Benjamin Dickson

Registration number 100104882

2018

Realistic Urban Modelling

Supervised by Prof Andy Day

University of East Anglia
Faculty of Science
School of Computing Sciences

Abstract

The aim of this project is to take a basic 3D model, that has been produced without the imperfections of materials in the real world, and process them so they look weathered, worn, and aged. Its aims can be summed up concisely in the project brief, provided by Prof Andy Day:

> Investigate and implement existing and new techniques to make virtual urban environments (in particular medieval) more realistic in terms of the effects of age, weathering, damage etc. Apply these techniques to example streets and buildings in Norwich. The project will be associated with our rapid urban modeling work and the student will be involved with existing projects (eg for Norwich HEART, Norman castles).

Acknowledgements

In addition to my supervisor, Prof Andy Day, who provided me with invaluable support and mentoring during this challenging project, I would also like to thank David Drinkwater both for providing me with the original models of Norwich Castle that I used for the basis of most of work, as well as aiding me with his expertise in the use of 3D Studio Max.

Contents

List of Figures

1 The Context

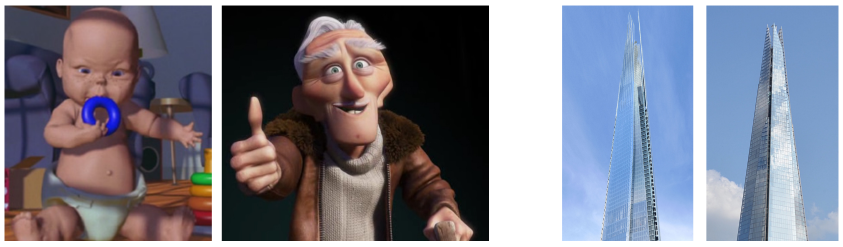
Be it through blockbuster movies, “Triple-A” video games or architectural/historical renderings, photorealistic 3D generated imagery is something that most of us encounter in our daily lives. It’s something that when done correctly is invisible, but when done poorly is impossible to miss(McNamara et al., 2000).

Also, as technologies improve, audience expectations shift with it - a lot of computer generated special effects age quickly - what was once seen as incredible can quickly look dated and stand out. However, with this increasing complexity comes an increasing cost. Computer games have seen their costs increase four fold over the last 10 years, and a large part of this is down to the additional costs for producing the higher quality graphics(Portnow, 2010; Schreier, 2017).

1.1 The “Uncanny Valley”

One of the biggest challenges with constructing 3D models is making them look natural to the human eye. This is consequence of many factors, but two of the most significant reasons are down to the fact that computers will perfectly execute instructions with no imperfections, and the human intuition to pick up on patterns in simulations of natural entities - more commonly known as the “Uncanny Valley.”

Figure 1: **Left:** Pixar originally tried realistic models before switching to stylised models. **Right:** The render of the shard (left) features uniformity in its elements and shading compared to the photograph (right)



The Uncanny Valley refers to the strong human rejection of visual facsimiles that look close to human (Mori, 1970), but are not quite a perfect replica. While this strong rejection is not a feature of inanimate objects, humans are still subconsciously able to pick up on the visual clues that something is artificial, particularly in computer graphics where optimisations often result in repeating patterns, or randomisation algorithms produce patterns inconsistent with our expectations from the real world.

1.2 Resources

As mentioned, high graphical fidelity is resulting in increasing costs for studios due to the increased number of work hours demanded of their animators. The reason for this is that creating realistic objects is time consuming. Consequently, the industry is always looking for techniques that can allow artists to focus on what is important rather than waste time animating individual strands of hair (Haran and DeRose, 2015b).

The Rapid Urban Modelling team aims to recreate a lot of Norwich's most significant buildings in 3D, ideally as lifelike as possible. Giving them tools that enable the production of higher quality models by allowing software to automate parts of the process would both help improve the speed and quality of their work, allowing them to produce more models at a faster rate.

1.3 Background Reading

In my background reading, I discovered there has been a lot of research done in the area of procedural generating the graphics for terrain, as this is very useful for computer games with large maps, however there has been far less research done in the procedurally ageing buildings.

There is an overlap, as procedurally generated terrain is essentially a surface (containing mountains, rivers, hills, fields etc) and ageing buildings requires the evolution of the surfaces of the model, to make them look older with cracks, dents etc... It is my intention to take some of the ideas from the terrain systems and which elements can be adapted for Urban Modelling.

2 My Approach

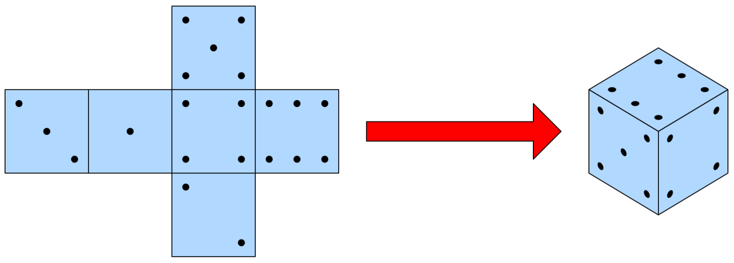
As it was a requirement that my project built on the work done by the Rapid Urban Modelling team, in particular the models of Norwich Castle created by David Drinkwater, it meant that I would be dealing with mesh based 3D models, created from 2D polygons positioned in 3D space - as opposed to alternative 3D modelling techniques such as voxels or point clouds.

This type of 3D modelling is the bedrock of 3D Graphics, having been the method used by Pixar for movies since the company's inception, and becoming mainstream in the computer games industry with the advent of games like Quake in the late 1990s.

The reason for this is that it is possible to create a complex 3D structure that is easy to animate with a lot less data than is required for alternative solutions.

3D dimensional objects are created in exactly the same way that 3D models can be constructed from a net cut from a piece of paper - flat polygons are arranged in 3D space so they wrap around, creating a representation of the desired 3D shape.

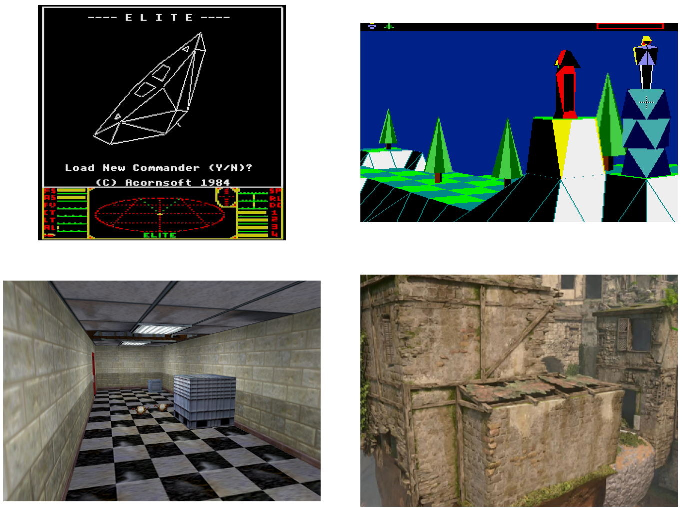
Figure 2: An example of a dice net transformed into a 3D shape. The flat net represents the 2D textures and the cube the texture mapped to the mesh



These polygons can be decorated with a texture - a 2D image of the surface - and other attributes such as shaders and bump maps can give the illusion of the surface being curved, rough or reflective.

The history of 3D computer games show an excellent rollout of these technologies as computer hardware become more powerful and able to render them in real time:

Figure 3: Top Left: Wireframe Graphics, Elite 1984 ; Top Right: Coloured Polygons, Sentinal 1986 ; Bottom Left, Textured Polygons: Half-Life 1998 ; Bump Mapping, Shaders, Uncharted 4, 2016



2.1 Previous Work On This Project

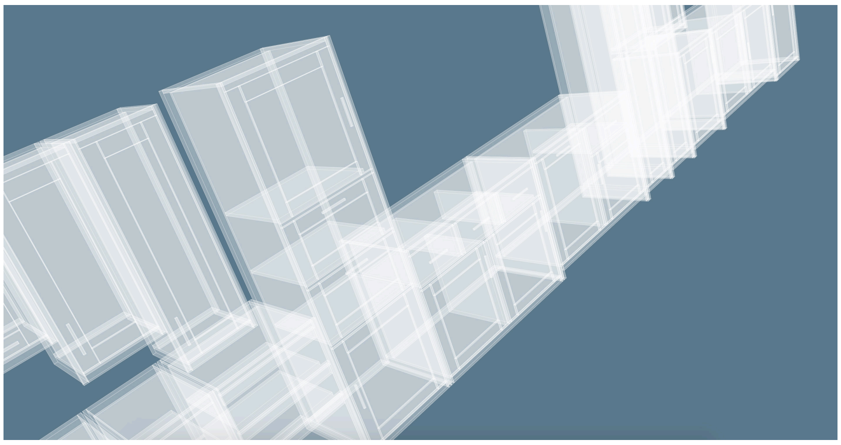
As this was a project previously undertaken by students, it gave me the opportunity to look at their approach. There were also a number of similar projects, such as “Reconstruction Of Historic Farm and Changes Over Time” , “Image-based and Geometry-based 3D Historic Urban Modelling: Magdalen Street (Norwich)” and “Historical Modelling of Dunwich Under The Sea”

However, as the project was open to both Computer Science students and Computer Graphics students, the previous students had all pursued the project from a graphics implementation point of view. While handcrafting 3D models has always been an interest of mine, having made maps of my childhood home in the Quake 3 level editor nearly 20 years ago, I was more excited and interested in tackling the project from a programming point of view.

I also had previously worked on programming code to procedurally generate 3D geometry when I worked on a 3D Kitchen Designer for a web page that created bespoke sized cabinets based on a set of rules.

I therefore decided that rather than take a 3D model and manual age it using tools

Figure 4: A screenshot of the kitchen designer I created



such as 3D Studio Max - I would develop software for automating the ageing process algorithmically.

2.2 A Geometric Solution

I decided from the outset that I would look at modifying the geometry, ie the model's mesh, rather than looking at the textures, shaders or bump maps. There were several good reasons for this:

- The mesh is the foundation of any 3D model, if this part isn't right then the rest won't be.
- Unlike textures, or bump maps, geometry requires no source material, such as large libraries of photographs.
- The mesh is the most complex part of the editing process for the artist to edit, due to the fact that it exists in 3D. Textures and bump maps are 2D elements that can therefore be more easily adapted by artists.
- Conversely, from a computer science point of view, the mesh is easier to intepret algorithmically - as textures are essentially 2 dimensional photographs, so simply recognising them would require advanced Image Classifiers, let alone adapting

them. Latest papers in the field Computer Vision have seen neural networks able to generate and interpolate between related photographs, however this field of research is very tangental to the area of 3D Graphics.

3 The Real World

Given that the aim of this project is to produce realistic 3D renders of a structure, and this means there is a second aspect to this project - in addition to the computer science - and that is investigating what factors in the real world affect the ageing process and how best to model these when procedurally generating the ageing and weathering effects.

As this is a computer science project, and one focusing on visual aesthetics rather than accurately simulating the environmental factors, it is not my intention to build a real world simulator down to the subatomic level - however considering the broad strokes, such as the causes of surface erosion and producing code that produces an accurate looking facsimile of that will require at least some consideration of environmental physics.

3.1 The Materials

One important property of polygon based 3D models vs alternatives such as voxels, is that they are essentially shells that only describe the surface of the shape they are representing. Given that real world materials do, and this affects key properties such as strength and rigidity, the system needs to model the internal structure at some level, in order to accurately simulate the environmental changes.

In the real world, this occurs at the atomic level, but clearly - without drastic advances in computational speed and memory storage - this is unfeasible in a simulation. As previously mentioned, voxels do contain volumetric data of structures, however these are not suitable for this project due to the fact that they produce quite blocky structures and take up large amounts of memory for large structures.

The computer game Minecraft, hints at one possible compromise. Minecraft simulates a world where everything is comprised of cubic metre elements.

Figure 5: Minecraft simulates the world by creating everything from 1 metre cubes.

Minecraft was able to simulate a very large complex world using these elements, with each element having unique individual properties. The elements could also interact with each other in complex ways - to the extent the system is Turing complete ('legomasta99', 2016) and people have constructed computers that run within the game world.

While graphically Minecraft is at the opposite end of the realism aesthetic, the game's mechanics of using macro sized elements to create unique, objects with material properties was a key influencer in helping me determine a solution to the problem.

