GENERATED VOXEL WORLD WITH PROCEDURAL SOUNDTRACK

J BRETHERTON 1800231

OVERVIEW

This application consists of an endlessly[1] generated voxel world and a procedurally generated soundtrack.

OUTLINE

HOW TO USE THE APPLICATION

ACTION	CONTROL
Move forward / backwards	W / S
Move left / right	A / D
Move up / down	E / Q
Change render distance	ImGui slider
Enable wireframe	ImGui tickbox

NOTES ON USE

The application uses the standard wsadeq movement provided by the framework.

Wireframe mode is also provided by the framework.

An additional slider was added to control the “render distance”. The render distance varies the number of chunks loaded, with this number of chunks being loaded in each cardinal direction around the player . This does not affect the camera’s “render distance” (far clipping plane) which remains unchanged at 1000 units no matter how the player sets this slider.

Note: it is not advised to set the render distance too high as it can significantly affect performance. The (admittedly not overly powerful) machine this was developed on can handle a render distance of four ($(4*2+1)^2$ = 81 chunks loaded) comfortably but begins to struggle when loading in chunks at higher render distances. The application is not particularly designed to run at these higher settings. They are included mainly for demonstration.

The application may also take some time to load initially.

[1] The world is split up into chunks whose coordinates are hashed as integers, so after a player walks a number of chunks past `INT_MAX`, this would begin to repeat, so the limit is realistically 2.1×10^9 chunks, effectively endless to a casual player.

GENERAL FEATURES

The application uses the directX-11 framework provided (Robertson, n.d.) and the instance cube shader that is a part of this. This shader combines a list of cube positions (render queue[2]) into a single render call allowing for very efficient rendering of a large amount of geometry.

The soundtrack makes use of the SFML library (Gomila, n.d. (a)) to output the samples generated to speakers.

TERRAIN

The world is split up into 16x64x16 voxel "chunks". These chunks can be loaded and unloaded to the render queue very efficiently, allowing for a player to walk about the world as it loads in and out around them in real time.

Chunks are cached in memory after generation, allowing for recently traversed areas to load in faster and more efficiently.

The chunks use fractional Brownian Perlin noise to generate a coherent yet varied terrain, with large hills and deep valleys.

The voxel's texture is generated by blending between three textures based on the height of the voxel. Hills are covered in grass, which smoothly turns to silty sand, which turns to rock as the terrain gets to its deepest.

SOUNDTRACK

The soundtrack generates note samples from a sine wave and passes these samples through a buffer to an object deriving from SFML's `sf::SoundStream` (Gomila, n.d. (b)), which plays them.

Notes of a given pitch are created through modifying a standard 440hz A440 pitch (ISO, 1975) through 12-TET tuning to generate each note in an A Major scale.

The generated notes are combined into triads or seventh chords in root, first, or second inversion, and randomly move through the octaves to provide variation without changing the chord's harmonic function.

The system uses a Markov chain with each chord being a state and moves between states through rules based on a functional harmonic progression (Hutchinson, 2020).

The system generates a melody through a system of rules. Either stepwise or "leaping" motion is randomly chosen to lead from the previous note. Stepwise will move the note one to two notes up or down in the scale whereas leaping motion will move to a random note within the scale, favouring notes from the more stable and more consonant pentatonic scale (which is a subset of the major scale) associated with the key (A Major pentatonic).

[2] Strictly speaking this is not a render queue, it is a simple array of positions that are sent to the instance shader. However, for simplicity and readability, I will be referring to this as "the render queue" for the remainder of this document.

TECHNIQUES USED

An in depth explanation of the procedural techniques used, the theory behind them, and why they were chosen.

TERRAIN

TERRAIN GENERATION ALGORITHM

MOTIVATION

The terrain needs to vary in height, as natural terrain does. For this, some kind of random procedure is needed to produce a heightmap, which can then be sampled or calculated per texel on the fly to give the height of each voxel coordinate. This function needs to have two main properties:

1. The function must be pseudorandom. It must be unpredictable over any range of inputs but must always map the same input to the same output. This way we can simply pass the position of our voxel in as an argument, and the terrain will remain at the same height even if the chunk is unloaded and re-generated.
2. The function must be continuous. Any step in any direction must not result in a large jump in the output value. This way there will be no gaps in our terrain surface, even along the borders of chunks.

