research into a novel method of procedural world and city generation, with real-time creation, rendering and exploration. The procedural content creation uses GPGPUs for acceleration of all world geometry generation and object placement. Tessellation with displacement mapping is used for terrain, allowing for simple geometry to be deformed into required shapes with minimal CPU overhead, and a new algorithm that exploits some underused features of DirectX 11 is used to procedurally place objects in the world in real time.

The focus is not on the plan of the city, which exists primarily to create an interesting data set, but instead on the underlying algorithms and rendering techniques that leverage the power of the GPU to dynamically create and draw the world as the user explores it.

The procedural generation method consists of two parts - the level of detail (LOD) system and the Generator. These two parts work in tandem to create the procedural world in real time. Both parts are built upon a simple graphics engine that handles the tasks of dealing with user input, rendering the virtual world onto the screen, and generally making it as easy as possible for the research and implementation of the procedural generation systems. These two parts are put into practice by creating a procedurally generated world within a demo application.

1.1.1 The Graphics Engine

As complete control over the rendering pipeline was required for this project, there was no choice but to create a custom graphics engine. The engine uses DirectX 11, and as a result the demo application only runs using Windows.

1.1.2 The Level of Detail System

In real life, the amount of detail that an eye or a camera can resolve on an object decreases as the object moves farther away. The same principle applies within a virtual world (assuming a perspective projection), where the number of pixels occupied in screen space by an object decreases with increasing distance. As a result, rendering terrain or an object with the same amount of detail at all distances is a waste of computational power, as there will be no visible difference between rendering a high and low quality version of an object at a large distance.

The level of detail system prevents this waste of resources. A set of radii around the user camera is defined using Chebyshev (or Chessboard) distances. Objects within the first radius are created and displayed at a maximum level of detail, and as objects move from a near to a far radius their level of detail is decreased, improving performance with, ideally, a negligible change in image quality.

In the context of the procedural generation system, the radius a given object or terrain tile falls in defines the parameters used in its creation. For instance, texture resolution and the number of polygons can be reduced the farther an object is away.

1.1.3 The Generator

When the level of detail system has realised that a tile or an object needs to be created (during the initial loading phase) or the level of detail changes (when the user moves through the world), the object or tile is added to the generator queue.

At the start of every frame, the generator processes a number of items on the queue by running a compute shader kernel on the GPU. Upon completion, the item on the queue is removed and can be rendered by the engine.

1.2 Target Platform

The development platform for this project is a 2 year old Lenovo Thinkpad E320. It has a 2.3GHz Dual Core with HyperThreading Intel Core i5-2410m (35W TDP), 6GB of DDR3 RAM, plus an AMD HD 6630M GPU with 1GB of DDR3 VRAM (26W TDP).

This machine has full DirectX 11 support, and is representative of the low end of the performance scale. Modern laptops with integrated graphics (for instance those running with the Intel Haswell micro-architecture) have approximately equal graphics capabilities as this machine, and a high end desktop can be expected to have upwards of 10x the graphics performance. The next generation of game consoles (Xbox One, Playstation 4) are estimated to have the graphics capabilities of a mid-high end PC, and performance could be estimated at 6-8x of the development platform. [1]

As a result, if the demo can be run smoothly (30-60 frames per second) on the development platform, GPU load on a modern desktop or next generation console will be practically negligible. The procedural generation technique could be scaled up to stress these high end platforms, or run in tandem with more traditional rendering techniques to add to the experience with a minimal performance hit.

1.3 Tools

The Visual Studio 2012 IDE is used due to its excellent support for DirectX and its graphical C++ debugger. In addition, AMD's GPU PerfStudio 2 is used for debugging and profiling DirectX API calls, and AMD's GPU ShaderAnalyzer is used for optimisation of DirectX HLSL shaders.

2 Background

2.1 General Purpose Computing on Graphical Processing Units

While GPUs have traditionally been used primarily for significantly accelerating algorithms used in computer graphics, recent developments have allowed them to be used for more general purpose computations (leading to the term General Purpose computing on a Graphics Processing Unit, or GPGPU).

The parallel nature of GPUs allow certain types of problems to be solved significantly faster than on a CPU. In particular, problems which can be sped up the most are those defined as embarrassingly parallel, where the workload can be trivially split into a set of parallel tasks, often where all tasks have little to no interdependencies.

Initially, engineers used GPUs to perform non-graphical calculations by mapping problem sets to graphical concepts, such as mapping 2-dimensional matrices to 2D textures, and then passing them through the graphics pipeline, using shader programs on the GPU to perform the required calculations and returning the result in a frame buffer.[2]

More recently, Application Programming Interfaces (APIs) have been developed that do not require the engineer to manipulate their problem into a form that can be processed in the regular graphics pipeline and instead can be expressed in a C-like language. The three main APIs used for GPGPU are NVIDIA's CUDA, Microsoft's DirectCompute and Khronos's OpenCL.

As of the release of Intel's Ivy Bridge micro-architecture in Q2 2012, all three major desktop GPU manufacturers (Intel, AMD and NVIDIA) have mainstream products that support both DirectCompute and OpenCL. In addition, hardware support of OpenCL is beginning to become available on some ultra-mobile platforms, though driver and operating system support is practically non-existent. The significance of this is that software engineers of consumer products can begin to use GPGPU APIs to significantly speed up specific tasks that would have previously run solely on the CPU, resulting in an improved user experience.

One area where this could lead to a significant jump in user experience is in the realm of real-time computer graphics, in particular in video games where high performance is all important. While the GPU is already extensively used in these for traditional graphical tasks, other problems can also be accelerated on the GPU such as physics calculations or large scale procedural content creation.

Some non-graphical tasks in video games are already performed on the GPU, but purely as an optional enhancement to the experience rather than as an integral part of the game engine. For instance, NVIDIA's PhysX technology is used in a not insignificant amount of modern PC games for advanced physics calculations (such as for fluid or cloth simulation), but as it is a proprietary API with support only on NVIDIA GPUs it is used strictly as an optional extra to enhance the visual experience rather than as an essential part of the game. See Figure 2.1, where the enhancement between PhysX being on or off is purely superficial.

Figure 2.1: Comparison between PhysX on and off in Batman: Arkham City

***NVIDIA** [3]*

2.2 Tessellation

With the most recent major updates to the graphics APIs, DirectX (11.0) and OpenGL (4.0), the ability to use programmable tessellation units has been introduced. In the context of computer graphics, tessellation is the process of subdividing a geometric primitive into several smaller primitives.

If you were to view the wireframe of a tesselated 3D model, as in Figure 2.2, there would be many more vertices. However, the frequency of detail is exactly the same as the original model. This means a shaded tessellated model will look identical to the original.

In order to add detail to a tessellated model, the newly generated vertices need to be moved, typically in a process called displacement mapping. In this process,

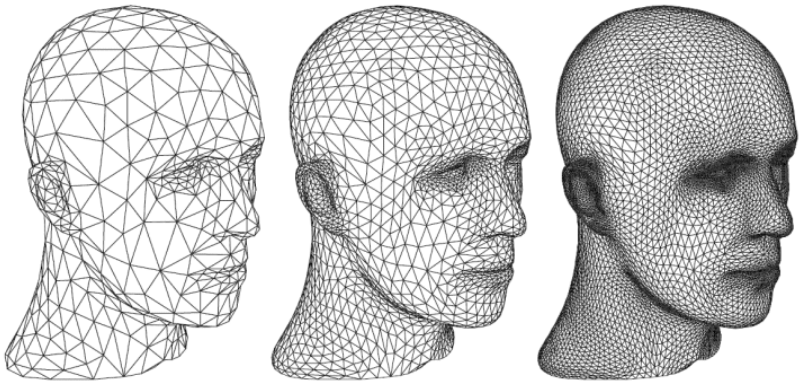


Figure 2.2: A coarse model (left) subdivided once (center) and twice (right)

all vertices evaluate a 2D texture called a displacement map which contains high resolution information about the shape of the 3D model, and as a result each vertex is displaced a specific amount. The results of such a process can be seen in Figure 2.3. Notice, in particular, the far more detailed silhouette of the displacement mapped model. While the lighting of the model could be emulated through bump mapping, an silhouette can only be improved through adding extra vertices.

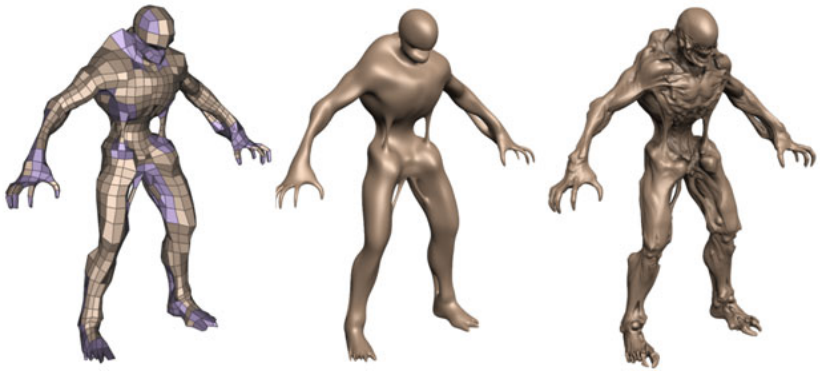


Figure 2.3: A coarse model (left) smoothly tessellated (center) and displacement mapped (right)

Kenneth Scott, id Software 2008

The benefits of using a coarse model with displacement mapping over a traditional high resolution model are numerous. A coarse model can be transformed with fewer operations than a high resolution model when it is being animated. A dynamic level-of-detail (LOD) algorithm can be used to smoothly decrease the number of vertices a model has as the scene camera moves away from it, preventing jarring transitions between high and low resolution models. The memory usage of

a coarse model with a displacement map is lower than that of a high resolution model, which is especially significant as compute performance scales at a faster rate than memory bandwidth.[4]

Tessellation, however, is a very new technology, and adoption of GPUs with this feature, while increasing, is still not high enough to justify having it as a hardware requirement for a video game. As a result, all uses in major video game titles have, again, been a purely superficial optional extra.

2.3 Procedural Generation

2.3.1 What is Procedural Generation?

Procedural generation is a broad term for using an algorithm to generate some kind of content. This can range from using a noise function (such as Perlin noise) to generate a random texture to complex generators capable of creating entire universes. Procedural generation does not necessarily need to be random, as the arrays of random numbers that are typically used for such algorithms can easily be generated through pseudo-random (and thus completely reproducible) means, resulting in identical outputs over multiple invocations.

2.3.2 Benefits

The benefits of algorithms for content creation are numerous, and include the following:

- Reduced storage requirements. If an algorithm can be used to generate a large proportion of a game's assets then the space savings can be significant. The downloadable game Roboblitz (2006) used 3MB of procedural texture information instead of 480MB of compressed textures, allowing the entire game to be just 50MB.[5] The game Fuel (2009) has a completely open 5000km^2 world, yet due to the use of procedural terrain generation it was still able to be shipped on a single DVD.[6]
- Requires fewer man-hours for repetitive tasks. By using a procedural generation algorithm for placement of lower priority objects in a hand-crafted game world, the level designer can spend their time on more important things. For example, the placement of every tuft of grass in a large world does not need to be specified manually if an algorithm can be designed that will automatically place them at run time in a realistic manner.

- Can generated a new, unique, potentially infinite world each time. Something simply impossible without procedural generation is a brand new world on each play-through of a game. By invoking the generator with a new random seed and parameters, a never seen before world can be created.
- Can be used for quicker loading and streaming of data into a game level. A typical bottleneck of modern games is disk I/O speed. By loading non-procedural data from disk in parallel with using procedural generation for other content, overall loading speeds can be improved.

2.3.3 Procedural Building Generation

For real time procedural city generation, visual fidelity needs to be balanced with performance, both of the generator itself and of the rendering of the buildings. Buildings must thus be designed in such a way that they look realistic yet are simple to create and render.

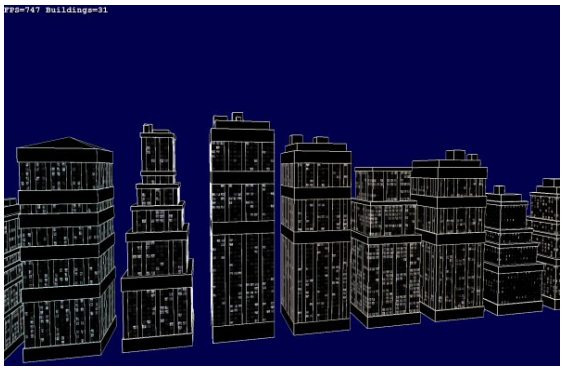


Figure 2.4: Procedurally generated buildings from Pixel City

Shamus Young [7]

A relatively recent implementation of building generation was used in the screensaver/tech demo Pixel City.[8] For this project, one of the algorithms created involved simply stacking multiple geometric primitives on top of each other. The result can be seen in figure 2.4. Given the simplicity of the algorithm, the result is very attractive and requires very little processing time. Unfortunately, such simple buildings will not be attractive when viewed from close up, as actual buildings tend not to have perfectly flat surfaces. By utilising displacement mapping on these flat surfaces, the amount of detail could be significantly increased when the camera is close enough to appreciate them.

3 The Graphics Engine

The graphics engine was created to make researching and programming the procedural generation systems as simple as possible.

There was the option between two competing graphics APIs, DirectX 11 and OpenGL 4.3, as both gave the ability to use Compute Shaders (DirectCompute and ARB_compute_shader, respectively). Compute Shaders allow for much better interoperability between GPGPU operations and normal graphics rendering operations compared to standalone GPGPU APIs such as OpenCL or CUDA. In the end, DirectX was chosen over OpenGL due to significantly better driver support at the start of the project. This does, however, lead to Windows being a requirement for running the demo application.

The engine is based off Microsoft's DXUT, a lightweight framework written in C++ that allows a simple DirectX 11 rendering loop to be set up with minimal effort, and provides a number of utility libraries. In addition, support for scripts written in Lua was added, using the Luabind library to expose C++ functions to the Lua interpreter. As a result, Lua scripts can directly invoke native C++ code, allowing for parameters and settings to be changed without requiring a recompile. Some usage of the Boost C++ library was necessary to get Luabind to work. [9]

A developer console was written to be able to run these Lua scripts from within the application. All input to the console is sent directly to the Lua runtime, meaning, should the user desire it, that any command line Lua application can be run from within the demo application. Figure 3.1 shows the developer console being used as an interpreter.

The engine also supports run time shader recompilation, meaning shaders can be rapidly prototyped and tested without having to restart the application.

The DirectXMath API is used for matrix and maths functions, and its usage of SIMD instruction set extensions mean it is very quick. Some unresolved issues with SIMD instructions and memory alignment mean that, while the application will successfully compile in both 32-bit and 64-bit modes, the 32-bit executable will almost immediately crash. The proliferation of 64-bit multiprocessors and operating systems mean that only being able to provide a 64-bit executable is not really a handicap.

All application profiling showed that, on the development platform, the GPU was almost always the bottleneck. As a result, very little CPU side optimisation was done.

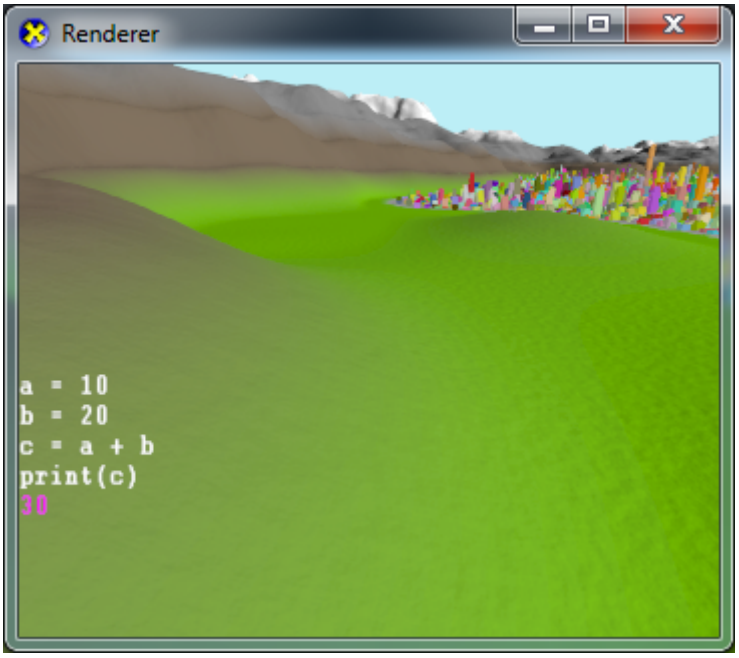


Figure 3.1: Lua interpreter within the application

3.1 The Rendering Loop - A High Level Perspective

Figure 3.2 shows a high level diagram of the rendering loop. Each iteration of the rendering loop deals with user input, updates all objects, then draws all objects.

3.2 User Input

User input is processed at the very start of the frame. By default, Windows queues up all user input (mouse movement and key presses) that occur when an application window is in focus. An application can then, at its own pace, process these chronologically. As drawing a frame can take any amount of time, and all objects rendered in a frame must be in a coherent state (i.e. they must have the exact same simulation time and camera position), all user input must be processed between frames, in order to prevent issues such as the camera moving mid frame.

Lua scripts are typically invoked by console command, so it is during the user input phase that they are executed. This is only possible due a lack of support for Lua scripts that run asynchronously to the native C++ rendering; all Lua scripts run directly on the rendering thread. While not an ideal solution for an actual game engine, it was more than adequate for the needs of this project, where Lua

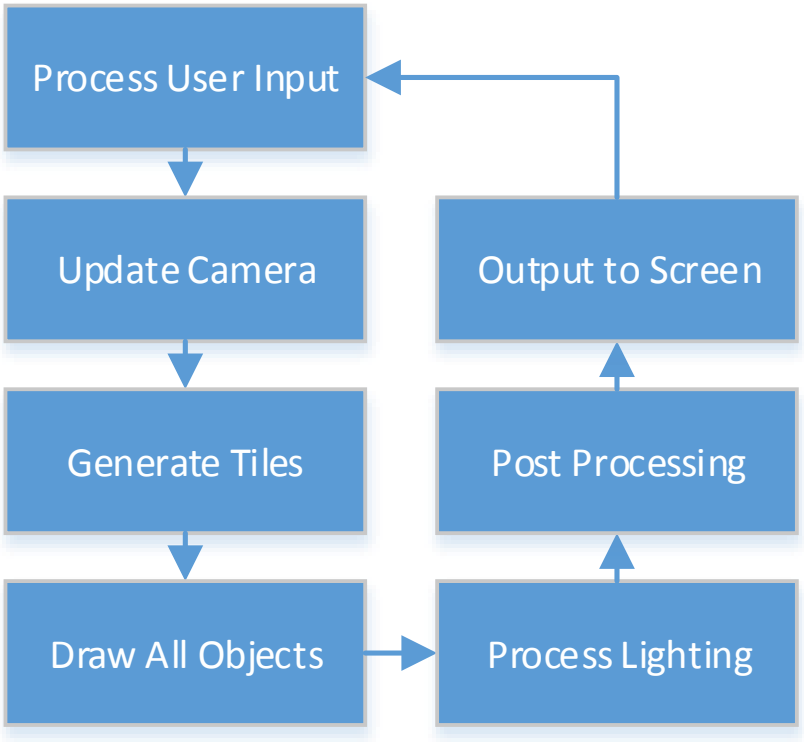


Figure 3.2: The rendering loop

scripts are primarily used to toggle settings or add some objects to the list of drawables.

Any user input that occurs after the current list of user input is processed is deferred until the next frame.

3.3 The Camera

After all user input is dealt with, any changes to camera position and viewing angle are processed, and the planes of the viewing frustum are recalculated. These planes

are especially important, as they are used for frustum culling, an optimisation that prevents any objects that will not be visible on screen from being sent to the GPU, wasting processing power on both the CPU and the GPU. Assuming a 90°horizontal viewing angle, approximately one in four objects in the world will be within the viewing frustum, so frustum culling could potentially cut down on three quarters of vertex transforms and DirectX API calls.