As Norwich Castle is made from stone, my primary goal will be to try and recreate the aesthetics of weathered stone, however I felt it prudent to ensure the system would be able to model other common materials seen in buildings, such as:

- Bricks and Mortar
- Wood
- Flint/Stone and Mortar Walls
- Structural Metal
- Sheet Metal / Metal Cladding
- Tiles

- Glass

3.2 The Environment

The other important element in the equation is the environment that the building is located in. Clearly a castle at the North Pole will endure different conditions to one in a desert on the equator, so it was important to look at the factors that would affect a building over time.

I considered a few of the main contenders:

- Weather
  - Precipitation (Rain / Hail / Snow etc)
  - Wind
  - Sun
- Damp
- Major Disasters
  - Earthquakes
  - Flooding
  - Fire
- Human Interference
  - Grafitti
  - Modification
  - Accidental Damage
  - Wear and tear
  - Damaged in Armed Conflict
- Plant Growth (Grass / Moss / Weeds)
- Animal Damage (Rodents / Colonies)

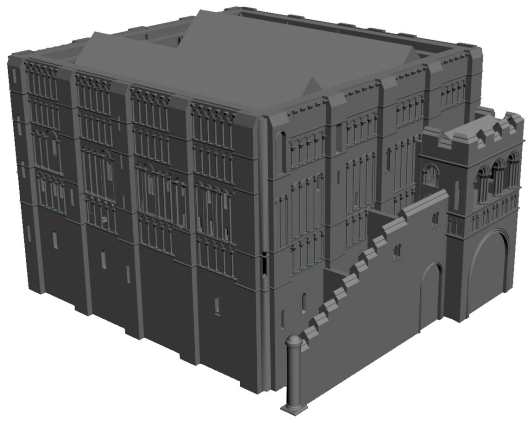
Of these they could broadly be divided into three categories: Rare, events that caused substantial and unpredictable changes - such as natural disasters; intermittent events, that caused localised changes - such at Human Interference; and finally, the continuous changes, such as the weathering.

Given that my focus was on procedurally generated modification, and that all buildings are affected by the continuous changes, it made sense to concentrate on these types of environmental factors. As Norwich Castle is a large building situated on top of a hill, I decided to set my goal to be simulating the effects of weathering - although I would also design it with the idea it could be used to simulating other predictable changes as well.

3.3 Norwich Castle

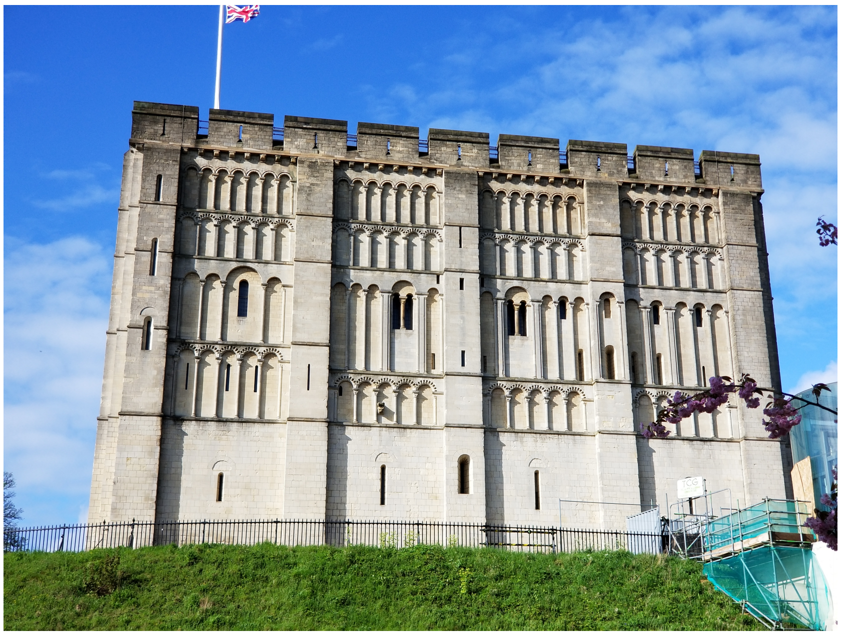
I was fortunate to be provided with two highly detailed models of Norwich Castle that had been created by David Drinkwater - one was a model of the castle as a whole, while the other was a very highly detailed brick by brick reconstruction of the south wall, comprising of over 17,000 elements.

Figure 6: The model of Norwich Castle



Although, as a Norwich resident, I was very familiar with the castle, I repeatedly visited it throughout the course of the project to take reference photographs, look at details of the structure and so its physical presence was always fresh in my mind when evaluating the results.

Figure 7: A photograph I took showing the south wall of Norwich Castle as it appears now



Over the course of the project I would also take photographs of examples of materials erosion I saw that could be relevant to my project, for example to concrete paths, or brick walls. Building up a library of data was useful to look at when configuring the final models. I have included some of these reference photographs at the end of this paper.

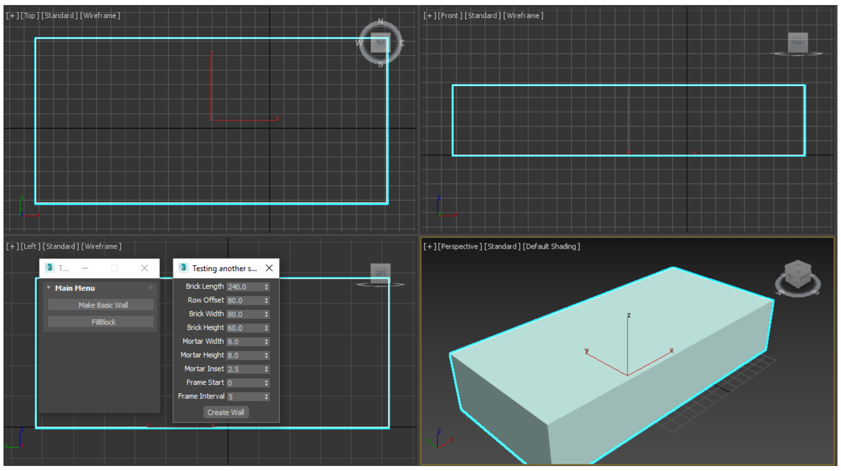
4 Overview of the Development History

When I first started the project, my original intention was to build a plugin for 3D Studio Max that an artist could use to age their model. However, as I became more familiar with the strengths and weakness of 3D Studio Max this evolved into a standalone piece of Python code for processing OBJ files.

4.1 MAXScript

When David Drinkwater first introduced me to the problem, he showed me tools that he had created with MAXScript and I immediately saw their potential for procedural generation. With their native integration to the software, ability to make graphical user interfaces for the artist to use, it seemed like the logical choice to use and I immediately tasked myself with getting to grips with it.

Figure 8: Example of a MAXScript tool I made for creating a building animation of bricks.


[+] [Top] [Standard] [Wireframe]
[+] [Front] [Standard] [Wireframe]
[+] [Left] [Standard] [Wireframe]
[+] [Perspective] [Standard] [Default Shading]
T...
Main Menu
Make Basic Wall
FillBlock
Testing another s...
Create Wall

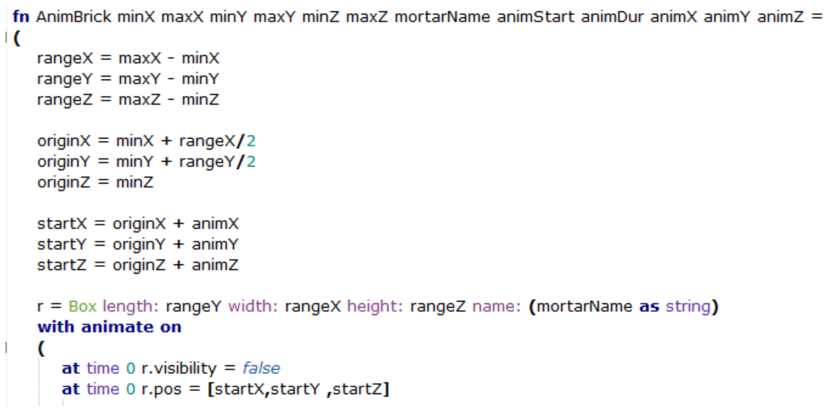
Brick Length 240.0
Row Offset 80.0
Brick Width 80.0
Brick Height 60.0
Mortar Width 8.0
Mortar Height 8.0
Mortar Inset 2.5
Frame Start 0
Frame Interval 5

The first thing I created was a tool for creating procedurally generated animations of bricks being laid on a building. The user could create a cuboid in 3D Studio Max, select it, input the brick dimensions and the plug in would create all the bricks required for the animation and all of the animation keyframes.

However, this mini project revealed two major problems with MAXScript - the first was that the code itself was not very readable, nor suitable for any level of complexity, and the second was that it was very slow if tasked with anything with a high number of iterations - and in models containing potentially thousands of elements, the code was going to require a very high number of iterations.

My conclusion was that while MAXScript was ideal for crafting simple tools for automating the production process, developing a realistic weathering model would be difficult given the limitations of the language and the readability of the code.

Figure 9: Example of a MAXScript function I wrote for the prototype, the arguments appear as a list after the function name, making them difficult to distinguish compared to traditional languages



```
fn AnimBrick minX maxX minY maxY minZ maxZ mortarName animStart animDur animX animY animZ =
(
    rangeX = maxX - minX
    rangeY = maxY - minY
    rangeZ = maxZ - minZ

    originX = minX + rangeX/2
    originY = minY + rangeY/2
    originZ = minZ

    startX = originX + animX
    startY = originY + animY
    startZ = originZ + animZ

    r = Box length: rangeY width: rangeX height: rangeZ name: (mortarName as string)
    with animate on
    (
        at time 0 r.visibility = false
        at time 0 r.pos = [startX,startY ,startZ]
```

4.2 Python in 3D Studio Max

Like many other advanced computer tools, 3D Studio Max has been updated to run plugins and macros written in Python. As I had experience coding in Python and it was a full object oriented language, capable of handling custom data structures, it meant it would be much easier for developing and implementing more advanced procedural algorithms(Grant, 2013), as well as being easier to develop in.

The MAXScript code library had been implemented as a Python library to interface with the models. I experimented by writing Python scripts to generate meshes from scratch, generating simple shapes such as cubes and spheres. Once I had done this, I was ready to start modifying existing meshes.

To achieve this, I wrote code to read the 3D Studio Max model and put it into a data structure that I designed that would make it easier for me to process. The code would then modify the model, and once it was complete export it back as a 3D Studio Max shape.

After completing this, I was able to focus exclusively on the code for procedurally modifying the structure of the model. And I started implementing a system for increasing the complexity of the surface of a mesh by sub dividing the polygons and applying convolutional smoothing. The system worked perfectly, and I was able to transform a very simple polyhedron made of 36 triangles to a smoothed version made of over 4,000.

4.3 Running Python Externally

The problem I encountered once the shapes reached triangle counts in the thousands is that the code started to run very slowly. I realised that as Python was running through an interpreter within 3D Studio Max it was running much more slowly that if it was running natively on the machine.

However, as all of the complex code was running entirely in my own data structures, it had no need to be confined to the 3D Studio Max environment. In fact, the only code dependent on the 3D Studio Max environment was the part that extracted the data from the 3D Model and then converted it back, so split my code into two - I had the script running in 3D Studio Max extract the data and save it in a text file. The rest of my code, running natively in Python, would read this file, process it much faster, and then save an updated text file that could be imported back into 3D Studio Max using the script.

This worked exactly as expected, and my native scripts ran complicated operations in a few seconds, compared to a few minutes in 3D Studio Max.

I later optimised this process even further, as the OBJ file format - a standard file format in 3D Graphics - is text based, and by updating my Python code to read and write OBJ files, I could just import and export OBJs through the standard 3D Studio Max GUI, which was much faster than importing and exporting via a script.

4.4 Distributing the problem

Due to the fact that my aim was to produce highly detailed, photo-realistic models, this meant that components that previously had very low poly-counts would become several orders of magnitude more complex - a brick that was once 12 triangles could become several hundred or even a thousand.

As the model was already significantly complex when comprised of these simple shapes, scaling it up by this order on a single machine would be unfeasible. However, as I had pipelined my system, this meant that I could break the problem down into components and distribute the work over several machines.

I used 3D Studio Max to export the shape with each component as a single OBJ - for the Norwich Castle South Wall this comprised of over 17,000 individual OBJ files. I then configured a web server to check out the unmodified OBJs and check in that OBJ after it had been processed. My original code was then set to connect to the server, download the next OBJ in the queue, process it, upload it back to the server and repeat.

