Generated Voxel World with Procedural Soundtrack

J Bretherton 1800231

# Overview

This application consists of an endlessly[[1]](#footnote-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 number of chunks away from the player in a square with side length of double this render distance (as the distance is the number of chunks past where the player is standing in each cardinal direction). 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 it was developed on can handle a render distance of three (36 chunks loaded) comfortably but begins to struggle when loading in chunks at higher render distances. The application is not designed to run at these higher settings, they are included only for demonstration.

The application may also take some time to load initially.

### 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”) 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, 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, 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. Randomly, either stepwise or “leaping” motion is chosen, to lead from the previous note. Stepwise will move the note one to two notes up or down in the scale, 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).

# 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:



Where a random number is generated using the X and Y position of the vertex of a square in the grid, and a vector is constructed using this value to lie on a point on a unit circle.

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

The grid square a point resides in is calculated 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.

  
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.

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 in Figure 1.

Dot = 0.86

Dot = -0.06

Dot = 0.14

Dot = -0.90

Figure 1

These four values are then interpolated together by an interpolation function based on the non-linear function: 0 = 3t2 – 2t3

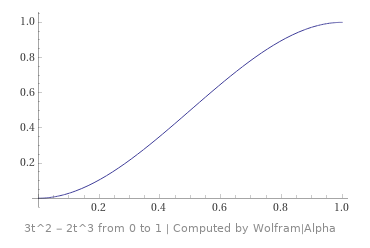


Figure 2

It is notable that this function has a zero derivative at 0 and 1.

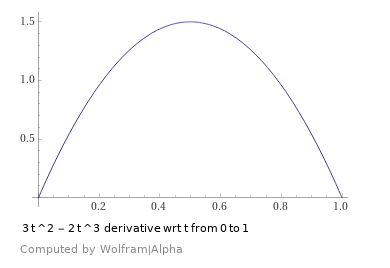


Figure 3

Meaning the rate of change near a grid vertex approaches 0, making input values close to a grid vertex have values very close to 0.

In code this function is expressed:



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, 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 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 and these have yet smaller bumps on them.

This application uses constant values of 18 for the amplitude, and for the frequency, decreasing the amplitude and increasing the frequency by a factor of four for 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:
   1. For each voxel in the column:
      1. If the voxel is solid:
         1. If the voxel is visible (not entirely surrounded by solid voxels):
            1. Add a cube in this position to the render queue.
         2. Else if the voxel is not visible:
            1. Do nothing.
      2. 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, height based texture blending shader was implemented.

Due to the restrictions of instanced rendering, a voxels position could not be passed to be pixel shader directly. So instead, the world position of a vertex is calculated in the vertex shader and passed down the pipeline to the pixel shader.

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, sand for in-between, and rock for the lowest voxels.

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 good sounding to most users. A Minor key was considered but the concept was discarded as the functional harmony in minor is more complex and harder to resolve in a satisfying way due to the Natural Minor scale lacking the leading tone (which gives Major it’s 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 delay 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 sin 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 package”, which is generated through an inverse lerp of the current index and the value of the attack and decay time:



### 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:

Where n is the number of semitones above (or below) the base pitch. The reason 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 Major triads in a 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 (the A Major chord). This gives the key the ability to create a large amount of harmonic motion, which keeps a listener interested and engaged with the piece.

## 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 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. For a given state, a node is associated with that state, and for any state 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 continuously called which 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.

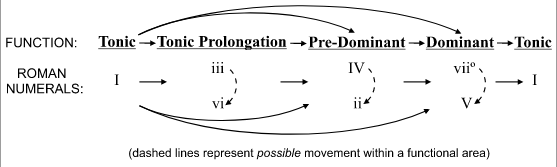


Figure 4

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 is varied and interesting.

## Melody Generation

The melody was generated using a slightly simpler method 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
   1. If It will be a rest, do not progress the note
   2. Else, it is not a rest. Decide randomly if the next note will progress stepwise or leaping
      1. 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
      2. 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 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, but will also generate melodies to the full potential of the key. The combination of both stepwise and leaping motion provide interest while keeping the melody followable and allow it room to move freely though the scale.

## Chord Improvements

These above methods provided good sounding chord progressions, but it 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 couple of 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, and not be 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, 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

# Critical Appraisal

Umm all those threads

Mimecft runs like ass anywas

# Reflection

# 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., a. *SFML.* [Online]   
Available at: https://www.sfml-dev.org/  
[Accessed 2020].

Gomila, L., 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.

* 12 tone even temperament
* Functional harmony
* Why major not minor (3 types of minor)

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 × 109 chunks, effectively endless to a casual player. [↑](#footnote-ref-1)