For this, the best candidate is a noise function (Bevins, 2003).

NOISE

A noise function is a pseudorandom function that is coherent. Such that, for a given input, a small delta will result in a small but unpredictable change in output value, and a large delta will result in a random output value. Perlin noise is a famous example of this (Perlin, 1999) using a gradient noise.

KEN PERLIN'S ALGORITHM

The application uses an adaptation of Perlin's (1997) original code. Perlin noise generates a noise value based on an input coordinate, depending on the dimensions of the noise function. In this case, 2D noise.

Perlin noise (in 2D) creates a grid and, at each vertex of this grid, assigns a unit vector with a pseudorandom direction. This is achieved in the application through the following function:

```
PerlinNoise::Vector2 PerlinNoise::RandomGradient(PerlinNoise::Vector2 position)
{
    float ix = position.x;
    float iy = position.y;
    float random = 2920.f*sinf(ix*21942.f+iy*171324.f+8912.f)*cosf(ix*23157.f*iy*217832.f+9758.f);
    return Vector2 { cosf(random),  sinf(random) };
}
```

Where a random number is generated using the X and Y position of the vertex of a square in the grid, and this number is then used to construct a unit vector pointing in a random direction.

When the noise function is called with its "position" parameters, the square of the grid that this point would fall in is calculated. It is at this stage the four random vectors at the vertexes of the square are calculated.

The grid square that a point resides in is deduced by simply flooring the x and y coordinates, thereby giving the coordinate of the bottom left vertex of the square, which can be used to identify which square the point is in.

```
// grid cords
int x0 = floorf(x);
int x1 = x0 + 1;
int y0 = floorf(y);
int y1 = y0 + 1;
```

This newly-calculated bottom left position is then subtracted from the position of the passed-in point to get the position of the point within its square (that is, relative to the bottom left corner).

A vector from each vertex to the passed-in point is created, and the dot product of this vector and the random vector at each vertex is calculated. This gives each vertex a random value between -1 and 1, as can be seen below in Figure 1.

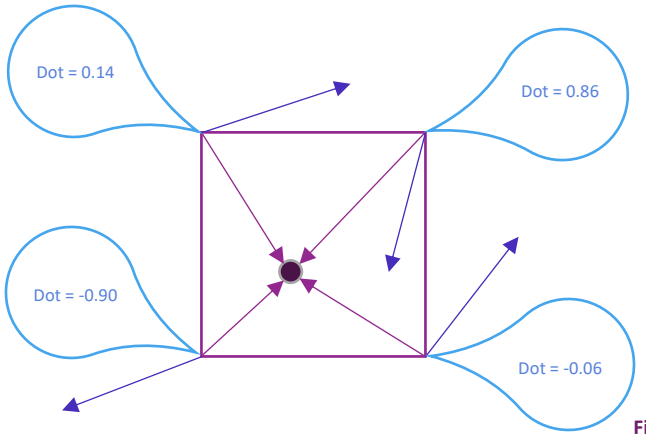

Figure 1

These four values are then interpolated together by an interpolation function based on the non-linear function: $0 = 3t^2 - 2t^3$

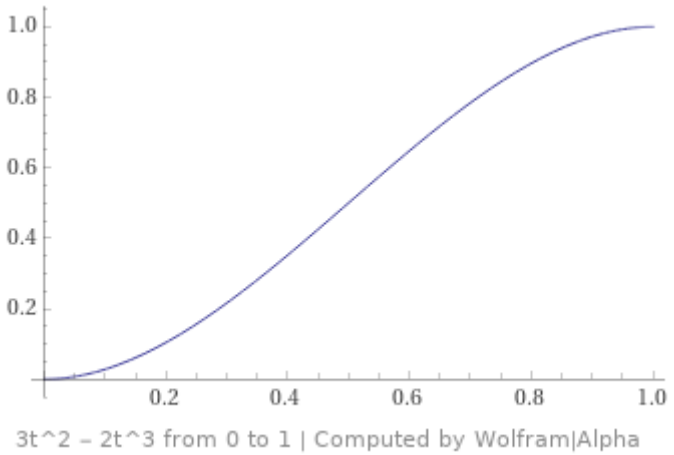

Figure 2

It's notable here that this function has a zero derivative at 0 and 1.