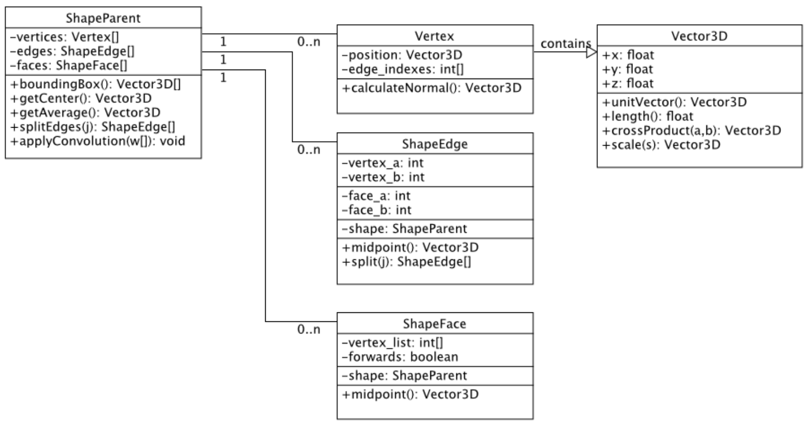
I ran my Python code on four PCs (and also on my phone!) and the set up was able to process the entire Norwich Castle South Wall model in less than 24 hours.

5 The System

The system I implemented consisted of three fundamental parts:

The first part was a data structure for handling the mesh data, in a format that would allow all of my modification operations to be easily and efficiently performed upon.

Figure 10: A UML class diagram, outlining the main components of my shape data structure



The second part I developed was a system for subdividing faces and smoothing a mesh using convolutional smoothing. This was important because as the shapes became more complex, the meshes would need more detail to show these changes.

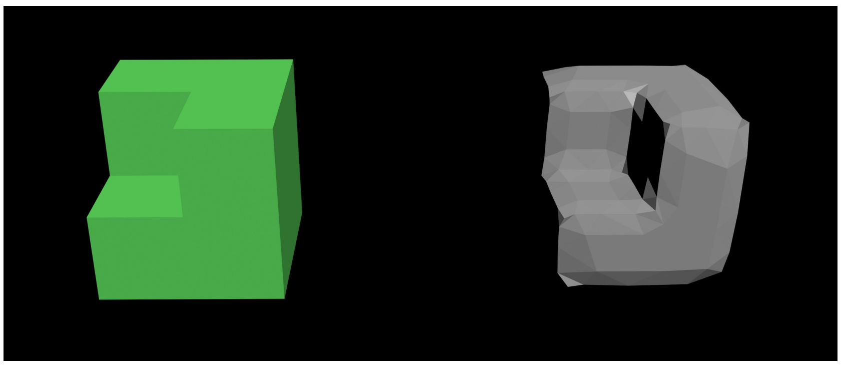
The final part was a material and environment model, used to inform the updates needed to the model.

5.1 Subdivision and Convolutional Smoothing

For my smoothing algorithm, I implemented my own system based on a method developed by Pixar (Haran and DeRose, 2015a; DeRose et al., 1998). The system works by subdividing the faces and edges, and then using weighted averages of a point and its adjacent nodes.

This consisted of two algorithms, one for subdividing the edges, and another for applying the convolutional weighted point averaging. The edge splitting algorithm essentially added a mid point on every edge, and then connected it to the adjacent mid point along each face, creating new edges and faces in the process, while the convolution averaging is explained below in Algorithm 1.

Figure 11: A simple shape (left) with two different examples of the shape after edge splitting and convolutional operations applied.



By feeding in an array of weights, as doubles, the new vertex position comes a sum of weighted averages, with each weight corresponding to the average position of nodes n points out, where n is the position in the array. This has the effect of ’pulling’ a point towards its neighbours, which results in a smoother surface.

I found that good smoothing weights to use were to bias original node with a large weight and bias the weights accordingly. It was also important to make sure that the mesh was at a high polygon count before applying the smoothing as for low complexity polyhedrons - such as a primitvie cube, all points in the cube are within 3 nodes of each other, meaning every vertex would be affecting each other.

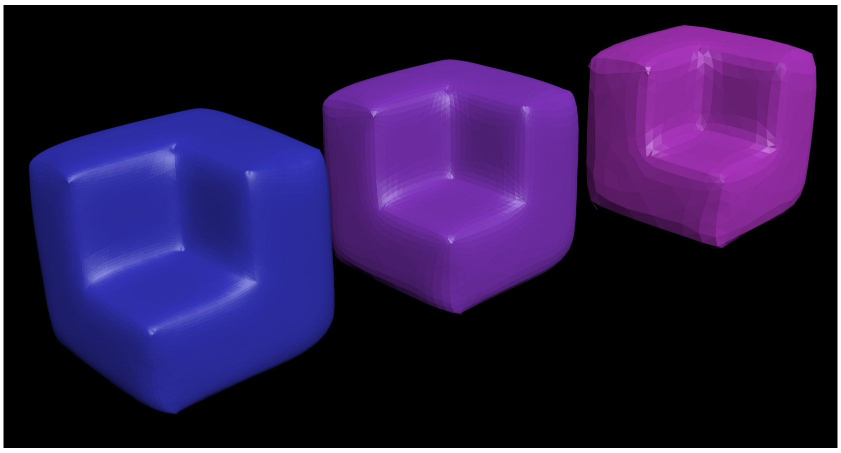
I typically used the weights array of [3.0, 2.0, 1.0] when doing my smoothing oper-

Algorithm 1 Vertex Convolution

```
1: function ApplyConv($v[], w[]$)                      ▷ Array of Vertices / Array of Weights
2:     $r\_v \leftarrow$ VertexArray()                   ▷ initialise return vertex array
3:     for $i$ in length($v$) do                       ▷ loop through every vertex
4:         $r\_v[i] \leftarrow$ Vertex(0,0,0)             ▷ Initialise return vertex as zero
5:         $totalWeight \leftarrow 0$                     ▷ For tracking the total weight
6:         $checked \leftarrow VertexList()$              ▷ for logging vertices already checked
7:         $spider \leftarrow VertexList(v[i])$           ▷ List of just this vertex
8:         for $i\_w$ in $w$ do                        ▷ loop through every weight
9:             $convWeight \leftarrow$ Average($spider$) * $i\_w$   ▷ Get average vertex position
10:            $r\_v[i] \leftarrow r\_v[i] + convWeight$    ▷ multiply by weight
11:            $totalWeight \leftarrow totalWeight + i\_w$  ▷ update totalWeight
12:            $nextSpider \leftarrow VertexList()$        ▷ Empty VertexList for next spider
13:            for $i\_s$ in length($spider$) do         ▷ update spider
14:                $checked.append(spider[i\_s])$
15:                $nextSpider.append(spider[i\_s]$.neighbours not in $checked)$
16:            $spider \leftarrow nextSpider$              ▷ reassign spider
17:        if $totalWeight > 0$ then
18:            $r\_v[i] \leftarrow r\_v[i] / totalWeight$   ▷ scale down by weights
19:    return $r\_v$
```

ations. The process of edge splitting and applying the convolution could be repeated until the resulting surface had so much detail the mesh itself looks smooth. When I applied the same smoothing operation recursively to my test shape it produced very smooth looking results. What started out as a 24 polygon, simplistic polyhedron was transformed in a a 4,000+ polygon that appeared perfectly smooth. This smoothness, though, was an illusion, as the shape actually comprised of very small, but perfectly flat surfaces.

Figure 12: The smoothing operation, repeated recursively on the same shape (right, least smoothed; left, most smoothed)

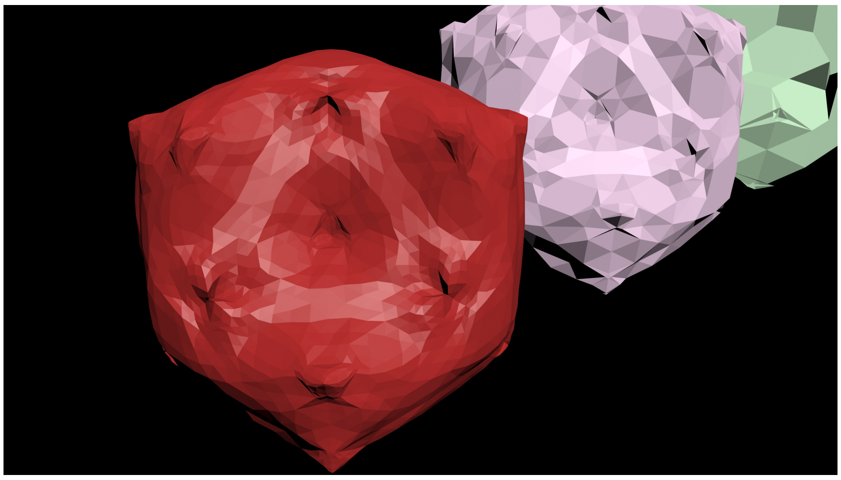


While I ultimately used it only for smoothing, the convolution operation could get used for other mesh modifying behaviour, such as adding complexity, shrinking the mesh. I did experiment with other values, producing interesting effects that gave irregular, organic-like surfaces:

When I saw these results, I did consider using the convolution averaging system for developing an erosion system, however I quickly decided against it for a few key reasons:

- While this particular result looked great, it was a result of trial and error - I wanted to develop a system where the user could predict the results.
- The changes are only a consequence of the mesh properties, and a mesh only describes the surface of a shape. Erosion and weathering is dependent on the

Figure 13: Convolution weights that didn't following the featuring parameters, such as [1.0,0,2.0] produced interesting results, such as this hooked-like deformed points.



internal construction as well as its surface.

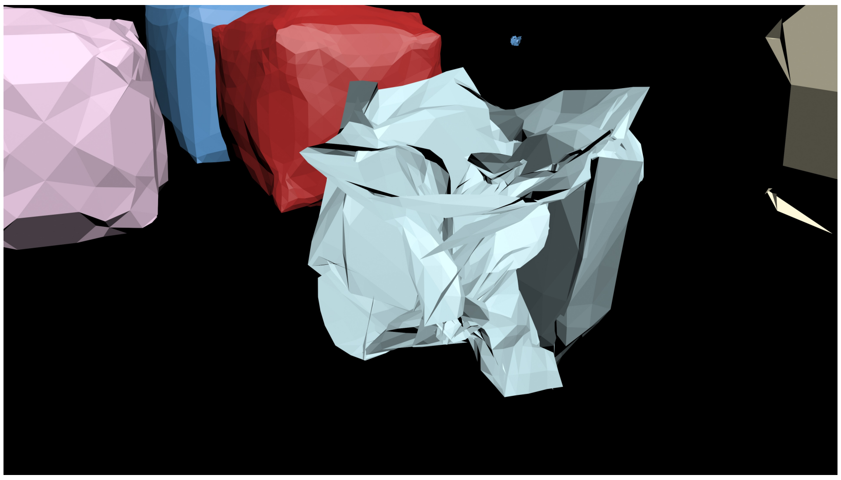
- As it was not part of my original plan to use it for this purpose, I had not done any background reading or thinking about how best to implement it.

The point about trial and error was the most significant. When developing a weathering system it was important to be able to deterministically influence the desired result. For example, if eroding stone the mesh should move towards the centre of the shape, not grow outwards. While I knew that using feathered weight values would produce smooth surfaces as I had based it on the system used by Pixar, I would need to do further reading or research the mathematics of the manipulation myself, in order to determine why particular patterns of values produced certain results.

For example, when I tested the system with negative values, to see what happened, the shape became a complete mess (which made sense, as a negative value effectively turns a vector inside out)

Having said that, this is an area that could have further research and help future development of this project. With a firm understanding of the principles of the mathematics, it would be possible to find patterns of weights that produce specific types of results beyond just smoothing. It could also be possible to inform the type of convolution applied,

Figure 14: Trying some of the weights with negative values turned the shape inside out...



based on the specific erosion or material present at the location of the mesh, rather than just apply a uniform convolution across the entire mesh.

Such a system would be very similar to fractal procedures, which have proven effective for generating realistic landscapes (Musgrave et al., 1989; van Pabst and Jense, 1996), although they have drawbacks (P., 1990), and the refinement of the systems requires a high level of mathematical systems. As I wanted to produce something that could be tuned more intuitively manipulated by a 3D Graphics artist, it was my preference for a system without this barrier to entry.

5.2 Material and Ageing System

The final stage in the development process was to design and implement the weathering an ageing system. This would require modelling the materials and the environment.