The GPU, using the depth buffer, can typically do a good job of preventing pixel shader operations on occluded pixels. However, if an object is drawn in front of another object, a lot of pixel operations can be wasted through overdraw. Sorting objects by distance from the camera, from near to far, can improve performance with absolutely no effect on the final image (ignoring any transparency related issues). The `Camera` class exposes frustum culling through the methods shown in figure 3.3, and allows any objects that need to be drawn to call these functions to find out if they will be visible in the current frame.

```
class Camera {
public:
  //Test a sphere against the viewing frustum. Returns TRUE if passes
  BOOL testFrustum( DirectX::XMFLOAT3 pPos, DirectX::XMINT3 pCoords,
      FLOAT pSphereRadius ) const;
  //Test an Axis Aligned Bounding Box against the frustum.
  BOOL testFrustumAABB( DirectX::XMFLOAT3 pPos, DirectX::XMINT3
      pCoords, DirectX::XMFLOAT3 pLenXYZ ) const;
};
```

Figure 3.3: Camera class frustum culling methods

3.4 The Generator

The procedural generation system will queue up a certain number of jobs on the GPU, dictated by a simple scheduler. The scheduler and the generator itself is covered in section 6.

3.5 Drawables

Objects that need to be drawn onto screen are typically clustered into `Drawable` instances, a convenient wrapper used for drawing objects that are related to one another. A drawable needs to adhere to the abstract interface shown in figure 3.4. The Level of Detail System is implemented as a drawable, allowing it to be

added to the list of drawable objects and processed without the engine knowing its implementation details, resulting in much more modular and reusable code.

```
class Drawable {
public:
  Drawable(Camera* pCamera) : mCamera(pCamera) {};
  virtual void Draw(ID3D11DeviceContext* pd3dContext) = 0;
  virtual ~Drawable() {};
  Drawable(const Drawable& copy);
  Camera* mCamera;
};
```

Figure 3.4: Drawable interface

The full list of `Drawable` instances is iterated through, and the `Draw` method is invoked. Each `Drawable` then updates their state, uses the `Camera` class to determine if they are visible, and then typically uses a `DrawableShader` object to draw a single mesh one or more times.

Clustering of similar objects into a single `Drawable` is necessary as it allows instancing to occur, where a single DirectX API call can be used to draw multiple objects, significantly reducing the CPU overhead of drawing lots of identical objects. Instancing is used heavily for the `CityTile` tile type in the level of detail system, covered in more detail in section 5.2.

3.6 Shaders

`DrawableShader` objects are used to draw `Drawable` instances. They implement the interface shown in figure 3.5. For the basic uninstanced `DrawMesh` method, the inputs required to draw an object are as follows:

- `ID3D11DeviceContext* pd3dContext`, the DirectX 11 device context. This is used to make API calls to DirectX.
- `DrawableMesh* pMesh`, the mesh object that contains the vertices and other attributes that make up a mesh.
- `DrawableState* pState`, the state object that contains information about the object that is about to be drawn, such as position matrix and colour. The instanced draw methods in the `DrawableShader` interface take in an array of state objects, each one corresponding to each object that needs to be drawn.

- `Camera* pCamera`, the camera object that defines the camera's position and angle in space. This is used to create any matrices used for vertex manipulation inside the vertex shader.

```
class DrawableShader {
public:
  std::wstring mShaderHandle;

  virtual void DrawMesh(ID3D11DeviceContext* pd3dContext, const
      DrawableMesh* pMesh, const DrawableState* pState, const Camera*
      pCamera) = 0;

  virtual void DrawInstanced(ID3D11DeviceContext* pd3dContext, const
      DrawableMesh* pMesh, DrawableState* const* pState, const Camera*
       pCamera, size_t pCount) = 0;

  virtual void DrawInstancedIndirect(ID3D11DeviceContext* pd3dContext
      , const DrawableMesh* pMesh, const DrawableState* pState, const
      Camera* pCamera) = 0;
  virtual HRESULT OnD3D11CreateDevice(ID3D11Device* pd3dDevice) = 0;
  virtual void OnD3D11DestroyDevice() = 0;

  DrawableShader(const WCHAR* pShaderHandle) {
    mShaderHandle = std::wstring(pShaderHandle);
  };
  virtual ~DrawableShader() {};
};
```

Figure 3.5: DrawableShader interface

Within the `DrawableShader` objects, DirectX 11 API calls are made to set up any shader states, perform the DirectX Draw call that draws the object, and finally clean up the shader state so that it is ready for the next call.

3.6.1 Deferred Rendering

Deferred rendering, as opposed to the more traditional forward rendering, is used for drawing objects. With forward rendering, lighting is applied in the same pass as the initial render of an object. In contrast, deferred rendering renders all geometry into a special kind of framebuffer called a geometry buffer, or G-Buffer, in a first shader pass. Only after all objects have been rendered into the G-Buffer are lighting effects applied, allowing these effects to only affect pixels that will be

visible on screen. For complex lighting effects, this can improve performance, but it is particularly memory bandwidth intensive due to the intermediate buffer stage.

The G-Buffer consists of 3 textures, the formats of which are as follows:

- `mNormSpec`, of texture type `R16G16B16A16_FLOAT`. Each pixel contains 4 channels of half precision floating point numbers. The R and G channels encode the 3D surface normal into 2 channels using spherical coordinates, and the B and A channels store specular amount and specular exponent.
- `mAlbedo`, of texture type `R8G8B8A8_UNORM`. Each pixel contains 4 channels of 8 bit values, each of which are mapped between 0.0 for 0x00 and 1.0 for 0xFF. The R, G and B channels are used for surface albedo, and the A channel is unused.
- `mEmittance`, also of texture type `R8G8B8A8_UNORM`. The R, G and B channels are used for surface emittance, and the A channel is an encoded brightness factor.

Figure 3.6 shows the `mNormSpec` and `mAlbedo` G-Buffers of some terrain. The wireframe of a a `DistantTile` instance is superimposed on the bottom left of the `mNormSpec` buffer. The steps used to get these G-Buffers and the output after a lighting pass are shown in section 5.1.

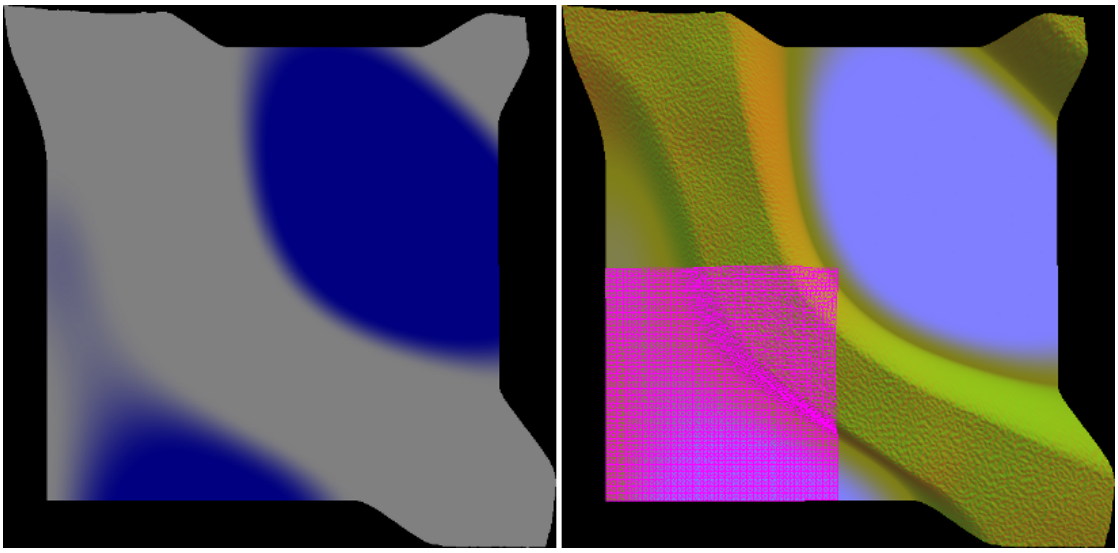


Figure 3.6: Contents of the G-Buffer, mAlbedo (left) and mNormSpec (right)

As the procedural generation system creates many, many objects that overlap with each other, forward shading would produce huge amounts of wasted pixels that are lit but never seen, due to occlusion. Deferred rendering does the absolute minimum amount of work in the initial pass to prevent this overhead, thus allowing for a more advanced lighting system in the second pass. The deferred lighting system is heavily based off code by Intel and supports up to 1024 dynamic lights, though this capability was never used. [10]

3.6.2 DrawInstancedIndirect

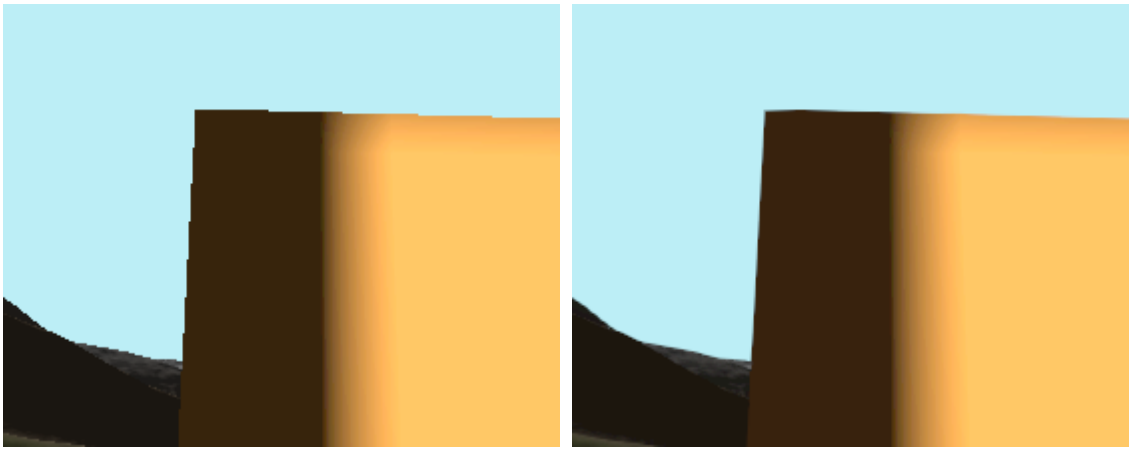
In the `DrawableShader` interface in figure 3.5, you can see a method called `DrawInstancedIndirect`. While not visible in the extract, The normal `DrawInstanced` function calls the DirectX 11 method `DrawIndexedInstanced`. This function takes in the following inputs: [11]

- `UINT IndexCountPerInstance`, the number of indices read from the index buffer per instance.
- `UINT InstanceCount`, the number of instances to draw.
- `UINT StartIndexLocation`, the location of the first index read by the GPU from the index buffer.
- `INT BaseVertexLocation`, a value added to each index before reading a vertex from the vertex buffer.
- `UINT StartInstanceLocation`, a value added to each index before reading per-instance data from a vertex buffer.

The DirectX 11 method `DrawInstancedIndexedIndirect`, called by the `DrawableShader` function `DrawInstancedIndirect` takes in the same arguments as `DrawInstancedIndexed`. However, it takes them in the form of an on-GPU buffer. This allows GPU generated buffers of objects to be drawn without any CPU side knowledge of the size and contents of the buffer. The use case for this is explained in more detail in section 6.5.

3.7 Post Processing

After the deferred lighting pass, FXAA, a post process anti-aliasing filter, is applied. Figure 3.7 shows the result of an FXAA pass on a typical scene. This is an easy way to improve visual quality with a minimal performance hit. [12]



(a) FXAA Off (b) FXAA On

Figure 3.7: FXAA comparison

3.8 Floating Point Error

Modern GPUs are optimised for single precision floating point maths, and often have no support for double precision. As a result, the original engine design used a vector of 3 floats to define camera and object position in 3D space. For ease of conversion from virtual world coordinates to real coordinates, a one to one mapping was used (i.e. 1 metre in real coordinates was equivalent to a floating point value of 1).

Figure 3.8 shows that, with a single precision floating point value of 10^7, the maximum error nears a value of 1. This means that, at a distance of 10^7m away from the origin, the minimum distance that the camera could reliably move and be guaranteed to change position is approximately 1m, as intermediate values could potentially not be representable. Severe issues with camera movement in space and object placement arose before this distance.

In order to get around this problem, a new format for representing camera and object precision was devised. A vector of 3 32-bit signed integers was used for coarse position information (to the nearest metre), and a vector of 3 single precision floating point numbers was used for fine position information. As the maximum representable integer is equal to 2^{31}-1, or 2,147,483,647, this is the maximum distance traversable parallel to an axis in the world. As the maximum floating point error at a value of 1 is approximately 10^{-7}, distances can be represented at a resolution of 10^{-7}m all the way up to the ends of the world.

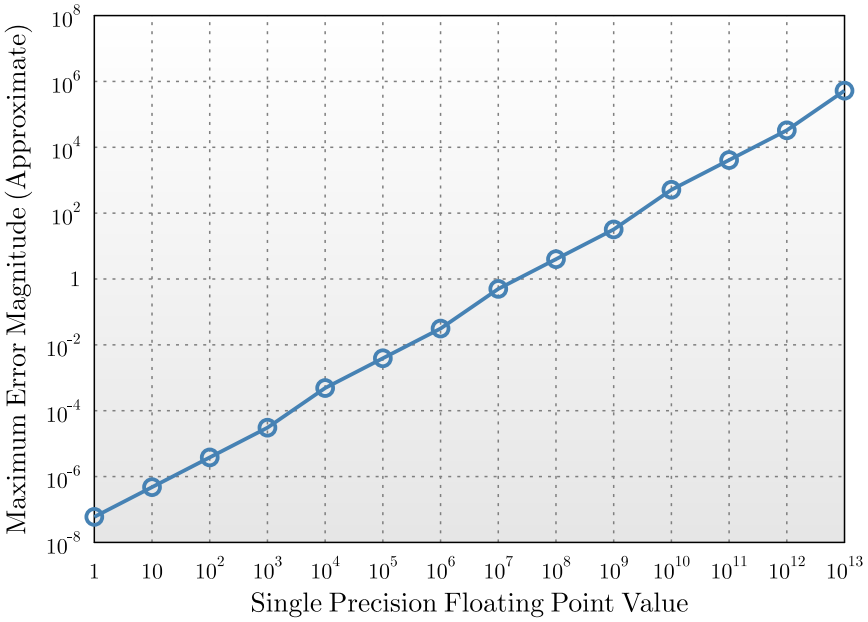


Figure 3.8: Single precision floating point values against maximum error magnitude

3.9 Potential Optimisations

As the target platform GPU was relatively low powered, the GPU ended being the bottleneck the vast majority of the time. Minimal CPU optimisation was done as there was no essentially no effect on performance.

Profiling of the application shows that approximately 65% of CPU time was spent inside the GPU driver, where DirectX 11 API calls are processed. As this is not user code, this cannot be sped up. However, what can be done is reduce the amount of DirectX 11 API calls.

For instance, terrain tiles are rendered individually rather than using instancing, so there are far more draw calls than are explicitly required. Instancing ended up being harder to implement in this case than expected due to the requirement of texture arrays, something which, due to a lack of foresight, the engine never had support for. Dependent on level-of-detail settings, a typical scene could render upwards of 400 terrain tiles, which could quite easily be consolidated into one single draw call, reducing CPU overhead by a significant amount.

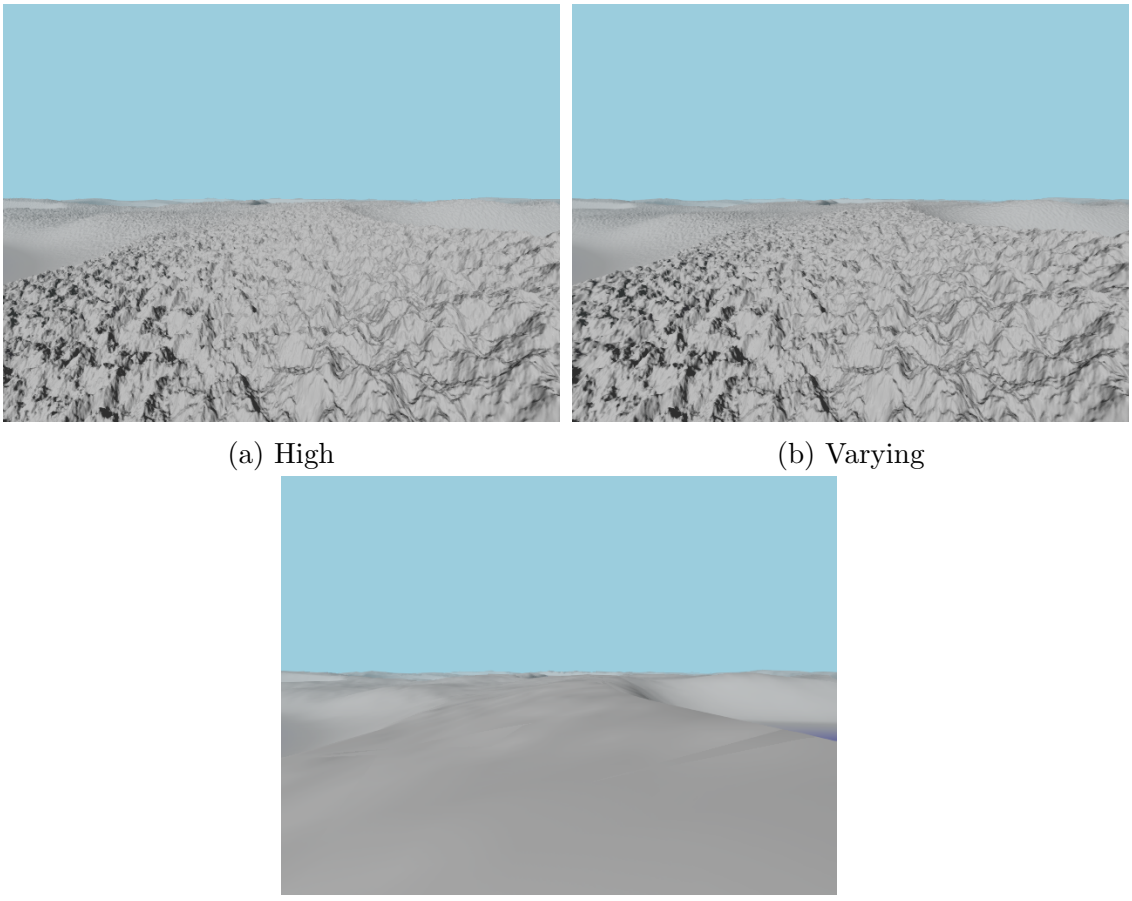
Frustum culling is also quite poorly optimised with respect to cache usage. Optimal frustum culling solutions will use contiguous arrays of input data and cull

large batches of objects in one go. The current solution has each object call the frustum culling code individually, resulting in quite a lot of function call overhead and vast amounts of pointer chasing and thus cache misses. However, only 3% of CPU time spent in user code is spent frustum culling, as the procedural generation system automatically puts objects into large batches, preventing the need for the code to be called all that often. Optimising this code would have minimal, if any, impact on performance.

4 The Level of Detail System

The level of detail system is built on the principle that an object that is further away from the camera can be drawn with less detail that a nearby one without any impact on visual quality. Typically, a lower quality object will have fewer vertices, a lower resolution texture or a shader with fewer instructions, meaning it requires less GPU time to draw.

Figure 4.1a shows a terrain only scene drawn with a high level of detail throughout, figure 4.1c shows the same scene drawn with a low level of detail, and figure 4.1b shows the scene with a varying level of detail, where the high and low levels of detail are visible nearest and furthest from the camera, respectively.



(a) High (b) Varying

(c) Low

Figure 4.1: Levels of detail

While the difference between the high and low levels of detail are plain to see,

a smoothly varying level of detail looks practically identical to the high level of detail. The GPU load and memory usage, however, is significantly lower, resulting in improved performance and loading times.

The level of detail implementation in this project divides the world into tiles, each of which has a given size and level of detail. For a terrain tile, the level of detail can be thought of as the number of texels and polygons per square meter. Defining the level of detail for a tile of discrete objects is more difficult, but it could be thought of as the number and complexity of objects per square meter.

Ideally, we want the world complexity to be as even as possible across the entirety of screen space; there should be approximately the same number of polygons in any given square centimetre of screen space as any other square centimetre of screen space, regardless of what is displayed within that area. For terrain, this is actually very straightforward, as we can simply make terrain tiles that are twice the distance away have half as many texels and polygons per square meter, and the perspective projection takes care of the rest.

