TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Computer Science

ITV40LT
Fred-Eric Kirsi 135216 IAPB

# VOXEL GAME ENGINE USING BLENDER GAME ENGINE

Bachelor's thesis

Supervisor: Jaagup Irve
Master of Sciences
Software Engineer

Tallinn 2016

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutiteaduse instituut

ITV40LT
Fred-Eric Kirsi 135216 IAPB

# VOXEL-MÄNGUMOOTOR BLENDER GAME ENGINE ABIL

Bakalaureusetöö

Juhendaja: Jaagup Irve
Magister
Tarkvarainsener

Tallinn 2016

# Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Fred-Eric Kirsi

23.05.2016

# Abstract

The objective of this thesis is to study various aspects of developing a voxel style game and in the process create a prototype engine to evaluate the possibilities and performance of the algorithms. The algorithms evaluated are Greedy Meshing, A* search, Goal Oriented Action Planning (GOAP), Finite state machine (FSM), and Perlin noise. The used 3d engine is the Bender game engine. The scripting is done in the Python coding language.

The result of this work is a working voxel game prototype, where the agents are able to navigate, manipulate, and make decisions in real time. The aim of the prototype is to quantize the performance and feature experience on for the user.

The work is divided into four main stages. Firstly evaluation of the work environment and its limitations. Secondly the development and optimization of the voxel implementation. Thirdly the creation and comparison of different pathing implementations. Lastly the evaluation different possible decision systems.

The thesis describes the implementation of the voxel generating system using greedy meshing and Perlin noise, how voxel aware pathfinding is executed and compares different AI decision making systems.

The result of this project is a functional prototype, its codebase, and algorithm design descriptions.

This thesis is written in English and is 59 pages long, including 7 chapters and 46 figures.

# Abstract

## Voxel-mängumootor Blender Game Engine abil

Selle töö eesmärk  on uurida 3d voxel mängude koostamisel kasutatavaid algoritme ja selle alusel töötada välja prototüüp. Seal hulgas uurida nii voxel keskkonna loomisest tulenevaid probleeme kui ka agente juhtivaid süsteeme. Algoritmid mida uuritakse on Greedy Meshing, A* tee otsing, Goal Oriented Action Planning (GOAP), Finite state machine (FSM), and Perlin noise. Kasutatud 3d mootor on Blender game engine. Koodimine on tehtud Python programmeerimise keeles.

Töö käigus loodud agendid peavad töö tulemusena olema võimelised reaalajas orienteeruma ja otsuseid vastu võtma. Loodud prototüübil peab olema võimalik tulenevalt iga algoritmi keerukusest viia läbi tööjõudluse ja võimekuse analüüs.

Töö jaguned nelja põhilisse peatükki. Esmalt tutvutakse töökeskkonna nõuetega ja parameetritega. Teiseks teemaks on prototüübi voxel lahenduse kirjutamine ja optimeerimine. Järgmiselt viidakse läbi erinevate teeotsingu algoritmide analüüs. Viimasena viiakse läbi agentide otsuste vastuvõtvate süsteemide katsed.

Selles töös kirjeldatakse voxel-lite loomist ja maastiku muutmist arvestavat teeotsingut ning võrreldakse erinevaid otsuste tegemis süsteeme. Töö tulemuseks on töötav prototüüp, selle koodibaas ja kasutatud algoritmide kirjeldused ning hinnangud.

Lõputöö on kirjutatud Inglise keeles ning sisaldab teksti 59 eheküljel, 7 peatükki ja 46 joonist.

# List of abbreviations and terms

| | |
|---|---|
| 2D | Two dimensional |
| 2d filter | Post processing image filter |
| A* | "A star" search algorithm |
| ANN | Artificial neural network |
| Brownian motion | Random motion of particles |
| chunk | Voxel organization unit |
| Dijkstra's algorithm | search algorithm |
| FPS | Frames per second |
| FSM | Finite state machine |
| GOAP | Goal oriented action planning |
| graph | data structure In computer science |
| Greedy algorithm | A greedy algorithm is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage |
| K-map | Karnaugh map |
| nodes | A node is a basic unit used in computer science graphs. |
| Perlin noise | Perlin noise is a type of gradient noise developed by Ken Perlin. |
| polygon | Polygon is a collection of vertices and edges that form a face. |
| quads | A polygon with 4 vertices. |
| rational agent | An agent that chooses to perform the action with the optimal expected outcome for itself from among all feasible actions. |
| Ray-tracing | Computer graphics technique. |
| RLE | Run length encoding |
| scene | Game engine environment where objects, cameras, lights and scene attributes are located. |
| shader nodes | Tool for creating custom shaders. |
| voxel | A voxel represents a value on a regular grid in three-dimensional space. |
| z-fighting | Z-fighting, also called stitching, is a phenomenon in 3D rendering that occurs when two or more primitives have similar values in the z-buffer. |

# Table of Contents

# List of Figures

# 1 Introduction

Voxel engines are everywhere. With the advent of Infiniminer, Minecraft and plethora of other voxel games a new game concept started to emerge. The world where the players now find themselves is not one of illusion of agency but one that actually responds and changes. It is a novel experience but as such still in its infancy. There much work that goes into making a voxel game and it is a craft worth further pursuit.

The objective of this thesis is to explore the technology that goes into making a voxel game. The main focus is on three main topics: voxel generation, pathfinding, and agent AI. Voxel generation is the main feature of voxel games and also the main concern when it comes to optimization. There are many approaches when it comes to generating voxels, with each offering variety of benefits and concerns. In most games pathfinding is dealt by the game engine which the game is built on itself. It is generated before the game and mostly static in nature. But voxel games require much more tailored and dynamic pathfinding which is why pathfinding is an aspect that needs to be revisited and evaluated. Finally, there is not much of a game without its inhabiting agents having their own mind. The illusion worlds of agency is something without which players can not invest themselves into the game. The agents AI must be able to make decision and respond with competence that is expected given the context of the world.

Addressing these core subjects will enable us to create better and more compelling experiences for the user.

The prototype on which the algorithms are studied on is developed on the Blender 3d engine using Python as the scripting language. Blender 3d has a well documented small scale game engine on which rapid prototyping is possible. The prototype is developed on an Intel® Core™ i7-4790K Processor at 3.8GHz (underclocked).[1] The graphics card used is an ATI Radeon™ HD 5770 Graphics card. [2]

## 1.1 Problem statement

The problem is to explore ways to give the user more challenging game environment while not jeopardizing the system performance and consequently the user's experience. For this purpose there is a need to explore ways how to generate procedural environments and how to bring those environments truly alive by complementing them with capable AI.

The voxel rendering part of the engine must enable the user to make changes in the scene without dropping the frame rate below 30 which equals to around 33ms frame time. The generated voxel landscape must be large enough to enable bird's-eye view while not impacting performance significantly.

The actors must be able to traverse and perform actions on the whole voxel terrain, while not exceeding the 16ms of logic time.

# 2 Performance goals

Any user experience can negatively impacted if the application is not responsive. There is no point in having a huge feature set if the user is unable or unwilling to use it. This is why there is a need to understand the different performance aspects of the engine used for the prototype and set reasonable performance goals for the mechanics.

## 2.1 What are the main performance factors?

The main performance factors of the Blender game engine are shown in Figure 1. Aside from the FPS, the profile shows physics, logic, animations, network, scenegraph, rasterizer, services, overhead and outside. The most notable are the physics, logic, and rasterizer statistics which contribute the most time spent for the frame time.

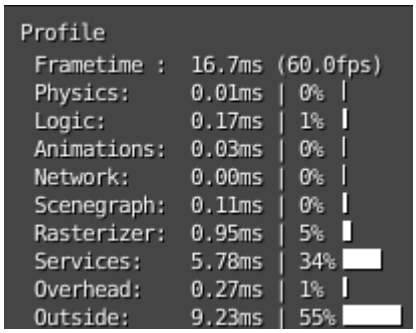

Figure 1: Blender game engine profiler

Physics represents the time spent on physics code. Blender uses the Bullet Physics Library for its physics calculations. Biggest causes for physics load are complex mesh based collisions, physics mesh updates, and large numbers of game object instances using physics. In this project's scope there is no need for Bullet physics as it is handled by the agent logic itself.

The logic encompasses everything user scripted, either in Python or Blender logic bricks. This also includes the procedural generation and updates to the voxel terrain and actor logic. The rasterizer is responsible for actually rendering the game. This includes

rendering geometry, shaders, and 2D filters. Aside from them, the other profile statistics that are either legacy or not important features in this project's scope. The most critical aspects of this project are the logic time and polygon count.

## 2.2 Engine graphics and polygon count

First variable to check would be the maximum possible polygon count. By subdividing a plane the game engine can show well beyond one million polygons while never exceeding 16ms of frame time. According to initial tests on the voxel rendering engine, the Blender game engine can handle displaying a maximum of 50 000 faces with shader nodes and post processing filters. Without shader nodes 100 000 polygons. See Figure 2. This is ample for a prototype voxel game.

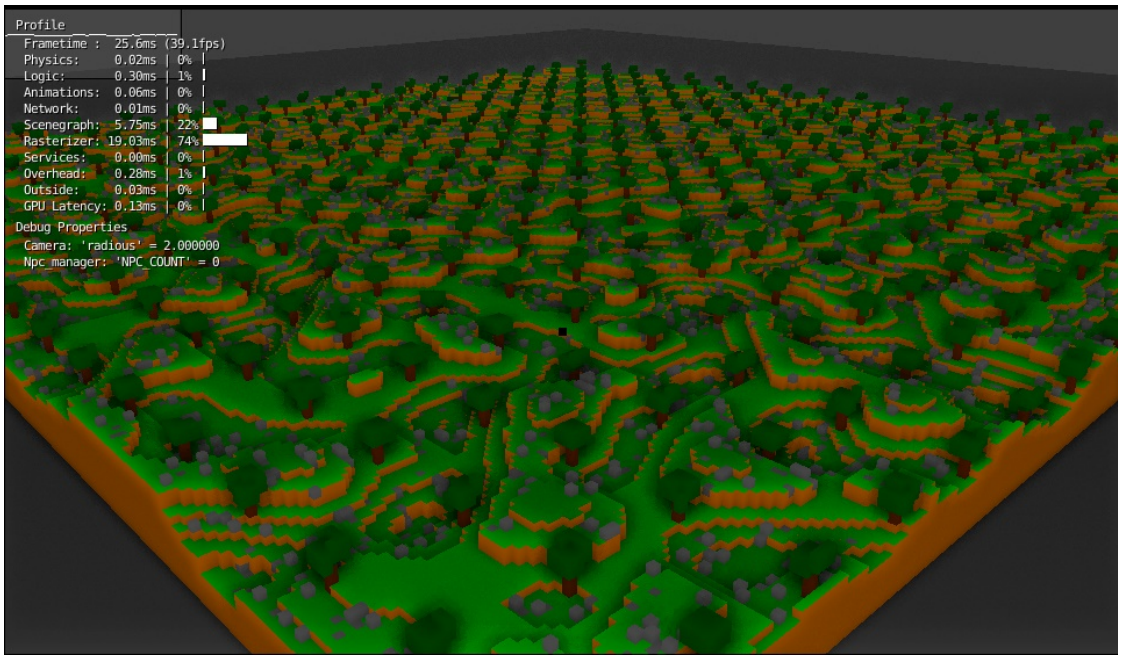


Figure 2: Example: effects of polygon count in Blender game engine

Besides FPS, it is notable that the Blender instance uses around 2-4GB memory which is considerably high and may by itself prove to be a bottleneck, but currently the engine ignores the memory aspect of performance.

## 2.3 Engine logic and scheduler

The voxel creation and actor logic should not impact the frame rate. The game logic time should be under 16ms to enable smooth gameplay at 60 FPS, but considering that this time is shared with the rendering time in the frame time. Given a ratio of one to one, the expected frame rate would be around 30 FPS which is still considered acceptable.

Considering that all calculations can not be made in 16ms, but are not real time dependent, there is a possibility to spread the load over several frames. This problem can be reduced by a scheduler. But considering there is a limit of how long you can postpone a reaction to an action there must be a maximum limit set.

The scheduler must ensure a response for user interaction in under a second to minimize user disarray and complete in 2 seconds in order not to disinterest the user.

The scheduler has more leeway with actor interaction as thinking pauses and semi optimal choices are less predictable and at best case at best case enforce the idea of the actors being conscious and bound to the rules of the world the same as the user would be.

The scheduler should not create situations where certain task are executed at a faster pace than other while the logic and mechanics require them to be synchronized.

# 3 Writing the voxel system and its performance factors

By analogy with pixel a voxel is a volume element. It is the smallest distinguishable space occupying element of a three-dimensional space. Voxels as data structures normally do not possess their own position, but are relative to their parent chunk, which is a larger space dividing element. This enables more efficient representation of voxels in memory and logic handling.

## 3.1 General ways of memory representation

In memory there are many ways to represent data and voxels are no exception. The different implementations follow the common trend that data access time can be exchanged with data compression. When building a voxel game, it is important to choose a data structure for representing the world early on. This decision more than any other has the greatest impact on the overall performance, flexibility, and scale of the game.

The simplest choice is a flat array, where all the voxels are stored sequentially in an array. This approach has the benefit of being the simplest solution and offers great access time. An added benefit is that simple neighbourhood checks can be done with index calculations instead of iteration. The flat array model's greatest drawback is its disinterest in compressing data. The amount of memory consumed grows with the cube of the linear dimension. It is a viable solution for small scale games, but for larger games the memory impact might overwhelm the hardware capacity.

A popular choice for voxel representation is an octree. An octree is a tree that recursively subdivides space into equal sized octants as shown in Figure 3. As of most of the voxel worlds are occupied by constant regions it is expected to reduce the size significantly. Unfortunately, the tree traversal creates a lot of overhead for random access and neighbourhood checks.

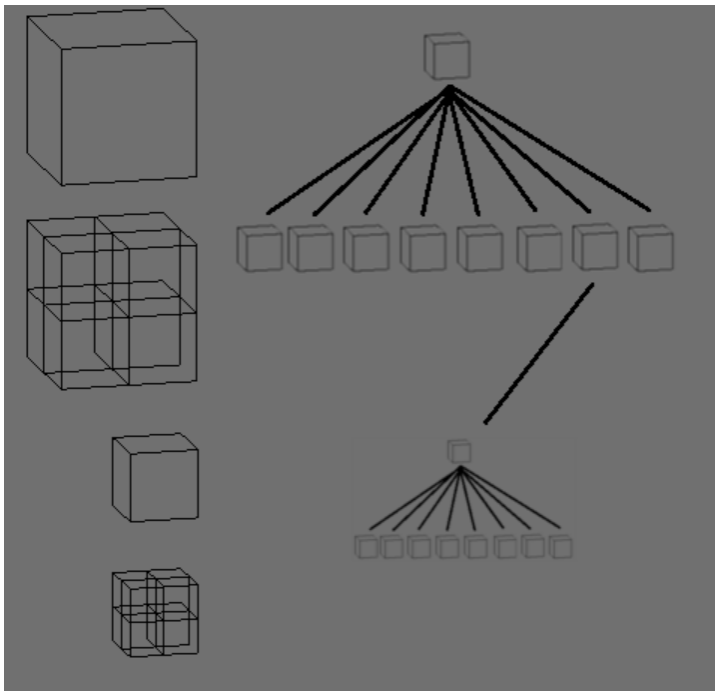

Figure 3: Example: space subdivison of an octree

Finally what [3] suggested is to use run-length encoding for storing voxels. RLE is a very simple of lossless data compression which benefits greatly from large homogenous sequences of data which is expected from a game environment anyway. Sequences in which the same data value occurs in many consecutive data elements are stored as a single data value and count, rather than as the original run. This increases the iteration time as the iteration over the voxel set in units of runs instead of units of values. For example in RLE “AAAAABBZZZZYYYTTTTTTTTT” would be encoded into “5A2B4Z3Y9T”.