5.2.1 Material System - Concept

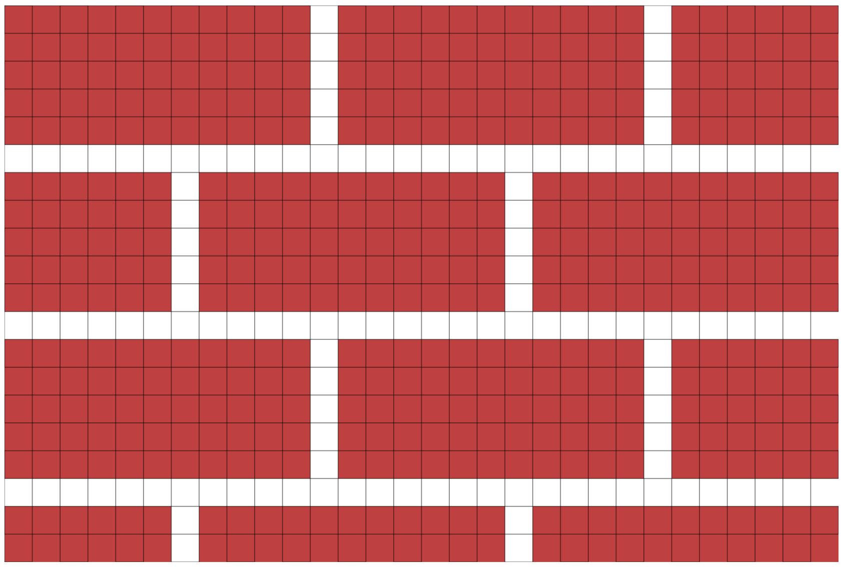
The basic idea of my system was to partition the volume within a mesh into cells, each cell would then be assigned material properties. All of the vertexes contained within a cell would inherit those properties and it would affect how they were eroded. A simplistic example could be that if vertexes were within a stone cell then they would erode less than if it was soft mortar.

However, as each cell is unique, they can be configured to have some random variance in their initial properties. For example, when created, each stone cell in a rock could be set to have a +/- 2.5% value on its strength properties. Using a gaussian distribution to calculate these random variances would mean that as a whole - stone would behave predictably (a brick of 100 cells would have the same overall properties to another brick of a 100 cells) but close up the detail of the erosion would be subtly different, and it is these subtle differences that will make the structure appear more realistic when glanced at by a human observer and help eliminate some of the Uncanny Valley effects.

The cells making up a mesh's volume need not all be of the same source material, although in most cases this would probably be the case. For example, if modelling a structure brick by brick, it would make sense to make the bricks sourced from one type of material and the mortar elements from another.

However, the system would allow for less detailed modelling - for example, a wall could be modelled from a single cuboid (it would need to be a highly detailed mesh) and then position clay and mortar cells in a brick work pattern:

Figure 15: A material could be configured to have brick work pattern and each section would erode according to the material's properties.



The material properties would be programmed using a class system. There will be a basic implementation of the class, with generic properties would apply to all materials, however for more complex materials this class could be extended and the extended class used instead.

The basic properties I decided to use for each material were: Strength, Hardness and Health

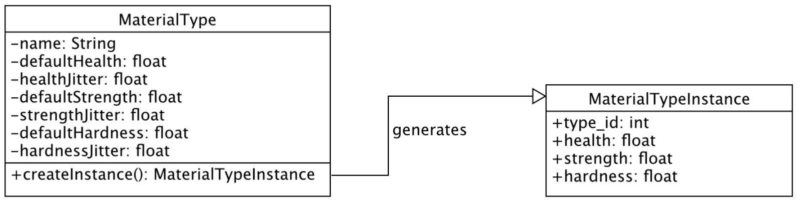
While it is ultimately up to the user to configure how these properties would affect the erosion, the standard implementation would be as follows:

Strength is a measure of how much the material resists the erosion. From a graphical sense, erosion manifests in the vertices of a mesh being pushed towards the centre of the shape, so the strength is essentially how much the vertex would push back against that.

Hardness is a measure of how resistant a surface is to flexing, in the basic model it would essentially be a threshold the erosive force has to exceed before it will start pushing the vertex back.,

Health, in the basic implementation, would be a dynamic variable that would reduce as the material is eroded. This can then be used as a multiplier to increase or decrease the strength and hardness. Therefore, as a material starts to erode it can be modelled to erode faster as it becomes more damaged.

Figure 16: MaterialType and MaterialTypeInstance Class UML Diagram

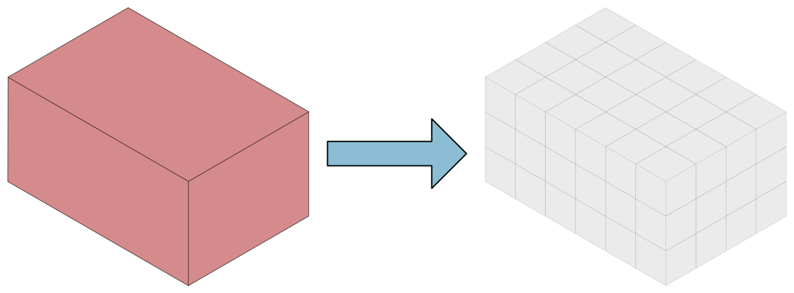


With this system in place, the next stage was to partition the volume of a mesh into cells, to assign the MaterialTypeInstances to.

5.2.2 Material System Space Divison - First Approach: Block Based Space

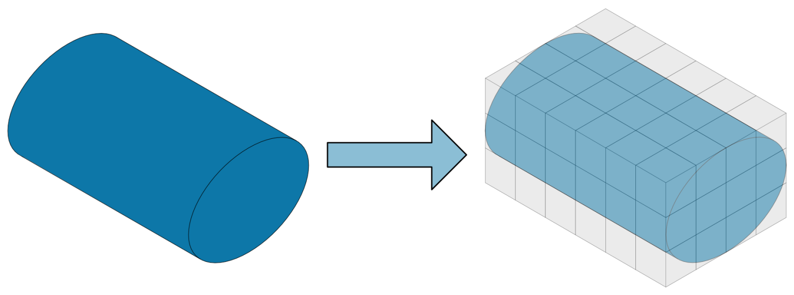
Inspired by Minecraft's block based environment, I decided this would be a good framework for modelling the material. I could partition the 3D space of an object into small cube blocks, assign each region a property and the vertexes that fell within a particular block would inherit the properties of that block.

Figure 17: A cuboid maps to a block map very easily.



However, unlike Minecraft where all blocks of one type are equal, I would use gaussian distributions to vary the parameters of the block. This meant that by coming up with properties for a particularly material - say stone - and setting a standard deviation, while a large block may be made up entirely of ’stone’ elements, they would have slight variations to each other, thus producing subtle differences in the wearing profile brick to brick.

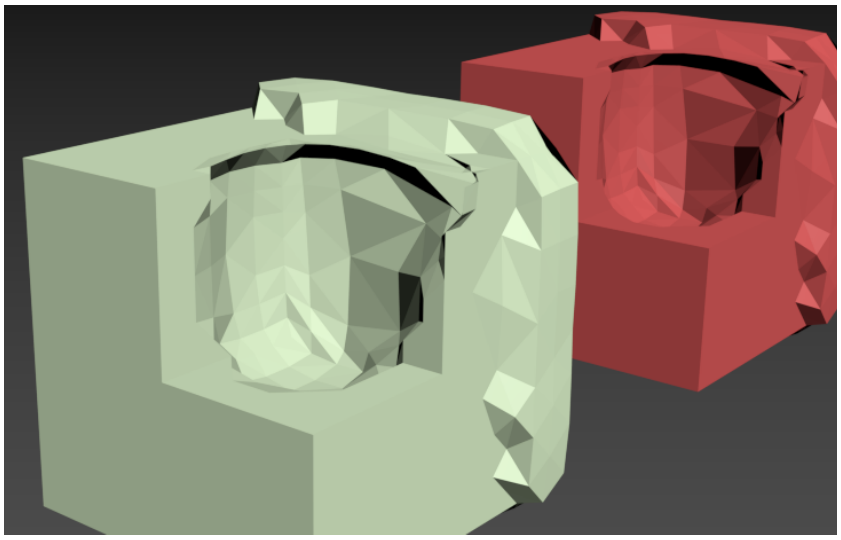
Figure 18: Non cuboid shapes will have an overfitted block map created, and then the vertices are assigned to the block they are contained within



This was important, since many buildings are made up of uniform elements, such as identically sized bricks made of the same material. However, in the real world, no two bricks will wear the same, yet if in a computer model two bricks had identical parameters then they would wear exactly the same, and these sorts of patterns would be obvious to a human observer.

This system seemed perfect, and my early attempts modelling with it saw me able to put in values, with a prediction for the approximate result and then see a result in line with my predictions.

Figure 19: Early implementation of the erosion model, showing the core hollowed out and the back face deformed.

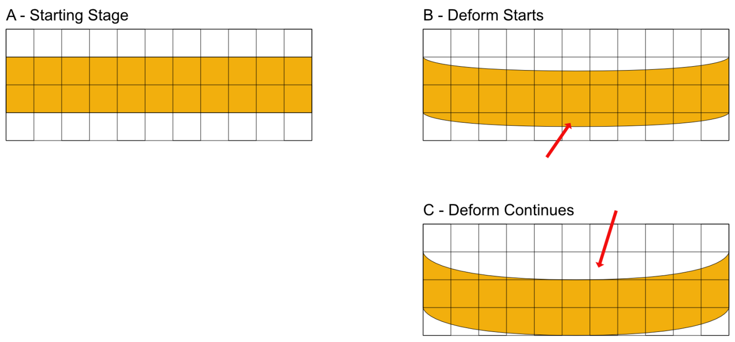


However, when I started to consider materials other than stone, I realised there was a drawback that I hadn't considered. Due to the fact stone is a hard, rigid material, it doesn't warp or deform. This means that other than being eroded inwards, it is static.

Most other common materials do not behave like this, something like wood or metal will bend, sometimes quite dramatically, and the consequence of this is that the internal structure of the material will move rather than be eroded away. In the block based model, the material internal structure exists independently of the mesh skin - the mesh simply moves through its space. Also, because the space is calculated as blocks in a 3D array, it is not possible to deform them - just like a pixel in a pixelated image, its position is discrete.

What was needed was a data structure for the material that could co-exist with the mesh, and deform and move with it if necessary. However, given the encouraging results

Figure 20: As this shape deforms, the vertexes on the bottom move into a different region, meaning their associated properties would change. As the shape continues to deform, some voxels that were previously occupied will become complete vacant, despite their original vertexes still existing.

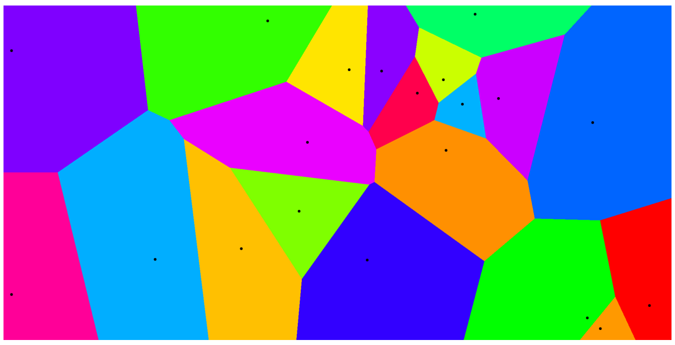


of the block based system I ideally wanted a solution that kept all of the advantages of this, and could replicate its results.

5.2.3 Material System - Second Approach: Voronoi Cells

The solution was quite straightforwards, and was arguably more computationally efficient. The solution was to use the principles from Voronoi diagrams.

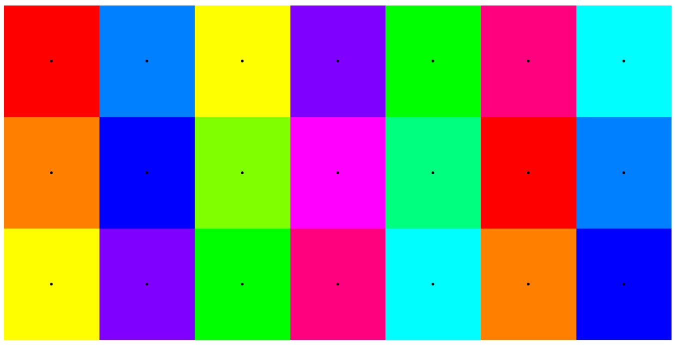
Figure 21: A Voronoi Diagram in 2 dimensions



A Voronoi diagram partitions space into areas (called Thiessen polygons) which are closest to specific points. While they are typically used in 2 dimensions, the principles work in any number of dimensions, dividing 3 dimensional space into Thiessen polyhedra.