For discrete objects, this is significantly more difficult. Objects cannot simply be subdivided, so a different approach needs to be taken. Objects that are unlikely to be seen from afar can be removed, such as those that will take up very little, if any, screen space, and those that are likely to be occluded by others. Mesh complexity can be reduced with distance, but there is only so far that this can go. A minimum number of vertices is always required to display an object.

For this project, mesh complexity is already at a minimum; buildings are simulated through deformed cubes, as explained in section 5.2. The only way to reduce level of detail is through removing objects that are too small to be visible, or likely to be occluded by others. Object tile level of detail is explained in more detail in section 6.5.

4.1 DistantDrawable

In order to implement the level of detail system, the class `DistantDrawable` was created. This class implements the simple `Drawable` interface described in section 3.5, but wraps up a huge amount of implementation on the inside. The constructor for `DistantDrawable` is as follows:

- `Camera* pCamera`, a pointer to the user camera.
- `ShaderManager* pShaderManager`, a pointer to the shader manager, a class that holds references to all `Shader` objects and ensures they are compiled

and updated as necessary.

- `MeshManager* pMeshManager`, a pointer to the mesh manager, a class that holds references to all `Mesh` objects.
- `Generator* pGenerator`, a pointer to the generator, the class that creates objects and terrain using the GPU and DirectCompute.
- `UINT pTileDimensionLength`, the number of tiles in each direction that each level of detail will be split up into. This needs to be an even number greater than 2.
- `UINT pNumLods`, the number of levels of detail.
- `DOUBLE pMinTileSize`, the tile size (in meters) that the smallest level of detail will be. Each level of detail above the smallest has a tile size twice that of the previous tile size.

The area of the world, and the maximum view distance in any direction parallel to one of the world axes can be calculated using the code in figure 4.2.

```
DOUBLE maxTileSize = pMinTileSize * pow(2,pNumLods-1);
DOUBLE worldDistance = maxTileSize * pTileDimensionLength;
DOUBLE maxViewDistance = worldDistance / 2;
DOUBLE worldArea = worldDistance * worldDistance;
```

Figure 4.2: Code snippet to calculate max view distance and world area

Typical construction parameters for `DistantDrawable` are `pTileDimension-Length = 6`, `pNumLods = 9` and `pMinTileSize = 200`. The resultant maximum view distance is 153600 meters, with a total world area of 94371 square kilometres.

4.2 LodLevel

Each level of detail within `DistantDrawable` is an instance of the inner class `LodLevel`, each instance of which has its own set of tiles. This class is a template, allowing different different types of object to be tiled. For the demo application, two classes are used with the template, `DistantTile` , the class used for drawing terrain tiles, and `CityTile` , the class used for drawing city tiles.

When constructing `DistantDrawable`, only LodLevel instances of type `Dis-tantTile` will adhere to the construction parameters. `CityTile` instances are

significantly more expensive to draw, so in order to prevent performance issues, a hard cap is set on the number of `LodLevel<CityTile>` instances that are created, and the size of each tile within each instance. This cap is easily modified through compile time constants in `DistantDrawable.cpp`.

The constructor for an instance of `LodLevel` is as follows:

- `DOUBLE pTileSize`, the size of each tile in metres. For the highest level of detail, this value will be identical to the `pMinTileSize` parameter in `DistantDrawable`. Every lower level of detail should have a tile size double that of the previous one.
- `UINT pTileDimension`, the total number of tiles in a dimension that the level of detail will be split into. This number should be identical to the `pTileDimensionLength` parameter in `DistantDrawable` for all created levels of detail.
- `FLOAT pOverlapScale`, the amount that each tile should overlap the one next to it, where an input of 1 is no overlap at all. The reasoning behind this parameter is explained in more detail in section 6.4.2.
- `DrawableMesh* pMesh`, a pointer to an instance of the object that this tile will be drawing. DirectX 11 API and algorithm limitations mean that it is only possible for a given tile to draw a single type of object within each tile, though there are potential workarounds that were not explored, something that is explained in more detail in section 8.

  Terrain tiles will use a plane as a mesh, and city tiles will use a cube.
- `DrawableShader* pShader`, a pointer to the shader used to draw the objects within a tile. Terrain tiles use the `DistantGBufferShader`, explained in section 5.1, whereas city tiles use the `CityGBufferShader`, explained in section 5.2.
- `Camera* pCamera`, a pointer to the user camera.
- `Generator* pGenerator`, a pointer to the generator.
- `LodLevel* pHigherLevel`, a pointer to the higher `LodLevel`. The lowest level of detail will have a pointer to the second lowest, which will have a pointer to the third lowest etc. For the highest level of detail instance of `LodLevel`, `pHigherLevel` will be `NULL`.

- `INT pPriority`, containing the priority of the level of detail. Higher priorities are larger numbers. This parameter is explaned in more detail in section 6.2.

- `TileCreatorFunc pTCF`, a function pointer of type TileCreatorFunc. Within `LodLevel`, this function is called to create and initialise each individual tile within the level of detail. Different settings must be used for each level of detail, so using a function pointer gives the flexibility that makes this possible. In practice, the function pointer is in the form of a C++11 lambda, using `std::function` to store it as a function pointer.

- `TileUpdaterFunc pTUF`, a function pointer of type TileUpdaterFunc. Within `LodLevel`, this function is called to update a tile when the tile is moved due to movement of the user camera. Different priorities are given to different levels of detail, hence the use of a function pointer. The lowest level of detail is given the highest priority to ensure there is always some terrain visible within the world, even if it is of the lowest quality.

- `TileDrawerFunc pTDF`, a function pointer of type TileDrawerFunc. Within `LodLevel`, this function is called to draw a tile if it is visible. Different methods are used to draw different types of tiles, hence the requirement of a function pointer. Terrain tiles use `DrawNoCull`, which draws an object without doing a frustum cull (these are done manually within `LodLevel`), and city tiles use `DrawInstancedIndirect`, which also doesn't frustum cull and is explained in more detail in section 3.6.2.

- `TileUniqueFunc pTUniqueF`, a function pointer of type TileUniqueFunc. This function returns true if only one `std::shared_ptr` shares ownership of a given tile. This is important as a shared pointer to a tile is passed to the generator when a tile needs updating, and only released when the tile has been processed by the GPU (which can take any amount of time). Tiles held by the generator must not be drawn as they are in an undefined state, hence the requirement to be able to test if there is more than one shared owner of a tile.

Full code listings for `DistantDrawable` and `LodLevel` classes can be found in `DistantDrawable.cpp`. These include the signatures for the `LodLevel`'s argument function pointers.

4.2.1 Tile Placement

Let an instance of `DistantDrawable` have the parameters `pTileDimensionLength = 6`, `pNumLods = 2` and `pMinTileSize = 200`. Also let the user camera be positioned at the origin, [0,0,0]. Two instances of the inner class `LodLevel` are created, with a tile size of 200m for the highest level of detail and 400m for the lowest level of detail.

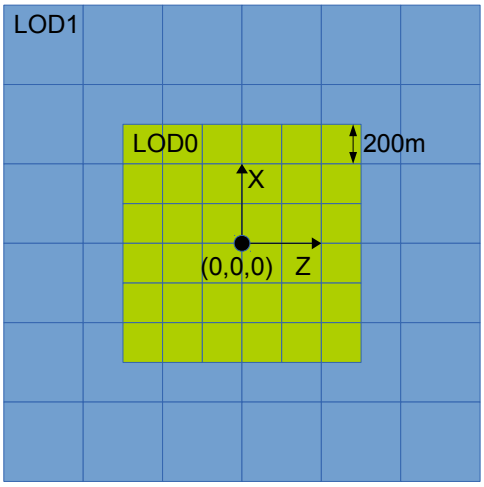
The created world tiles will be as arranged as shown in figure 4.3a, where LOD0 is the highest level of detail and LOD1 is the lowest level of detail. Not visible are the tiles in LOD1 that are completely overlapped by tiles in LOD0.

From the origin, moving the camera more than half of the tile size causes the highest level of detail level to reposition itself. Figure 4.3b shows the position of tiles after the camera moves to [0,0,101]. Note that while LOD0 has repositioned itself, LOD1 remains unmoved. Also note that the minimum distance from the camera to the edge of the visible world is slightly lower than when at the origin. The minimum visible distance is equal to the maximum visible distance minus half of the tile size of the lowest level of detail. A fog effect is used to hide the disparity between minimum and maximum visible distances, by cutting off view before the minimum distance.

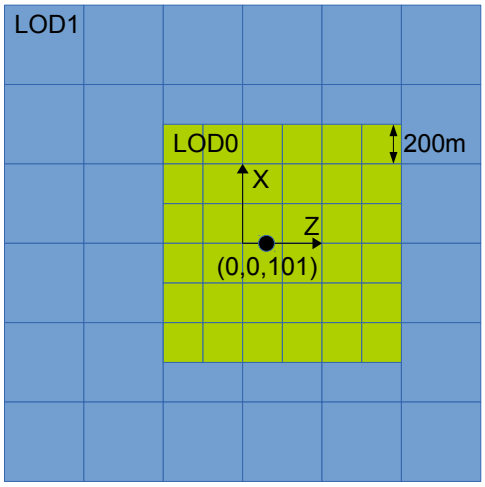
Moving the camera to [0,0,201] causes LOD1 to reposition itself in space, as can be seen in figure 4.4a. Finally, moving to [0,0,301] causes LOD0 to move again, resulting in a translated version of the tile layout at [0,0,0].

It is important to note that when a level of detail moves, it does not actually translate itself in its entirety in a certain direction. Instead, only the tiles furthest from the camera move in position. This allows all other tiles to remain unchanged in position, so their data does not need to be refreshed by the generator.

Figure 4.5a shows a 4x4 tile layout with the camera at the origin, where green tiles are the actual tile positions. Figure 4.5b shows the same layout with the camera translated 2 tiles in both the X and Z directions. The green tiles with IDs 2, 3, 6 and 7 did not change position, so they did not need to be updated. All other tiles have moved, so their data needs to be updated by the generator. Any 4x4 tile layout positioned anywhere on the grid will always have tiles with indexes from 0 to 15, and there is a mapping between a tile ID and a tile position in space for all user camera positions.

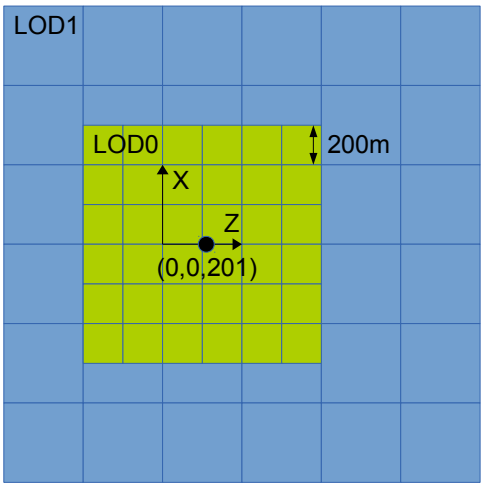


(a) Camera at [0,0,0]

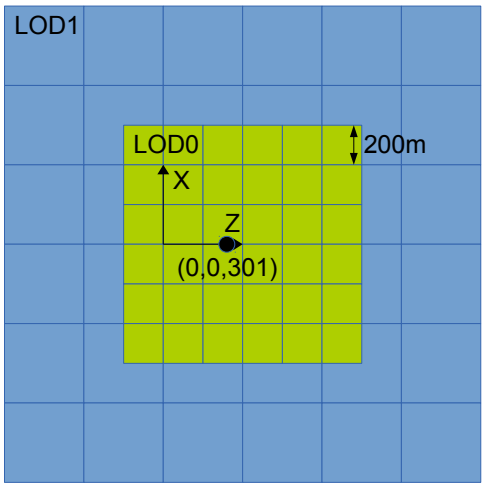


(b) Camera at [0,0,101]

Figure 4.3: Tile layouts

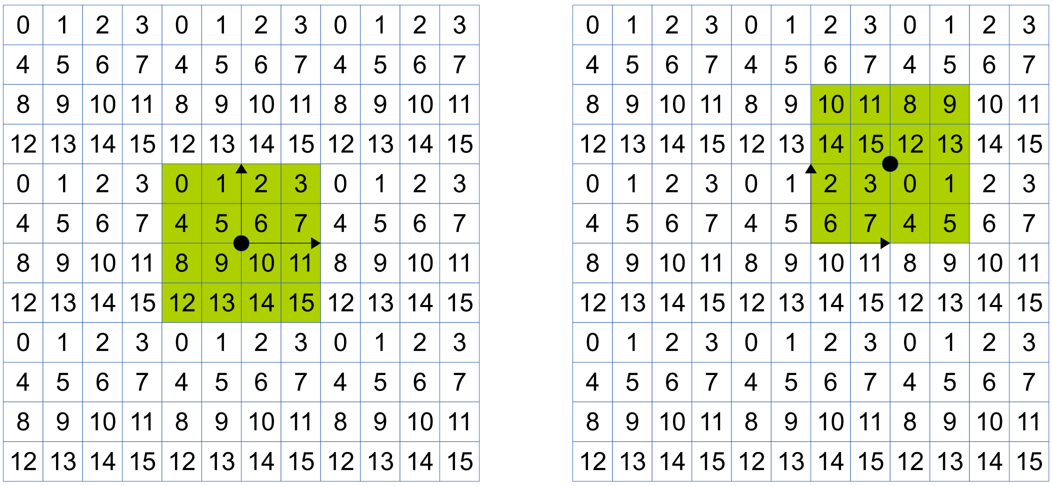


(a) Camera at [0,0,201]



(b) Camera at [0,0,301]

Figure 4.4: Tile layouts



(a) At the origin

(b) Translated 2 tiles in X and Z axes

Figure 4.5: Tile IDs with respect to camera position

4.2.2 Tile Updating and Creation

Upon construction of a `LodLevel` instance, tiles are placed as though the camera were at the origin. The tile creator function that was passed in as an argument is called, which creates the tile.

Immediately after, the positions of all tiles are updated to take into account the actual position of the user camera. The tile is also sent to the generator using the tile update function. The information stored in a tile, and how the generator uses and creates this information, is covered in section 6.3.

Every frame, each instance of `LodLevel` is updated to take into account any changes in camera position. If a tile has moved, it is sent to the generator to be processed.

4.2.3 Tile Drawing

To draw the world, an algorithm that recursively traverses through the stack of `LodLevel` instances is used. When the `Draw` method is called on the `DistantDrawable` by the graphics engine, the `DistantDrawable` calls `Draw` on the lowest level of detail instance of `LodLevel`, at which point the algorithm shown in the pseudocode below occurs.

```
void LodLevel::Draw() {

  //Tiles in the lowest level of detail are sorted by distance from
      camera to prevent overdraw
  std::sort(mSortedTiles.begin(),mSortedTiles.end(),distCompareFunc);

  //Iterate through all tiles
  for (UINT i = 0; i < mSortedTiles.size(); i++) {

    Tile t = mSortedTiles[i];

    //Check if the tile is unique. If a tile is non unique, the
        generator holds a reference to its data and thus the tile has
        not finished being created yet.
    if (isUnique(t)) {

      //Calculate the X and Z offsets of the tile. This maps the
          linear array of tiles to the two dimensional square layout,
          accounting for any camera translations
      INT offsetX = calculateOffsetX(t);
      INT offsetZ = calculateOffsetZ(t);
      this->DrawRecursiveAtPos(offsetX,offsetZ);
    }
  }
}

void LodLevel::DrawRecursiveAtPos(INT offsetX, INT offsetZ) {

  //From the offsets, calculate the index of the tile. For the lowest
      level of detail this is duplicated work, but the calculation is
      simple arithmetic which can be done in very few cycles, and it
      prevents large amounts of duplicated code
  UINT index = calculateIndexFromOffset(offsetX,offsetZ);
  Tile t = mTiles[i];

  //Test if the tile is inside the viewing frustum. If the tile is
      outside, return;
  if (!mCamera->testFrustumAABB(t)) return;

  //If there is no higher level of detail, draw the tile.
  if (mHigherLOD == NULL) {
    tileDrawFunction(t);
  }
  else {

    //Calculate X and Z offsets for the next highest level of detail
        at positions at the four quadrants of the tile. (Tiles in the
```

```
        next highest level of detail are exactly half the length of
        the current level of detail, so four tiles fit in very neatly)
        .
      INT2 blOff = this->getOffsetBottomLeft(t);
      INT2 brOff = this->getOffsetBottomRight(t);
      INT2 tlOff = this->getOffsetTopLeft(t);
      INT2 trOff = this->getOffsetTopRight(t);

      //Check if the given tile is drawable at the offsets.
      BOOL bl = mHigherLOD->DrawableAtPos(blOff.x,blOff.y);
      BOOL br = mHigherLOD->DrawableAtPos(brOff.x,brOff.y);
      BOOL tl = mHigherLOD->DrawableAtPos(tlOff.x,tlOff.y);
      BOOL tr = mHigherLOD->DrawableAtPos(trOff.x,trOff.y);

      //If all are drawable, recursively draw all four tiles
      if (bl && br && tl && tr) {
        mHigherLOD->DrawRecursiveAtPos(blOff.x,blOff.y);
        mHigherLOD->DrawRecursiveAtPos(brOff.x,brOff.y);
        mHigherLOD->DrawRecursiveAtPos(tlOff.x,tlOff.y);
        mHigherLOD->DrawRecursiveAtPos(trOff.x,trOff.y);
      }
      //Otherwise, draw the tile at the current level of detail
      else {
        tileDrawFunction(t);
      }
    }
  }

BOOL LodLevel::DrawableAtPos(INT offsetX, INT offsetZ) {

    //Check if the offsets are actually valid. As each level of detail
        has a tile size half of the lower one, the calculated offsets
        may fall outside the area of a given level of detail, and are
        thus not valid.
    if (areOffsetsValid(offsetX,offsetZ)) {
      UINT index = calculateIndexFromOffset(offsetX,offsetZ);
      Tile t = mTiles[index];

      //Ensure the tile has been created and isn't still lurking in the
          generator
      if (isUnique(t)) {
        return TRUE;
      }
    }
    return FALSE;
  }
```

Notice in particular the axis aligned bounding box frustum cull at every single recursion. This allows a very early exit for large chunks of the world if not within the viewing frustum. With a typical horizontal viewing angle of 90°, three quarters of the world could be cut out almost immediately. As the CPU knows nothing of the created world, the height of the bounding box is set to 10km, an estimated upper bound of the world. While not ideal, each tile is cheap to render so some redundant draws will not result in a meaningful performance penalty.

A minor flaw of the algorithm is that there is a lot of duplicated work between recursions, where maths is done to determine if a condition is met, then redone in the next recursion if it is actually met. This work is invariably trivial for a processor to do, however, as it will never require a search, simply some maths to find the correct index in an array. Future iterations of this algorithm could improve on this, but the overhead of this is very low down on the list of things that take up CPU time.

Figures 4.6 and 4.7 show tile layout and actual drawn tiles for two camera positions, with two levels of detail in a 6x6 grid formation. Under some circumstances, only a 4x4 section of the highest level of detail is drawn, whereas under other circumstances the entire 6x6 section is drawn, due to the fact that the current implementation cannot subdivide a tile (i.e it can either draw it or not draw it, there is no middle ground), something which could definitely be improved upon.

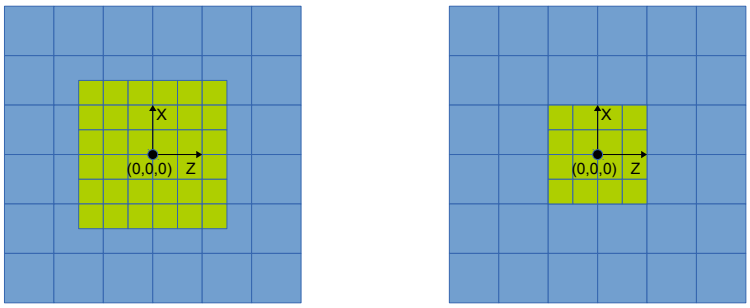


Figure 4.6: Tile layout (left) and drawn tiles (right) with the camera at [0,0,0]

4.3 A Failed Implementation

Before the final method was developed, an algorithm with a similar visual result but some serious flaws was used. Rather than the current recursive level of detail system, each level of detail was completely independent and could not communicate with the others.