Generally it is advised to add a chunk paging as another layer of data management. This way the management of contained spaces is improved. Using a hashmap to store the chunks allows one to maintain random access times, while simultaneously taking advantage of sparsity as in an octree.

As of time constraints the data structure used in the prototype is flat arrays divided into chunks. While not evident yet, the voxels generation method and pathing, which is discussed later, depended mainly on random access times, which was for the benefit of this choice.

## 3.2 General ways of visual representation

A common method to draw voxels is to use ray-tracing. This involves raster graphics where you simply raytrace every pixel of the display into the scene. See Figure 4. An added benefit for this type of rendering is the added feature of ray-traced illumination with shadows. Also ray-casting is known to scale much better with large numbers of voxels as the processing depends on the number of pixels which is constant. Downside is that ray-casting is feature that game engines normally don't support and such voxel systems are written from ground up by their users.

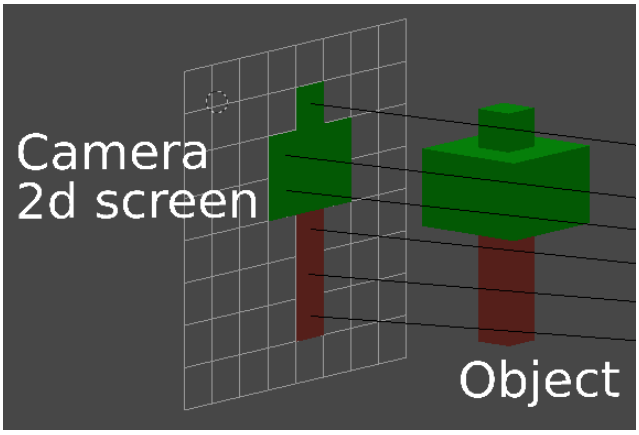


Figure 4: Raycasting example

What game engines are good at is drawing triangles, a lot of them. One of the main innovations in voxel games is that they use polygons instead of ray-casting to render their volumes. A volume described as voxels can be visualized by the extraction of polygon iso-surfaces that follow the contours of given voxels. This is called meshing.

## 3.3 Meshing

The main benefit to meshing is because it can be implemented in most game engines one way or the other and can be used side by side with other mainstream engine features out of the box. This includes texturing, shading, physics, and animations.

The main challenge in using polygons is figuring out how to convert the voxels into minimum amount of polygons efficiently. In a typical voxel game the voxels do not get modified that often compared to how frequently they are drawn. Which is why the main

heavy lifting happens at rendering the voxels. As a result, it is quite sensible to optimize the mesh upfront.

In the following I will explain some of the meshing possibilities.

### 3.3.1 Naive method

This is the worst case scenario. For each voxel a cube with 6 faces is generated. This means that a lot of polygons are wasted to create faces that are hidden between solid voxels as seen on Figure 5. One benefit for this method is that it does no neighbourhood checks and thus updating a voxel is contained only to that voxel.

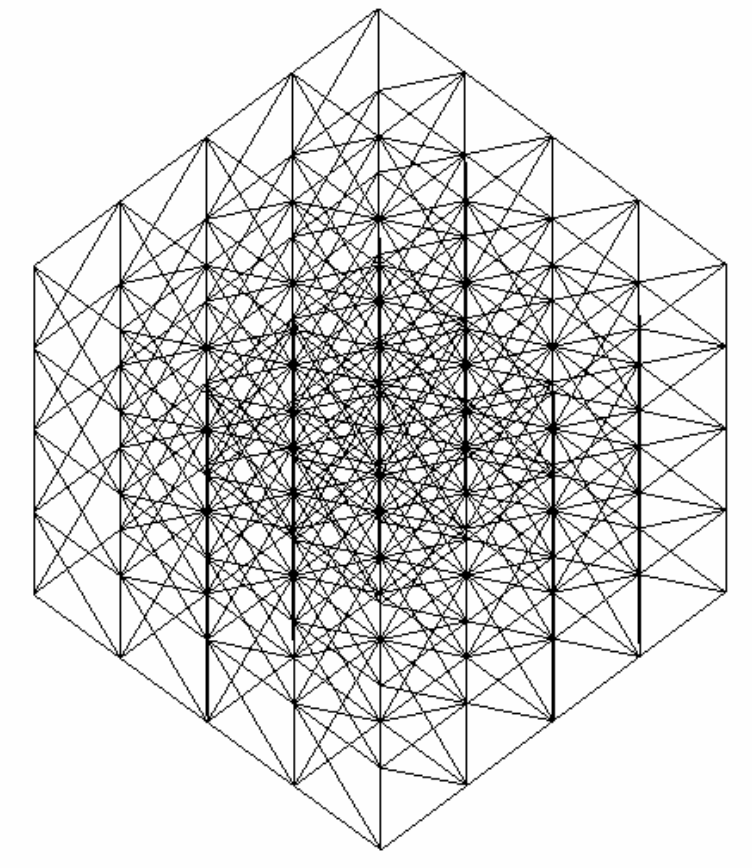

Figure 5: Naive culling example

For reference this method has a ratio of 6.

### 3.3.2 Culling

Clearly, in order to improve on the naive method is to simply not to draw the faces that are obscured by checking each cubes neighbours before creating a face. This means for each voxel 6 neighbourhood checks must be done. Of course those checks work both ways for the voxel and the neighbour so this can be improved to about 3 checks per voxel.

The reduction of face count can be in the best case, a solid chunk of n cubed voxels as seen on Figure 6, a factor n. At the worst case, namely a checkerboard chunk as seen in Figure 7, the same as the naive method. Earlier results suggest that the expected ratio for a Perlin noise map is somewhere in the 1 to 2 faces per voxel.

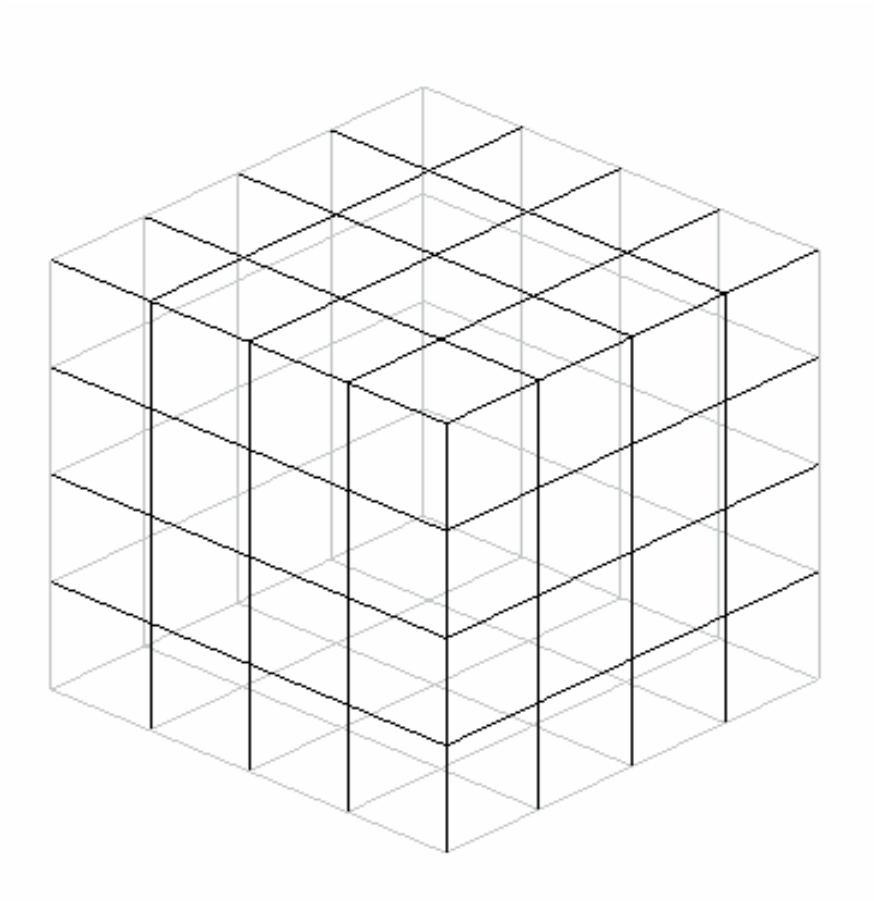

Figure 6: Example of culling voxel faces

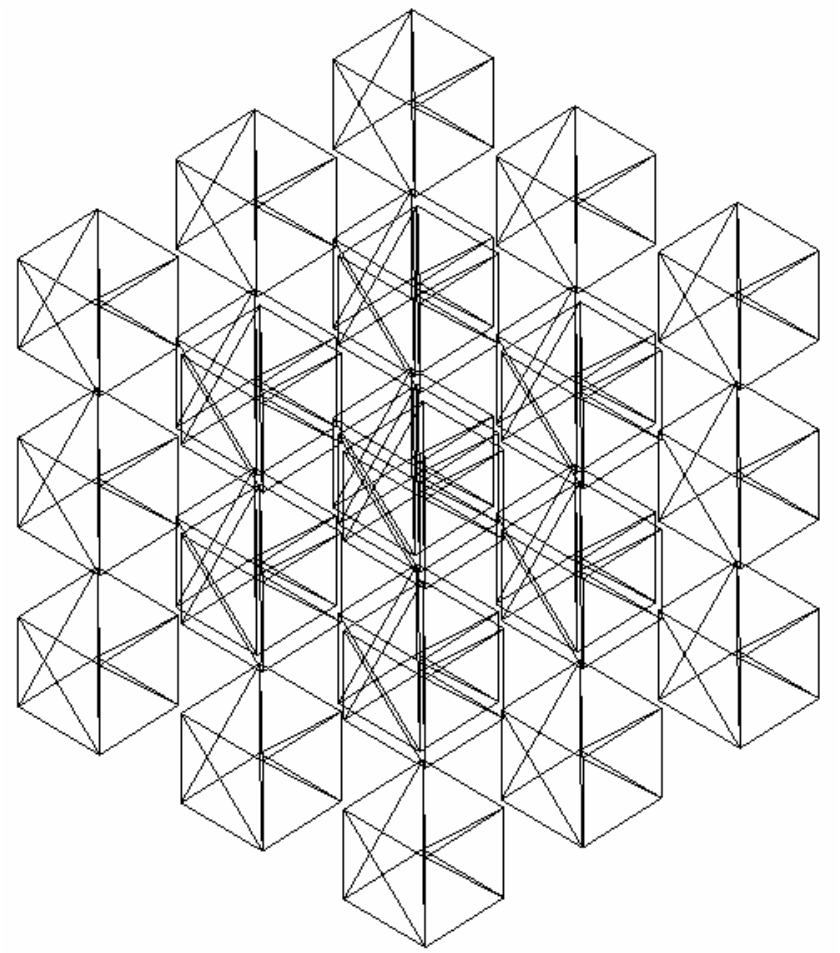

Figure 7: Example of worst case voxel chunk

### 3.3.3 Greedy Meshing

As the optimal mesh representation of a group of voxels is an open problem it is not practical to aim for that. While greedy approaches do not promise the best or even a good solution the more often provide a fast solution that will do.

The greedy meshing algorithm merges adjacent quads together into larger regions to reduce the total size of the geometry. For example see Figure 8.

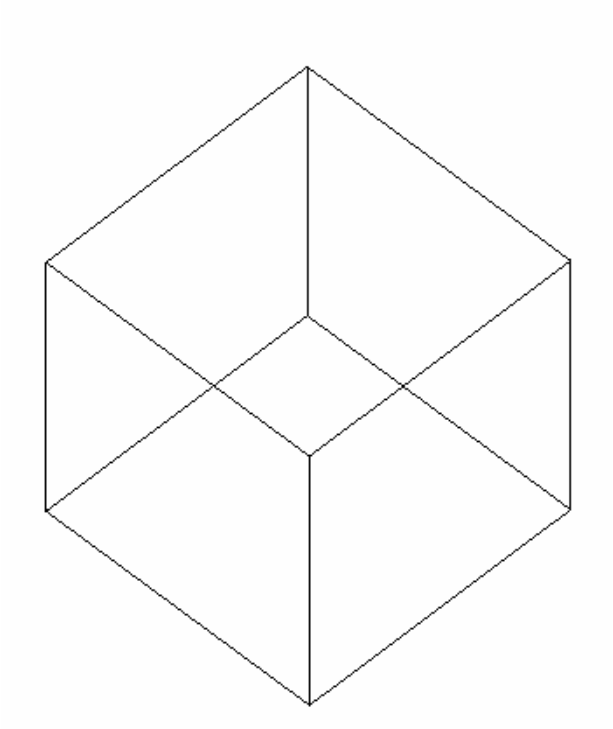

Figure 8: Example of greedy meshing on a solid chunk of voxels

Given that greedy meshing can improve the result of the culled method it is expected to give at least as good results if not better. Early test results show that the implementation has a ratio of 0.1 to 1 face per voxel.

## 3.4 Prototype implementation of greedy meshing

The idea was inspired by Karnaugh map method which is used to simplify boolean algebra exressions.[4] By analogy to K-Map, meshing can be improved by allowing faces overlap and extend over undefined (culled) space as shown in Figure 9. Given that the material (visual) is the same, there is no z-fighting or other artifact for this approach. There is no rule that faces can not intersect.