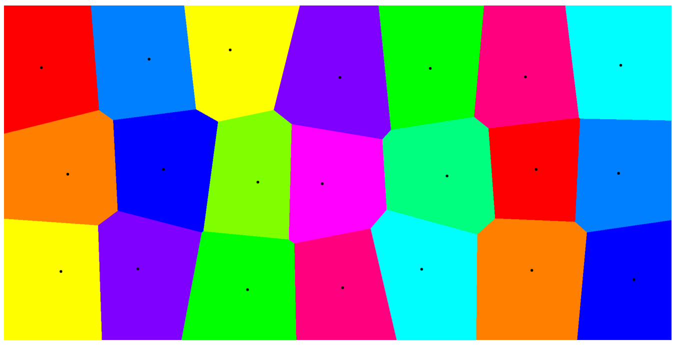
This can create very organic looking structures when the points are randomly distributed, however a Voronoi can also recreate the block based model simply by positioning them at equal distance in each axis:

Figure 22: A Voronoi Diagram can recreate the block based partitioning when the points are placed at equal intervals parallel to each axis



Instead of using bounding planes, the Voronoi algorithm uses least distance, and this has two advantages that proved useful. The first is that the by applying a small amount of jitter to the positions, the regions produced still had the relative positional order of a grid like structure, but the boundaries looked more organic, which would lead to more organic and less predictable erosion effects.

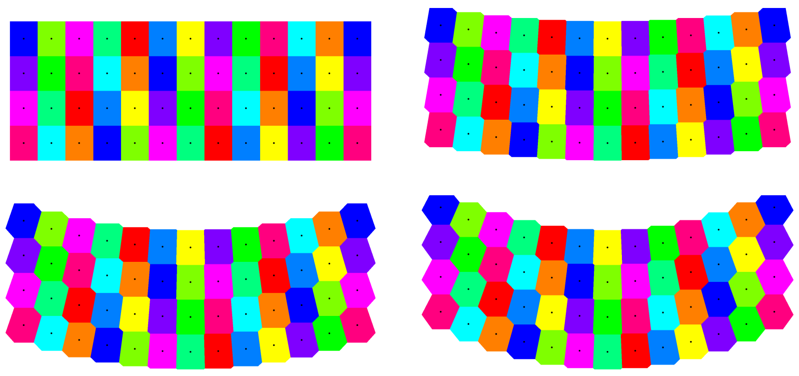
Figure 23: Applying a small amount of jitter produced more organic cell like structures



However, more importantly, by dividing the space by proximity to specific points, this meant that I could create a dynamic structure that could move and evolve. For modelling something like wood or metal - a material that may bend - this flexibility would be very useful.

Applying a parabolic distortion to the positions of the nodes of a regularly spaced Voronoi cell structure showed what would occur for a crossbeam sagging with a central load:

Figure 24: A regularly spaced Voronoi cell structure has a parabolic distortion applied (note, the cellular structure continues beyond the confines of the shape, hence the jagged edge).



The final advantage of this system is that is meant that I had the freedom to implement any initial structure I wanted - I was not limited to cube based structures, I could place nodes at any position in space and the space would be partitioned accordingly.

2 dimensional Voronoi diagrams are used extensively in a lot of procedurally generated terrain systems (Olsen, 2004; Musgrave, 1993; Korn et al., 2017; Patel, 2010), for partitioning a map into non-regular cells. In the early days of procedurally generated days, it was usual to create procedural maps using square or hexagonal tessellation, however they produce uniform structures. As computational power increased, it was possible to move to 2D Voronoi diagrams to produce more organic structures. Voronoi diagrams can be found everywhere in nature (Oudot, 2010), which means as well as being non-uniform, their irregularity 'looks right' to a human observer.

In the same way 2D maps moved from regular shapes for partitioning their space to Voronoi diagrams to produce more natural looking results, moving the partitioning from

3D voxels to 3D Thiessen polyhedra has the same effect.

With this structure fixing the problems of the block based approach, I decided to make it the foundation of my material model.

5.2.4 Material System - Implementing the Voronoi Cells

While the Voronoi structure would be the basis of my model, with each region being a different instance of a material, I did extend it with two additional parameters.

The first was to add the options to have a maximum radius for a given node, so if a point in space was further than this distance from a node, it would no longer be part of its region, even if it was the closest node. This was done to help simulate materials like unevenly distributed rocks in mortar. The stone rocks could have a capped radius, leaving the rest of the space would then be the mortar.

The second would be to include to option of connecting nodes to other nearby adjacent nodes. This would be similar to a skeletal structure, but rather than for animation is for transferring relevant information between connected nodes. For instance, if it was a needed to model moisture content, a the node connections could help with modelling the distribution of moisture within the material.

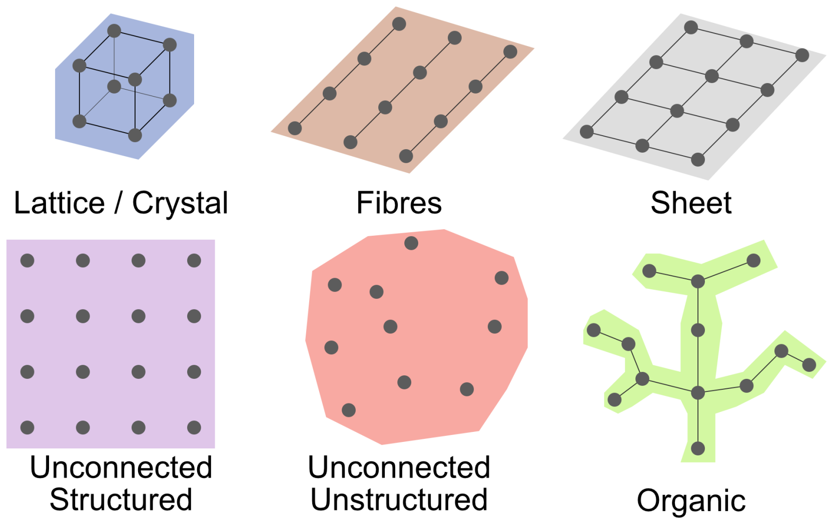
All this this meant I would be able to model many different types of material, with the cells interacting in different ways. Once set up the system would be automated and procedural, but it means that that the user is able to determine rules to guide the evolution.

I considered a few key structures that could be created:

Creating skeletal structures for 3D objects is commonly used for animation, but similar techniques have been used to produce deformable objects (Nealen et al., 2006). Therefore, it is a concept that animators and 3D artists are familiar with, as well as being a useful and relevant mechanism / data structure.

While I would focus on stone, given that was the material in Norwich Castle - which is not a flexible material, it was important to ensure the system could be used for other materials as well to prevent having to redesign the system from the ground up should the work be continued.

Figure 25: Example material structures



5.3 Mapping the Mesh the Material Model

The algorithm for mapping the vertices to is quite straightforwards. It loops through the vertexes, finds the nearest node that is behind it and that it is within the node's maximum distance, and gets bound to that.

The reason for making sure the node is behind the vertex is that the erosion process will move the vertex relative to its bound node, so to shrink, it will move towards it and to grow it will move away. To ensure this happens correctly the node must be behind the vertex.

5.4 Environment Modelling

The model for the environment is similar to the one used for the material. At the core, the model consists of nodes, each carrying properties to help model its affect. Just like the material nodes, they can be different types to describe different sorts of environmental factors - such as wind, rain, sunlight etc...

To model the environment, the model contains different regions of space containing clusters of nodes for that environmental effect. The nodes are distributed in the space (by default using a random gaussian distribution). Each environmental cluster is processed

Algorithm 2 Vertex Material Mapper

```
 1: function MAPVERTEXS($vertices[], materialNodes[]$)
 2:     for $v$ in $vertices$ do
 3:         $v\_norm \leftarrow$ Vector3D(0,0,0)
 4:         for $face$ in $v.faces$ do            ▷ the vertex normal is average of face normals
 5:             $v\_norm \leftarrow v\_normal + face.normal().unitVector()$
 6:         $nearestNode \leftarrow$ null
 7:         $dist \leftarrow 0$
 8:         for $node$ in $materialNodes$ do            ▷ find nearest node behind vertex
 9:             $n\_dist \leftarrow$ node.pos.distanceTo(v.pos)
10:             if $node.pos.behind(v.pos,v\_norm)$ and $n\_dist <= node.maxDist$ then
11:                 if $nearestNode$ is null or $n\_dist < dist$ then
12:                     $nearestNode \leftarrow node$
13:                     $dist \leftarrow n\_dist$
14:         $v.materialNode \leftarrow nearestNode$            ▷ will be null if none are found
```

against each object, with the nearest environmental node to each vertex being applied.

While this does run the danger of scaling up to **On^4** complexity, the effects of this can be mitigated by partitioning the space before processing so environments only get processed against materials they are close to. One property of the weathering nodes is that they have a maximum range - this can be used to create a bounding box or bounding sphere for the cluster, which can then be tested against the bounds of an object, or an object's oct-tree, to determine if it is necessary to perform a more detailed check.

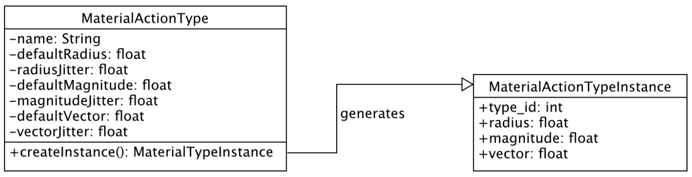
Just like the Material Type nodes, the environmental nodes would have a base class - which I called MaterialActionType - with a base set of properties. The cluster would then have instances created for each node with some randomisation of the values. It would also be possible to extend the classes with more parameters should more information be required for the simulation.

The properties I decided upon for the base class were: Radius, Magnitude and Vector.

Radius defines the maximum radius that the node can influence.

Magnitude is used to calculate the strength of the action, this value is used to calcu-

Figure 26: MaterialActionType and MaterialActionTypeInstance Classes



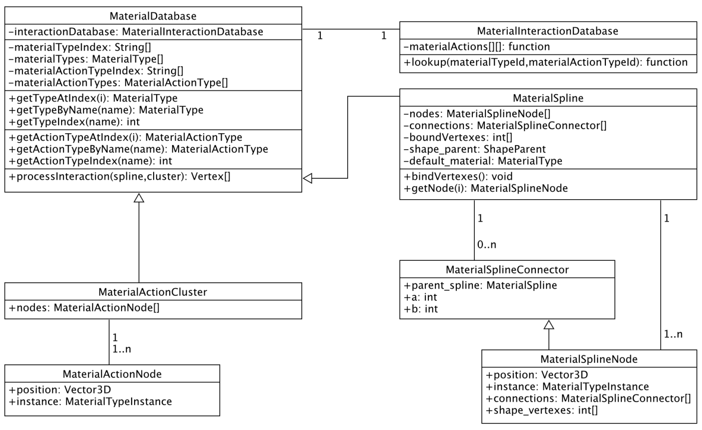
late the strength of the effect the property has.

Vector is an optional criteria, which can give direction to the effect - useful for properties like wind.

5.5 Interaction

The final part of the system, deals with the processing of the interactions. Interactions themselves were functions with the vertexes and cluster nodes passed as arguments. However there was one more step before this.

Figure 27: An overview of the Material Database datastructure


MaterialDatabase
–interactionDatabase: MaterialInteractionDatabase
–materialTypeIndex: String[]
–materialTypes: MaterialType[]
–materialActionTypeIndex: String[]
–materialActionTypes: MaterialActionType[]
+getTypeAtIndex(i): MaterialType
+getTypeByName(name): MaterialType
+getTypeIndex(name): int
+getActionTypeAtIndex(i): MaterialActionType
+getActionTypeByName(name): MaterialActionType
+getActionTypeIndex(name): int
+processInteraction(spline,cluster): Vertex[]
1
1
MaterialInteractionDatabase
–materialActions[][]: function
+lookup(materialTypeId,materialActionTypeId): function
MaterialSpline
–nodes: MaterialSplineNode[]
–connections: MaterialSplineConnector[]
–boundVertexes: int[]
–shape_parent: ShapeParent
–default_material: MaterialType
+bindVertexes(): void
+getNode(i): MaterialSplineNode
1
0..n
1
MaterialSplineConnector
+parent_spline: MaterialSpline
+a: int
+b: int
1..n
MaterialSplineNode
+position: Vector3D
+instance: MaterialTypeInstance
+connections: MaterialSplineConnector[]
+shape_vertexes: int[]
MaterialActionCluster
+nodes: MaterialActionNode[]
1
1..n
MaterialActionNode
+position: Vector3D
+instance: MaterialTypeInstance

Tying everything together is a parent class, which I called MaterialDatabase, which contains all of the MaterialTypes and Material Instances.