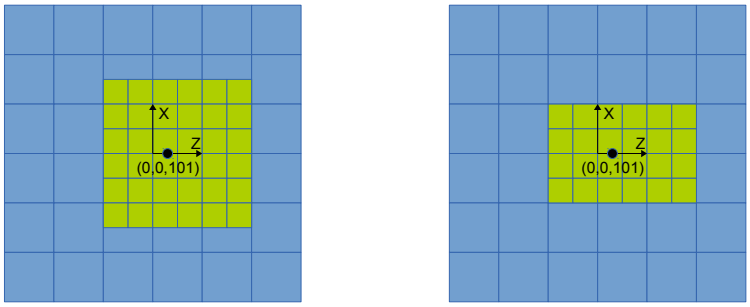


Figure 4.7: Tile layout (left) and drawn tiles (right) with the camera at [0,0,101]

Figure 4.8 shows tile layouts of three levels of detail (with positions moved for clarity). The highest level of detail (LOD0) was not unlike the current implementation, with an nxn area of tiles directly underneath the user camera. Each lower level of detail, however, had a hole the exact size of its higher level of detail in the middle of it. This worked fine if the camera was perfectly still, but as soon as the camera moved and some new tiles needed to be loaded in, there would be a massive hole in the terrain.

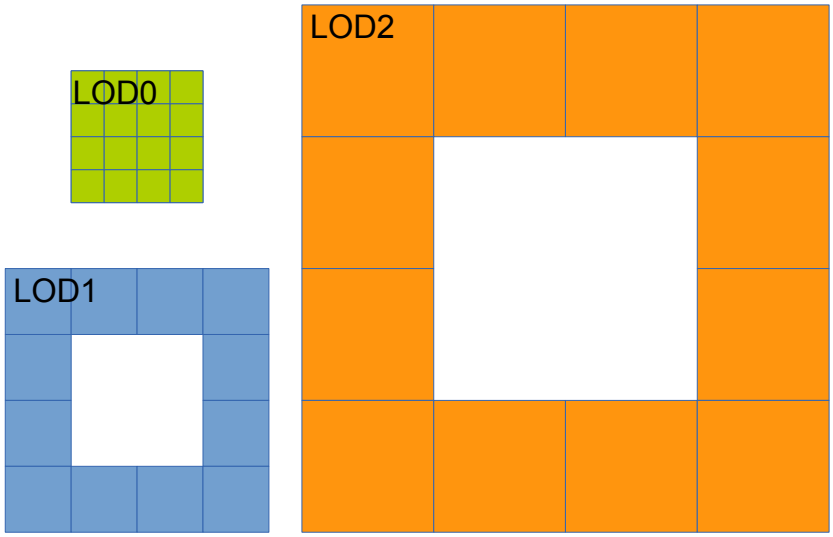


Figure 4.8: Tile layout (left) and drawn tiles (right) with the camera at [0,0,0]

Some experimentation was done with overlapping tile layouts, but moving quickly enough would easily foil this solution. Having overlapping tile layouts also caused the very unpleasant issue of Z-fighting. This occurs when two different

meshes occupy the exact same place in 3D space, resulting in one or the other mesh being visible dependent on draw call order, GPU job scheduling and general GPU architecture. The result is rapid flickering between the two meshes, a completely unacceptable visual effect on the scales it was occurring. The only option was to go back to the drawing board and create a new algorithm, the result of which is the method that is currently in place.

5 Tile Types

In the demo implementation, there are two types of tiles dealt with by the generator; the terrain tile `DistantTile` and the city tile `CityTile`.

5.1 DistantTile

The `DistantTile` class prototype is shown in figure 5.1. It contains a 3D vector of double precision floating point numbers to define the centre point of the tile, and one double to define the width of the square tile. Tiles are always aligned with edges parallel to the X and Z axes, with the Y axis pointing perpendicularly upwards, hence there is no need for a vector defining rotation. The `mPriority` value is the priority of the tile, and is explained in section 6.2. The constructor assigns the position and size of the tile, and also sets up the `Texture2D` instances.

```
class DistantTile {
public:
  DOUBLE mPosX;
  DOUBLE mPosY;
  DOUBLE mPosZ;
  DOUBLE mSize;
  INT mPriority;

  Texture2D mAlbedoMap;
  Texture2D mNormalMap;
  Texture2D mHeightMap;

  DistantTile(DOUBLE pPosX, DOUBLE pPosY, DOUBLE pPosZ, DOUBLE pSize)
      ;
  ~DistantTile() {}
};
```

Figure 5.1: DistantTile class prototype

There are 3 `Texture2D` instances within a tile, where a `Texture2D` instance contains a reference to a texture stored in GPU memory. Examples of all 3 texture maps for a single tile can be found in figure 5.2. The data stored in these texture maps is as follows:

- `mAlbedoMap` is an `R8G8B8A8_UNORM` texture map, meaning 8 bytes are used for each of the red, green, blue and alpha channels, with a linear mapping

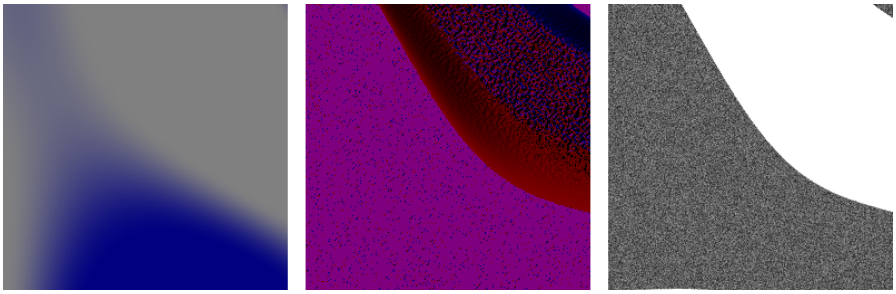


Figure 5.2: `mAlbedoMap` (left), `mNormalMap` (centre) and `mHeightMap` (right)

between 0.0 for a value of 0x00 and 1.0 for 0xFF. The R, G and B channels are used for the red, green and blue channels of the albedo map, and the A channel is used for specular reflection intensity.

- `mNormalMap` is also an `R8G8B8A8_SNORM` texture map. This is identical to `R8G8B8A8_UNORM`, except values are mapped between -1.0 and 1.0. The R, G and B channels are used as the X, Y and Z normal vectors for normal mapping, and the A channel is used for the specular reflection exponent.
- `mHeightMap` is an `R32_FLOAT` texture map, meaning there is one floating point number per pixel. This is used for height information when displacement mapping. As the values are floating point, it is particularly difficult to visualise the data, hence the image of the height map in figure 5.2 should be taken with a grain of salt. At one point, a half precision `R16_FLOAT` texture map was used for height data. However, at high terrain altitudes there were precision issues causing terrain quantisation, as can be seen in figure 5.3.

By applying these texture maps to a simple plane by use of normal mapping, tessellation and displacement mapping, complex terrain can be easily generated. Figure 5.4a shows a scene with 4 square tiles, with texture mapping, lighting and normal mapping. The bottom left quadrant uses the exact textures shown in figure 5.2. The wireframe of this scene is shown in figure 5.4b. Note the slight overlap between adjacent tiles, the reasons behind which are explained in section 6.4.2. Normal mapping alone gives a good sense of height in the scene, but the illusion is lost when viewing from anywhere other than directly above.

Figure 5.5a shows the same scene using displacement mapping, using the height map shown in figure 5.2. Figure 5.5b adds 4x tessellation. Finally, figure 5.6 shows the final product.

The code for the shader `DistantGBufferShader`, which is used to render each tile, can be found in `DistantGBufferShader.fx`.

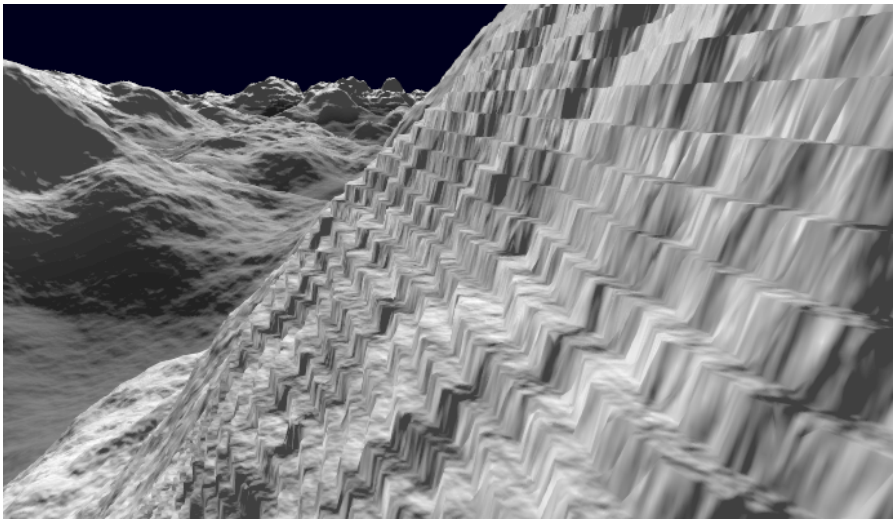


Figure 5.3: Terrain quantisation with half precision heightmaps

5.2 CityTile

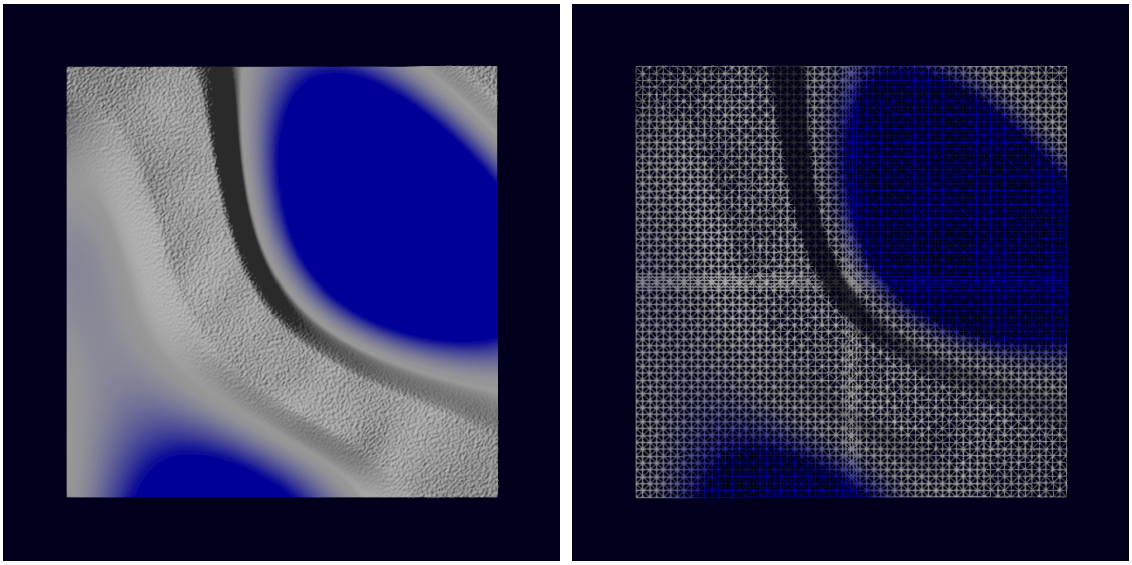
The `CityTile` class prototype is shown in figure 5.7. Like `DistantTile`, it contains 4 doubles to define tile position and size, and is also axis aligned. It also contains a priority value. The remaining class members are as follows:

- `CityLodLevel mCLL` is an enumerated type that defines the level of detail level of the tile, which affects the quality of objects created by the GPU. There are 5 possible values, from `CITY_LOD_LEVEL_HIGH` (0) to `CITY_LOD_-LEVEL_XXLOW` (4). Extra levels of detail are easily added through small modifications in the GPU kernel and by adding more members to the enumerated type, with even more X's in their names.

- `StructuredBuffer<InstanceData> mInstanceBuffer` is a reference to a structured buffer of type `InstanceData` the prototype of which can be seen in figure 5.8. A `StructuredBuffer` is held only in GPU memory.

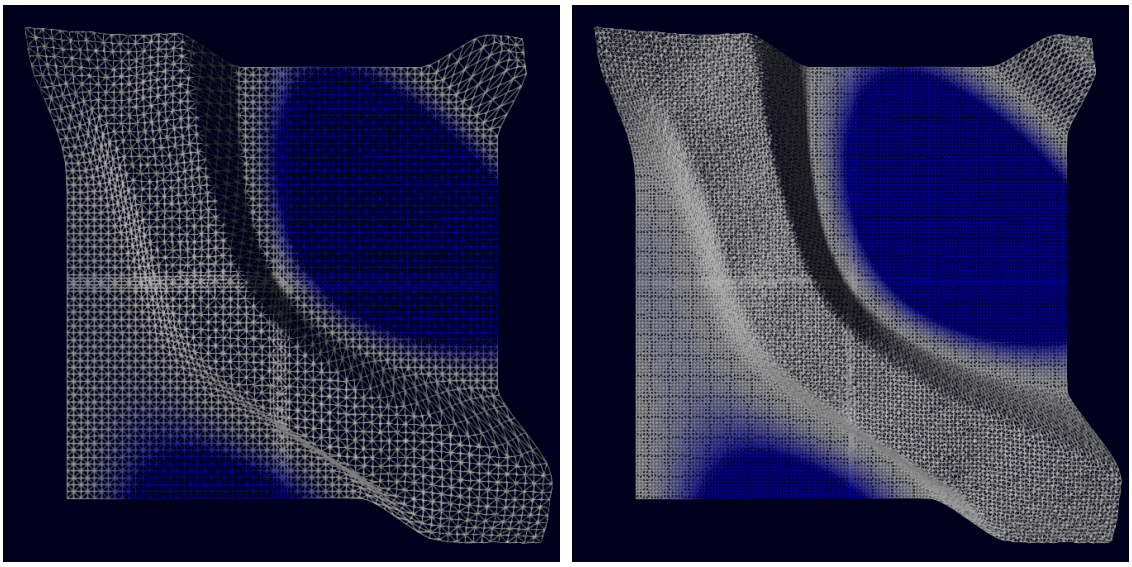
  For every object created by the GPU, an instance is added into the buffer. Each instance holds position, size, colour and rotation information. When drawing the city tile, the instance buffer is iterated through and each instance is used to create each object.

- `StructuredBuffer<UINT> mIndirectBuffer` is another reference to a GPU held structured buffer. This buffer is used for drawing the tile using the `DrawInstancedIndirect` method explained in section 3.6.2.



(a) Without wireframe (b) With wireframe

Figure 5.4: A 2x2 array of tiles, texture and normal mapped



(a) Without tessellation (b) With 4x tessellation factor

Figure 5.5: A 2x2 array of tiles, texture, normal and displacement mapped, with wireframe

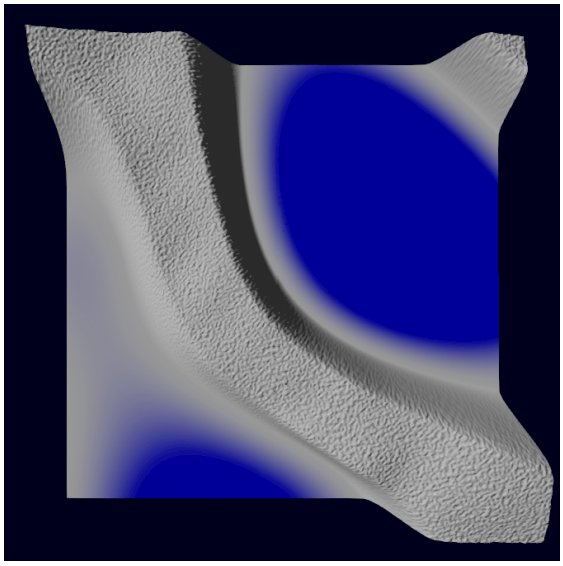


Figure 5.6: Final scene of 4 tiles, texture, normal and displacement mapped with 4x tessellation

```
class CityTile {
public:
  DOUBLE mPosX;
  DOUBLE mPosY;
  DOUBLE mPosZ;
  DOUBLE mSize;
   CityLodLevel mCLL;
  INT mPriority;

   StructuredBuffer<InstanceData> mInstanceBuffer;
   StructuredBuffer<UINT> mIndirectBuffer;
  UINT mIndirectData[5];
   //Position is the centre of the tile
   CityTile(DOUBLE pPosX, DOUBLE pPosY, DOUBLE pPosZ, DOUBLE pSize,
       DrawableMesh* pMesh, CityLodLevel pCLL);
   ~CityTile() {}
};
```

Figure 5.7: CityTile class prototype

- `UINT mIndirectData[5]` is an array holding the initial data that is copied into GPU memory for `mIndirectBuffer`.
- The constructor assigns the position, size and the tile level of detail. In addition, it takes in a mesh parameter. In order to draw using `DrawIn-`

`stancedIndirect`, the number of vertex indices in the mesh is required. This data is copied into `mIndirectData`, and late uploaded to the GPU.

```
struct InstanceData
{
   DirectX::XMFLOAT3 POSITION;
   DirectX::XMFLOAT3 SIZE;
   DirectX::XMFLOAT3 COLOUR;
  FLOAT ROTY;
};
```

Figure 5.8: InstanceData structure prototype

A single instance of `CityTile` can only hold instance information for a single type of mesh, which is quite a big limitation. A potential way to get around this problem is discussed in section 8.

Despite this limitation, absolutely any mesh can be used with `CityTile`. For this project and the sake of simplicity, a simple cube was used. This cube is resized, translated and rotated for each instance. By clustering multiple deformed cubes together and applying a simple texture map, the general shape and profile of a building can be emulated. Figure 5.9 shows an example city scene using this technique.

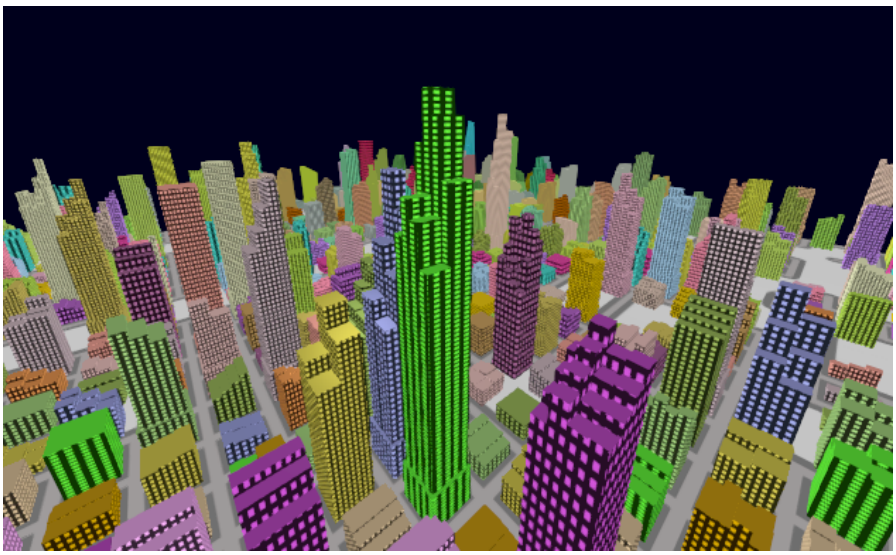


Figure 5.9: An example city scene at the highest level of detail

`CityGBufferShader`, the shader used to render these objects, reads directly from `mInstanceBuffer` within the shader for every object it draws. This is actually

a workaround for an API limitation of DirectX 11. For normal instancing, where the array of instances is created by the CPU, the instance buffer can be bound directly to the input assembler, outside of the shader, using the API function `IASetVertexBuffers`. This allows the GPU driver to optimise streaming of the instance data into the shader during execution. DirectX 11, however, does not like it when you try and bind a buffer that has the `D3D11_BUFFER_UAV_FLAG_APPEND` flag set on it, which allows it to be used as an `AppendStructuredBuffer`.

Instead, `mInstanceBuffer` must be bound as a vertex shader resource, using the API function `VSSetShaderResources`. Thus, instances must be manually pulled out during shader execution, rather than automatically streamed in by the driver. Fortunately, the performance hit for this is small, but the limitation is annoying nonetheless.