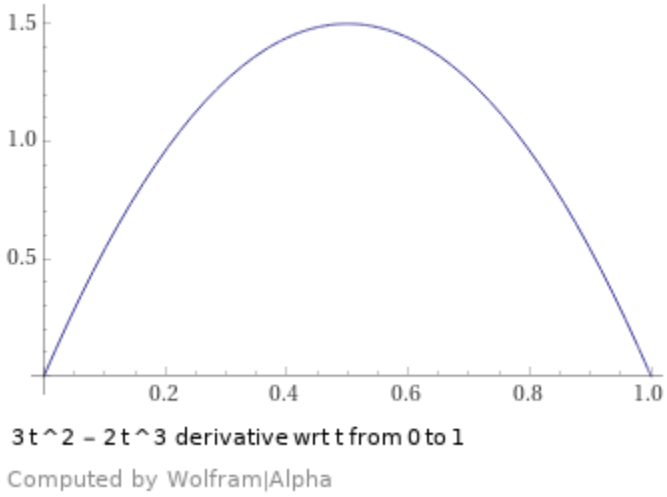

Figure 3

This means that the rate of change near a grid vertex approaches 0, making input values close to a grid vertex have values very close to 0 which do not suddenly change.

In code, this function is expressed:

```
float PerlinNoise::Interpolate(float a0, float a1, float w)
{
    return pow(w, 2.0) * (3.0 - 2.0 * w) * ((double)a1 - a0) + a0;
}
```

This results in a noise function which smoothly transitions between 0 and 1 across a 2D input space, allowing it to be sampled directly as a height-map without any discontinuities while giving a pseudorandom value for each input.

FRACTIONAL BROWNIAN MOTION

The Perlin noise at this stage is still too simple to create any kind of interesting terrains. However, several passes of Perlin noise combined can give far more interesting structures.

Using a Perlin noise function, two variables can be used to manipulate the resulting texture generated: frequency and amplitude. The frequency is a constant that the input to the noise function is multiplied by to increase or decrease the distance that one unit step along the grid is. Conceptually this is similar to zooming in or out of the output texture. Amplitude is the constant that the output of the noise function is multiplied by to map the output range from -amplitude to +amplitude.

By combining multiple passes of the noise function, starting with a high amplitude and low frequency, and decreasing the amplitude as the frequency is increased, it is possible to re-create the fractal behaviour existing in nature where large hills have smaller undulations on them which themselves have yet smaller bumps on them. This technique is known as Fractional Brownian Motion (FMB).

This application uses constant values of 18 for the amplitude and $^1/_{128}$ for the frequency, decreasing the amplitude and increasing the frequency by a factor of four over a total of six iterations.

USE IN THIS APPLICATION

At generation, each chunk passes the world position of each of its voxels to a function that tests if the Y position is below or above the height value given by the FBM function at that location. The operation is as follows:

1) For each column of voxels in the chunk:
    a) For each voxel in the column:
        i) If the voxel is solid:
            (1) If the voxel is visible (not entirely surrounded by solid voxels):
                (a) Add a cube in this position to the render queue.
            (2) Else if the voxel is not visible:
                (a) Do nothing.
        ii) Else if the voxel is not solid (above the ground level):
            (1) Move on to the next column early as nothing else will be solid.

See (Bretherton, 2020, pp. Chunk.cpp, Chunk::FillChunkData) for implementation.

CHUNK LOADING OPTIMISATIONS

Initially the chunk loading was very performance-impacting. This is perhaps not very surprising as the laptop this application was developed on struggles to run Minecraft (which has a similar, and likely very well optimised, voxel terrain generation algorithm). However even on very low render distances the application would freeze for up to a second whenever walking over a chunk border, which was unacceptable. The solution to this was in two parts.

CHUNK CACHING

After a chunk object is created for the first time, it fills its data by generating that chunk. However, when a player leaves the render-area of the chunk, it would be wasteful to delete this data. Instead, the chunk is simply disabled, and when a player re-enters the render-area of the chunk the already generated data is added back on to the render queue.