For speed, these are indexed in an array, with their original name in an array with matching index number. The string name is purely for the user to aid in human recognition - also for if the data is saved to a text based file such as OBJ - once loaded in the system, only the integer values are used as reference to help with speeding up the operations by using array index access.

These indexes are used as the type_id property in the MaterialTypeInstance and MaterialActionTypeInstance classes.

The final part of the jigsaw is the MaterialInteractionDatabase. This class is a lookup table for each pairing of MaterialType and MaterialActionType - this means that it is possible to specify what type of interaction occurs between them.

Algorithm 3 Algorithm for processing the interactions between a spline and a cluster

```
1:  function PROCESSINTERACTION(matTypeSpline, matActionTypeCluster)
2:      for v in matTypeSpline.boundVertices do
3:          n_nearest ← null
4:          n_dist ← 0
5:          n_interaction ← null
6:          for node in matActionTypeCluster.nodes do
7:              if interactions.exists(v.type_id , node.type_id) then
8:                  n_dist ← node.pos.distanceTo(v.pos)
9:                  if n_dist <= node.maxDist then
10:                     if n_nearest is null or n_dist < dist then
11:                         nearestNode ← node
12:                         dist ← n_dist
13:                         n_interaction ← interactions.lookup(v.type_id , node.type_id)
14:         if n_interaction is not null then
15:             n_interaction(v , n_nearest)          ▷ Only processed if a match is found
```

For example, if an MaterialActionType of Fire was created, it would be necessary to have a completely different effect on wood and stone, and this is how the system deals with that. It's also not a requirement to specify an action type for every pairing, if no action is listed then it will be ignored.

As previously mentioned, to avoid algorithmic complexity issues, this algorithm is best run after any space partitioning is done, so it is only done at a local level.

I included a basic "Erosion" weathering effect, which worked by calculating an overall erosion value based on the strength, hardness and health of the Material node, and the magnitude of the Material Action node, reduced according the inverse square law based on its distance to the vertex. The value generated would determine how far the vertex moved towards the Material node.

6 Implementation and Refinement

With the system finalised, the process of implementation and refinement began. I decided it would be best to start small, first implementing it on my test object, I would then try a brick, then a wall of bricks. After that I would extract a portion of the model, run the code on that before finally trying it on entire south wall model.

As well as being a natural progression, as the number of components increased so did the processing time. While a simple brick could take a few seconds, a high polygon component would take longer, and the entire castle model would take hours. It therefore made sense to make sure I was on the right path so I ended up with something useful.

6.1 The Test Model

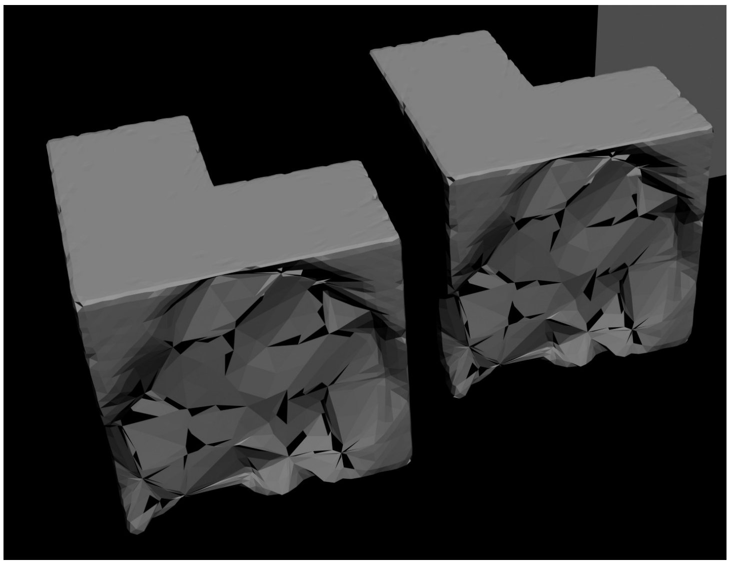
While my earlier implementations had been tests to see what would happen when I tried something, this stage of the testing process would be different. The whole point of the system was to try and cause something specific to happen, such as “make a brick look like it has been eroded”

This time, I would set an aim to each process I ran and see if my code generated the desired result. If it did, that would be great, but it was also important to see where it failed as I would then learn about any weakness in the algorithms, or input parameter combinations to be avoided.

the first task I set was to cause a massive crater like erosion in the back face of the test model:

After running the code, the results looked great - there was indeed the massive crater like structure in the back of the cube, just as I had hoped to model. However, there was another detail - notches had appeared across the edges of the cube.

Figure 28: A block has extreme erosion applied, the mesh is then subdivided and smoothed.



While these notches looked really nice, like wear you may expect to see in a stone block over time, it was not a behaviour I had intended to cause and as a result - no matter how nice it may look - I needed to understand why.

Ultimately, it didn't take me long to figure out - it was being caused by the convolution process applied to smooth the mesh at the end.

A property of the convolution process is that it averages vertices with its neighbours. Because some of the neighbouring vertcies were positioned along the two adjoining faces, the edge vertices were being pulled into the shape to create the notch. Implementing a solution to this was quite straightforwards - I only needed to apply the convolution to vertices that were affected by the erosion process. I updated the code to return a list of every vertex it affect, and then these were the only vertices that had the smoothing operation applied.

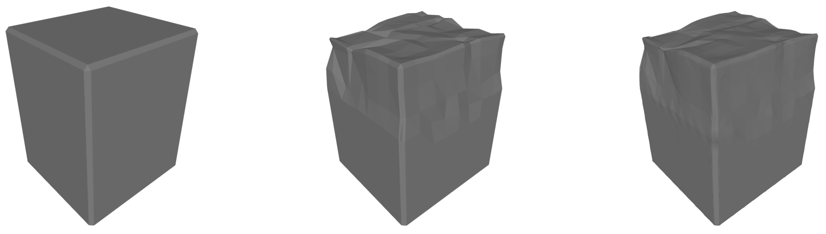
6.2 Initial results and fine tuning

It was now time to start applying the code to a model representative of the type of geometry I would be eventually be using it on. My test model was a simple primitive that had convex and concave features, however in practice, most of the world I would be doing would just be standard bricks, with a chamfered edge (as this makes up the majority of objects in the Norwich Castle model)

I wrote a small piece of Python code to generate a chamfered block as this would allow me to automate the testing process by generating many different blocks of different sizes.

The first thing I did was to configure an extreme erosion affect to half of the brick (to simulate an exterior brick - the outward facing side would see a lot more erosion to the interior)

Figure 29: From left to right, a block has extreme erosion applied to it and then the mesh is then subdivided and smoothed.



The results were better than I had expected. I ran the test multiple times, the outputs all looked the same, all showing bricks that looked like they had seen hundreds of years of weathering, each one unique. I was extremely encouraged by the results.

However, this level of erosion was “extreme” - most cases, and particularly Norwich Castle which has stonework in excellent condition, only warrant a very minor level of erosion. I therefore decided to see if I could create a much softer effect. This was quite easy to achieve, all I had to do was reduce the size of the erosive force, which could be achieved by reduced the magnitude of in the MaterialActionType properties.

I set my code to procedurally lay out some stone blocks in a rectangular pyramid, and then apply different levels of weathering to them. I set the code to generated chamfered edges, so they matched those in the south wall model. By doing this it meant I would get a controlled test to gauge what to expect when I applied it to actual model.

Again, I was pleasantly surprised by the results - the detail added to the model was incredible subtle, exactly what I had wanted. However, while every brick looked similar

Figure 30: A group of blocks with different levels of weathering applied to them.

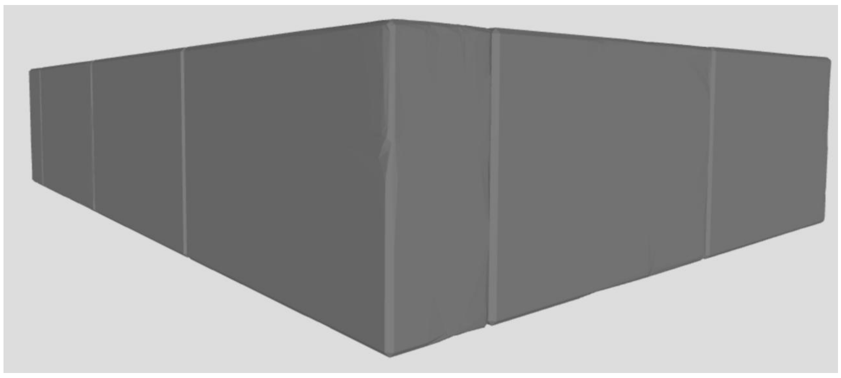
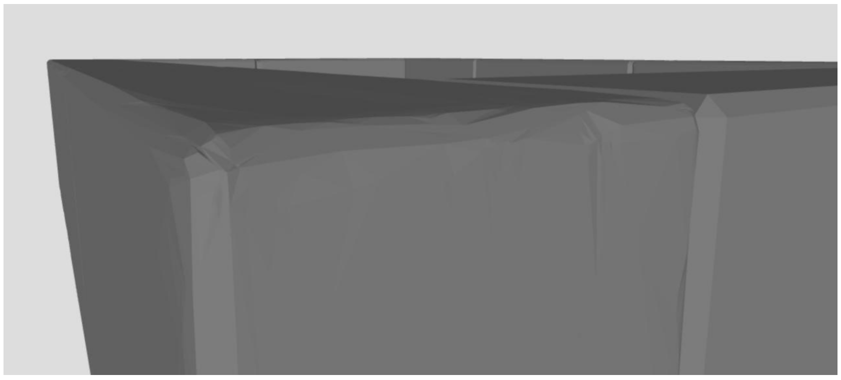


Figure 31: Close up detail of the weathering effect achieved. Note the small scratch on the corner and the larger one in the middle.

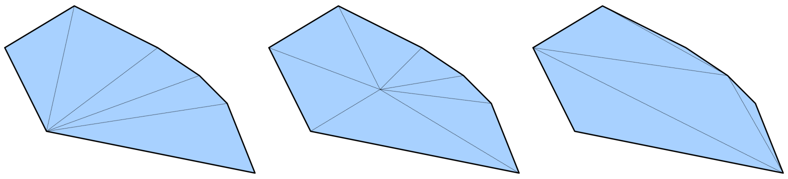


each one was unique with little details that looked natural. There were small scratches, dents and imperfections that looks exactly how a lightly weathered brick or stone would appear. What is important to stress is that this was all grown organically from a few basic parameters describing the material properties. It was not programmed to add some scratches or dents, this occurred purely down to the way the vertex manipulation responded to the few basic rules of the algorithms.

After getting it to work on my procedurally generated blocks, it was time to see how it would work on models created in 3D Max. While it would be imported into the same data structures so there would be no issue with the algorithms processing correctly, it was possible that the order of triangle sub division may cause problems.

From a rendering point of view, the way a polygon is subdivided makes no difference

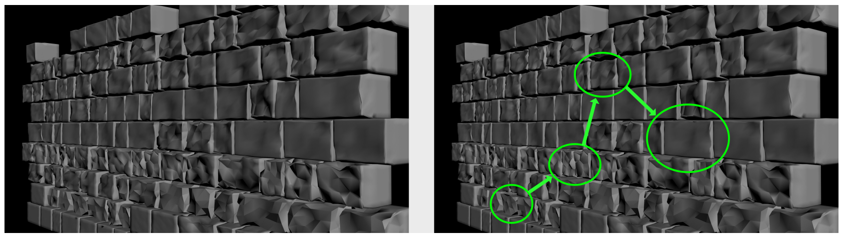
Figure 32: Examples of different methods of triangulating the same polygon



to the final visual output; however, for geometric manipulations there is the potential for it affecting the results. The procedurally generated shapes were created with the polygons whole, so this had no effect, however some 3D file formats, such as OBJ, give the option to split the faces into quads or triangles to save the renderer from having to do this at a later stage. The order of this splitting would affect the final output.