5.3 Meshes

In this project, a mesh contains two data structures that define its shape: an array of type `VertexData` that contains the position in space and the normal for each vertex, and an array of type `UINT` that contains the indices for every polygon within the mesh. Normally, each polygon is defined by three indices, making a triangle.

In the entirety of the project, only two meshes are used. Every single object drawn on screen uses one of these two meshes.

5.3.1 Cube

The cube mesh is, somewhat unsurprisingly, shaped like a cube. It consists of 12 triangles, 2 for each face of the cube. Figure 5.10 shows its wireframe. In order to draw all objects produced by `CityTile`, this cube mesh is deformed and translated to whatever shape and position it needs to be.

In the final version of the `CityTile` GPU kernel, at no point are the bottoms of any of the deformed cubes visible. By removing the two triangles that make up the cube, overall vertex transforms when drawing these tiles could be reduced by 16.6%, potentially improving performance when in particularly vertex heavy scenarios.

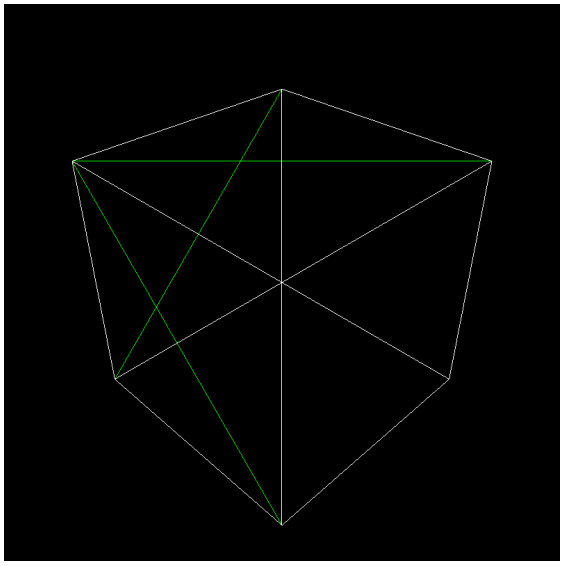


Figure 5.10: Cube mesh wireframe

5.3.2 Plane16

Plane16 is not a normal mesh, in that it does not contain triangles. Instead, it consists of polygons with 4 vertices - quadrilaterals - all of which actually perfectly square shaped.

The 16 in Plane16 defines the number of squares it has in each direction. It is produced algorithmically at run time, along with all planes of power 2, from Plane2 to Plane256. Figure 5.11 shows the structure of Plane16.

The reason behind using quadrilaterals instead of triangles is that this plane is only used to draw `DistantTile` instances using the `DistantGBufferShader` shader. This shader uses tessellation, which does not use regular polygons; instead it uses patches. Patches can range from 1 to 32 vertices, and in the case of `DistantGBufferShader` a patch size of 4 is used.

The tessellation parts of `DistantGBufferShader` (the hull shader, the fixed function tessellator, and the domain shader) will, from a high level perspective, take this patch and subdivide it into triangles, dependent on the tessellation factors input during the hull shader. For a tessellation factor of 1x each quadrilateral will be split in 2 triangles. For a factor of 2x, it will be 8, for 4x, it will be 32, and so on.

The original design plan was to, for the rendering of `DistantTile` instances, use a vertex density map to determine what tessellation factor to use at each point

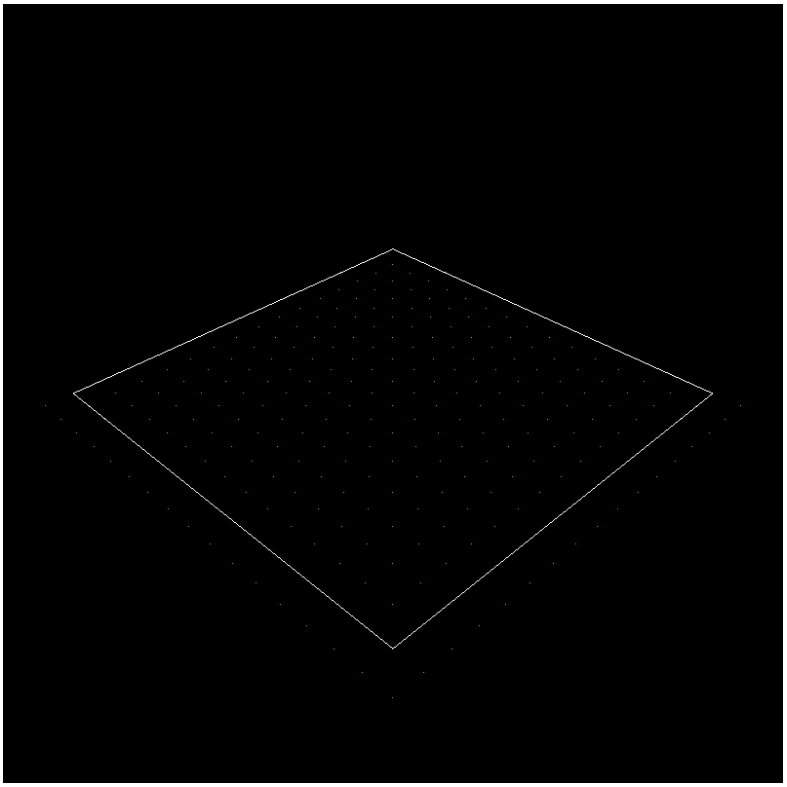


Figure 5.11: Plane16 mesh structure

in space. In practice however, simply directly setting the tessellation factor to 4x (and turning a Plane16 into a Plane64) resulted in no obvious hit to the frame rate and gave excellent results.

While it might seem strange that a Plane16 was used with 4x tessellation instead of a Plane64, the reasoning behind this choice is sound. A small benchmark was done on the development platform to get a rough idea of maximum triangle throughput using the tessellator. For all planes from Plane2 to Plane64, 500 instances were drawn with a tessellation factor that puts the number of triangles per plane at 8192, and thus the total number of triangles at 4096000. Table 5.1 shows the results.

The results shows that at some point between Plane32 and Plane64, it becomes impossible to fully satiate the vertex shader with vertices, due to there simply not being a high enough memory bandwidth.

Up to this point, however, the frame rate is at a perfectly steady 37, implying that the total triangle throughput of the GPU is approximately 152 million triangles per second. This, however, does not account for time spent not draw-

Plane	Triangles	Tessellation Factor	Frame Rate
Plane2	8	32	37
Plane4	32	16	37
Plane8	128	8	37
Plane16	512	4	37
Plane32	2048	2	37
Plane64	8192	1	11

Table 5.1: Tessellator triangle throughput benchmark

ing planes, such as during the deferred lighting stage, post processing and general frame composition.

In the end, Plane16 was chosen for flexibility. While for 8192 triangles it gave the same performance as most of its plane peers, the range of triangle subdivisions it gave, from 512 at a tessellation factor of 1x, to 2097152 at the maximum factor of 64x, meant it was more suited for the job were dynamic tessellation factors to be programmed in.

6 The Generator

When the level of detail system has a tile that needs creation or updating, it sends a reference of the tile to the generator. Every frame, the generator processes and releases its reference to a number of these tiles, signalling to the level of detail system that the tile can be drawn.

6.1 The Generator Class

The `Generator` class encapsulates all code to create tiles using the GPU. It has a few publicly accessible methods that can be used to add tiles to the queue, and to create tiles at the start of every frame. The methods are as follows:

- `void InitialiseTile(std::shared_ptr<DistantTile> const& pDistantTile)` takes a shared pointer to a `DistantTile` and adds it to the standard `DistantTile` queue.
- `void InitialiseTile(std::shared_ptr<CityTile> const& pCityTile)` takes in a shared pointer to a `CityTile` and adds it to the standard `CityTile` queue.
- `void InitialiseTileHighPriority(std::shared_ptr<DistantTile> const& pDistantTile)` takes a shared pointer to a `DistantTile` and adds it to the high priority `DistantTile` queue.
- `void InitialiseTileHighPriority(std::shared_ptr<CityTile> const& pCityTile)` takes in a shared pointer to a `CityTile` and adds it to the high priority `CityTile` queue.
- `void Generate(ID3D11Device* pDevice, ID3D11DeviceContext* pContext, FLOAT pMaxRuntimeSeconds)` takes in pointers to the DirectX 11 device and context, plus a `FLOAT` that contains the duration of jobs that should be queued up on the GPU.
- `void setInitialLoad()` sets a flag that indicates that it is the initial load of a `DistantDrawable` instance, and forces all the tiles to be immediately loaded upon the next call of the `Generate` method, rather than be spread across multiple frames. This is ideal when using a Lua script to load a new world.
- `BOOL hasGeneratables()` returns `TRUE` if any of the queues have any pending tiles to be processed.

6.2 The Generate Method

The `Generate()` method is directly called by the graphics engine at the start of every frame, and processes a number of items on the tile queues. Each of the four tile queues is given a priority, and tiles are processed in priority order.

The highest priority is given to the high priority `DistantTile` queue. Tiles in this queue are typically the lowest level of detail, and thus of the largest size. By prioritising these tiles, it means that at all times there will be some terrain at all places in the visible world, even if it is of the lowest quality.

The next highest priority is given to the high priority `CityTile` queue. Tiles in this queue are also typically the lowest level of detail and largest size. While less important than terrain, missing patches of buildings are still particularly unpleasant and some poor quality buildings are better than no buildings at all.

The final two queues are the standard priority `CityTile` and `DistantTile` queues. The priority for these two queues is the reverse of the high priority queues, with `CityTile` taking priority over `DistantTile`.

In order to service these final two queues, tiles within each queue are sorted based on their tile priority, which was set when creating the `LodLevel` instance that contains the tiles. Lower detail tiles are always given a higher priority than higher detail tiles, as a larger area can be created more quickly by servicing lower detail tiles first.

For a `DistantTile`, the subjective difference in quality between the highest level of detail and the lowest level of detail is smaller than for a `CityTile`. In addition, a typical scene will contain more `DistantTile` levels of detail than `CityTile` levels of detail. To even out quality in the world, `CityTile` instances are given a priority handicap that is manually configurable through the Lua scripting interface. This means that the highest `CityTile` level of detail will always be finished before the highest `DistantTile` level of detail.

The decision of giving extra priority to `CityTile` instances is also supported by the fact that a single `CityTile` takes significantly less time to create than a `DistantTile`, as far fewer threads are required, with a comparable amount of work per thread.

This priority hierarchy is always adhered to, so it possible that if the user moves through the world quickly enough, the lower priority queues are never serviced as there are always tiles being constantly added to the higher priority queues.

Something that is very important to note is that the `LodLevel` class will never

add a tile to the tile queue that is already in the tile queue. It will, however, modify a tile that is already in the queue. This means that, even if moving around the world at a ridiculous pace, work is never done on tiles that are outside of the visible world.

6.3 Tile Generation

The GPU kernels that produce both types of tiles are found in the source file `GeneratorShader.fx`.

From a high level perspective, the virtual world can be though of as a many dimensional function, with a number of inputs and a number of outputs. This function can be sampled at any point in 2D space, and by collecting enough points of data, a world can be created. Such an approach to world creation is ideal for a GPU algorithm, as each GPU thread can be set to sample a different point in space, with absolutely no dependencies. The result is a world that is massively configurable and incredibly quick to create.

As both types of tile sample the same world, a lot of code and inputs are shared. These inputs are as follows:

- `StructuredBuffer<uint> simplexBuffer`, the random seed for the world. This buffer contains 65536 entries and is used in simplex noise, the random noise that defines the world. This is explained in more detail in section 6.9.2.
- `uint cNoiseIterations`, the number of simplex noise iterations the world uses, up to a certain maximum defined by the compile time constant `MAX_NOISE_ITERATIONS`.
- `StructuredBuffer<float> scalesBuffer`, a buffer containing the scaling factors used for each simplex noise iteration. The length of this buffer is equal to `cNoiseIterations`.
- `uint cNumBiomes`, the number of biomes the world has. Each biome has different characteristics, such as terrain roughness, colour, and whether buildings may be placed upon them. There is no upper limit on the number of biomes, and there is no performance difference between having one and a thousand.
- `StructuredBuffer<float> coeffsBuffer`, the multiplicative factor for each iteration of noise in each biome. The length of this buffer is equal to `cNoi-`

seIterations * cNumBiomes. Multiplicative factors are smoothly interpolated on the borders of biomes to prevent jagged transitions.

- StructuredBuffer<float4> coloursBuffer, a buffer of length equal to cNumBiomes. The R,G and B components of this contain the terrain colour for a given biome. The A component is the transition factor. If two biomes next to each other have the same transition factor, the biomes will be smoothly interpolated together. Else, there will be a hard cut off between the two. A transition factor of less than 0 also means that this biome may have buildings placed upon it, and the magnitude of this number is the multiplicative factor for building heights.

- StructuredBuffer<float2> specPowBuffer, a buffer of length equal to cNumBiomes. The X component is used for the specular power of the colour of each biome, and the Y component is used for the specular intensity.

- CITY_GRID_SIZE, a compile time constant that defines the distance in metres between two city blocks. Its value also determines the number of threads dispatched for a CityTile, as each block requires a thread. This value must be a power of two.

The inputs scalesBuffer, coeffsBuffer, coloursBuffer, specPowBuffer, cNoiseIterations and cNumBiomes are completely configurable through Lua scripting, meaning different worlds can be designed and loaded at run time, without any compilation.

Both DistantTile and CityTile GPU kernels run the same two functions, getNoisesBoundsAccept and getTerrainInfo on each thread at the start of execution. All threads use the same inputs as described above, the only difference is the input position, a 2D floating point vector with the X and Z coordinates in space.

These two functions take in the above inputs and return the following outputs:

- float noises[MAX_NOISE_ITERATIONS], an array of the raw outputs of the simplex noise function, for each scaling factor defined in ScalesBuffer.

- float2 bounds[4], the X and Z coordinates in space of the four corners of the tile block. This is used for drawing roads and paved areas.

- bool accept[4], the road acceptance function at each of the four bounds, used for defining where roads and buildings should be placed.

- `float tileCoeff`, the biome index of the input position. The value of this is between 0 and `cNumBiomes` - 1, and is calculated by a weighted average of the simplex noise function at the first 3 scales in `ScalesBuffers`.
- `float terrainHeight`, the height of the terrain at the input position.
- `float tileCols`, the colour of the terrain the input position.
- `float tileSpec`, the specular power and exponent at the input position.

The DirectCompute compiler typically unrolls all fixed length loops and inlines all function code. As a result, if a kernel does not use all of the outputs of these two functions, the compiler will do dead code elimination to prevent wasted instructions and reduce general purpose register usage.

Various control semantics can change the behaviour of the DirectCompute compiler's optimisations, such as forcing conditional branching or predicated instructions, and preventing or forcing loop unrolling. It was noted that in certain cases, preventing some loops from being unrolled significantly sped up compilation time without any effect on performance, hence these control semantics have been liberally spread around the code.

6.4 DistantTile Generation

To generate a `DistantTile`, the `Texture2D` buffers for the tile first need to be created. The resolution of each texture buffer must be a multiple of 16 in both X and Y direction, as each texture is divided into 16x16 blocks of pixels for processing by the GPU kernel. When a tile is reused, such as when the camera moves, the textures can simply be overwritten with the new data; there is no need to recreate the texture buffers, which has a considerable CPU overhead.

In the demo implementation, texture resolution is set at 256x256, but this is easily configurable using a compile time constant. There is no requirement for all tiles to have identical resolutions, but as the level of detail system currently stretches textures of identical resolution over a different area chunk of terrain, the effect is that the texture resolution per square meter decreases with distance from the camera, reducing the need for any extra run time configurability. All 3 textures on a single tile must be of identical resolution, as all 3 are created in the same kernel.

Work is done in thread groups of 18x18, working on each 16x16 block of pixel. After finishing their calculations, all threads in a group load their terrain height

value into an 18x18 group shared buffer. Each thread, other than the outermost, uses the terrain heights of surrounding threads to calculate the surface normal for their texel. The number of work groups dispatched is equal to the amount of 16x16 blocks within the texture. For a 256x256 texture, this will be 16x16x1 (x, y, z).

The result of this 18x18 thread group structure is some duplicated work, but as the alternative is for all threads to write their heights to global memory, then do a second compute shader pass to calculate terrain normals. Adding in a second shader pass adds complexity, and would most likely decrease performance overall.

The code for the kernel is as follows:

```
#define TILE_DIM 18
#define MAX_CITY_COEFF 0
#define OVERLAP_SCALE 1.05f

#define ROAD_WIDTH 12
#define BUILD_WIDTH 23
#define PAVE_WIDTH 35

cbuffer CSGlobalTileCB : register(b0)
{
  uint cNoiseIterations;
  uint cNumBiomes;
}

groupshared float sDistantHeights[TILE_DIM][TILE_DIM];

[numthreads(TILE_DIM, TILE_DIM, 1)]
void CSDistantTile(uint3 groupID : SV_GroupID, uint3 groupThreadID :
    SV_GroupThreadID)
{
  //Calculate the texel this thread is working on
  int2 pixIndex = groupThreadID.xy - 1 + (TILE_DIM-2) * groupID.xy;

  //Calculate the X and Z coordinates in 3D space
  float2 pos = (float2)pixIndex/(cDistantResolution-1) - 0.50f;
  pos = pos * OVERLAP_SCALE * cDistantTileSize + cDistantTileCoords;


  float noises[MAX_NOISE_ITERATIONS];
  float2 bounds[4];
  bool accept[4];
  float terrainHeight;
  float tileCoeff;
```

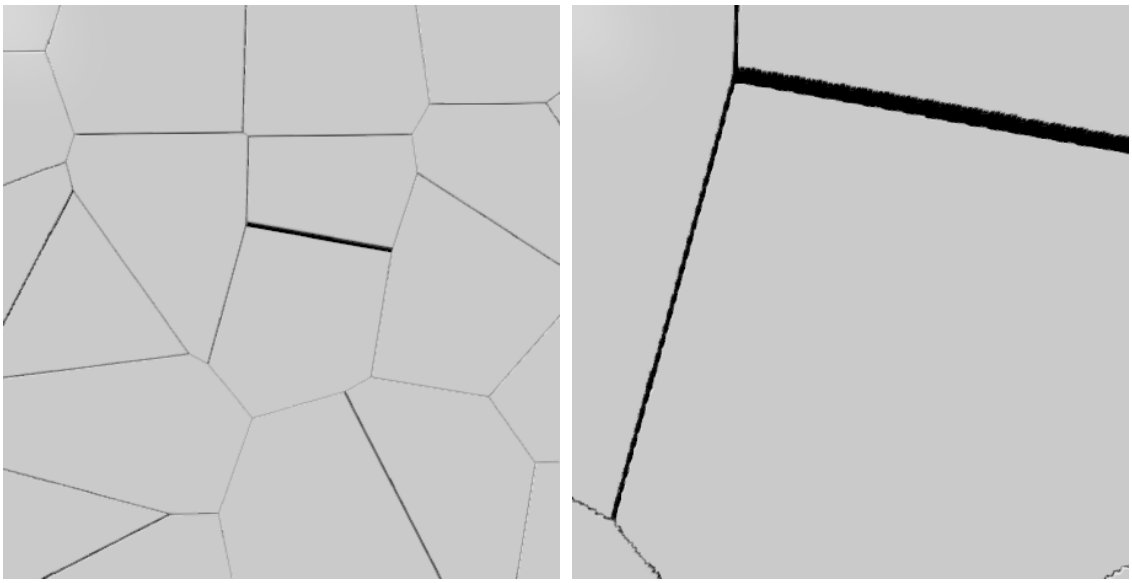
```
float4 tileCols;
float2 tileSpec;

//Calculate all the world information at the current position
//Some of the parameters to these functions are outputs
//Hence lack of initialisation
getNoisesBoundsAccept(pos,noises,bounds,accept);
getTerrainInfo(pos, noises, accept, tileCoeff, terrainHeight,
    tileCols, tileSpec);

//Calculate the number of roads we need to draw
uint numAccept = 0;
for (uint i = 0; i < 4; i++) {
  numAccept += accept[i] ? 1 : 0;
}

float3 colour = 0;

//If we need to draw roads, or we're in a city biome
if (tileCols.a <= MAX_CITY_COEFF || numAccept) {
  //Calculate the distance from the road
  float roadDist = 9999999.0f;
  for (int i = 0; i < 4; i++) {
    int ind1 = i%4;
    int ind2 = (i+1)%4;

    if (accept[ind1]) {
      float dist;
      if (accept[ind2]) {
        dist = minDist(bounds[ind1],bounds[ind2],pos);
      }
      else {
        dist = length(bounds[ind1] - pos);
      }
      roadDist = min(roadDist,dist);
    }
  }
  //Set the colour dependent on the distance
  //Magic numbers!
  if (roadDist < ROAD_WIDTH) {
    colour = float3(0.55,0.52,0.52);
  }
  else if (roadDist < PAVE_WIDTH) {
    colour = float3(0.259,0.259,0.259);
  }
  else {
    colour = tileCols.rgb;
  }
}
```

```
  //Otherwise set the colour we obtained in world information
  else {
    colour = tileCols.rgb;
  }

  int SpecPower = tileSpec.x;
  float SpecAmount = tileSpec.y;

  //Load height into group shared memory
  sDistantHeights[groupThreadID.x][groupThreadID.y] = terrainHeight;

  GroupMemoryBarrierWithGroupSync();

  float hp[4];
  hp[0] = sDistantHeights[groupThreadID.x-1][groupThreadID.y];
  hp[1] = sDistantHeights[groupThreadID.x+1][groupThreadID.y];
  hp[2] = sDistantHeights[groupThreadID.x][groupThreadID.y-1];
  hp[3] = sDistantHeights[groupThreadID.x][groupThreadID.y+1];

  //Calculate normal vector
  float epsilon = (float)cDistantTileSize/(cDistantResolution.x-1);
  float3 va = normalize(float3(2*epsilon,hp[1]-hp[0],0));
  float3 vb = normalize(float3(0,hp[3]-hp[2],2*epsilon));

  float3 normal = normalize(cross(vb,va));

  //Only write out to the textures if the thread isn't one of the
      outermost in the thread group
  if (all(groupThreadID.xy > 0) && all(groupThreadID.xy < TILE_DIM-1)
      ) {
    albedoTex[pixIndex.xy] = float4(colour,SpecAmount);
    //SpecPower is normalised between -1 and +1 for packing in an
        SNORM texture
    normalTex[pixIndex.xy] = float4(normal,SpecPower/64.0f - 1);
    heightTex[pixIndex.xy] = terrainHeight;
  }
}
```

After completion, the `DistantTile` is popped from its queue, reducing the number of `shared_ptr` instances with a copy of the data to just one, inside the `DistantDrawable` class. This signals that creation has finished, and the tile can be drawn.