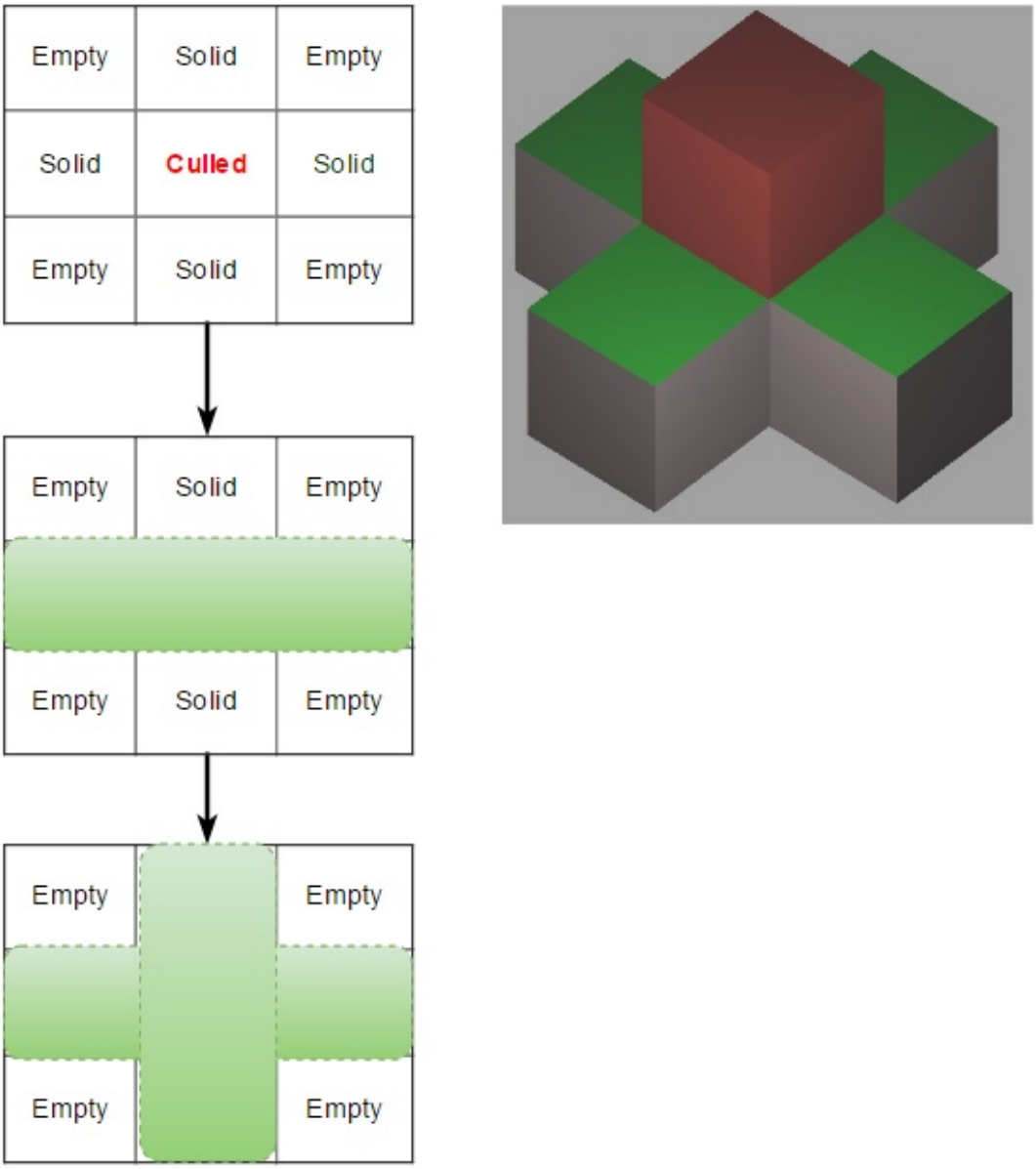


Figure 9: Example of same type faces allowed to pass through each other and over culled space.

As per [5] example, the culled approach would result in Figure 10 surface. While the greedy meshing would result in Figure 11. This implementation of this idea managed to improve on is to merge the floor tiles into one single big tile as shown in Figure 12. The greedy algorithm can generate reasonably fast results that are by a significant margin better than any simple culling. The main improvements can be seen in contrast of Figure 13 and Figure 14 where the improvements in vertex and face counts are listed.

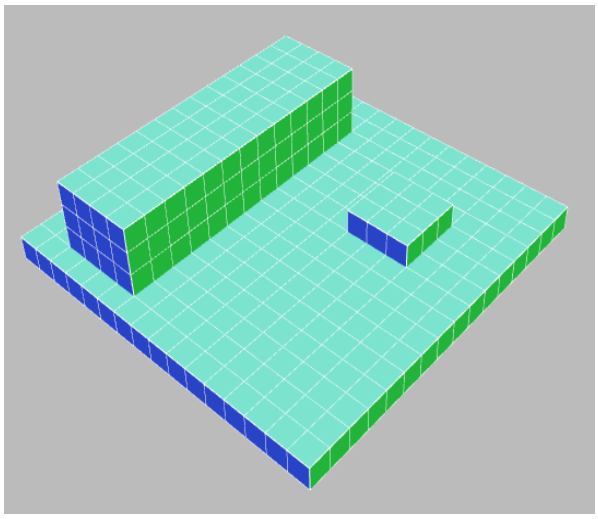

Figure 10: Mikola Lysenko's example of culling[5]

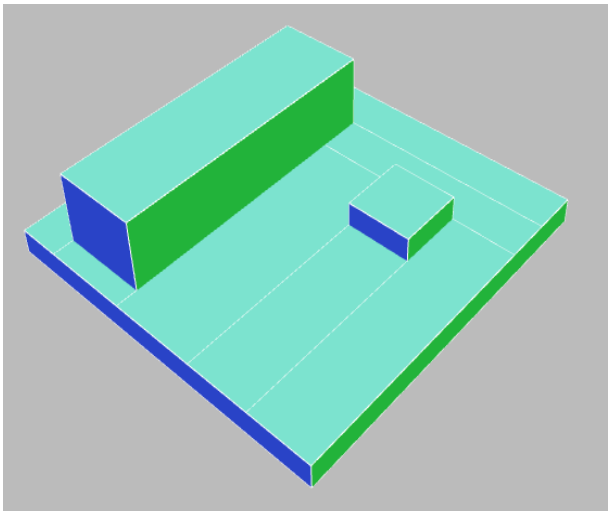

Figure 11: Mikola Lysenko's example of greedy meshing[5]

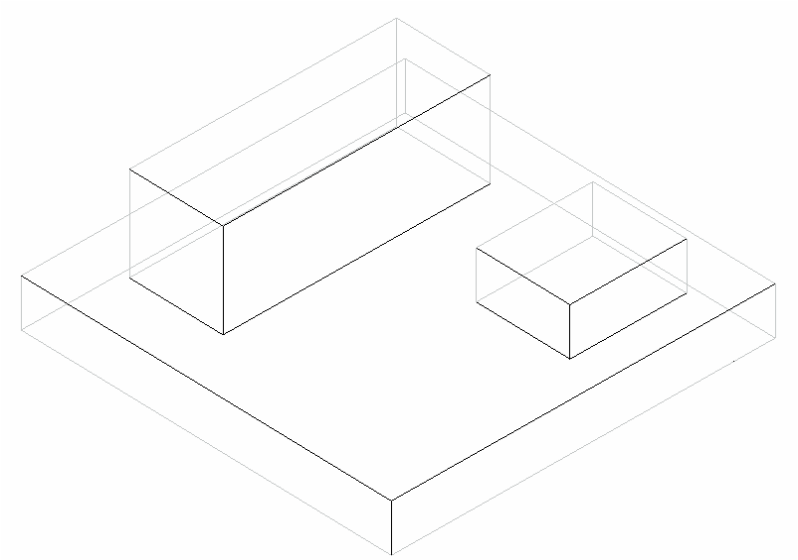

Figure 12: Improved greedy meshing result

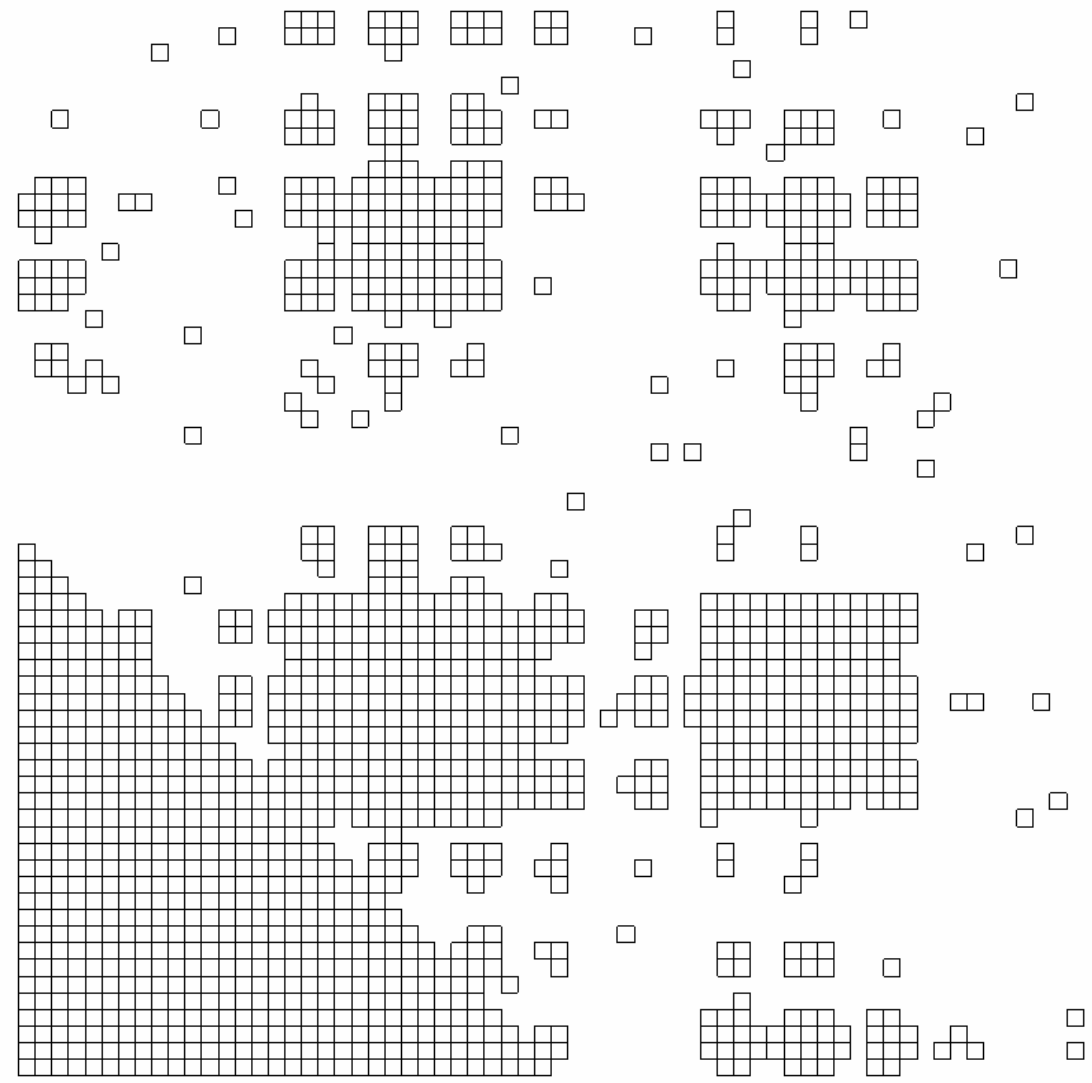

Figure 13: Example of a unoptimized plane with 5784 vertices and 1446 faces.

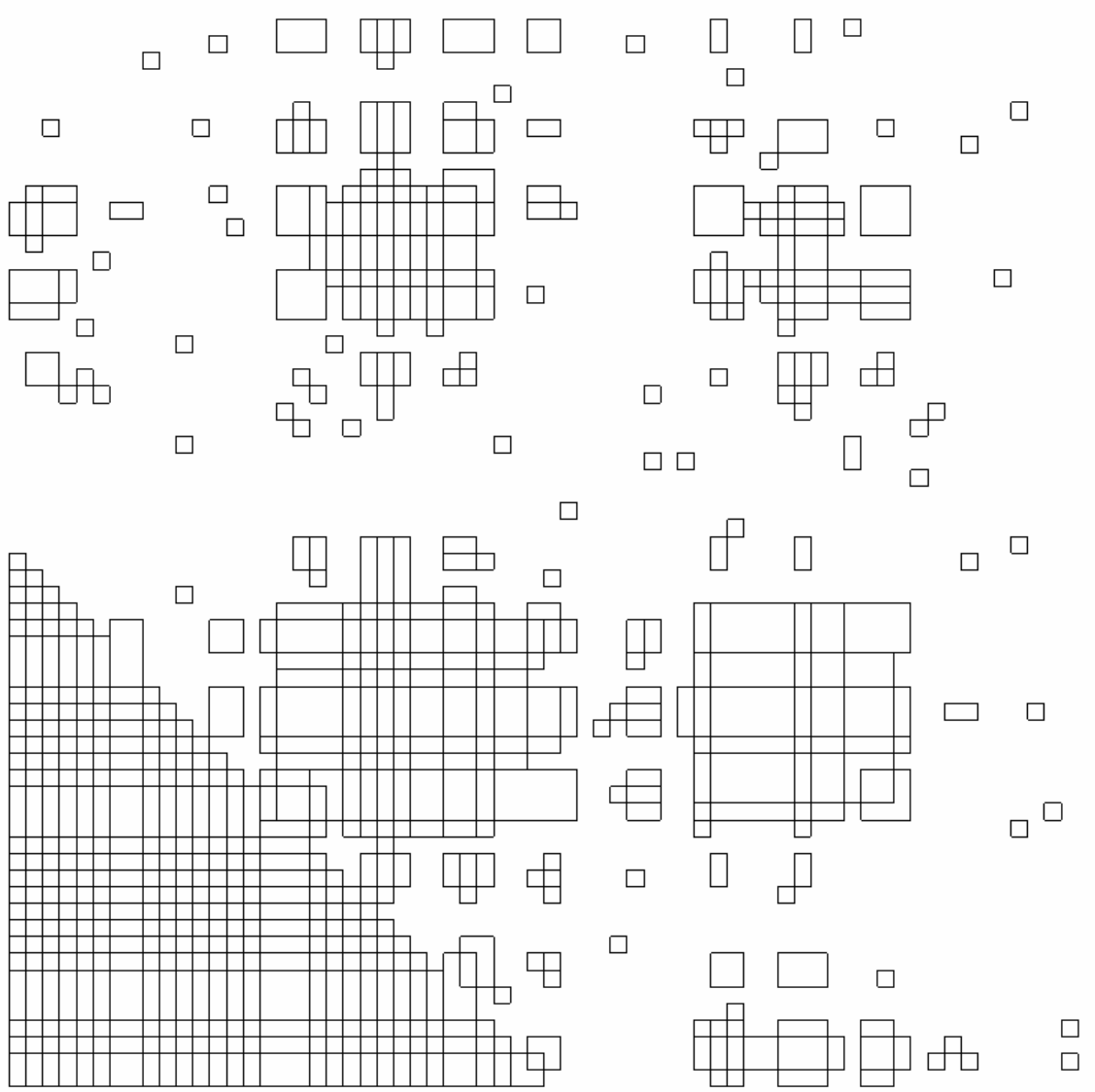

Figure 14: Example of greedy meshing a plane. Vertex count has been reduced to 792 and face count to 198.

Testing showed that the optimal chunk dimension is 8 units, in order for the meshing not to take too long. At higher dimension sizes the smallest schedulable work unit took more than 16 ms and was in violation with the set performance goals.

### 3.4.1 The basic algorithms

To begin, every chunk with $n^3$ voxels has 3(n+1) planes where faces can be drawn as seen on 8x8x8 chunk in Figure 15. By pre defining these planes there are no unnecessary neighbourhood checks. The result is a flat array representing a two dimensional slice of the voxel chunk, where each element has a value that represents either no face, desired face value or undefined value. Undefined values mean that it does not matter what kind of face polygon extend over this particular face.