I extracted a portion of the Norwich Castle south wall and split it into component form, so each object was saved in an individual OBJ file. I then set my code to load in each obj, process the erosion, and then save it in an output folder. That way I could monitor the output files as they were processed, stop the process, adjust the parameters, and keep tweaking until I achieved the desired result.

Figure 33: A section of wall .



Due to the fact the bricks had been manually positioned by David Drinkwater when he created the model, it meant their ordering bore no relation to their position in the wall, however my tweaks can be seen as I refined the settings - highlighted in green on the right hand side.

Initially, the changes to the bricks were much more substantial than in my original tests. This was due to the fact that the bricks were defined on a much smaller scale. My test bricks were approximately 100 units long, compared to 10 units for the bricks in the Norwich Castle model. The solution to this was just to add a scale variable, which would convert between different geometric scales. The materials database could then be configured for a standard scale, and the multiplier required to convert a model's scale to the standard scale would be the scale variable.

Once I adjusted the scale variable, I was able to reproduce similar results to those seen on my test bricks.

6.3 Feedback and refinement

I showed my results to David Drinkwater and Prof Andy Day and it was suggested that I apply the weathering to a complete building - or at least elements significantly different in shape to a cuboid block, and also demonstrate variability in the weathering, both from variability of material structure and variability of the environmental condition. This was possible to create without any modification of my system.

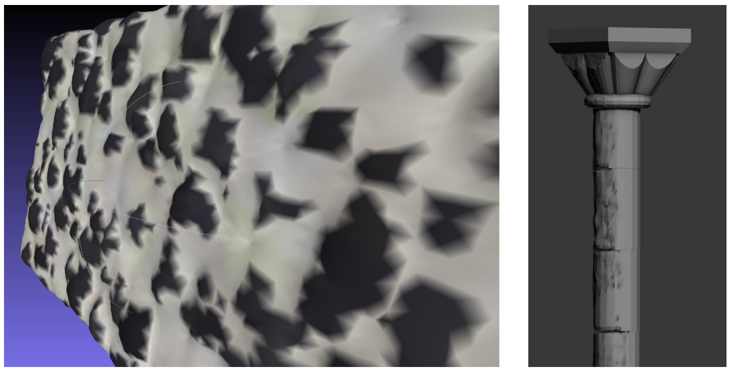
In order to create the variability of the structure I merely had to create MaterialType-Instances from a collection of different MaterialTypes (for example, I could create a class for StrongRock, RegularRock and WeakRock) and I could create variance in the structure. For my example I created a standard structure of "weak mortar" nodes, and then created a second structure of flint nodes. The flint nodes had a maximum radius, meaning that the weaker material behind it would be the inherited property until the vertex moved within its bounds. The has the effect of meaning the mesh would start out flat but erode back, forming around the harder lumps.

Another approach that could be applied would be to start out with a standard lattice spline and initially seed every node as an instance of RegularRock, and then add clusters of StrongRock and WeakRock within it.

Creating the variance of the weathering effects was even more straightforwards. For example, model greater erosion at the top of a building compared to the bottom, the weathering nodes nearer the top just need to be positioned closer to the building. The default erosion method I implemented used an inverse square law to adjust the erosion magnitude, this means that nodes closer to the structure will produce more pronounced erosion effects.

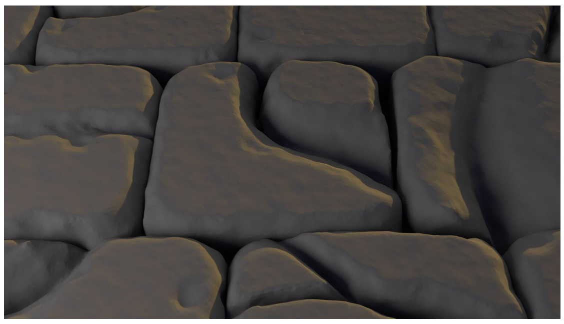
I also took the opportunity at this point to seek out feedback from 3D artists on the Internet, asking how it compared to tools they used. One user said that they used a

Figure 34: **Left:** A cuboid created with hard (dark) and weak (light) MaterialTypeInstances **Right:** A stone pillar is configured with weathering erodes closer at the top



tool called Houdini to produce similar effects (Zandehr, 2018) however it was not as sophisticated. It involved manual subtraction of meshes from other meshes - albeit with some degree of automation.

Figure 35: Example of weathering achieved with Houdini, produced by reddit user 'Zandehr'



The results also looked stylised, similar to a Pixar type of animation. Having said that, it was the best example of an existing tool that I came across during my time on this project.

6.4 Improving Performance

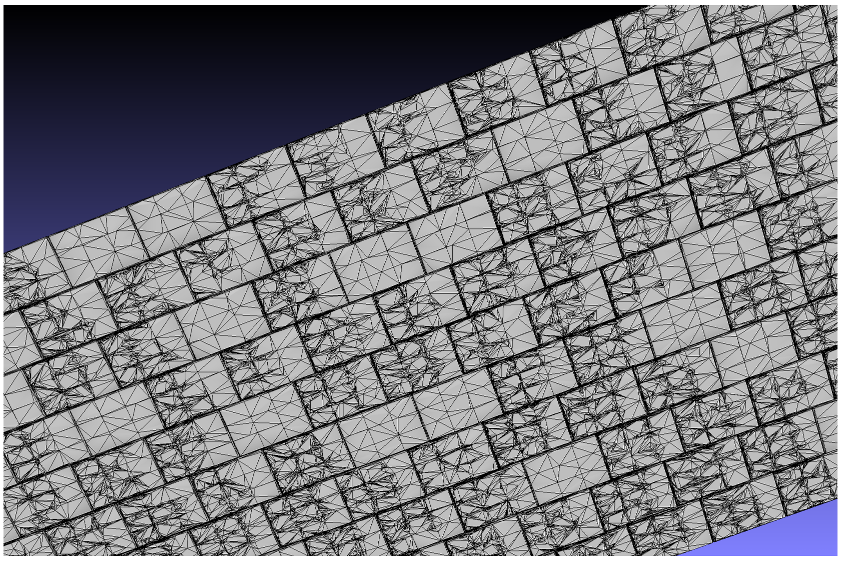
Although this was never intended to be a real time performance, as the objective was high quality models, as the complexity of the system grew, so did the processing time. There were several factors to this:

- **Model Complexity:** As the model becomes more detailed each pass of the edge splitting is being applied to an exponentially increasing number of edges.
- **Gaussian Randomisation:** In order to increase the natural look of the erosion, I applied gaussian randomisation to as many elements as possible - in particular the jitter applied to the position of the midpoint when edge splitting. Gaussian randomisation is a lot more computationally heavy than uniform
- **Python Performance Overhead:** In the end, despite giving a speed boost over MAXscript and the 3D Studio Max Python interpreter, there were still performance issues caused as a result of Python. Just to the weak typing and lack of a true array structure, it means that accessing information in the equivalent data structures is not as fast as it would be in languages such as C or Java. While this performance penalty was imperceptible with simple models, it quickly became apparently on large numbers of more complex models.

I did not want to reduce the quality of the resulting models, so I did not want to simplify the algorithms. I did consider porting the code to C++ but decided against this as in addition to taking a lot of time, it would have required restructuring some of the code and would have run the risk of adding errors, and consequently a lot of debug time, which given I was approaching the end of my project was not an effective use of my time.

There were things I could do though. The first of these was to reduce the number of times edge splitting occurred, as this was the most expensive operation. Edge splitting was only required where complexity was added, so it was only required to add complexity at the points where erosion occurred. When I had addressed the problem with the convolutional smoothing earlier, that solution returned a list of modified vertices, so I could just use this list to split the edges touching those vertices.

Figure 36: A stone wall, with the mesh overlaid showing complexity added only on the regions affected by erosion.



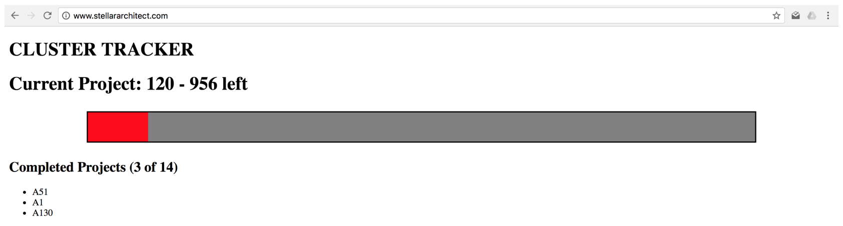
While this would reduce the overhead, it would not eliminate it - and the next stage in my plan was to apply the weathering process to the south wall model. I realised it could potentially take my laptop several days to process all of these elements which was not the most optimal solution.

I did consider using CUDA, as the pyCUDA library allows for Python code to make use of nVidia GPUs - however my algorithms I developed did not lend themselves to be easily ported to the GPU and I had no previous experience of using CUDA in any language. However, it did make me realise that it would be possible to implement another form of parallel computing.

As I was splitting the larger models into individual files it meant that I could have multiple computers process the files. I could even have one computer process multiple files at once if I ran multiple instances of Python to spread the load across all of the cores in the machine.

With OBJ files being simple text files, it was a simple process to set up a web server to check out and distribute the files. I reconfigured my Python code to connect to the web server, download the next available file, process it and upload it. The web server would save the new file and distribute the next one not yet checked out. This system meant I could have multiple computers connect to the server and help contribute to the workload.

Figure 37: A screenshot of the cluster tracker web interface



www.stellararchitect.com

CLUSTER TRACKER

Current Project: 120 - 956 left

Completed Projects (3 of 14)

- A51
- A1
- A130

The system worked effectively and using a network of computers (including my Android smartphone) from volunteers who kindly agreed to run the Python code on their computer (that reached as far away as Japan courtesy of my fiancee) I was able to process all 17,500 components in a few hours.

7 Results

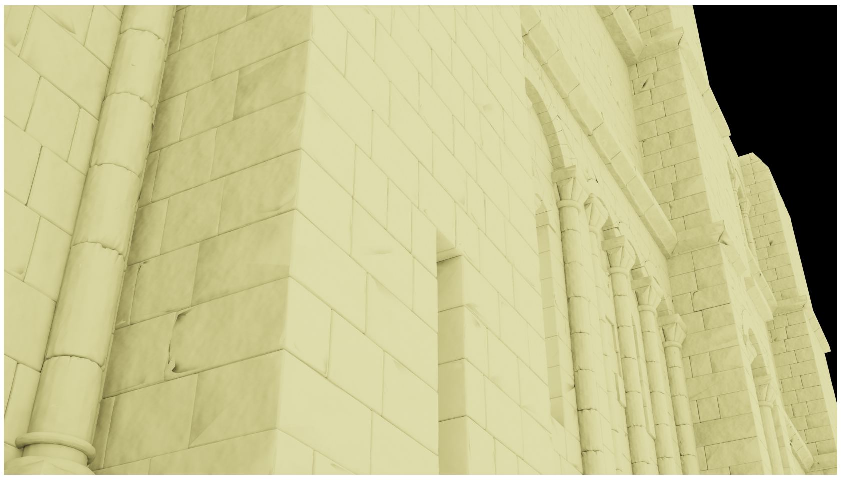
Once I had produced the new geometry it was time to import the produced OBJs back into 3D Studio Max and render the results. I decided against texturing the object, as I did not want to distract from the geometry. I did, however, decide to apply some procedurally generated bump mapping so the light diffusion was more similar to what would be expected with rock, and I coloured the model in a sandy yellow colour to more closely match the original castle.

7.1 South Wall Model

The renders exceeded expectations, the stonework looked very realistic. The castle model had a great presence, and the renderings gave it a true sense of scale. The weathering effect was very subtle - which had been my intention, as I just wanted the bricks to look slightly worn from the wind and rain.

However, after admiring the renders, I did wonder how much of it was down to the bump mapping and lighting affects I had applied, as after all, the model was highly

Figure 38: The South Wall model with weathering applied



detailed. The only way to test this was to render the original model with the exact same settings and compare them.

After rendering, the differences were obvious. The original model - with its geometrically perfect cuboid blocks - showed all the uniformity of computer generated content. I was surprised to see how obvious the flat and consistent surfaces highlighted the procedural nature of the generated bump map.