6.4.1 Roads

A large chunk of the code in the kernel is dedicated to just drawing roads. Some experimentation with different methods for drawing roads was done, but the current implementation ended up being the fastest and the simplest especially when, in the `CityTile` compute shader, trying to fit buildings correctly within these roads. One alternative option was to use Voronoi noise, which, while quite fast, was incredibly difficult to filter properly to get consistent road widths. Figure 6.1 shows the failed Voronoi noise implementation from far and near. Note the inconsistent road widths and jagged edges. [13]



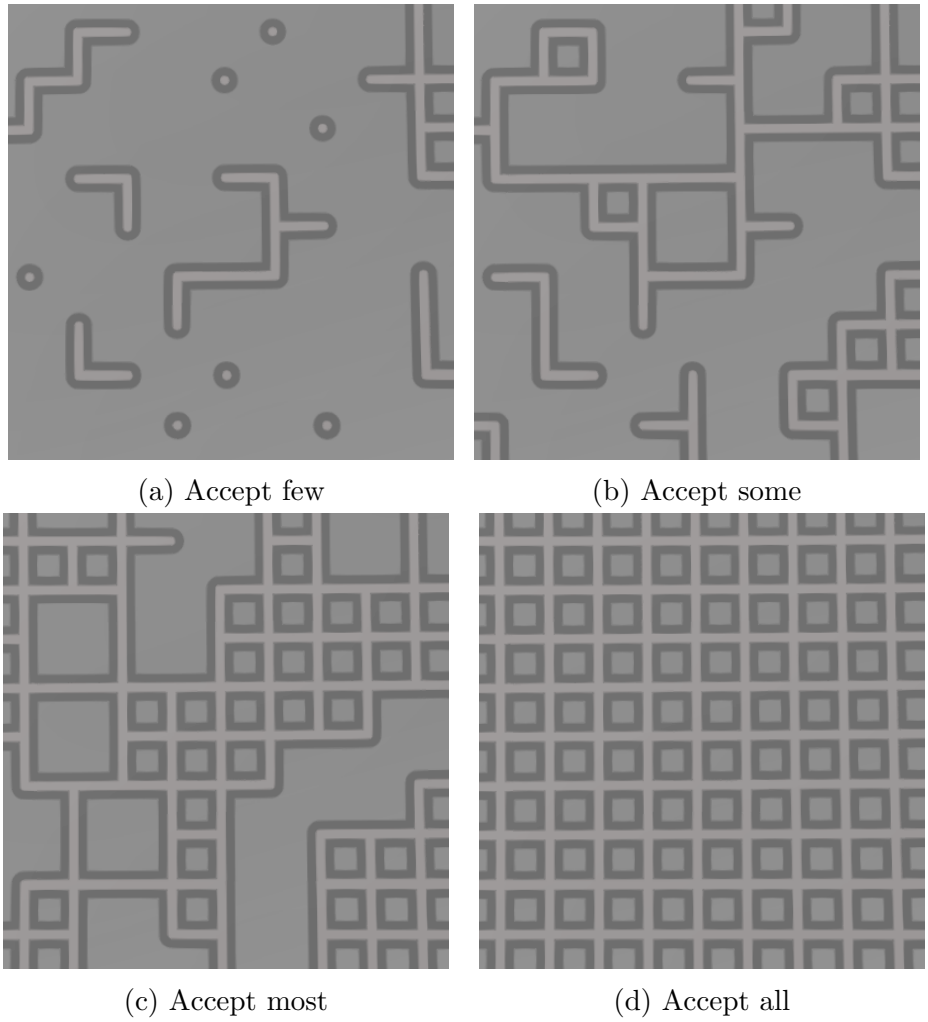
(a) Zoomed out (b) Zoomed in

Figure 6.1: Voronoi noise roads

Figure 6.2 shows the current implementation using different acceptance functions for different road layouts. All in all, the road layouts were a very minor piece of the whole project, so spending too much time on them would have been counter productive.

6.4.2 Seams

If two neighbouring terrain tiles are the exact same size and are created at the same level of detail, under most circumstances there will be no visible seam between the two. The heightmaps for both tiles should be identical at the very edges of each



(a) Accept few (b) Accept some

(c) Accept most (d) Accept all

Figure 6.2: Road acceptance functions

tile, so both should line up next to each other perfectly.

If, however, two neighbouring tiles are of different levels of detail, there is a very good chance that the two tiles will end up having a very small gap between them, as their vertices will not perfectly align; the smaller tile will have twice as many vertices in the same area of space as the larger tile. The result is an unsightly seam, where the skybox bleeds through the world. Figure 6.3 shows a seam between 4 different tiles where the skybox and some other geometry bleeds through.

The solution used to fix this is to have the edges of all tiles overlap by a small amount, defined by the value `OVERLAP_SCALE` in both the `DistantTile` generator

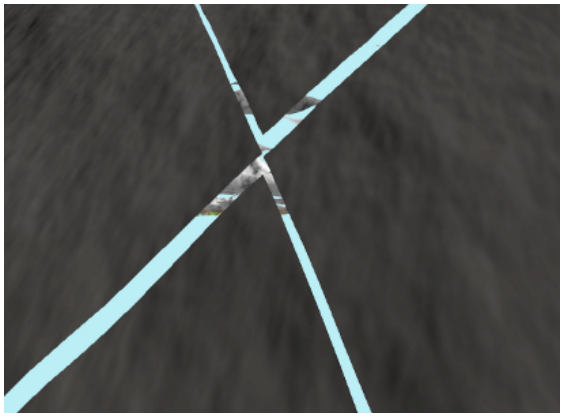
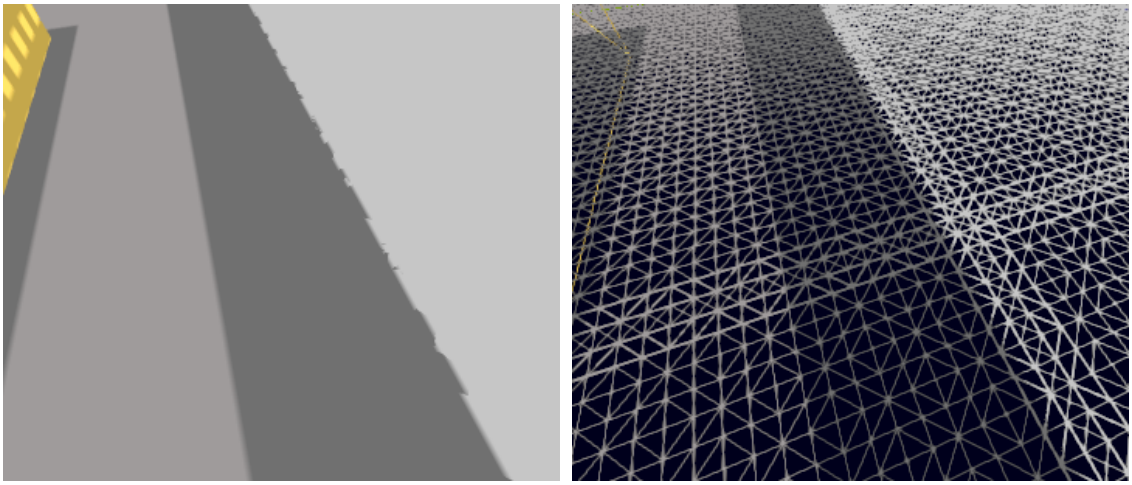


Figure 6.3: Seam between 4 tiles

kernel and the `DistantDrawable` class. Under most circumstances, this fixes the seams. However, this brings its own issue; z-fighting. Figure 6.4 shows z-fighting caused by this overlap.



(a) Without wireframe (b) With wireframe

Figure 6.4: Z-fighting

While z-fighting is certainly the lesser of two evils, there is actually an incredibly simple solution that avoids both of these problems, which was noticed while flying around Google Earth. Rather than extending tiles outwards, tiles should be extended downwards. This literally fixes everything, and this author is slightly ashamed that he didn't think of it first. Any z-fighting that occurs will happen below the surface of the world, and all seams will be avoided as it is guaranteed that all neighbouring tiles will meet.

By modifying the plane mesh used for drawing `DistantTile` instances to extend downwards a little bit around all of the edges, this solution can be implemented without even having to modify the shaders. The number of vertex transforms will slightly increase, but that was never a bottleneck to begin with.

6.5 CityTile Generation

To generate a `CityTile`, the `StructuredBuffer<InstanceData> mInstanceBuffer` first needs to be initialised. This buffer holds the data of all the objects created by the compute shader. This is a very special type of buffer, called an `AppendStructuredBuffer`, and this characteristic needs to be defined at initialisation time through a specific flag. An `AppendStructuredBuffer` allows all threads within a compute shader to append objects onto the end of it. There is minimal data available about the implementation of this buffer, but behind the scenes there is almost certainly an atomic counter that can give each thread a global memory location to place their object. [14]

The implementation of `CityTile` generation also relies on a special trick that `AppendStructuredBuffer` has. The DirectX 11 API method `CopyStructureCount` can be used to send a command to the GPU to copy the count of an `AppendStructuredBuffer` (i.e. the number of elements that have been appended to the buffer) to another GPU held buffer. In this case, after the GPU kernel has run, the count is copied from `mInstanceBuffer` to `mIndirectBuffer`, the 5 UINT `StructuredBuffer` that can be seen in section 5.2. [15][16]

Later, when the tile needs to be drawn, the method `DrawInstancedIndirect`, as seen in section 3.6.2, can be used with `mIndirectBuffer`, allowing exactly the number of objects held in the tile to be drawn with absolutely no CPU-GPU synchronisation. The output of the compute shader can be used in exactly the same frame as it was created in.

The length of `mInstanceBuffer` is defined by the class members `CityLodLevel mCLL` and `DOUBLE mSize`, as seen in section 5.2, and the compile time constant `CITY_GRID_SIZE`, the C++ code equivalent of the compute shader constant of the same name.

Using these values, we can calculate an upper limit on the number of buildings that will be in a given tile. The constant array `const UINT MAX_OBJECTS_PER_GRID_TILE[5]` can be indexed by the value `mCLL`, which has 5 possible values, from 0 to 4, and it will return the maximum number of objects in a a city grid at a certain `CityLodLevel`. For the highest level of detail, a building can be

created with anywhere up to 17 objects, and as there is a at most one building per city grid, the maximum number of objects per city grid tile is 17. For all other `CityLodLevel` values, the maximum number of objects per city grid tile is 1.

Unfortunately, the array `MAX_OBJECTS_PER_GRID_TILE` needs to be manually initialised at compile time, as values depend on the maximum number of objects that can be added to the buffer in the `CSCityTile` compute shader at each level of detail. There is no hard cap on the number of objects that the compute shader can produce, but there is a trade off that needs to be evaluated between compute shader performance and complexity, memory usage (the objects need to be stored somewhere), and rendering performance (the objects need to be drawn).

The code in figure 6.5 shows the calculation for the buffer size.

```
UINT maxObjects = MAX_OBJECTS_PER_GRID_TILE[mCLL];
UINT numBuildings = (UINT)(mSize / CITY_GRID_SIZE);
numBuildings *= numBuildings * maxObjects;
```

Figure 6.5: Code to calculate size of an instance buffer

When a tile is reused, the length of the `AppendStructuredBuffer` does not change, so the count can be reset to zero and the buffer reused.

As `mInstanceBuffer` needs to be large enough to hold the maximum number of objects the compute shader can append into it, there will be some wasted memory if the compute shader does not entirely fill up the buffer. A proposed method to prevent this wasted memory is detailed in section 6.6.

One thread per city grid tile is used, so the number of thread groups dispatched is equal to the number city grid tiles in each dimension divided by the number of threads the kernel works on in parallel. Given that the tile size is 1024x1024, the grid size is 128x128 and the kernel works with an 8x8 thread group, the number of work groups dispatched is equal to 1024x1024 / (8x8 $*$ 128x128) = 1x1x1.

The actual kernel itself is used to create a varied set of buildings. This is only really possible through using a lot of pseudo random numbers, created through either running the simplex noise function or through running a number through a hash function, explained in more detail in section 6.9. As a result, the structure of the code contains many magic numbers that exist only to prevent visible patterns from emerging through pseudo random number generation. In addition, a lot of strange code logic choices were also made to improve building variety and aesthetics without affecting performance adversely.

The code for the kernel is as follows:

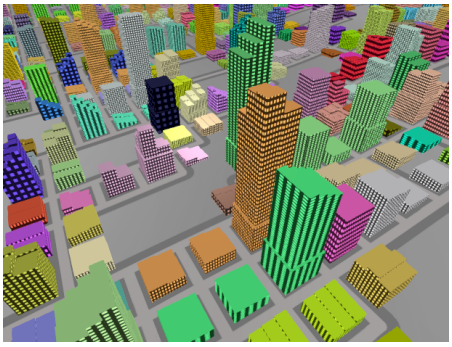
```
#define CITY_LOD_LEVEL_HIGH 0
#define CITY_LOD_LEVEL_MED 1
#define CITY_LOD_LEVEL_LOW 2
#define CITY_LOD_LEVEL_XLOW 3
#define CITY_LOD_LEVEL_XXLOW 4

#define CITY_TILE_GROUP_DIM 8
#define CITY_GRID_SIZE 128

#define ROAD_WIDTH 12
#define BUILD_WIDTH 23
#define PAVE_WIDTH 35

cbuffer CSCityTileCB : register(b1)
{
  int2 cCityTileCoords;
  uint cCityTileSize;
  uint cCityLodLevel;
}

AppendStructuredBuffer<Instance> sInstance : register(u0);

[numthreads(CITY_TILE_GROUP_DIM, CITY_TILE_GROUP_DIM, 1)]
void CSCityTile(uint3 dispatchID : SV_DispatchThreadID)
{
  //Calculate the position of the building in space
  float2 p1 = CITY_GRID_SIZE * dispatchID.xy - ((float)cCityTileSize
      /2);
  float2 pos = cCityTileCoords + p1;

  float noises[MAX_NOISE_ITERATIONS];
  float2 bounds[4];
  bool accept[4];
  float terrainHeight;
  float tileCoeff;
  float4 tileCols;
  float2 tileSpec;

  //Calculate all the world information at the current position. Some
       of the parameters to these functions are outputs, hence lack of
       initialisation
  getNoisesBoundsAccept(pos,noises,bounds,accept);
  getTerrainInfo(pos, noises, accept, tileCoeff, terrainHeight,
      tileCols, tileSpec);

  //Check if we need to put a building down here. Buildings are only
      placed where there is more than one road within the grid tile
```

```
uint numAccept = 0;
for (uint i = 0; i < 4; i++) {
  numAccept+=(accept[i] && accept[(i+1)%4]) ? 1 : 0;
}

if (numAccept <= 1) {
  return;
}

//The negative of the tile colour alpha is used as the seed for
    building height
const float heightSeed = -tileCols.a;
//Buildings will be a minimum of 20m, but half is sunk into the
    ground to ensure the building will not be floating on uneven
    terrain
const float minHeight = 20;

//Use Simplex noise to calculate a base height
float baseheight = (noise2D(pos.x/100,pos.y/72)/2)+0.5f;
baseheight = minHeight + pow(baseheight,4)*(heightSeed);

//Use the terrible hashing function to generate a terrible pseudo
    random number from our current position in space
uint2 prng = poorRNG(pos);

//Use the terrible random number to generate a pretty colour
//Blue is suppressed to make colours brighter
float3 col;
col.r = prng.x%256/256.0;
col.g = prng.y%256/256.0;
col.b = pow((prng.y+prng.x)%1024/1024.0,2);

//Use another noise function with some magic numbers to generate a
    pseudo random number used for building rotations. Only for tall
    buildings, though. If a building is tall enough, an extra block
    will be placed underneath it because it looks better.
float rotY = 0;
float heightOffset = 0;
if (cCityLodLevel == CITY_LOD_LEVEL_HIGH && baseheight > 100) {
  heightOffset = 50;
  rotY = 0.10f * noise2D(pos.x/1000.0f,pos.y/1000.0f);
}

//The maximum possible footprint a building can have
const float maxFootPrint = (CITY_GRID_SIZE - (BUILD_WIDTH*2))/2;

//Generate 2 pseudo random numbers between 1 and 4 inclusive
uint2 rn = prng%4 + 1;
//Calculate the footprint of each object within the city grid tile
```

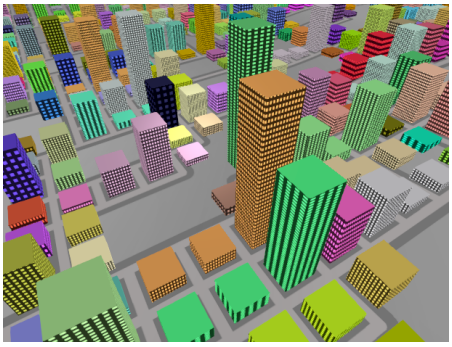
```
float2 fp = float2(maxFootPrint/rn.x,maxFootPrint/rn.y);

//The bottom left point in the buildable area
const float2 bl = float2(BUILD_WIDTH,BUILD_WIDTH);

//Store the maximum height after each iteration. This is used on
    the lower levels of detail to approximate a high level of detail
     building with just one block
float maxHeight = 0;

//Loop through all the objects in both x and y dimensions
for (uint i = 0; i < rn.x; i++) {
  for (uint j = 0; j < rn.y; j++) {
    //Calculate the position of the given object in space
    float3 p;
    p.xz = p1 + bl + 2*fp * float2(i+0.5f,j+0.5f);
    float2 pos2 = cCityTileCoords + p.xz;

    //Calculate the height of this new object through another
       random number. The height is based off the previous height
       or height seed, to prevent crazy looking buildings
    uint2 rnh = poorRNG(float2(pos2.x/120,pos2.y/98));
    float height = ( ((int)((rnh.x + rnh.y)%1024)) - 512)/512.0f;
    baseheight = baseheight + height * 0.15f * baseheight;

    //Set the y position of the building, based off terrain height
       and building height. The building is sunken slightly to
       account for changes in terrain height over the building area
    p.y = terrainHeight-10 + baseheight + heightOffset;

    maxHeight = max(maxHeight,baseheight);

    //Append the object to the buffer, but only if in the highest
       level of detail
    [branch] if (cCityLodLevel == CITY_LOD_LEVEL_HIGH) {
      Instance i0;
      i0.mPos = p;
      i0.mSize = float3(fp.x,baseheight-heightOffset,fp.y);
      i0.mColour = col;
      i0.mRotY = rotY;
      sInstance.Append(i0);
    }
  }
}

//If we are either in not the highest lod level or we need to add
    an extra block underneath a really tall building from the
    highest level of detail
[branch] if (cCityLodLevel != CITY_LOD_LEVEL_HIGH || heightOffset)
```

```
      {

    //break if the building is too short given the level of detail
    if (cCityLodLevel >= CITY_LOD_LEVEL_LOW && maxHeight < heightSeed
        /8 + minHeight) {
      return;
    }
    if (cCityLodLevel >= CITY_LOD_LEVEL_XLOW && maxHeight <
        heightSeed/4 + minHeight) {
      return;
    }
    if (cCityLodLevel >= CITY_LOD_LEVEL_XXLOW && maxHeight <
        heightSeed/2 + minHeight) {
      return;
    }

    //recalculate the footprint and position
    fp = float2(maxFootPrint,maxFootPrint);

    float3 p;
    p.xz = p1 + bl + 2*fp * float2(0.5f,0.5f);

    float bHeight;

    //If we are adding the new block, set the height to the offset
        from earlier
    if (heightOffset) {
      bHeight = heightOffset;
      fp+=5;
    }
    //Else it is the maximum height we got from earlier
    else {
      bHeight = maxHeight;
    }

    //Set the y coordinate
    p.y = terrainHeight-10 + bHeight;

    //Append the object
    Instance i0;
    i0.mPos = p;
    i0.mSize = float3(fp.x,bHeight,fp.y);
    i0.mColour = col;
    i0.mRotY = 0;
    sInstance.Append(i0);
  }
}
```

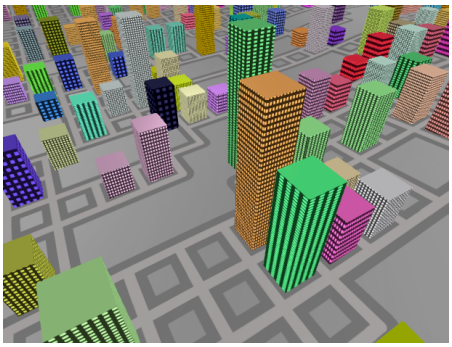
Each building level of detail can be seen in figure 6.6.