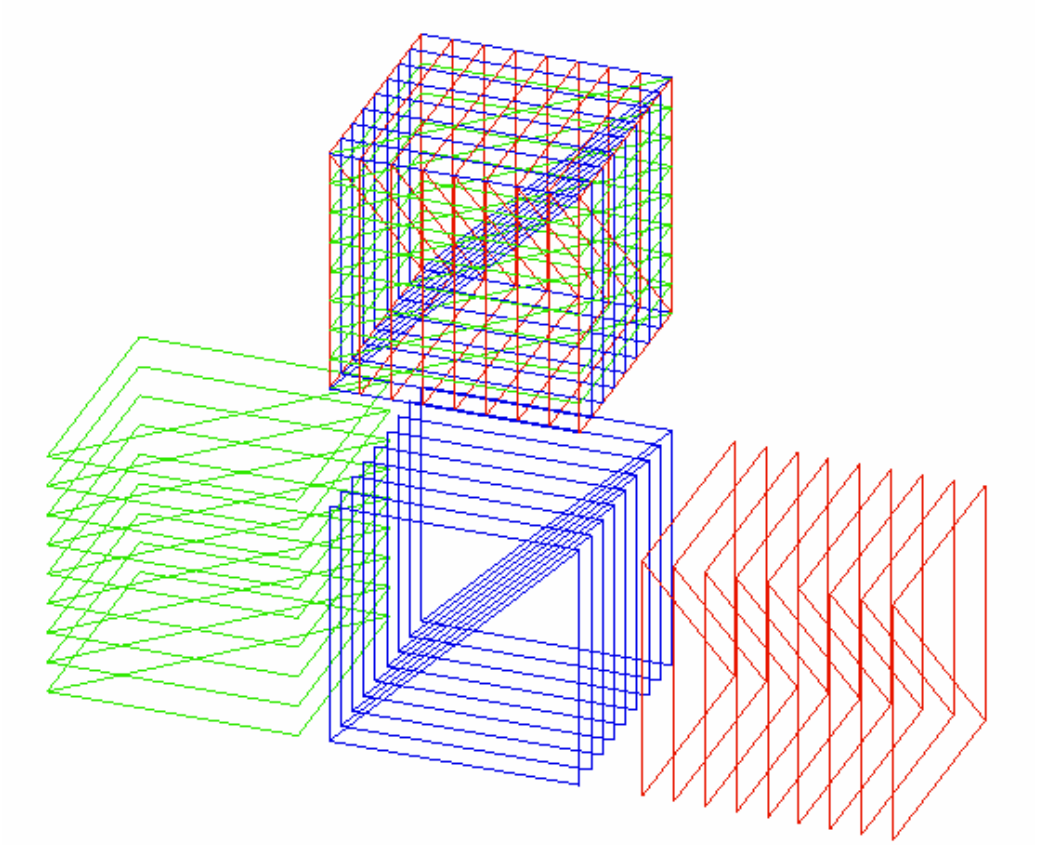

Figure 15: Axis aligned planes of a 8x8x8 chunk.

To illustrate, consider the example shown in Figure 16. There is shown a simple chunk with two layers of voxels and where the middle plane is occupied by green, red (obscured) and none type of faces.

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | -1 | -1 | 1 | -1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

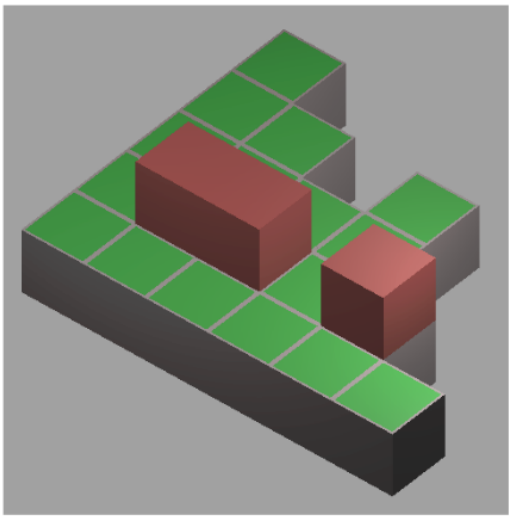

Figure 16: Example of the initial state of greedy meshing.

The algorithm starts by iterating over each plane's faces till the first undiscovered face is found as shown in the example Figure 17.

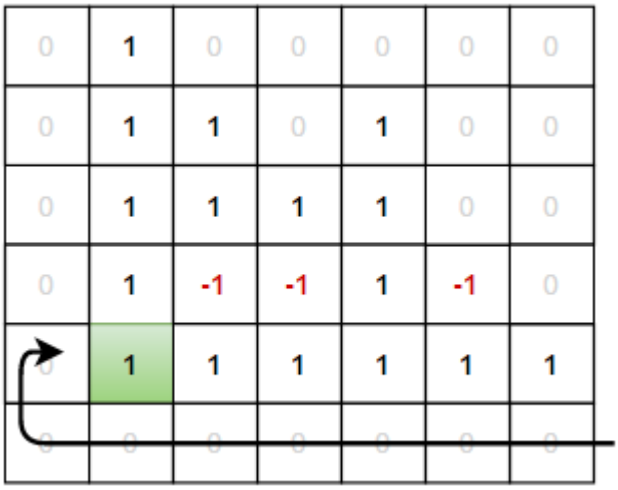


Figure 17: Step 1 of greedy meshing

Starting form that face the first axis is chosen. In the example of Figure 18, the axis is chosen to be the positive Y.

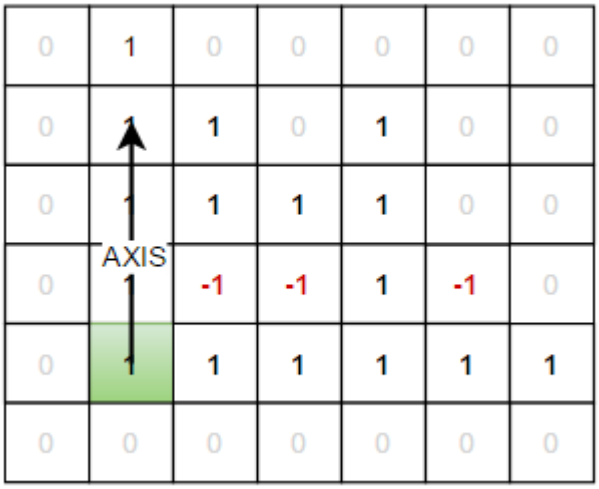


Figure 18: Step 2 of greedy meshing

The next step is shown in Figure 19. The plane from that starting field till the first obstructing field which can't be stretched over is found. This is noted as the height of the first column.

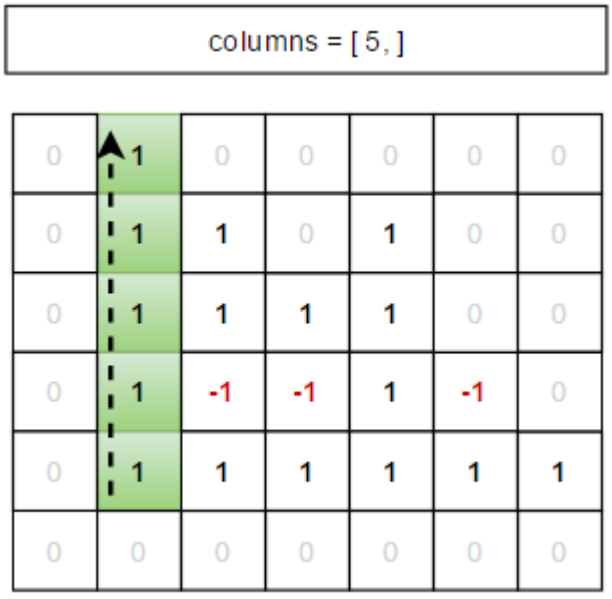


Figure 19: Step 3 of greedy meshing

Then this process is repeated on the same axis on the next faces again till the first obstructing field which can't stretched over is found or it has gone past the height of the shortest column as shown in Figure 20. It finally stops at the end of the plane or if the column height is 0.

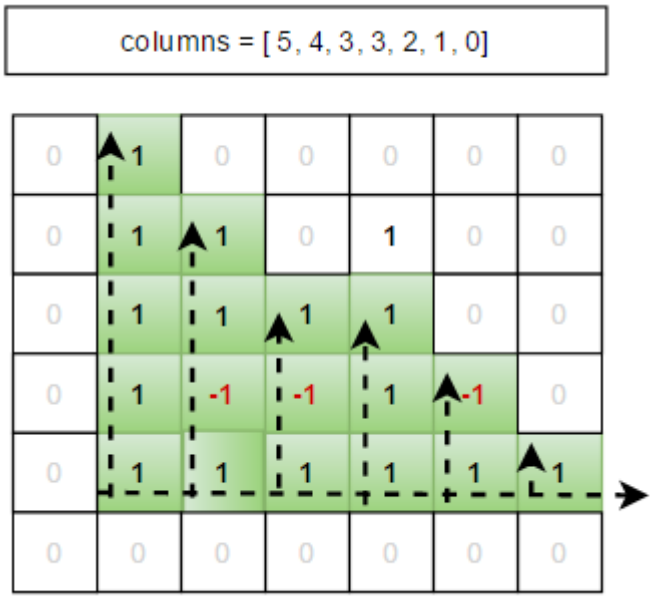


Figure 20: Step 4 of greedy meshing

The resulting list now has the column heights in decreasing order. Not stepping further than the shortest column ensures that the repeating numbers can now be ignored and the generated polygon can be stretched from the furthest repeating number column to the first column. Those unique numbers and their indexes mark the corners of the polygons to be drawn as is illustrated in Figure 21 and Figure 22.

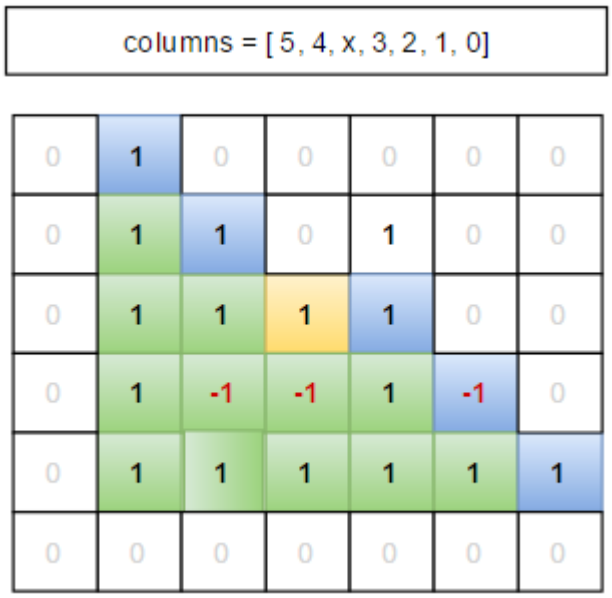


Figure 21: Step 5 of greedy meshing. The marking of the corners.

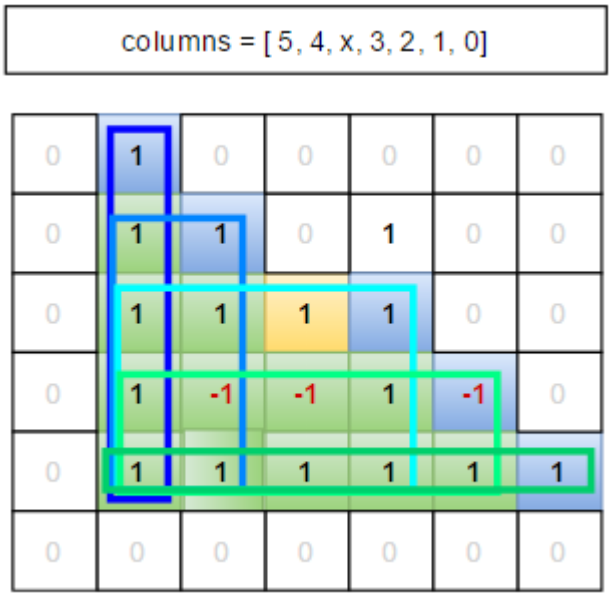


Figure 22: Step 5 of greedy meshing. The creation of polygons from corners.

This is repeated on all 3 remaining rotations and after that all covered faces are considered to be visited. Visited means, that they are not considered as new starting points. After that the faces are marked in a similar temporary array where the reference count of how many faces it covered is noted. The overwrite rule is based on the idea that a faces are covered by the most referenced or recent equal face. This is illustrated by Figure 23, where an old face D was overwritten by E. While that number does get dirty (invalid) over time it gives a quick hack to reduce the face count with little performance impact. Later, when all the faces are visited, all faces still referenced in the temporary array are to be drawn.

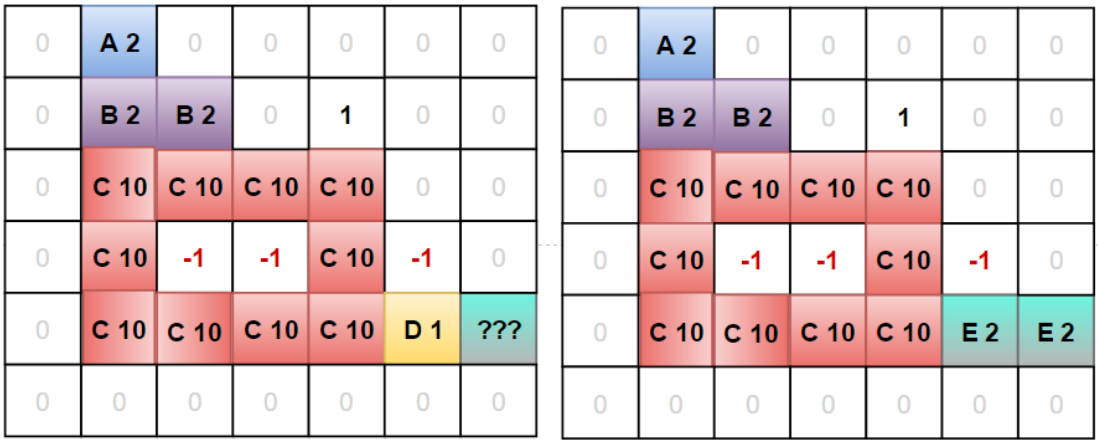


Figure 23: Example of greedy meshing reference overwriting

This method has certainly some shortcomings. For example corner faces should get preferential treatment. Faces from different orientation passes could be merged together on the axis edge. Even parallelization can be applied as the planes are not dependent on each other. There are many possible improvements that can be done, but for the time being those can be postponed.

## 3.5 Afterthought on meshing

During the writing of this algorithm a third option was conceived. While sharing polygons is a proven viable idea, all voxel faces on a plane can be drawn with a single polygon where the polygon's texture pixels are the faces. So if at least 1 face exists on a plane there has to be at least one polygon. This would hint that polygon-wise this would be the optimal solution for the least amount of polygons possible. This would work by stacking those textured chunk planes on all 3 axes like in Figure 24. More detailed textures could be added in a material shader as the pixels values, world position and normal can be combined into a texture UV coordinate. The concept render of this approach is Figure 24.