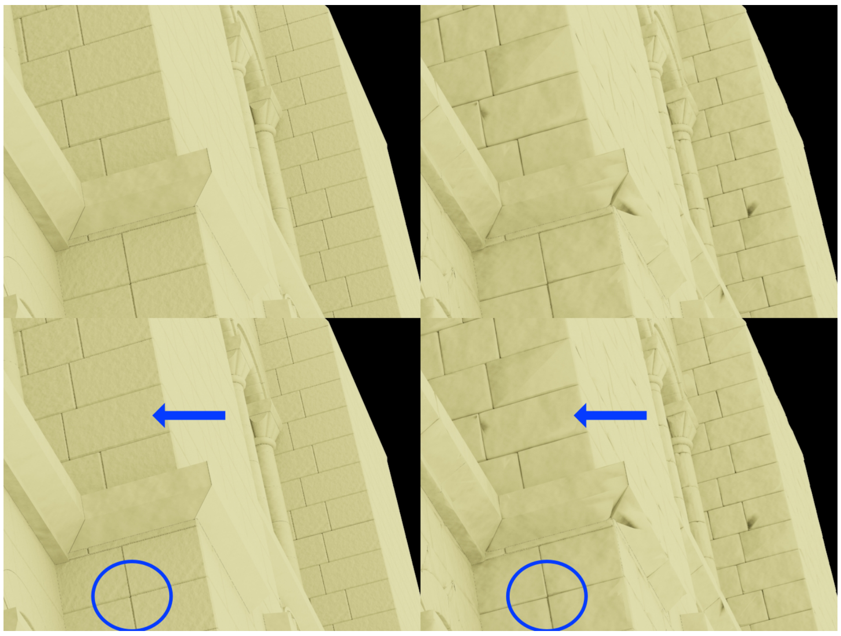
The weathered model did not show any of this uniformity. One of the consequences in very small adjustments to the vertexes was that the faces of the bricks had very subtle variations across their surfaces. These differences in surface angle were impossible to see with the human eye, however the resulting change to the light profile made the surfaces appear far more natural. (see blue arrow in comparison renders)

Side by it was possible to note some of the tiny adjustments the code had produced. The erosion around the chamfered edges of the blocks meant the shadows of the spacing between blocks appeared to vary (see blue ring in comparison renders)

7.2 Other Results

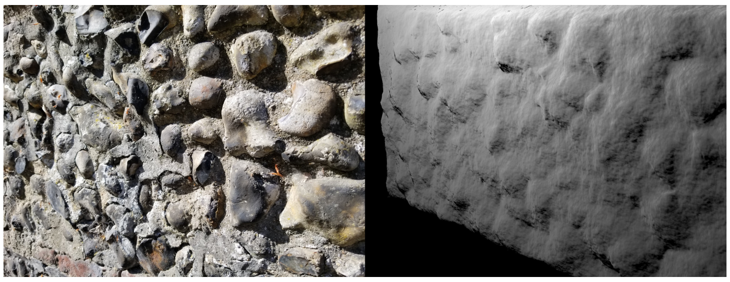
While the model of Norwich Castle was the focus of my project, I felt it important to demonstrate some of the potential of the system by showing how it could be developed to simulate other materials. I refined the flint and mortar model and was able to produce

Figure 39: The original model (left) and the weathered model (right)



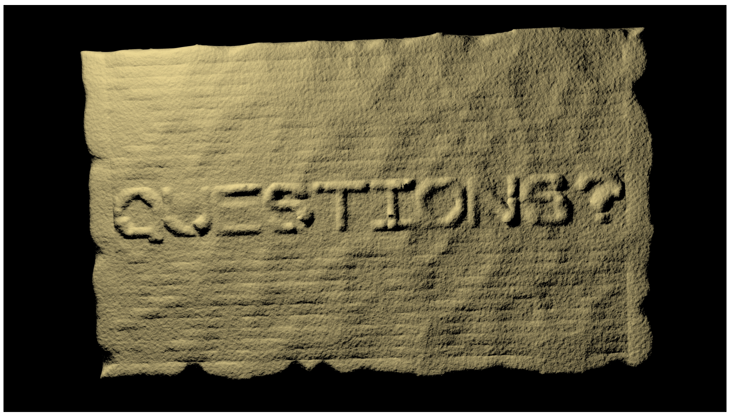
meshes that were very similar in profile to the wall type I was trying to create.

Figure 40: Flint/mortar wall in the real world compared to my simulation.



I realised that I could adapt this and by specifying where the flint stones were located within the model (rather than using randomised locations) I could use it to create messages in the material. I decided to render the word "Questions" into a surface with the intention to use it in my presentation as the final slide. While this did start out as a trivial experiment, the results again looked better than I imagined, and looks like embossed lettering that has been eroded over time:

Figure 41: Using this system it was possible to control the erosion of an object or surface



8 Analysis and Future Development

When I chose the direction of my project, I was surprised to learn how little research has been conducted in this area. While procedural generation has existed since the start of 3D Computer Games (the video game Elite had a procedurally generated universe) this has usually been limited to terrain, or for producing models from a bag of components. This has meant over the course of my project I have been able to develop my own solutions and ideas.

A lot of techniques I implemented, such as Voronoi diagrams, I later discovered had been used to make procedurally generated terrain, which I felt vindicated my decision to go down this route.

Ultimately, I am very pleased with the results of my implementation; while I knew that the system I came up with would work, I never expected to get the subtle variations it was able to produce.

8.1 What went well

- **Configurability:** It was very easy to be able to develop new materials with predictable results even with the small number of parameters in the basic set up. I was able to make new materials such as soft mortar, hard rock and strong flint and get them to work as expected with very little tweaking.

- **Realism:** My system could produce unlimited stone blocks eroded to a certain level, every one would look the same at a glance, but be totally different when looking at the tiny details. This is was exactly what I wanted to achieve as it made repetitive elements look far more realistic.
- **Scalability:** I was able to apply a tiny level of erosion that took the sharp edges off, or make it look like it brick work that was thousands of years old. Thanks to my cluster I was able to process thousands of elements. I could get the system to work on models of different sizes .

8.2 Where it can be improved

- **Performance:** The system is very slow. I think most of this is down to Python and would strongly encourage a rewrite in C++ to gain performance, however looking at the internal algorithms for handling the data structures for the 3D shapes would also make sense.
- **Glitches:** There were a few glitches where some bricks had darker regions that stood out. This was caused by the triangulation of the shape in 3D Studio Max when converting to OBJ. There are different solutions to this - one is to export as quads or polygon meshes, the other is to procedurally generate as many of the components as possible.

8.3 Conclusion

While I am very pleased with the final renders I produced, and the quality of some of the procedurally generated weathering effects, I believe that my greatest accomplishment on this project is showing that there is a lot of research opportunity in this area of 3D graphics.

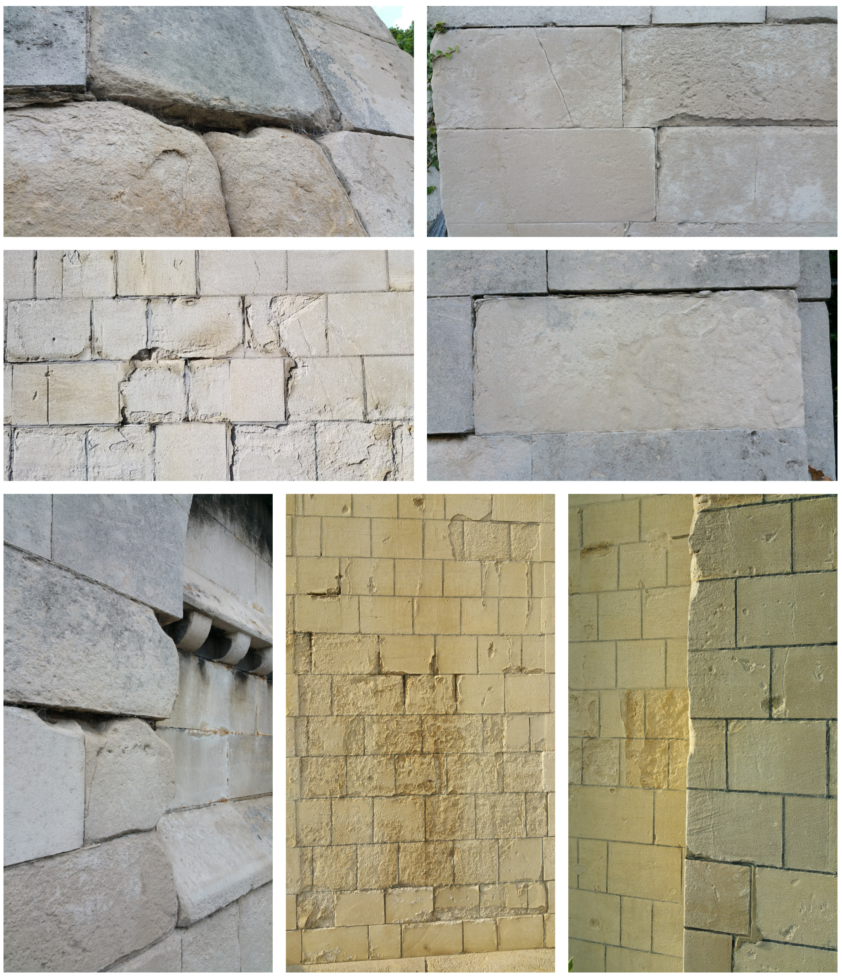
Procedural generation is vital to the future of this field, particularly in film, TV and computer games. All of these industries are seeing escalating costs as a consequence of having to hire more and more animators, models and designers. Procedural generation is the key to reducing those costs, so long as it doesn't impact the quality.

Systems like this are critical to achieving that, as they focus on repetitive tasks that are very time consuming and produce better results, freeing up the artists to work on more creative parts of the project.

9 Reference Photographs

The following photos show close up details of the erosion on Norwich Castle

Figure 42: Photographs I took showing erosion and weathering of the stonework on Norwich Castle



References

DeRose, T., Kass, M., and Truong, T. (1998). Subdivision surfaces in character animation. *Computer graphics and interactive techniques*.

Grant, C. (2013). Python in 3ds max - what does it mean? Blog. http://christophergrant.com/blog/2013/10/4/python-python-more-python.

Haran, B. and DeRose, T. (2015a). Numberphile - math and movies (animation at pixar). YouTube. https://www.youtube.com/watch?v=mX0NB9IyYpU.

Haran, B. and DeRose, T. (2015b). Numberphile - more from numberphile's pixar video. YouTube. https://www.youtube.com/watch?v=2NzTAaYgk4Q.

Korn, O., Blatz, M., Rees, A., Schaal, J., Schwind, V., and Gorlich, D. (2017). Procedural content generation for game props? a study on the effects on user experience. *Computers in Entertainment*, 15:1–15.

'legomasta99' (2016). Minecraft quad-core redstone computer v4.0. YouTube. https://www.youtube.com/watch?v=SPaI5BJxs5M.

McNamara, A., Chalmers, A., Troscianko, T., and Gilchrist, I. (2000). Comparing real and synthetic scenes using human judgements of lightness. *Rendering Techniques 2000*, pages 207–218.

Mori, M. (1970). The uncanny valley. *Energy*, pages 33–35.

Musgrave, F. K. (1993). *Methods for Realistic Landscape Imaging*. PhD thesis, Yale University.

Musgrave, F. K., Kolb, C. E., and Mace, R. S. (1989). The synthesis and rendering of eroded fractal terrains. *ACM Siggraph Computer Graphics*, 23(3).

Nealen, A., Muller, M., Keiser, R., Boxerman, E., and Carlson, M. (2006). Physically based deformable models in computer graphics. *Computer Graphics Forum*, 25(4):809–836.

Olsen, J. (2004). Realtime procedural terrain generation.

Oudot, S. (2010). Delaunay triangulation. Lecture Slides.

P., L. J. (1990). Is the fractal model appropriate for terrain? *Disney's The Secret Lab 3100*.

Patel, A. (2010). Polygonal map generation for games. Website.

Portnow, J. (2010). Extra credits - games should not cost $60 anymore. YouTube. https://www.youtube.com/watch?v=VhWGQCzAtl8?t=1m24s.

Schreier, J. (2017). *Blood, Sweat, and Pixels: The Triumphant, Turbulent Stories Behind How Video Games Are Made*. Harper Collins.

van Pabst, J. V. L. and Jense, H. (1996). Dynamic terrain generation based on multi-fractal techniques. *High Performance Computing for Computer Graphics and Visualisation*, pages 186–203.

Zandehr (2018). Reddit comment on brief overview of my university project - procedural weathering/erosion for 3d models. reddit. https://www.reddit.com/r/3Dmodeling/comments/8hilhn/brief_overview_of_my_university_project/dyo6z0m/.