CHUNK MULTITHREADING

The main bottleneck for the application was the generation of the voxel data in a chunk with the repeated testing against the FBM. To solve this, it was decided that the task of generating a chunk should be dispatched to separate threads. As the application was designed to have a render distance of four chunks, it was decided that the best solution was to simply construct and detach a thread per chunk generation. This allowed the program to be multithreaded without completely altering the architecture to create a producer / consumer pattern, but did have its drawbacks, discussed further in Critical Appraisal. However, the performance improvement was remarkable, with a four chunk render distance running perfectly on this machine.

TEXTURE BLENDING

The terrain generated had an interesting shape, but with every voxel having the same texture, the result was ultimately unconvincing. For this reason, a height based texture blending shader was implemented.

Due to the restrictions of instanced rendering, a voxel's position could not be passed to the pixel shader directly. Instead, the world position of a vertex is calculated in the vertex shader and passed down the pipeline to the pixel shader.

```
output.worldPosition = mul(input.position, worldMatrix).xyz;
```

In the pixel shader, the world position y value is used to test against the ground level, and three textures are lerped between. Grass for above ground level, rock for the lowest voxels, and sand for those that were in-between.

SOUNDTRACK

The application also features a completely procedural soundtrack with a melody and chord progression.

It was decided the application will write the melody in a major key, as it will be familiar and sound good to most users. A minor key was considered but the concept was discarded as the rules of functional harmony in minor are more complex and harder to resolve in a satisfying way, partially due to the Natural Minor scale lacking the leading tone (which gives Major some of its strongest resolutions).

SOUND GENERATION

The sound generation consists of creating samples to fill a buffer. This buffer is then passed to SFML's audio thread and the sound is played.

The algorithm gives an index within the buffer to begin the note, and an attack and decay time, which are set to imitate the sound of a piano, with a short attack and long decay.

During a note's generation, an index is swept over. This index is the sample index in the buffer and is fed into a sine function to produce a sine wave. This sine wave also imitates the sound of a piano, as when a string vibrates (as one does in a piano) while fixed at both ends, it vibrates in a sinusoidal manner.

The output of this sine wave is then multiplied by the "amplitude envelope", which is generated through an inverse lerp of the current index and the value of the attack and decay time:

```
if (i < attack) {
    ampEnvelope = Lerp(0, 1, InvLerp(0, attack, i));
}
else if (i < (attack + decay)) {
    ampEnvelope = Lerp(1, 0, InvLerp(attack, (attack + decay), i));
}

audio *= ampEnvelope * envelopeFactor;
```

NOTE GENERATION

Note pitches are generated using the western twelve tone equal tempered tuning system. This means each note's pitch is calculated by a function on the base pitch: concert A440 (ISO, 1975).

This frequency is then calculated by the function:

$$pitch = A440 * \left(\sqrt[12]{2}\right)^{n}$$

Where n is the number of semitones above (or below) the base pitch. The reason that this horrific function is necessary goes beyond the scope of this document, but it is impossible to create a perfectly tuned 12 note system where each musical interval is exactly represented (Encyclopædia Britannica, 2019). This tuning system provides a very good compromise, where every interval (except the Unison and Octave) is slightly but evenly out of tune.

CHORDS

The application generates chords to go along with the melody. These chords are all based off Major, Minor or Diminished triads in the key of A. That is a Root note with a Major or Minor Third and a Perfect or Diminished Fifth above this Root note. These triads are the most common types of chord, and there are seven triads in each Major key. In A Major, these are A Major, B Minor, C Sharp Minor, D Major, E Major, F Sharp Minor and G Sharp Diminished (Cazaubon, 2017). In the case of the diminished chord, the Diminished Fifth is a Tritone above the Root, and the Root of the chord is a half-step below the Tonic note of the key, giving this chord a large amount of tension and pull to resolve to the Tonic chord of the key. This gives the key the ability to create a large amount of tension in its harmonic motion, which keeps a listener interested and engaged in the piece and was a further reason why A Major was chosen as the key.