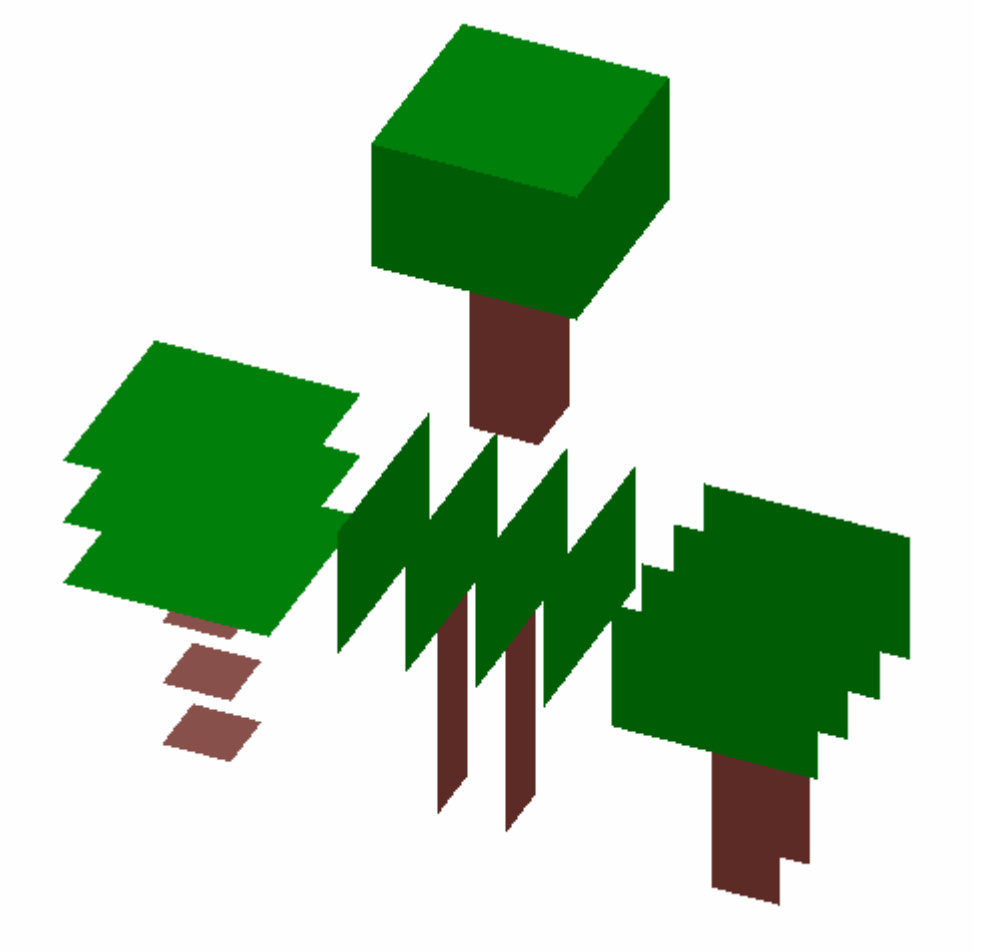

Figure 24: Example of stacking textured planes

The the worst case checkerboard of a 32x32x32 chunk would be 99 polygons with a 320 by 320 texture as shown in Figure 25. The best case of a single voxel chunk would be only 6 polygon planes done with a 6 pixel image. While an alpha channel image (4 bytes per pixel) with 101376 pixels would take about 406 kilobytes, whether a graphics engine can handle large amounts of chunks with unique textures is something to be looked into.

$$chunkSise = 32$$
$$planeCount = 3 \cdot (chunkSize + 1) = 99$$
$$pixelCount = planeCount \cdot chunkSise^2 = 101376$$
$$imageDimension = \sqrt{pixelCount} \approx 320$$

Figure 25: Calculation for worst case pixel count and image size.

## 3.6 Physics

It is notable that physics is major factor in voxel engines. The simplest way to implement physics is to use mesh based physics, where the game engine would calculate the intersection all physics enabled faces. As the polygon count is the main concern this would also double the impact. This is why mesh based physics are a bad option. An alternate method is to only spawn invisible physics enabled cubes only in the Moore neighbourhood of each actor. This method is what was used in an earlier project with minimal performance impact Figure 26.

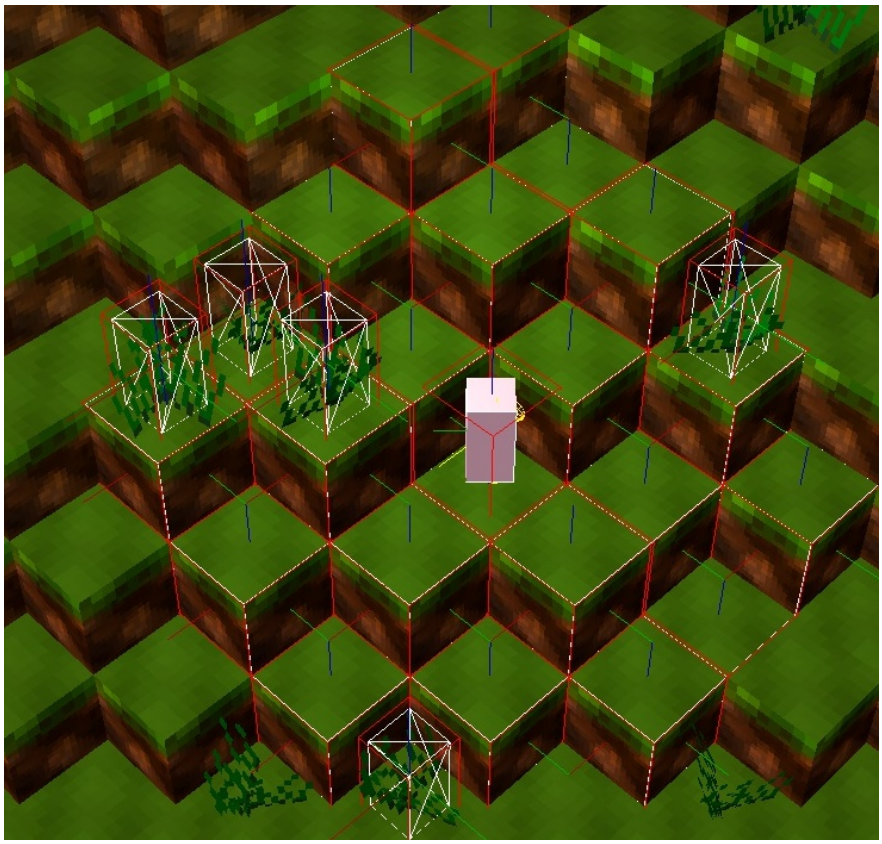

Figure 26: Example of cube based physics from an earlier project

The solution which was used in the prototype was to force the behaviour of the actors to move only in a chess like unit motions and calculate the physics in the voxel engine code. Because the random voxel access time was low, the performance overhead even with a thousand actors was miniscule and schedulable.

## 3.7 Terrain generator

To generate a playable world out of voxels, some form of terrain generation is required. One such way is to use Perlin noise. “An implementation typically involves three steps: grid definition with random gradient vectors, computation of the dot product between the distance-gradient vectors and interpolation between these values.“[6]

For the prototype's purpose, the algorithm produces a pseudo-random heightmap which can be used to spawn mountains and valleys. The appearance of randomness is enhanced by the fact that multiple octaves of noise are used. In order to eliminate Perlin noise as a performance factor only the first octave was used in the prototype. The resulting terrain can be seen in Figure 2.

# 4 Pathfinding

What is pathfinding, its concerns and why is it relevant? Pathfinding as the name implies describes the process to find a path between two points. This field of research is heavily based on finding the shortest path in a weighted graph. At its simplest a pathfinding method starts at a node and explores its adjacent nodes in least cost order till the destination node is reached.

Two primary problems of pathfinding are finding a path and the shortest path problem. The basic approach is to exhaust all possibilities. This of course is far from practical. Combined with the more complicated problem of finding the optimal path all system resources will be quickly exhausted before finding a solution. This is where algorithms such as Dijkstra and A* improve on. They strategically eliminate paths through dynamic programming and heuristics. By elimination the search space drastically shrinks.

But why voxel games are different? There is inherent expectation of the world's capability to perform alteration. Like the user is able to manipulate the state of the world also the inhabiting actors must be applicable to the same expectations. This means that the pathfinding not only has to traverse nodes but also be able to alter the state of the world. By bringing this new dimension of complexity to the pathfinding problem the prospect of finding the optimal path becomes seemingly unreachable.

As often pathfinding problems are more concerned of efficiency or being shortest possible than focusing on features. There is a tendency to forget that the true aim is to provide the user with unique experiences. A game is not the real world. What games mostly do is to approximate and fake features found in real life. There is nothing wrong with sub-optimal solutions if they are believably reasonable.

## 4.1 General pathfinding methods

### 4.1.1 Pseudo-random pathfinding.

Simplest would be the Brownian motion inspired pathfinding. This algorithm just expects the actor to move around randomly. With any set of simply random vectors there won't be any path or progress made normally. But if one can offset the probability of the vectors appearing by some dynamic function the resulting biased motion or drift will try to reach the direction. If the motion to the direction of the target is more probable, it may get there eventually.

For example we can use this to call the sheep to a location or scare them with wolves away from a position. As these actions can be executed without any foresight there is little to none performance impact. This is why approach is very good at giving a sense of activity to a large group of actors without impacting performance as illustrated by Figure 27.

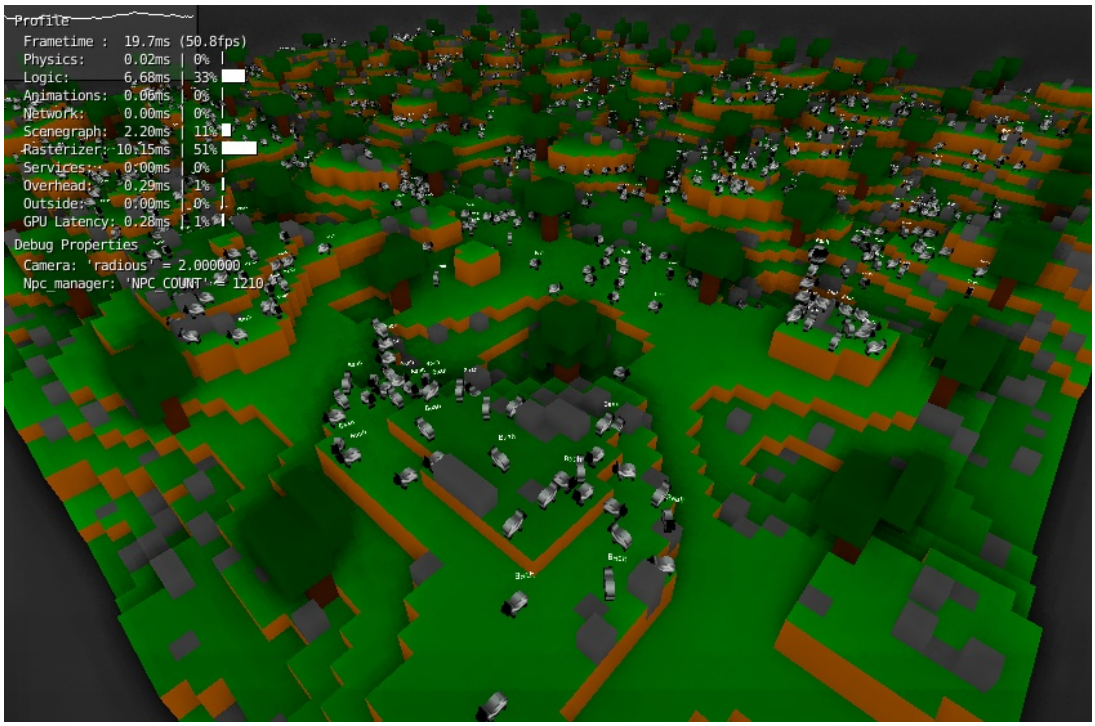

Figure 27: Example of activity and impact of 1210 sheep actors.

### 4.1.2 Dijkstra's algorithm

Dijkstra’s algorithm works by visiting nodes in the graph from a starting point. While it examines each newly discovered not-yet-examined node in a lowest cost order, it expands outwards from the starting point until it reaches the goal. This algorithm fails if there is a negative edge weight. Otherwise Dijkstra’s algorithm is guaranteed to find a shortest path.

The main shortcoming if Dijkstra’s algorithm is that is does not make use of all the information present. It does the pathfinding solely on the cost function while avoiding any attempt to make assumptions. See Figure 28.

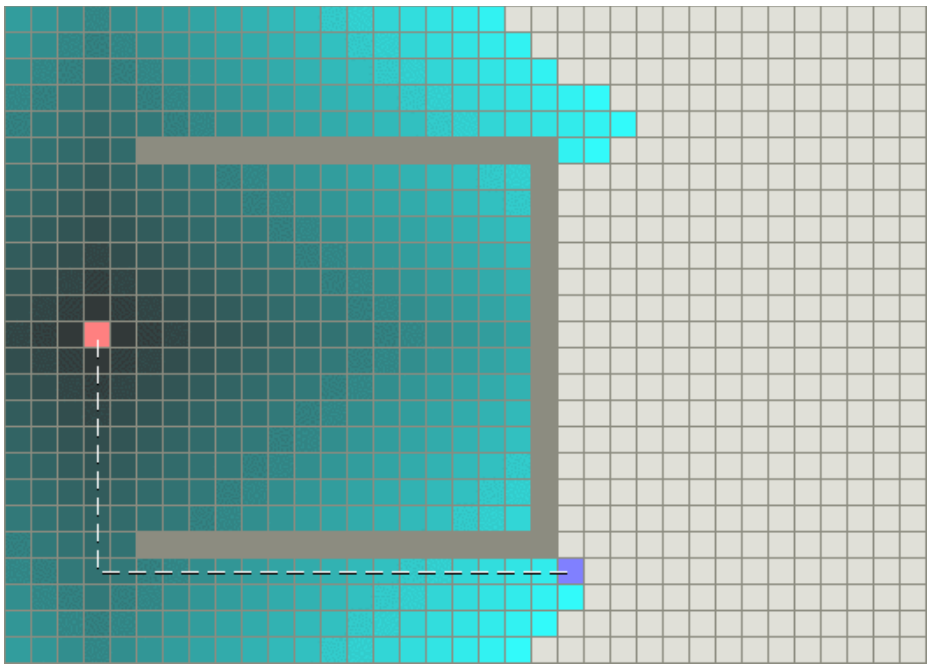

Figure 28: Example of Dijkstra’s algorithm wasteful nature. [7]

This can be used as an advantage when multiple undiscovered endpoints are present and there is no preference of targets. Without exploring any alternatives the first shortest path to a valid target is chosen. In the example Figure 29 it is shown how this applies to the wolf and sheep situation.