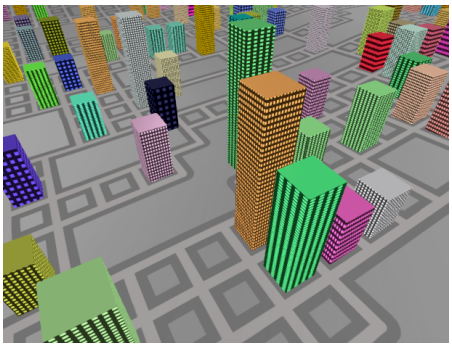
(a) High



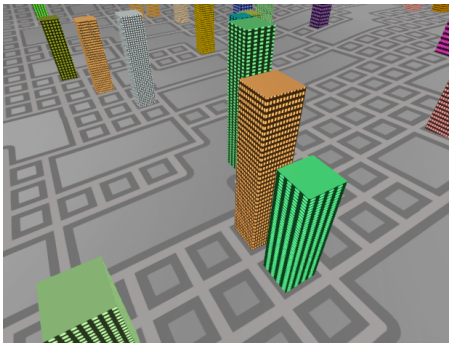
(b) Medium



(c) Low



(d) XLow



(e) XXLow

Figure 6.6: Building levels of detail

6.6 Potential Optimisations

The main problem with these algorithms is memory usage. For a world with a texture resolution of 256x256, 15 levels of detail and a tile dimension of 6, the GPU memory usage is over 600MB including mipmaps, just for `DistantTile` instances.

Even worse is `CityTile` memory usage, where, due to the requirement of having a buffer than can accept the maximum number of possible objects, a huge percentage of memory is allocated and then never used. In the demo application, 223MB is allocated for buildings and, in the worst case scenario where there are absolutely no objects created in the world, every last byte of this will be wasted.

Fixing `DistantTile` is quite easy, as the texture could potentially be put through one of the DirectX block compression algorithms, such as BC1. BC1 can store 16 pixels of `R8G8B8A8` texture data into just 64 bits of output, in a lossy `R5G6B5A1` format. This means an 8MB texture file can be compressed to just 1MB of data. The resultant texture will lose some detail, but the saved memory would be worth it. Fast GPGPU BC1 compressors exist, potentially allowing textures to be compressed with a minimal performance hit. [17]

BC7 is another `R8G8B8A8` compressor, but uses more computationally intensive encoding techniques to get a much higher quality image than BC1, though with half the compression ratio (an 8MB texture would only go down to 2MB). A GPGPU compressor does exist, but some preliminary benchmarks show that it is far too slow on the development platform to be feasibly implemented, where a 512x512 texture took 50ms to compress. [18]

Fixing `CityTile` memory usage is a little trickier as, in the current implementation, the utilised amount of the instance buffer is only known by the GPU. Some DirectX 11 functions allow memory to be copied about from a GPU buffer to another GPU buffer, but the CPU will both have to find the required size of the new buffer through querying the GPU about the utilised chunk of the old buffer, and it will have to actually allocate the buffer.

These tasks have quite a bit of CPU overhead and introduce some CPU-GPU synchronisation, something that will need to be intelligently planned to prevent any performance issues from arising. In the best case scenario, objects created by the GPU may only be drawable several frames after creation.

6.7 World Generation as a Whole

Throughout the duration of the project, the world function implementation was in a constant state of flux. Creating anything more than a somewhat mediocre implementation proved very difficult, especially with the limitations that come with programming the GPU. Above all else, performance needed to come first, and it was easy to write code that caused a sudden and tragic performance hit. In one instance, simply changing some parameters to be defined at run time rather than at compile time caused a 15% performance hit on the development platform.

The current implementation is adequate, but has some flaws and rough edges, and lacks a lot of optimisation. Fortunately, its main job was to create an interesting data set to work with, and it has done this very well. A lot of improvements can be made through subtle tweaks to the kernel, and fiddling with the many, many, unavoidable magic numbers that make up its behaviour.

Both the `DistantTile` and `CityTile` kernels have been designed to be as modular as possible. So long as the `DistantTile` kernel writes data out to the three textures, and so long as the `CityTile` kernel appends objects to the buffer, absolutely anything could occur within the kernel and all of the other systems will work exactly as required, with at most some trivial changes to some compile time constants. The job scheduler will adapt to any increases or decreases in execution time to ensure a smooth experience.

6.8 Scheduling

In computer graphics, the smoothness of an application depends on two factors: the frame rate, and the frame jitter.

Frame rate is a measure of how many frames an application can push out over a given time period. Frame jitter is the delta in the period between subsequent frames. For an optimal experience, the frame rate needs to be as high as possible, and the jitter needs to be as low as possible.

By running compute jobs on the GPU, some computational resources are taken away from rendering things to screen, reducing the average frame rate. This in itself is fine, but only on the assumption that the duration of jobs on the GPU is constant across all frames. If the duration changes adversely between frames, the result is a large amount of frame jitter, causing frame stuttering. [19]

The task of the scheduler is to maintain a constant duration of compute jobs per frame. The scheduler itself is not a discrete object or class; instead, its logic

is spread across the main `Renderer` class and the `Generator` class.

In figure 6.1, you can see that the method `Generate()` takes in `FLOAT pMaxRuntimeSeconds`, defining the amount of time in seconds the generator needs to run for. In the main rendering loop, the `Renderer` class will call this function, giving the required time.

The method used to calculate this time is incredibly straightforward. A value called `FLOAT savedFrameTime` is initialised to 1/60, or the frame period of 60 frames per second. Every frame, the `Renderer` class checks if there is anything in any of the `Generator` queues, by calling `BOOL hasGeneratables()`. If there are none, the current frame period is stored in `savedFrameTime`. Else, the `Generate()` function is called with a `pMaxRuntimeSeconds` equal to `savedFrameTime` $*$ `FrameTimePercentage`.

`FrameTimePercentage` is a floating point value between 0 and 1. It is the percentage of time per frame (and by extension, per second) that the `Generator` should run for. Through some subjective and thoroughly unscientific testing, this value was set 0.15, or 15%, as it was deemed to provide the best balance between giving enough time for things to load without causing a noticeable hit to the frame rate. As this value is so subjective, it will be different for different people on different machines, as the duration of any given job will vary based on the machine. Ideally, a more intelligent algorithm that dynamically calculates `FrameTimePercentage` should be implemented, that accounts for the capabilities of different machines while ensuring as little frame jitter as possible.

The reasoning behind only updating `savedFrameTime` when the generator has no tiles queued is because the act of generating will increase the frame period. By storing and reusing a generation free baseline time, `pMaxRuntimeSeconds` should stay relatively consistent, even after many continuous frames of generation.

Every frame, the `Generate()` method will continually dispatch jobs until either the tile queue is empty or until the estimated duration of jobs that have been dispatched in the current frame is greater than `pMaxRuntimeSeconds`.

Estimating job duration on a GPU is particularly difficult, as the CPU and GPU work asynchronously. There is an indeterminate time between a job being dispatched to the GPU and the work being performed, and simply having the CPU wait around and directly measure the time taken causes stalls in the GPU pipeline. Instead, what is used are DirectX 11 GPU queries.

Within the `Generator` class there are two methods, `ProcessDT()` and `ProcessCT()`, which dispatch `DistantTile` and `CityTile` jobs to the GPU, respectively. Inside these methods, DirectX 11 timing queries are used to time jobs.

DirectX calls are quite expensive to make, so creating and dispatching a query for every single job sent to the GPU would consume quite a lot of CPU time. Instead, a maximum of one query is in flight for each type of tile.

Upon the very first call of a `Process__()` method, the DirectX queries are created, and they are used to time the very first job dispatched. However, as the GPU works asynchronously, the CPU cannot simply find out how long the job took. Instead, it continues to send out jobs without queries until the next frame, where it checks if the query has been returned. If it has, it then uses the job duration to update the estimated job duration. Every job that runs returns this estimated duration, which is conservatively initialised to 0.01s for both types of tile before the first query is complete.

Due to differing workloads, the time estimation equation is slightly different for each of the tile types. For `DistantTile`, each tile will always produce 3 textures of identical resolution, and each will always have the same amount of work groups dispatched at a time. As a result, the time estimation equation is simply a weighted average, currently calculated as shown in figure 6.7. On the other hand, `CityTile` instances will have a variable number of work groups, dependent on tile size. The equation to calculate the estimated time is slightly modified, as shown in figure 6.8.

```
estimatedTime = 0.8 * estimatedTime + 0.2 * queryTime
```

Figure 6.7: Code to update estimated time for a Distant Tile job

```
estimatedTime = 0.8 * estimatedTime + 0.2 * (queryTime/numThreads)
```

Figure 6.8: Code to update estimated time for a City Tile job

When `ProcessCT()` returns an estimate of the time the job took, it multiplies the saved time estimate by the number of threads required to create the tile. `ProcessDT()` does not require this extra step.

6.9 Random Number Generation

The GPU kernels have two methods of pseudo random number generation; simplex noise and hashing. Each generator has different characteristics, making them suited for different tasks.

6.9.1 The Hash Function

The hashing function, affectionately named `PoorRNG`, is not very good. Its code can be seen in figure 6.9.

```
uint2 poorRNG(float2 xy)
{
  uint m_z = asuint(asuint(xy.x) - xy.y);
  uint m_w = asuint(asuint(xy.y) + xy.x);

    m_w = ((m_z >> 16) ^ m_w) * 0x45d9f3b;
    m_w = ((m_w >> 16) ^ m_w);

  m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);

  uint r1 = (m_z << 16) + m_w;

  m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);

  uint r2 = (m_z << 16) + m_w;

  return uint2(r1,r2);
}
```

Figure 6.9: PoorRNG

When designing the function, the requirements were as follows:

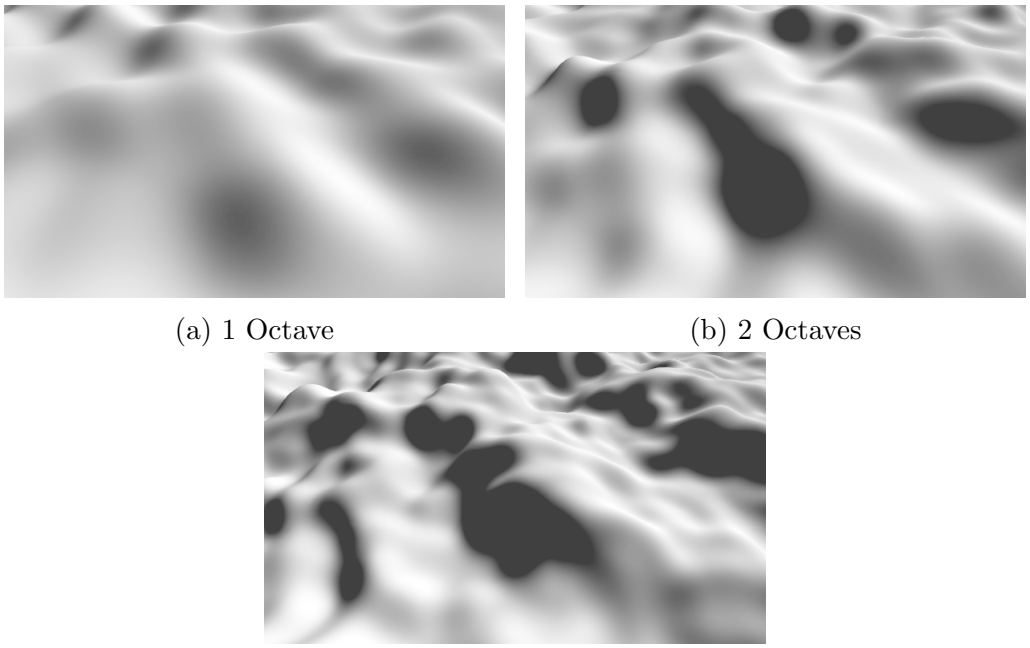
1. The function must take in a vector of two floating point numbers (i.e. a position in 2D space) and return a vector of two unsigned integers.
2. The function must mix together the inputs; each output integer cannot simply be a hash of a single one of the input floats.
3. The function should be completely deterministic. The same two inputs will always produce the same two outputs.
4. The function must be fast, with absolutely no memory lookups.
5. All inputs should have a one to one mapping with all outputs.
6. The function must actually hash the inputs.

Unfortunately, due to a serious lack of experience with anything to do with cryptography, `PoorRNG` does not succeed at actually hashing the inputs all that well. When viewing the output of `PoorRNG`, there are quite obvious patterns and clusters of numbers with neighbouring inputs values. Unexpectedly, due to the uses of the random numbers (colours, building heights, etc.), these characteristics actually provide some interesting texture to the world. Despite these useful characteristics, its fatal flaw is its extreme susceptibility to floating point error, even at relatively short distances from the origin, manifesting as very repetitive roads and buildings.

6.9.2 Simplex Noise

Simplex noise is a multi dimensional continuous noise function. The implementation used is a cut down, ported and optimised version of a C++ implementation by Eliot Eshelman. [20]

By combining multiple octaves of Simplex noise together in different ways, terrain can be simulated. Figure 6.10 shows 3 worlds created with a different number of octaves of simplex noise.



(a) 1 Octave

(b) 2 Octaves

(c) 3 Octaves

Figure 6.10: Octaves of simplex noise

Porting of the algorithm from C++ to DirectCompute was very straightforward, as the syntax of DirectCompute and C++ is very similar, plus DirectCompute supports the multidimensional static const arrays that the original implementation used as a look up table. Performance of this straight port, however, was terrible. Figure 6.11 shows a short section of the original C++ implementation. All 15 memory lookups for the algorithm are contained within this section. For a CPU, this many lookups are trivial, as the two look up tables, at a grand total of 2560 bytes, can easily fit inside the L1 cache of just about any modern processor. For a GPU this is tragic, as it is near impossible to hide the latency of this many dependent memory lookups in such a short section of code.

Optimisation was surprisingly straightforward. An article by Simon Green, at NVIDIA, explains how he optimised 3D Simplex Noise for the GPU by replacing the recursive memory lookups by a single lookup, by creating a buffer with all the permutations of these recursive lookups. In the 3D case, the result was reducing 22 memory lookups to just 9. Memory usage is higher, but for a GPU this is not really an issue. [21]

For 2D noise, we can do even better. Figure 6.12 shows the ported version of the same section of code as figure 6.11. By encoding all permutations of memory lookups into a single uint buffer of length 65536, the previous 15 integer memory lookups were reduced to just one single unsigned integer lookup, plus lots of bit shifts to extract the encoded data. In addition, conditionals were replaced with predication to prevent any unnecessary branching. The code for this ported version can be found in `SimplexNoise.fx`, and the code used to generate the buffer can be found in `Generator.cpp`. The result of removing this many memory lookups was an overall 3x speedup of the entire `DistantTile` GPU kernel, meaning the speedup just from within the simplex noise function was even greater than this. There is still some room to optimise this code, as vectorisation has not even been explored.

```
//Contains values that equal either -1, 0 or 1
static const int grad3[12][3];
//Contains only values between 0 and 255 (i.e. a uint8)
static const int perm[512];
{
  //...Do some work ...//
    int gi0 = perm[ii+perm[jj]] % 12;
    int gi1 = perm[ii+i1+perm[jj+j1]] % 12;
    int gi2 = perm[ii+1+perm[jj+1]] % 12;

    float t0 = 0.5 - x0*x0-y0*y0;
    if(t0<0) n0 = 0.0;
    else {
        t0 *= t0;
        n0 = t0 * t0 * dot(grad3[gi0], x0, y0);
    }

    float t1 = 0.5 - x1*x1-y1*y1;
    if(t1<0) n1 = 0.0;
    else {
        t1 *= t1;
        n1 = t1 * t1 * dot(grad3[gi1], x1, y1);
    }

    float t2 = 0.5 - x2*x2-y2*y2;
    if(t2<0) n2 = 0.0;
    else {
        t2 *= t2;
        n2 = t2 * t2 * dot(grad3[gi2], x2, y2);
    }
    //...Do more work ...//
}

float dot( const int* g, const float x, const float y ) { return g
   [0]*x + g[1]*y; }
}
```

Figure 6.11: C++ simplex noise code snippet

```
//Length = 65536
StructuredBuffer<uint> simplexBuffer : register(t0);
{
  //...Do some work ...//
  uint simpData = simplexBuffer[256*ii+jj];
  simpData = simpData >> (16*j1);

  float t0 = 0.5 - x0*x0-y0*y0;
  t0 = (t0<0) ? 0 : t0;
  t0 *= t0;
  n0 = t0 * t0 * dot20(simpData,x0,y0);

  float t1 = 0.5 - x1*x1-y1*y1;
  t1 = (t1<0) ? 0 : t1;
  t1 *= t1;
  n1 = t1 * t1 * dot21(simpData, x1, y1);

  float t2 = 0.5 - x2*x2-y2*y2;
  t2 = (t2<0) ? 0 : t2;
    t2 *= t2;
  n2 = t2 * t2 * dot22(simpData, x2, y2);
  //...Do more work ...//
}

float dot20(uint simpData, const float x, const float y) {
  int gx = (int)(simpData & 3) - 1;
  int gy = (int)((simpData & 12) >> 2) - 1;
  return gx*x+gy*y;
}

float dot21(uint simpData, const float x, const float y) {
  int gx = (int)((simpData & 48) >> 4) - 1;
  int gy = (int)((simpData & 192) >> 6) - 1;
  return gx*x+gy*y;
}

float dot22(uint simpData, const float x, const float y) {
  int gx = (int)((simpData & 768) >> 8) - 1;
  int gy = (int)((simpData & 3072) >> 10) - 1;
  return gx*x+gy*y;
}
```

Figure 6.12: Ported and optimised simplex noise code snippet

There is one issue with the current implementation, and that is to do with floating point error. As shown in figure 3.8, as the magnitude of a floating point number increases, so does the magnitude of the error. In the context of the procedural world, this can manifest in some unfortunate ways, such as quantisation of terrain in the world and terrain tiles that simply don't fit together properly, as can be seen in figure 6.13.