CHORD GENERATION

A NOTE ON MUSIC THEORY

This application takes as its basis for constructing chord progressions the Theory of Functional Harmony. This theory is based on the framework that particular chords in a key hold a particular function within that key. That is to say, each chord can be assigned a classification of a function type depending on an “action” the chord does to the music.

Each key has a key centre, where the music feels “at home” or “at rest”. In functional harmony, there are three broad types of function a chord can do, all in terms of this key centre.

Tonic function provides this feeling of rest. It creates or prolongs a release of tension that may have been built up earlier in the progression.

Sub-dominant (sometimes called Pre-dominant) function provides a pull away from the Tonic or key centre. It creates tension and provides a feeling of motion within the music.

Dominant function creates a strong pull back towards the Tonic or key centre. This provides the most tension in a progression and has a strong call for resolution, by moving back to the Tonic.

(Open Music Theory, n.d.)

MARKOV CHAIN

A Markov chain is a finite state machine that moves between its states probabilistically. Each given state has a node associated with it, and for any state that this state can move to (including itself) a probability is associated with this movement.

To run a Markov chain, an initial state is needed, after which a function to progress the state is periodically called. This moves the state randomly along each node’s connecting paths, based on the probability weighting of each movement.

CHORD PROGRESSIONS

Functional harmony provides a rules system for moving between harmonic functions in a way that will sound consonant and create satisfying harmonic motion. This example (Figure 4) from Hutchinson (2020) was used as the basis to build the rules that the Markov chain controlling the chord progression follows.

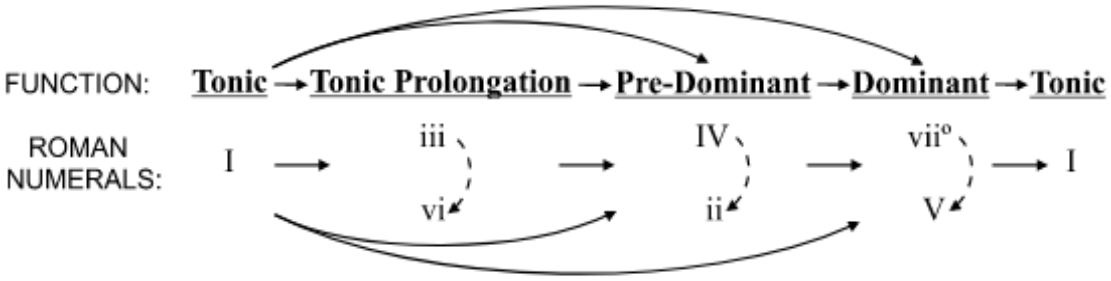

Figure 4

(dashed lines represent *possible* movement within a functional area)

Each chord was associated with a node in the Markov chain and, following this diagram, the connections between nodes were set up. Each exact probability was fine-tuned through trial and error, but a fairly even distribution was maintained to ensure the soundtrack remained varied and interesting.

MELODY GENERATION

The melody was generated using a slightly simpler system of rules. In music, a melody will progress in one of two manners: stepwise, where the next note is one or two notes above or below the previous, and leaping, where the next note is further away in the scale.

Whenever a note in the melody progresses, the following algorithm is run:

1) Randomly decide if the next note will be a rest
   a) If It will be a rest, do not progress the note
   b) Else, it is not a rest. Decide randomly if the next note will progress stepwise or leaping
      i) If it is stepwise, decide which direction to step in, and to step by one or two notes
         (1) Add this next note to the buffer
      ii) Else it is leaping. Pick a random note from the scale
         (1) If this note is in the pentatonic scale associated with the key, add this note to the buffer
         (2) Else, if the note is not in pentatonic scale, pick another note and add this new note to the buffer (this new note might also not be in the scale, but this is fine)

See (Bretherton, 2020, pp. Chord.Cpp, Chord::GetRandomNote) for implementation.

This provides a varied and interesting melody that will favour stable consonant notes while still generating melodies that use the full potential of the key. The combination of both stepwise and leaping motion provide interest while ensuring the melody remains easy to follow yet allows it room to move freely though the scale.