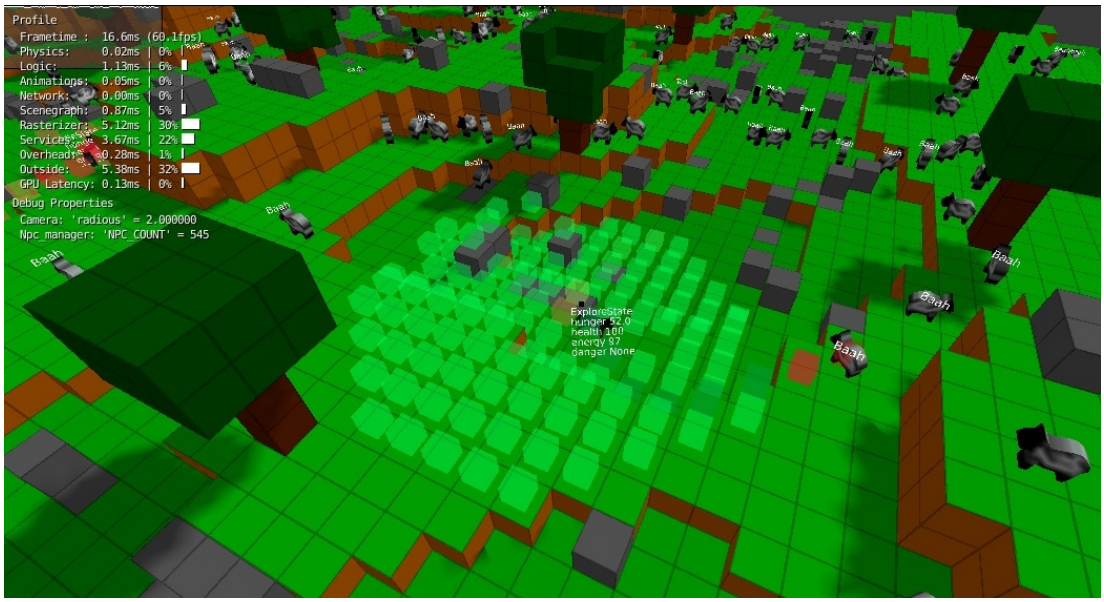

Figure 29: Example of multiple valid targets being available to the actor.

While this is a neat feature, there is a pre-processing trade-off. If the amount of targets is low or they are positioned relatively far away, a better approach would be to list all the valid targets and to perform heuristic search algorithm for each target. One of such algorithms is A*.

### 4.1.3 A* search algorithm

More commonly than Dijkstra's algorithm A* is used in games. It is evident in the example Figure 30 that the A* is an improvement as the number of nodes probed is visibly reduced.

While being a variant of Dijkstra's algorithm the secret to its success is that it combines the travel cost information that Dijkstra’s algorithm uses and the remaining distance to the target. This approximate distance is found by the heuristics, and represents a minimum possible distance between that node and the end. When the weight of heuristics in the evaluation approaches zero, A* becomes equivalent to Dijkstra's algorithm. As the focus on heuristic increases, the result becomes more greedy and no longer guarantees an optimal path. In many applications like video games this is acceptable and even desirable, in order to keep the algorithm running quickly.

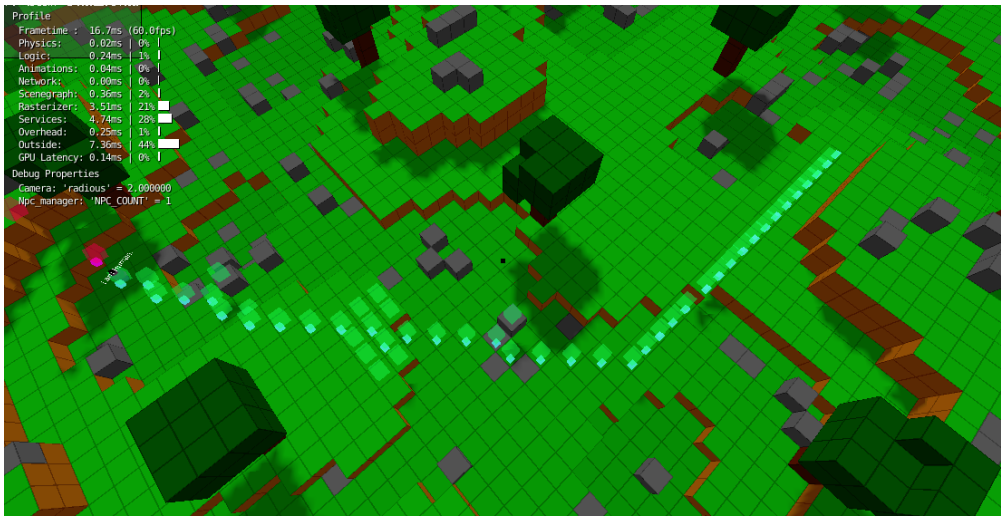

Figure 30: Improvement in visited node count in A* pathing

## 4.2 Prototype's implementation of world altering pathfinding.

The goal is to enable the actors to deliver the experience that the users expect from a capable AI. The main such capability is show voxel conscious path planning. This requires to handle world state tracking per node which memory inefficient and leads to further branching. In the following are shown possible options that were tested.

### 4.2.1 Branching A* altering path

This algorithm follows the A* pathfinding pattern. The main change to the algorithm is the added functionality for passing through solid voxels where the travel cost is calculated with the necessary changes in mind. The most significant adjustment is that after each change to the graph the current node chain is considered to have moved to another branch of graph. This results in nodes being visited multiple times on branched graphs which is detrimental. The algorithm will find the shortest path if possible, as each branch paths are considered equally in the queue and after the fact can be viewed as a path emerging on a pre modified graph. Most concerning drawback is the uncontrolled branching which quickly saturates the system resources. The concept is viable only on short distances as the path shown in Figure 31 and executed in Figure 32, required 20000 nodes being visited which while visually unseen caused major lag in the prototype.

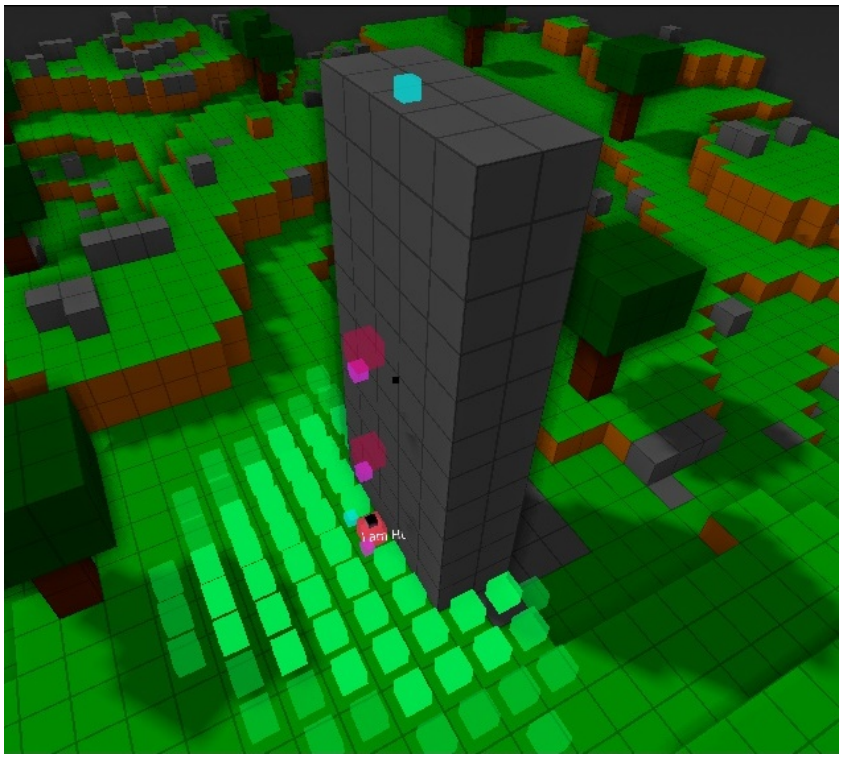

Figure 31: Branching pathfinding search visualization

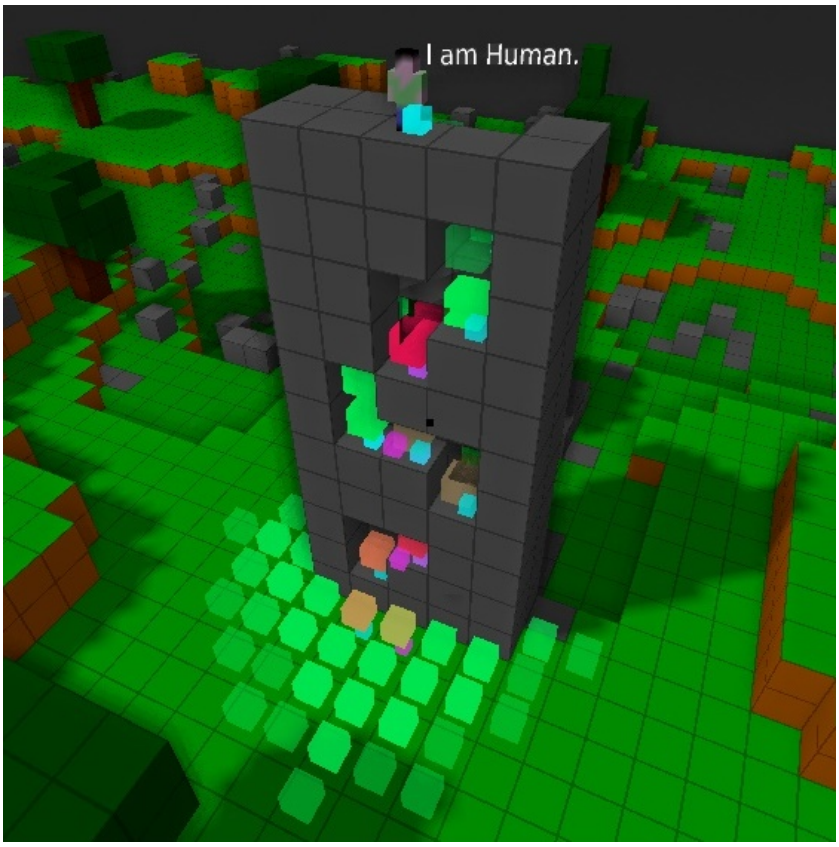


Figure 32: Branching pathfinding search execution

### 4.2.2 Greedy A* altering path

Previously it was assumed that after each change in the world the node on that chain continues in a branched graph contrary to other currently evaluated nodes as any change applied in the current chain of nodes may not apply to other. In order to avoid branching the nodes are limited to be evaluated only once. This is achieved by first come first served basis. This serves the purpose of avoiding node overlap between branched graphs. The child nodes inherit the invalidated nodes list of its own path chain which in addition to the initial graph servers as the branched graph.

The resulting path is capable of building stair structures into solid spaces as was the branching algorithm. The main difference is that while the path is not optimal, it manages to explore more node with zero overlap which in turn reduced the visited node count to 2000 in the path shown in Figure 33 and Figure 34.

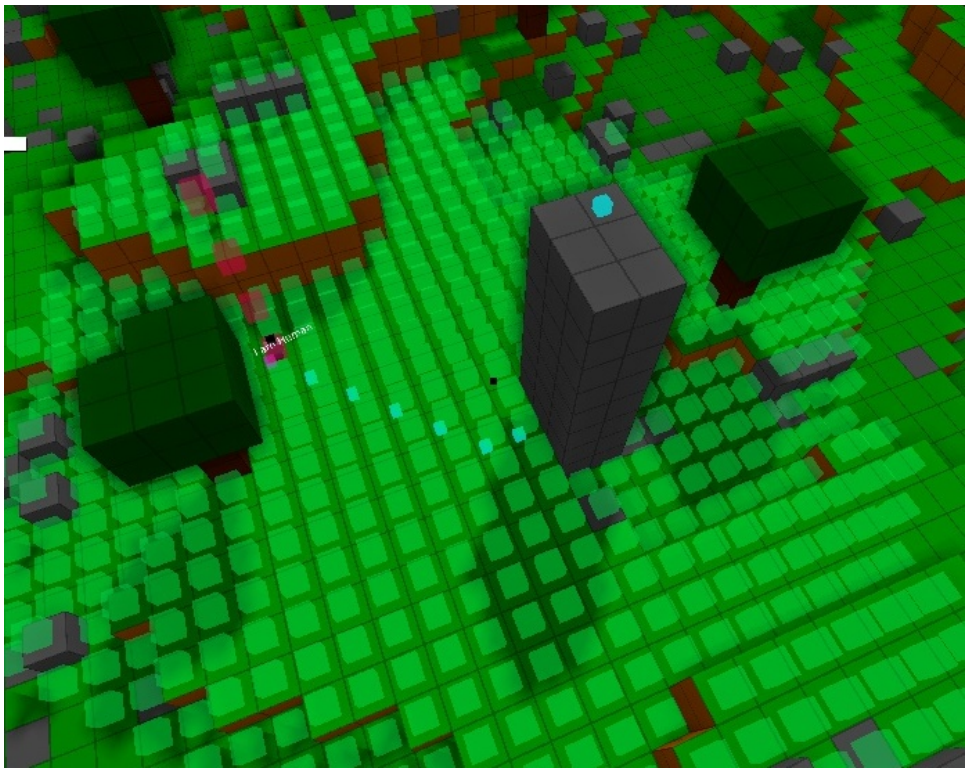

Figure 33: Greedy A* altering path generation visualization

Figure 34: Greedy A* altering path exccution visualization

While during testing no test case was found where this could not produce a path, this algorithm still discards a lot of possible paths as first choice basis eliminates otherwise perfectly fine nodes. What is more, while the path that this algorithm serves is acceptable through solid space, it makes no effort to generate a better standard surface path.

### 4.2.3 Hybrid greedy altering path

While the greedy algorithm didn't do exceptionally well on normal unobstructed graph a simple improvement is to first explore the graph with standard A* algorithm while noting any starting points for paths that require voxel alteration. When the first pass does not yield a path then all the noted points are entered back into the pathfinding stack which then is passed to the greedy pathfinder. This results in an easy improvement on the greedy path with no drawbacks for paths that do not require alteration. The first pass can be seen in Figure 35 where the actor tries to climb the cube structure by first searching any possible paths around it and then resorts to more destructive approach in Figure 36. Of course, the issue of getting sub-par results for any paths after the first stage is still present.