One possible way around this is by implementing 3D or 4D simplex noise, and folding the 2D position in space into multiple dimensions such that it wraps around onto itself, creating a perfectly continuous world. An actual implementation of this requires far more thought, however.

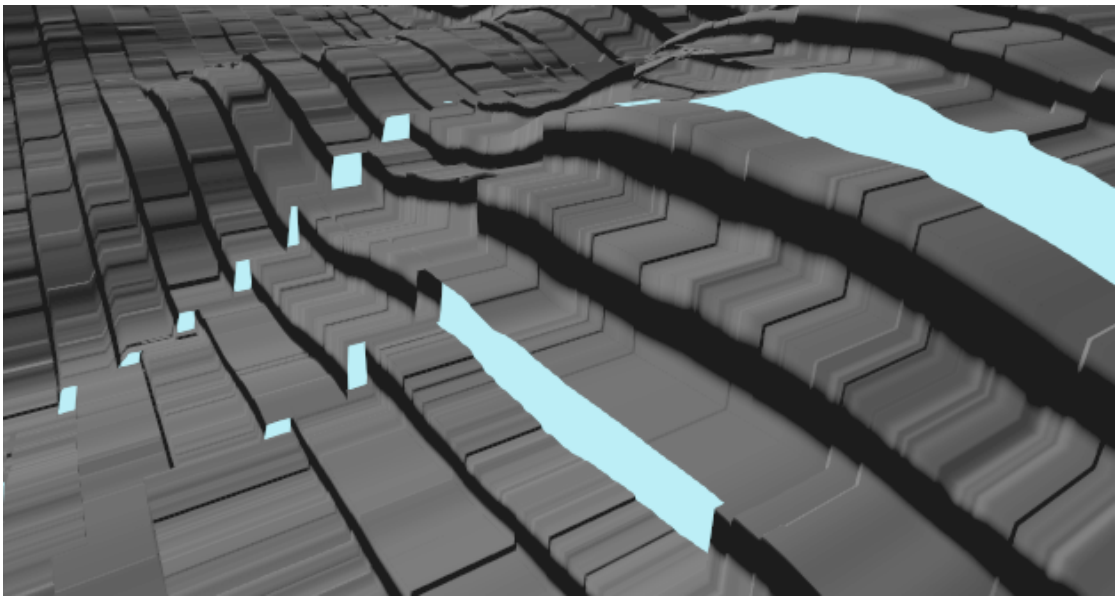


Figure 6.13: Floating point error with the camera at [2000000000,0,0]

7 Conclusion

Analysing the algorithms and the demo application in any meaningful way is very difficult, as the results are so subjective and depend heavily on the target platform. Measuring throughput of the GPU kernels in terms of megapixels or objects per second is not too useful without any direct comparisons to similar applications that implement both procedural terrain and object generation, and running the application on different platforms only serves to compare these platforms against each other.

What can be done is look at some of the features and traits of the demo application, a screenshot of which can be seen in figure 7.1, and determine what they mean.

- The world in the demo application and scene runs always at the target frame rate (30-60 fps) on the low end development platform, at a 1366x768 resolution. The platform is actually pixel fill rate limited, with the frame rate skyrocketing by running the application at lower resolutions.
- For the demo world, the maximum view distance is 2088km, with 15 different levels of detail, a base tile size of 42.5m, and a tile dimension of 6. The texture resolution is also set to 256x256. This means that, at the highest level of detail, the resolution is 6 pixels per metre, and at the lowest it is 3.68×10^{-4} pixels per metre. The level of detail system means that there really is no upper bound on view distance, though some issues do arise with the Z-buffer not having a high enough precision.
- On the development platform, the camera speed can be set to 2000km/s and still maintain the 2088km view distance in the demo application at the lowest level of detail, without any noticeable hit to performance. Above this point, tiles cannot be created quickly enough.
- The time taken to load the demo world, from the time the Lua script begins to the time every single `CityTile` and `DistantTile` is created is approximately 1.6 seconds. On average, each `DistantTile` takes 2.5ms to create (with 3 256x256 textures), and in a single second 170,520 buildings can be created. Shader compilation time is actually the limiting factor in application start up speed.
- Due to the scheduler, the frame rate remains stable under all circumstances, regardless of GPU load or the number of tiles that need to be created.

- The application can scale across platforms. By modifying the level of detail parameters, larger worlds with higher resolution textures and denser meshes can be created to stress high end GPUs, and vice versa for particularly low end GPUs.
- The created worlds are totally configurable through simple Lua scripts, from the colour of the land to the layout of the terrain. In addition, the GPU kernels are incredibly modular, and so long as some conditions are met, they can be modified with minimal, if any, changes to the underlying application. They could even be dynamically modified and recompiled at run time.
- The maximum world size is approximately $1 \times 10^{10}\,\mathrm{m}^2$, defined only by the point that floating point error becomes subjectively unacceptable. This is about 20 times the surface area of the earth.
- CPU utilisation is consistently low, even with little optimisation. This is expected, as the primary job of the CPU is to tell the GPU what to do. On the development platform, CPU utilisation sits at approximately 35% of a single processor when running at 1366x768 resolution. However, this number is particularly unreliable as the processor was observed consistently switching between low and high power states, so actual CPU utilisation is slightly lower.

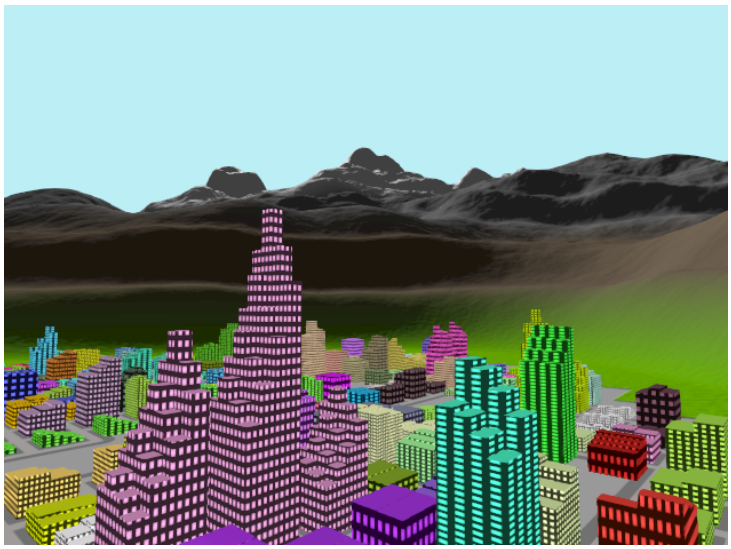


Figure 7.1: A screenshot of the demo application

8 Future Work

As the scope of the project was quite large, and the available time was quite short, there are many ways to improve upon the work done.

- Though the scheduler works quite well, it needs to be manually tuned platform by platform for optimal results. A more intelligent scheduler could be written to ensure the best possible experience for the user.
- Memory utilisation can be decreased by a massive factor through simply implementing the techniques described in section 6.6.
- `CityTile` instances currently only support a single type of mesh at a time. After the memory utilisation issues are resolved, adding in support for more types of objects simply requires adding more appendable instance buffers per tile, each with a different type of mesh.
- The world is currently very flat. Planets typically are not. A method needs to be determined that can tile a sphere with quadrilaterals. One possible method is through use of a "spherificated" cube, as can be seen in figure 8.1, plus replacing 2D noise with 3D noise to allow the created terrain to correctly wrap around on all sides.
- While already quite fast, the simplex noise implementation could be optimised further through vectorisation. In addition, the many issues with floating point error need to be resolved.
- The hash function `PoorRNG` is quite awful and needs a complete overhaul.
- More research needs to be done into road generation and building placement, as the current system has massive constraints put upon it for performance and simplicity reasons.
- Research into tessellation stopped very early due to excellent results from just setting a simple constant factor. However, the heightmaps produced are typically of significantly higher resolution than the actual wireframe mesh, so there is a lot of wasted data. More research can definitely be done to dynamically modify tessellation factors to improve detail where necessary, such as for at the silhouettes of objects, and reduce detail where it just isn't needed. The result would be fewer vertex transforms, better performance and even better results.

- Individual tiles can neither be drawn nor created in smaller chunks. Either the tile is drawn in its entirety or it isn't, and either the tile is created in its entirety or it isn't. There simply is no middle ground. Being able to draw a subset of a tile would improve overall quality, for the benefit shown in section 4.2.3, and being able to create a subset of a tile would allow the tile to be created over multiple frames, spreading the load if necessary.

- In the initial project plan, the intention was to use displacement mapping for improving quality on the objects created in the world. Due to time constraints this wasn't possible, but is certainly something that could help break the current monotony of the city. Buildings also only have a very basic texture map applied to them that simulates windows and lights. This could be massively improved.

- All Lua scripts run synchronously on the rendering thread. Adding the ability to let them run asynchronously on a different thread could allow for a lot more functionality, though certain safety measures would need to be taken to prevent any concurrency bugs.

- On the CPU side, the entire application uses just one thread. There are many opportunities to use simple task based parallelism to spread the load across multiple threads, though as the application is so GPU limited the performance improvement would be mostly non existent.

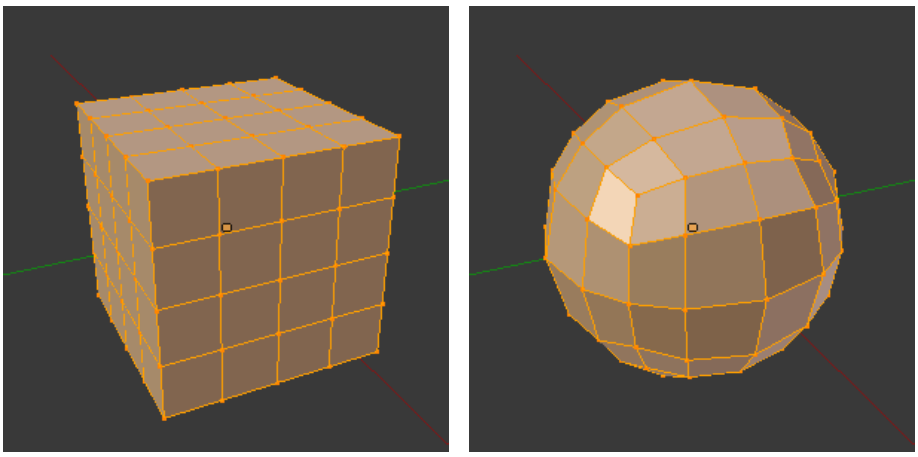
- The graphics engine supports up to 1024 simultaneous light sources visible in a scene. Letting the procedural generator place lights, such as street lights, could be an interesting way to utilise this currently unused functionality.

- Due to the focus of the project being primarily on the algorithms that support the procedural generation, not much time was spent making the actual output graphics all that "shiny". Adding more post processing, improving the lighting systems and fiddling with world generation parameters could definitely improve aesthetics.

- Some research was done into combining satellite images and maps with the generator. A proof of concept demo was made from the textures in figure 8.3, the result of which can be seen in figure 8.2, but time constraints prevented any further work.

- It would be nice to try and implement this project on a hUMA architecture, with a shared CPU and GPU memory space, allowing for much better interoperability between the parallel GPU code and branchy CPU code.

- An occlusion culling implementation would allow for many more objects to be placed in the world without affecting the frame rate too much. This was initially a part of the project but had to be cut due to time constraints.
- Due to shader compilation, the initial load time is quite slow. By saving the shader binary to disk on the first run, this initial load could be almost completely eliminated.

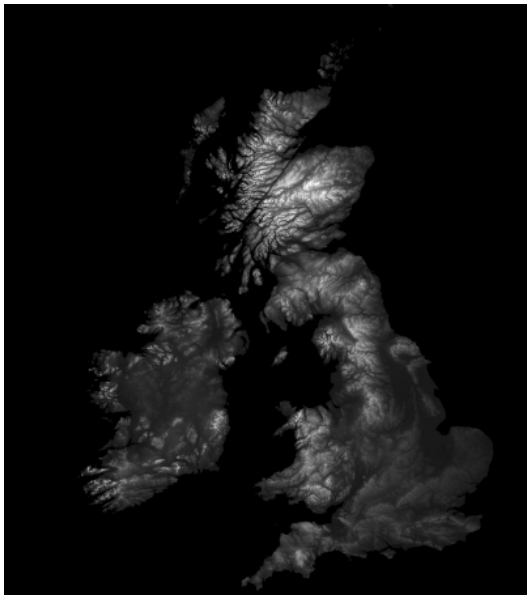


(a) Original (b) Spherificated

Figure 8.1: Cube spherification

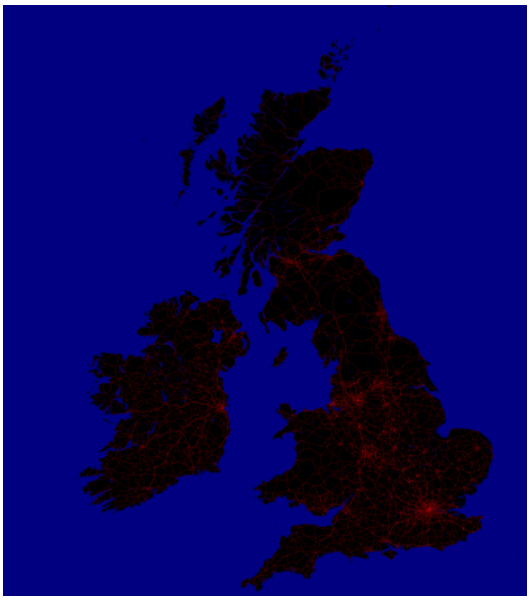
Figure 8.2: Generator combined with mapping data

(a) City density map



(b) Height map

(c) Satellite map



(d) Water and road map

Figure 8.3: UK Mapping dataset

References

[1] A. L. Shimpi, *The Xbox One: Hardware Analysis & Comparison to PlayStation 4*. 2013. Retrieved 2013-06-20 from http://www.anandtech.com/show/6972/xbox-one-hardware-compared-to-playstation-4.

[2] M. Houston, *General Purpose Computation on Graphics Processors (GPGPU)*. Stanford University, 2005. Retrieved 2013-01-07 from http://graphics.stanford.edu/~mhouston/.

[3] A. Burnes, *Exclusive: PhysX In Batman: Arkham City - First Look*. 2011. Retrieved 2012-12-30 from http://www.geforce.co.uk/whats-new/articles/exclusive-physx-in-batman-arkham-city-a-first-look.

[4] T. Ni, *DX11 Tessellation*. NVIDIA, 2010. Retrieved 2013-01-07 from http://www.nvidia.asia/content/asia/event/siggraph-asia-2010/presos/Ni_Tessellation.pdf.

[5] R. Garside, *Procedural Textures: Gaming's Future*. 2006. Retrieved 2013-01-03 from http://www.bit-tech.net/gaming/2006/11/09/Procedural_Textures_Future_Gam/1.

[6] J. Rossignol, *Interview: Codies on FUEL*. 2009. Retrieved 2013-01-03 from http://www.rockpapershotgun.com/2009/02/24/interview-codies-on-fuel/.

[7] S. Young, *Procedural City, Part 5: Fixes*. 2009. Retrieved 2013-01-04 from http://www.shamusyoung.com/twentysidedtale/?p=3059.

[8] S. Young, *pixelcity - A procedural city-generating program*. 2009. Retrieved 2013-01-04 from http://code.google.com/p/pixelcity/downloads/list.

[9] R. Software, *Luabind*. Retrieved 2013-06-20 from http://www.rasterbar.com/products/luabind.html.

[10] A. Lauritzen, *Deferred Rendering for Current and Future Rendering Pipelines*. 2010. Retrieved 2013-06-20 from http://visual-computing.intel-research.net/art/publications/deferred_rendering//.

[11] Microsoft, *DrawIndexedInstanced method*. Retrieved 2013-06-20 from http://msdn.microsoft.com/en-us/library/windows/desktop/ff476410%28v=vs.85%29.aspx.

[12] T. Lottes, *FXAA*. NVIDIA. Retrieved 2013-06-20 from http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf.

[13] P. Euphoria, *Voronoi Noise*. 2011. Retrieved 2013-06-20 from http://www.pixeleuphoria.com/node/34.

[14] Microsoft, *AppendStructuredBuffer*. Retrieved 2013-06-20 from http://msdn.microsoft.com/en-us/library/windows/desktop/ff471448%28v=vs.85%29.aspx.

[15] N. Darnell, *Chaining Compute Shaders*. 2011. Retrieved 2013-06-20 from http://www.nickdarnell.com/tag/appendstructuredbuffer/.

[16] Microsoft, *CopyStructureCount method*. Retrieved 2013-06-20 from http://msdn.microsoft.com/en-us/library/windows/desktop/ff476393%28v=vs.85%29.aspx.

[17] N. Reed, *Understanding BCn Texture Compression Formats*. 2012. Retrieved 2013-06-20 from http://www.reedbeta.com/blog/2012/02/12/understanding-bcn-texture-compression-formats/.

[18] C. Walbourn, *BC6H/BC7 DirectCompute Encoder Tool*. 2013. Retrieved 2013-06-20 from http://code.msdn.microsoft.com/BC6HBC7-DirectCompute-35e8884a.

[19] R. Smith, *AMD Comments on GPU Stuttering, Offers Driver Roadmap & Perspective on Benchmarking*. 2013. Retrieved 2013-06-20 from http://www.anandtech.com/show/6857/amd-stuttering-issues-driver-roadmap-fraps.

[20] E. Eshelman, *Simplex Noise for C++ and Python*. 2012. Retrieved 2013-06-20 from http://www.6by9.net/simplex-noise-for-c-and-python/.

[21] M. Pharr and R. Fernando, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005. Chapter 26, Simon Green, NVIDIA.

9 User Guide

9.1 Compilation

The source code repository for the project can be found at `https://github.com/n1kunj/eie-ic-renderer`.

Compilation requires Visual Studio 2012, the Windows 8 SDK and the DirectX SDK.

Loading the eie-ic-renderer.sln in the source folder will open Visual Studio, and the code can be easily compiled. Ensure that 64-bit compilation is used, as the 32-bit version crashes due to SIMD alignment issues.

9.2 Running the Code

To run the code, eie-ic-renderer.exe needs to be run in the same directory as the Media folder, which contains the shaders and the Lua scripts.

The initial load is quite slow as all the shaders need to be compiled at run time. Once compilation is finished, the application window will appear and the Lua script "mountains.lua" will automatically be run. Dependent on GPU performance, loading time could range from instant to 3 seconds. Within the application, the controls are as follows:

W, S, A and D will move the camera forwards, backwards, left and right, respectively.

Space and Ctrl will move the camera up and down, respectively.

Holding the Left Mouse Button will change the camera angle.

F will toggle locking the mouse to the window, allowing for the camera angle to be changed without holding down the left mouse button. There is a minor bug where, if the window is not positioned over the centre point of the screen, mouse locking will not work correctly.

R will recompile all shaders. This will lock up the program for a short amount of time.

Tilde will toggle between the developer console and the frame rate counter.

The developer console is a fully functional Lua interpreter, though only a small number of libraries have been loaded in. From within the developer console, the following commands can be used to change some settings:

SET.wireframe = {1,0} will switch wireframe on or off.

SET.cameraspeed = 100 will set the camera speed to 100m/s. The default speed is 1000m/s.

SET.fxaa = {1,0} will turn FXAA on or off.

move(x,y,z) will set the camera position to (x,y,z). The y axis points directly upwards.

print(x) will print the value or variable x to screen.

run("water.lua") will run the script water.lua in the Media\Lua folder, which creates a simple water texture.