CHORD IMPROVEMENTS

These above methods provided pleasant chord progressions, but the result was a little repetitive. For this reason, chords are sometimes modified before being added to the audio buffer. These modifications are primarily decorative and do not change the chords function from a functional harmony perspective. The decorations are as follows:

- Transposition:
  The chord may move up or down an octave and stay there for a few chords, providing additional interest and movement to the piece.
- Inversion:
  The chord may be in root position, first, or second inversion. A chord inversion is where the notes are re-arranged within a chord. The root (default) inversion is where the earliest note in the scale is the lowest note, first inversion moves the third of the chord down an octave to be the lowest note. Second inversion moves the fifth down an octave to be the lowest note. This changes the colour and timbre of the chord without affecting its function.
- Extension:
  The chord may gain an additional scale degree, the seventh. This turns the chord into a seventh chord, a chord with the same function but a more complex colour and timbre.

Any combination of these modifications is possible, providing far more interest and variation to the soundtrack.

AUDIO MULTITHREADING

As the soundtrack needed to run in the background without being affected by any lag in the game application, the sound system needed to be on a separate thread to the rest of execution. On top of this, the SFML `sf::SoundStream` (Gomila, n.d. (b)) class also needed to run on its own thread, so memory access and thread synchronisation needed to be implemented to prevent threading issues.

ARCHITECTURE

A discussion of the code used to implement these features.

OVERVIEW

The application was developed attempting to use only the most modern and best C++17 practices. Any time heap allocation was needed, a smart pointer was used to prevent any possible memory leaks. RAII locks were used wherever possible when mutex access to data was required. Range based and iterator for loops were preferred throughout the solution. As much as possible, const references to objects and data were used, and functions that did not affect the logical state of an object were marked as const (Standard C++ Foundation, n.d.).

I believe these practices lead to the production of better code with fewer bugs and produced code that was more efficient and easier to understand.

TERRAIN

The terrain generation is accomplished through four main classes

CHUNK MANAGER

The chunk manager is responsible for loading/unloading and activating/de-activating chunks as the player moves around the world.

The chunk manager is also responsible for gathering the positions of all of the voxels of all of the active chunks and adding them to the render queue.

It is further responsible for creating and destroying chunk objects. Created chunks use the chunk manager's member hash functor to hash their position into a unique ID.

The chunk manager then holds a map of these chunks, accessed by their ID. This makes loading a chunk at any position scale O(1) with the number of chunks in memory.

The chunk manager also stores the Terrain Generator, which is passed to each chunk to give them access to the FBM functions.

CHUNK

A chunk is responsible for generating and storing the positions of its voxels.

The chunk's world position is passed to it along with a reference to the chunk manager's Terrain Generator object. It then uses this world position to learn the height that the terrain should be at. It then generates a surface of voxels based on this data.

It holds a smart pointer to its data, a std::vector of XMFLOAT3s, so that the chunk is sperate from its data and is therefore far more movable.

The generation runs on a separate detached thread, and the chunk holds a mutex to access this data, so the generation is not interrupted.

TERRAIN GENERATOR

This class holds data used in the generation of Fractional Brownian Motion and provides an interface to the output of this repeated Perlin noise function. It holds the starting amplitude, frequency, and ground level (the Y-level which is treaded as 0 in the output of the Perlin Noise function).

The main function of this class is to test if a cube at a given position is solid or not. A function takes a 3D coordinate and returns if the y position of that coordinate is above or below the height at that x and z location.

PERLIN NOISE CLASS

This class consists of a single static function to perform a Perlin noise calculation at a given 2D coordinate.

It also contains some private helper functions, alongside a definition for a vector and a grid.

SOUNDTRACK

There are three main classes involved in audio generation and two classes for the Markov chain. The audio generation uses a producer/consumer pattern, where the audio generator produces samples that are consumed by the procedural sound stream class.

MARKOV CHAIN

The Markov chain class is the interface to the chain. It contains and manages the Markov nodes.

It is a templated class and so can hold nodes representing any class. It has functions to get the data of the current node, as well as progress the chain.

It also provides the interface for adding nodes and setting up connections between nodes.

MARKOV NODE

The Markov node is a templated class that holds a piece of data of this type (the node's implicit state) and a map of other nodes.