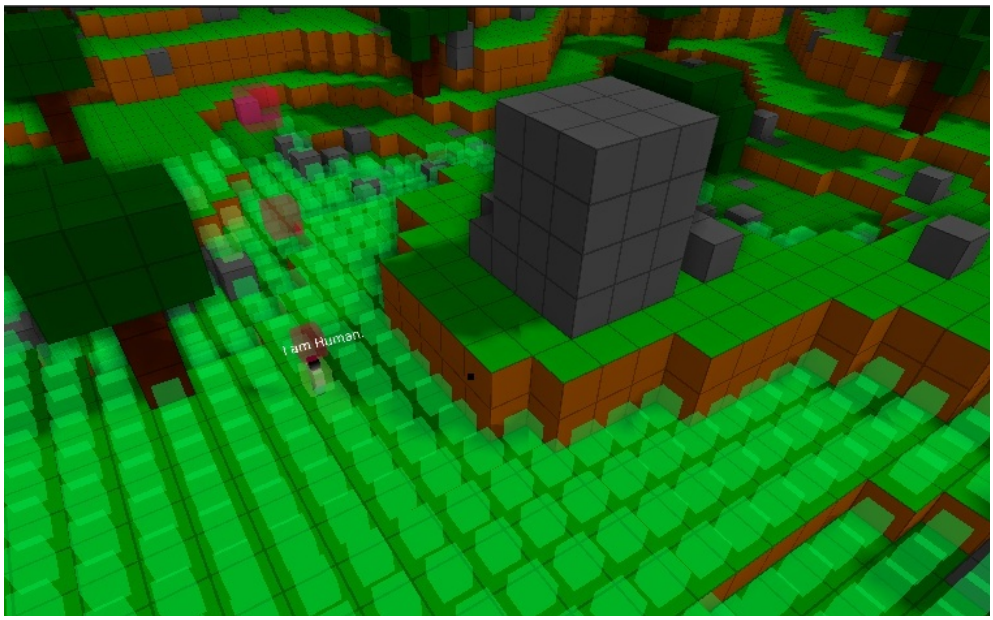

Figure 35: Hybrid greedy altering path first stage: searching for a simple path

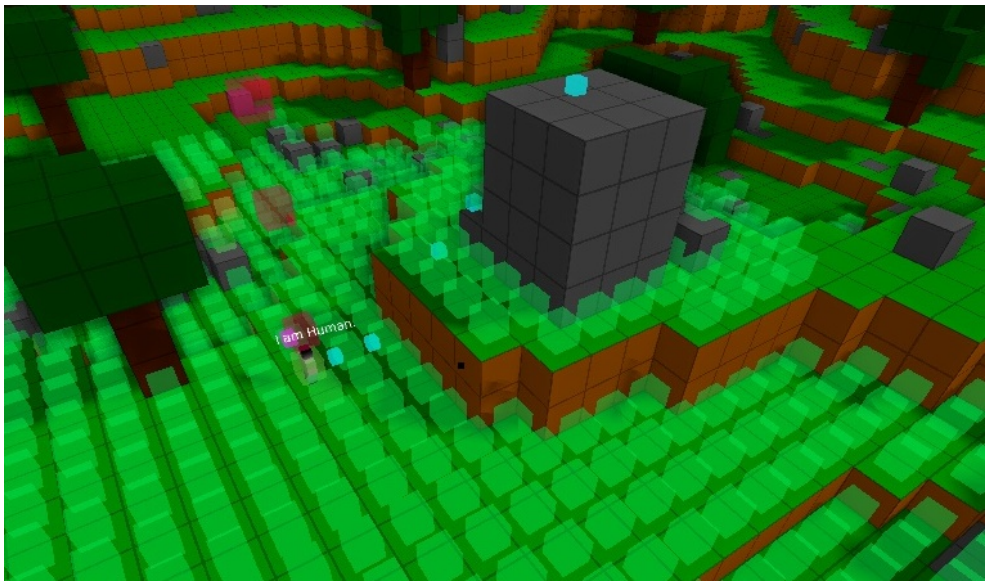

Figure 36: Hybrid greedy altering path second stage: continuing with greedy pathing

### 4.2.4 Possibility of improvement

While branching search is a broken algorithm, the pursuit of that idea was not without merit. The preservation of the node updating nature of the A* algorithm after the alternating stage of the greedy is something that should have been a result of this study. Relaxing the “no update” criteria mentioned in 4.2.2 could merge the two staged process into one that is similar to the branching algorithm. For example by allowing updates on any node which branch is same, but otherwise enforce first choice rule.

# 5 AI - Intelligent agents and decision handling

An AI system is composed of an agent and the environment in which they act. “An intelligent agent is an autonomous entity which observes and acts upon an environment and directs its activity towards achieving goals. Intelligent agents may also learn or use knowledge to achieve their goals.”[8]

Such behaviour is achieved by the use effectors to perform actions in response to perception. Effectors are what the agent uses to perform actions. Those include legs and hands in real life and by analogy movement logic actuators in Blender. Actions are what results from the use of effectors. For example change of its location or damage to enemies. Sensors are what the agent uses to perceive the world. In Blender this is done by using logic sensors or scripting.

Agents are considered rational if they work towards the optimal outcome. The rational actor should be concerned with results of its actions in response what the agent has perceived. This way the actor performs actions that causes the agent to be most successful the knowledge of the world it has acquired.

By only having an interactive world and means of exploration the agents capable of action but not reaction. This on its own does not count to being a good experience for the user. As the goal is to inhabit the game world with agents that are rational in order for the world to be perceived interesting and challenging there is a need to explore ways to program such agents.

## 5.1 Stateless

The easiest form of a rational agent is a simple stateless agent. It can only act on the basis of the current perception as it has no information of its past observations. “The agent function is based on the condition-action rule: if condition then action.” [8]

As there is no state or perception history to evaluate the performance and response time is great. While the decisions lack foresight and appear naive it can avert the attention from that by the sheer numbers of actors that can be simulated. This is great for giving the user impression of an alive and populous world. For example it is easy to implement sheep logics as such as they walk around randomly or while seeing a wolf run away as shown in the transition from Figure 37 to Figure 38.

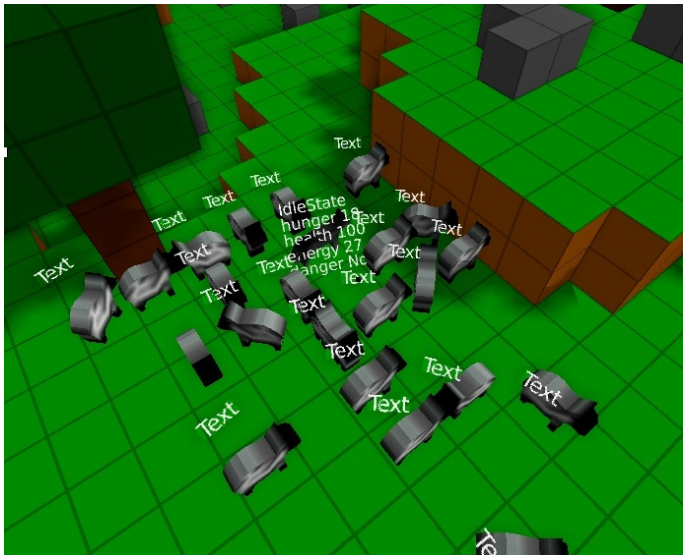


Figure 37: Sheep being spawned next to a threat

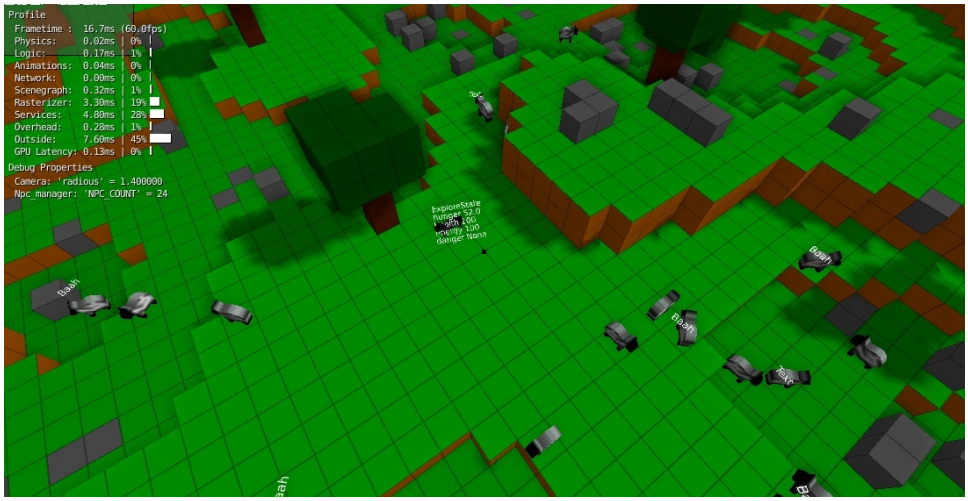


Figure 38: Sheep running away from threat

## 5.2 Finite-state machine

A finite-state machine is in only one state at a time. While being in a state the actor performs an action of that state and checks the perceived state of the world. It can transition from its current state to another when initiated by a triggering event or condition.

This is great for developing smarter agents compared to stateless logic. By being state aware the actor can complete rational sequences of actions. An added benefit is that there is not much difference in performance compared to stateless as the logic is comparable and the overhead is small. As the states and transitions are finite in number and written by hand the resulting choices have easily understandable logic to them.

For example a wolf might have four main states and the corresponding transitions. See Figure 39. The wolf can execute rational sequence of actions. For example it knows that in order to feed it has to explore for food or prey while protecting itself from danger and fatigue. The transition from being inattentive Figure 40, to being hungry Figure 41 and lastly eating Figure 42 is illustrated below.

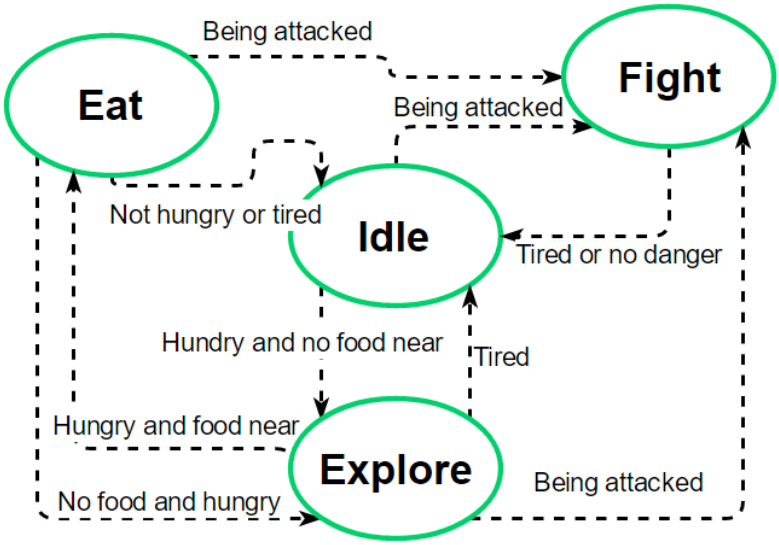


Figure 39: A wolf's states and transitions

Figure 40: Wolf in idle state

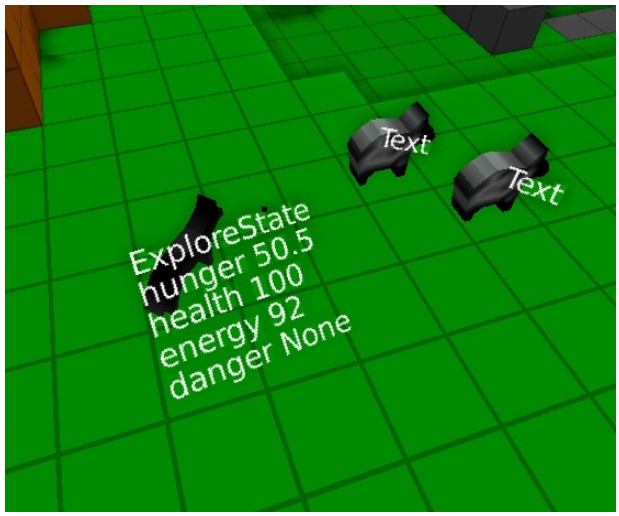

Figure 41: Wolf having changed to explore state as of being hungry

Figure 42: Wolf having found food (a sheep) and changing to the eating state.

## 5.3 Goal oriented action planning

Goal oriented action planning is a system that allows to plan a sequence of actions to satisfy a particular goal. The planner finds the appropriate combination of actions based on what preconditions are available to fulfil the goal that is self-consistent. This is achieved with A* pathfinding. The use of A* with the appropriate heuristic avoids visiting overly expensive action routes.

“A GOAP system does not replace the need for a finite-state machine, but greatly simplifies the required FSM. A plan is a sequence of actions, where each action represents a state transition. By separating the state transition logic from the states themselves, the underlying FSM can be much simpler. Actions define when to transition into and out of a state, and what happens to the game world as a result of this transition. Ultimately, the Goto and Animate states cover most of things that agents do. GOAP offers a much more elegant structure that better accommodates change. The addition of design requirements is handled by adding actions, and preconditions to related actions.” [9]

The states that the actor can be in are described as positive valued vectors while the queries and transitions can have negative values. The query is formed by the difference of the current state and the desired state. A goal state is found when a sequence of actions is found that transforms this query back into a positive vector. The sequence is found by working backwards from this goal query by starting with the last action and moving to towards the starting action and representing the unfulfilled pre-requirements of each transition as negative values in the query. A simple example scenario is the acquisition of food rations and weapons, namely the creation of cake from ingredients and picking up swords as seen in Figure 43 and Figure 44. The starting state would represent the actors inventory. The desired state would be the desired state of 3 cakes and 3 swords with other values being 0 in order to be ignored. This would result in a query where the actors state is some number of negative swords and negative cake in its inventory. From there the transitions are applied to the query state in the same way as possible moves are evaluated for the current node the A* pathfinder.