When adding a connecting state to the node, a corresponding probability must also be given. This probability is then accumulated, and each subtotal value is set as the key in this map. When progressing the chain, a random number is chosen between zero and this accumulated probability. Each map element is then iterated through and the first key to be greater than this number is chosen. This allows for a weighted random connected node to be chosen very efficiently, with this progressing function scaling O(n) with the total number of states.

AUDIO GENERATOR

The audio generator is responsible for filling the sample buffers with the music. This object runs entirely from a child thread of the procedural audio stream and its operation is controlled by condition variables altered by the procedural audio stream.

The class has two buffers, the hot buffer and the back buffer. When signalled to, the generator fills the back buffer, swaps the back with the hot buffer, and then fills the new back buffer. At this point it checks to see if the hot buffer has been read and , if not, it suspends execution here until signalled. When signalled that the hot buffer has been read, the stream swaps the buffers again and fills the new back buffer.

PROCEDURAL AUDIO STREAM

This object derives from SFML's sound stream object, and so overrides the object's pure virtual function: `virtual bool onGetData(sf::SoundStream::Chunk& )` (Gomila, n.d. (b)).

The class uses its parent's `void play()` function which detaches an audio thread. This thread periodically calls the `onGetData` function to copy samples into the audio thread's audio buffer.

This implementation takes the audio buffer generated by the audio stream and loads it into the audio thread's buffer. If there are no more samples then it requests a new buffer from the audio generator. As there are two buffers, it is effectively guaranteed that the hot buffer will be ready immediately. The audio generator temporarily moves the buffer into the audio stream under a mutex lock, the samples are copied, and the buffer is moved back.

CHORD CLASS

The chord class stores data about the notes, chords, and their pitches. It is responsible for generating a random note and has a number of helper functions for getting the scale degree of a note, or the function of a chord, in a key.

CRITICAL APPRAISAL

CODE QUALITY

Overall, I am very happy with the quality of the code I produced.

It conforms well to modern C++ practices and is well organised. Some of the names I chose could be improved, such as perhaps re-naming `ChunkManager` to `World`. At times, some functions become too long and overly complex, particularly in the note and sound generation algorithms.

I am glad I stuck to the modern safer smart pointers, but at times their use may have been unnecessary, especially considering how verbose they can make code, requiring scopes, names, type parameters and functions where a single * or & would otherwise have been.

DESIGN

The solution, I think, is generally well designed. There is good separation of tasks and for the most part the code is very maintainable. The classes allowed each part of the solution to be developed and tested separately before being combined into the larger system.

Some of the classes did get away from me somewhat. The chords class is hopelessly over-engineered and far too future-proofed – allowing a system that can change keys is way beyond the scope of what it needed to be and was ultimately a waste of time.

If I had designed the chunk generation system with multi-threading in mind from the start, I think the outcome would have been far better. Simply detaching a thread for each chunk is a verry poor solution and making a proper producer / consumer pattern would have been far better.

EFFICIENCY AND PERFORMANCE

The efficiency of the code is reasonably good in terms of time and space. As previously said, having a separate thread per chunk is a very inefficient solution, particularly as the render distance gets higher. At high render distances the number of threads the CPU has to switch between leads to the very important threads that are running the game, audio, and taking input beginning to spend significant amounts of time queuing for access to the CPU. This eventually causes the entire machine to slow down, frames to drop, and the music to stutter.

As for the actual generation algorithm, I think it is reasonably efficient. Needing to make multiple memory hops through various objects to discover if a block is solid could definitely be improved by having each chunk have its own access to Perlin functions.

Improving generation, however, could be done in two ways. These improvements would likely significantly improve performance. Firstly, calculating the hight for many positions at once and caching the result would save the overhead of each call, as repeated calls within a column (which happens often) would be cached. The second would be checking if a block is solid at the ground level first (rather than the bottom of each column) and checking above/below if the block were/weren't solid until it reaches ground. This would vastly reduce the number of Perlin calls.

The Markov chain and chord progression algorithms are very tidy, and the methods that fill the audio buffer follow the very efficient producer / consumer pattern where each thread suspends operation efficiently when they are not doing work. I think these are about as efficient as could reasonably be.