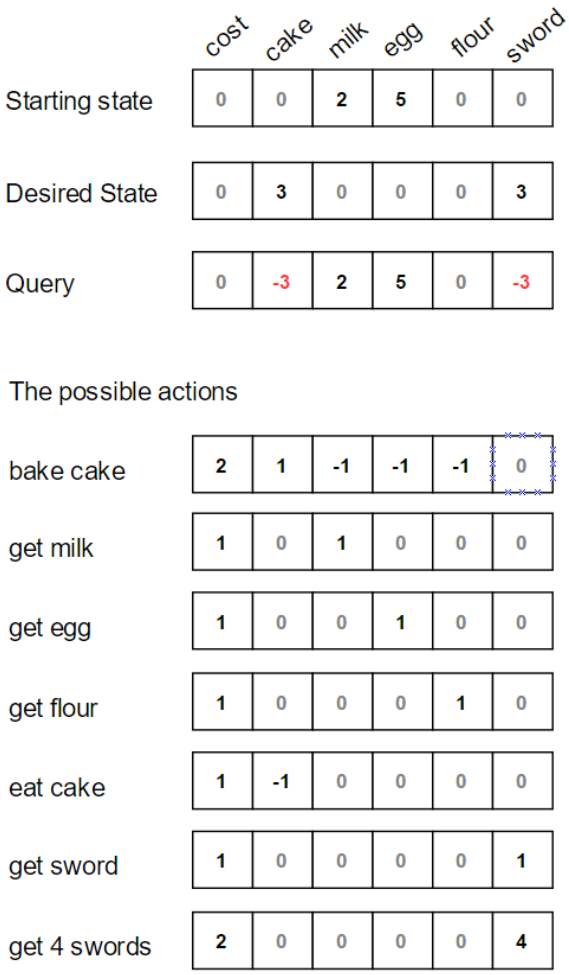


Figure 43: Base variables of the GOAP example

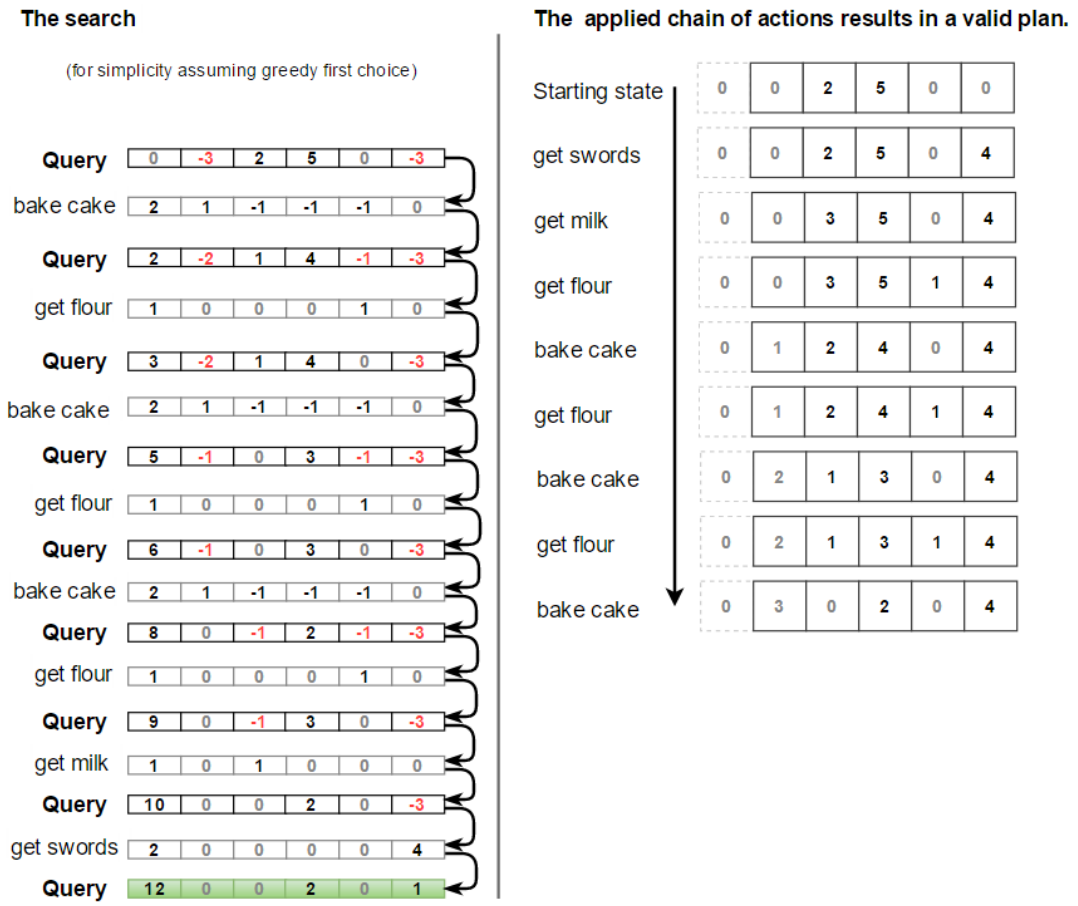


Figure 44: The search and result of the GOAP example

While this was not fully implemented into the game itself, the benefits speak for themselves. Characters in the game can exhibit more varied, complex, and interesting behaviours using GOAP while the code behind the behaviours is more structured, reusable, and maintainable.

## 5.4 Neural network assisting

A novel idea would be the use of neural networks to assist agents. Neural networks are good a pattern recognition and as there are certainly interaction patterns between actors that the agent could learn to better evaluate its situation, which is why it is something worth to be looked into.

Artificial neural networks are generally presented as systems of interconnected "neurons" which exchange messages between each other. The connections have numeric weights that can be tuned based on experience, making neural nets adaptive to inputs and capable of learning. [10]

Initial test in writing a neural network turned out to be successful, but dreadfully slow. Event after increasing the learning rate coefficient the learning took several seconds to group the data from six inputs nodes to three output nodes. Main problems were caused by the backpropagation which really could have used a more memory lookup and iteration efficient language like C++. There is probably a good reason why till now little to no game has used this and why the likes of Google are producing custom “Tensor Processing Unit” units for their AlphaGo machine which is also why this idea was not investigated further in the prototype.[11]

# 6 The prototype

The goal of this work was to explore the most crucial topics of creating a voxel game. For the sake of testing various algorithms of their performance characteristics, a prototype was in order. This prototype served as a voxel game engine where the aforementioned features could be written and explored first hand.

“A game engine is a software framework designed for the creation and development of video games. The core functionality typically provided by a game engine includes a rendering engine, a physics engine or collision detection, sound, scripting, animation, artificial intelligence, networking, streaming, memory management, threading, localization support, scene graph, and may include video support for cinematics.“ [12]

While this engine is not complete or yet user friendly it has managed to address the first core subject of rendering, pathfinding and agent artificial intelligence.

All the testing data was acquired on an outdated ATI Radeon™ HD 5770 Graphics card and a Intel® Core™ i7-4790K Processor at 3.8GHz (underclocked).

## 6.1 Features and performance

One of the main features realized in the prototype was the meshing of the voxel chunks. The testing was done on 8x8x8 voxel chunks in 193 chunk terrain. The terrain generation took an average of 9 seconds for 61660 voxels of the standard terrain seed and size that was used. The voxels were reduced to 5850 polygons which is a ratio of ~0.095 polygons per voxel. The maximum time spent to initiate a chunk was 0.45 seconds. This also includes the Perlin noise generation with rock and tree placement. The meshing itself took an average of 2ms and a maximum of 10ms. This time is important that it describes the response time to the change in environment. What is more, this time can be scheduled into per plane smaller tasks. Meshing of a chunk can be seen in Figure 45. The individual squares seen on the left are a shader trick to illustrate those virtual faces of voxels and not real polygons.

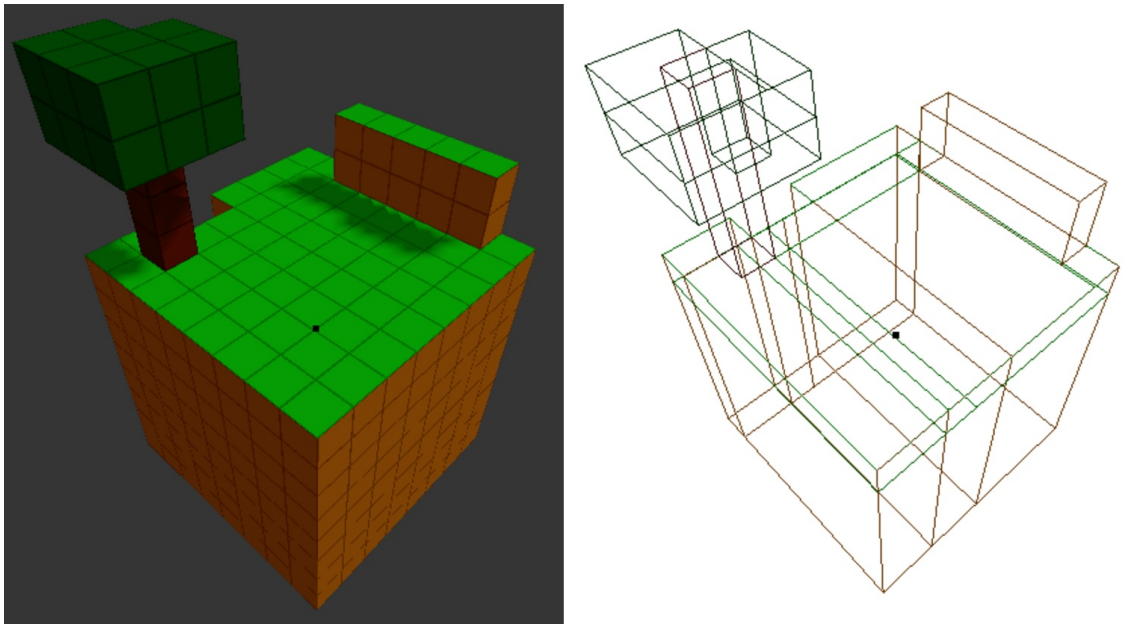

Figure 45: Example of polygon reduction of meshing a chunk

As a side note, as the Blender game engine does not support polygon creation by itself, the polygons are borrowed from various sized donor objects. This means that there is a slight disarray between the voxel polygon count and actual polygon count present in scene, which results in extra strain for the rasterizer.

The user and actors are able to manipulate the voxel terrain varying influence radiuses. The aiming for the user was done with ideas borrowed from Bresenham's line algorithm which result was a list of passed voxels.[13] The last voxel of the list was the solid voxel the ray hit and the hit normal, which is required for relative placement, was found by subtracting the position of the one before last and the last voxel. The user is free to change landscape as shown in Figure 46 with placing and removing up to 1000 voxels at a time while not exceeding 16ms logic spikes. The alterations made by the actors are minor in scale and shown in previous topics.

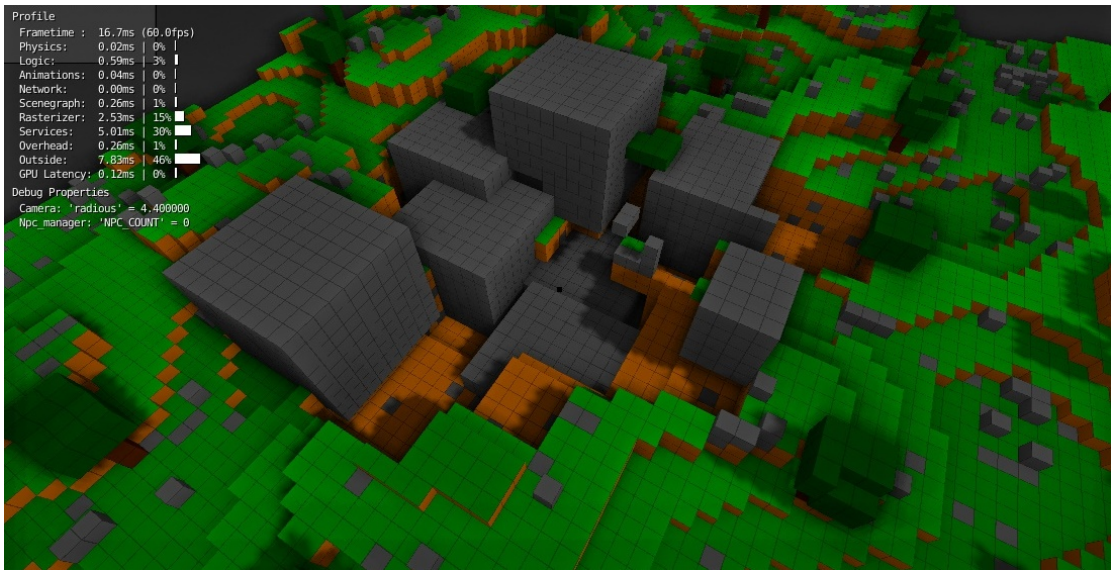

Figure 46: Example of user's terrain manipulation

The prototype used a scheduler to minimize the strain on the time budget of a frame. The shown individual 1000 voxel changes in Figure 46 were spread over multiple chunks and their planes. All responses happened under a second which was acceptable. The actors were handled in a separate scheduler with a time budget of 8ms. This enabled to update over 1000 actors per seconds with an average strain of 5ms. Also a scheduler was used for pathfinding with a 3ms limit per logic tic. All those schedulers were not aware of each other's consumed time which led to frame rate drops when all aforementioned features were used at the same time. This was a concern that due time constraints was not further addressed.

The prototype featured voxel aware pathfinding. The pathfinding implemented was able to find solutions to all presented test cases while being very scheduling friendly. The actors were able to traverse, search and manipulate the environment. The biggest downside was that pathfinding required up to 80000 voxel lookups which took about a second on average. This would normally be acceptable if it weren't for multiple actors. Each actor had to wait in queue for its path which caused obstruction to all actors.

The actors actions were handled by stateless logic and FSM. The implemented logic did not cause any significant processing overhead compared to pathfinding or meshing. While not implemented into the actors code for GOAP and ANN was included. The

implementation of GOAP planning took under 1ms for a 20 step plan. Neural network performance was subpar and was included for proof of concept.

The prototype voxel game engine was able to provide a testing environment for various features while performing mostly to specification. Besides performance the technological debt was also a significant problem. This was mostly caused by lack of time and discipline. This is a concern that needs to be addressed in the future.

# 7 Summary

The aim of this work was to explore everything concerning the creation of a voxel game engine. The result was a prototype environment where the algorithms could be evaluated and demonstrated. The main topics covered were voxel graphics, voxel conscious pathfinding and actor decision making systems.

First main goal addressed was the data and visual representation of voxels. Various implementations were discussed and new solutions regarding voxel meshing was developed and tested. The result was a responsive voxel generation and manipulation system.

The second main goal was to develop a voxel aware pathfinder. The found solution was not perfect, but was able to create paths that required alteration to the world state while doing so with acceptable levels of system load.

The last goal was to create actors that could use the terrain and pathing in an intelligent manner. The proposed solutions delivered acceptable performance and expected capability. The main issue was that not all solutions were not fully implemented into the agent logic system.

The discussed algorithms were implemented into the prototype voxel game engine and put into perspective. While the engine performed mostly to the specification the main issue was the development time limit and the technological dept it caused. Overall the prototype voxel engine was a good first step towards being able to create a unique experience for the end users.

# References


[1] Intel core i7. [WWW] http://ark.intel.com/products/80807/Intel-Core-i7-4790K-Processor-8M-Cache-up-to-4_40-GHz (16.05.2016)

[2] AMD HD5770. [WWW] http://www.amd.com/en-us/products/graphics/desktop/5000/5770 (16.05.2016)

[3] An Analysis of Minecraft-like Engines. Mikola Lysenko [WWW] https://0fps.net/2012/01/14/an-analysis-of-minecraft-like-engines/ (16.05.2016)

[4] Karnaugh map. [WWW] https://en.wikipedia.org/wiki/Karnaugh_map (16.05.2016)

[5] Meshing in a Minecraft Game. Mikola Lysenko [WWW] https://0fps.net/2012/06/30/meshing-in-a-minecraft-game/ (16.05.2016)

[6] Perlin noise. [WWW] https://en.wikipedia.org/wiki/Perlin_noise (16.05.2016)

[7] Introduction to A*. [WWW] http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html (16.05.2016)

[8] Intelligent agent. [WWW] https://en.wikipedia.org/wiki/Intelligent_agent (16.05.2016)

[9] Applying Goal-Oriented Action Planning to Games. [WWW] http://alumni.media.mit.edu/~jorkin/GOAP_draft_AIWisdom2_2003.pdf (16.05.2016)

[10] Artificial neural network. [WWW] https://en.wikipedia.org/wiki/Artificial_neural_network (16.05.2016)

[11] Google reveals the mysterious custom hardware that powers AlphaGo. [WWW] http://www.theverge.com/circuitbreaker/2016/5/19/11716818/google-alphago-hardware-asic-chip-tensor-processor-unit-machine-learning (16.05.2016)

[12] Game engine. [WWW] https://en.wikipedia.org/wiki/Game_engine (16.05.2016)

[13] Bresenham's line algorithm. [WWW] https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm (16.05.2016)