HOLISTIC EVALUATION

Overall, I am very, very happy with the application. The game looks far better than I initially thought it would and the soundtrack is, in my honest opinion, simply lovely. I think in terms of terrain generation I achieved what I set out to make, and in terms of the soundtrack I surpassed what I thought would have been possible for me in the time allocated. I am somewhat disappointed I could not have spent more time fixing the inefficiencies with the generation algorithm, which might have allowed my machine to run the game smoothly at higher render distances. I am, however, glad I made the improvements I did make that allow it to run smoothly at all. In an ideal world it would have been nice to add some kind or trees or similar to my terrain, but as a terrain alone I think it does a very convincing job.

REFLECTION

The main thing I have learned from this procedural generation module is that generation of any kind is *expensive*. From the very beginning, and for more than in other programming areas, and particularly if your generation is going to be real-time, you need to be thinking about efficiency and spreading computational work. Both sections of my application make heavy use of threads, but only one did so with multithreading in mind from the beginning, and it very much shows.

The other thing I realised was that in some cases, a small amount of procedural methods use can go a very long way. The last minute alterations to my chord generation algorithm where I added the transposition, inversion and extensions made my music sound deeper by orders of magnitude. It needs a solid basis already, but relatively simple procedures like this can be very easy to implement and can add multiple layers to whatever you are producing.

Generally, I became far more comfortable and confident with modern C++ features (such as smart pointers) and learned of their common pitfalls. I particularly became far more comfortable linking external libraries up inside visual studio.

Generally, I think I definitely will be making heavy use of the knowledge I have gained in this module to create procedurally generating systems. More than any specific technique, I have learned that procedural generation is not in fact as daunting as it might first seem. That it is often a very good use of time to automate the generation of something that might otherwise be *incredibly* tedious to create any other way. I have learned that writing procedural methods is an intensely creative experience and can be just as, if not more, rewarding than the process of creating the same things by hand.

REFERENCES

Bevins, J., 2003. *Libnoise.* [Online]
Available at: http://libnoise.sourceforge.net/coherentnoise/index.html
[Accessed 2020].

Bretherton, J., 2020. *CMP305 Project Source Code: Generated voxel world with procedural soundtrack.* Dundee: Abertay University.

Cazaubon, M., 2017. *piano-keyboard-guide.* [Online]
Available at: http://www.piano-keyboard-guide.com/key-of-a.html
[Accessed 2020].

Encyclopædia Britannica, 2019. *Equal temperament.* [Online]
Available at: https://www.britannica.com/art/equal-temperament
[Accessed 2020].

Gomila, L., n.d. (a). *SFML.* [Online]
Available at: https://www.sfml-dev.org/
[Accessed 2020].

Gomila, L., n.d. (b). *Documentation of SFML 2.5.1, sf::SoundStream Class Reference.* [Online]
Available at: https://www.sfml-dev.org/documentation/2.5.1/classsf_1_1SoundStream.php
[Accessed 2020].

Hutchinson, R., 2020. *Music Theory for the 21st-Century Classroom.* [Online]
Available at: http://musictheory.pugetsound.edu/mt21c/HarmonicFunction.html
[Accessed 2020].

ISO, 1975. *ISO 16:1975, Acoustics — Standard tuning frequency (Standard musical pitch).* s.l.:ISO.

Open Music Theory, n.d. *Harmonic functions.* [Online]
Available at: http://openmusictheory.com/harmonicFunctions.html
[Accessed 2020].

Perlin, K., 1997. *Noise and Turbulence.* [Online]
Available at: https://mrl.cs.nyu.edu/~perlin/doc/oscar.html#noise
[Accessed 2020].

Perlin, K., 1999. *Making Noise.* [Online]
Available at: https://web.archive.org/web/20071011035810/http://noisemachine.com/talk1/
[Accessed 2020].

Robertson, P., n.d. *Abertay directX-11 framework for education in various modules.* Dundee: Dr Paul Robertson.

Standard C++ Foundation, n.d. *Const Correctness.* [Online]
Available at: https://isocpp.org/wiki/faq/const-correctness#logical-vs-physical-const
[Accessed 2020].