Bachelor’s thesis

Information and Communcations Technology

2023

Christy Green

Optimizing Game Models: A Comparison of Geometry Nodes and Traditional Modelling Techniques in Blender

How the use of geometry nodes in Blender affects the polygon count, file size, and performance of 3D models when exported to Unity, compared to traditional techniques and best practices for optimizing these models.

Bachelor’s / Master’s Thesis | Abstract

Turku University of Applied Sciences

Information and Communications Technology

2023 | 59 pages

Christy Green

Optimizing Game Models: A Comparison of Geometry Nodes and Traditional Techniques in Blender

This study investigates the impact of using geometry nodes in Blender on the polygon count, file size, and performance of 3D models when exported to Unity, compared to traditional modelling techniques for the purposes of discovering their optimal use in games or applications in terms of memory usage/performance. The project also includes an analysis of different techniques available when using geometry node networks. Ten models were created, with four using traditional box-modelling techniques and six generated from geometry node systems, their performance was tested individually, and each three times using Unity's built-in profiler to measure the CPU /GPU load time in milliseconds to render the mesh in a blank scene. The results indicate that geometry node-generated meshes are essentially identical to traditionally made models, if utilised correctly.

It was found that particle systems in geometry nodes do not translate into Unity’s prefab system, requiring additional steps that should be considered depending on the needs of the user. The geometry node system is a versatile tool with the potential to save significant developer time under the right circumstances. However, understanding its limitations and suitability for specific workflows is essential to maximise its utility and optimize 3D model performance/usability in Unity.

Keywords:

3D modelling, blender, unity game development, modelling, geometry nodes

Content

List of abbreviations (or) symbols **5**

1 Introduction **7**

2 Literature Review **11**
2.1 Polygon Count: 12
2.2 File Size: 12
2.3 Performance Optimization: 13

3 Methodology **14**
3.1 Methodology Introduction 14
3.2 Methodology – Model Making Process 15
3.2.1 Default Cube 18
3.2.2 Geometry Node Cubes 19
3.2.3 Default Torus 24
3.2.4 Geometry Node Torus 24
3.2.5 Particle System Comparison 26
3.2.6 Geometry Node Particle System 28
3.2.7 Geometry Node Helmet 30
3.2.8 'Applied' Geometry Node Helmet 33
3.2.9 Modelling Adjustment Helmet 34
3.3 Methodology – Unity Testing Process 37

4 Results **39**

5 Discussion – Geometry node performance **45**
5.1 CPU 45
5.2 GPU 47
5.3 Overall 48

6 Discussion – Geometry Node Instances **50**

7 Evaluation **54**

8 Conclusion and Recommendations **57**

References **58**

List of abbreviations (or) symbols

.blend, or ‘blend file’	The save file format for a Blender project.
Boolean Method	A modelling technique that combines or subtracts one mesh from another based on their overlapping geometry.
CPU	Central Processing Unit - the primary component of a computer that carries out instructions of a computer program.
CPU/GPU usage	The percentage of a CPU/GPU’s processing power that is being used.
Edge	A connection within a mesh that connects two vertices.
Face	A polygon created from 3 or more edges, forming a surface.
.fbx	A file format used for exchanging 3D models between different software applications. Stands for ‘Filmbox.’
File size	The amount of space a digital file takes up on a storage device.
Geo node	An abbreviation for ‘geometry node’ or ‘geometry node system.’ A ‘geo node cube’ is a cube generated using geometry nodes.
Geometry/mesh	The actual 3D shape of the object. Lists the network of interconnected vertices, edges, and faces.
GPU	Graphics Processing Unit - a specialized electronic circuit designed to rapidly manipulate and alter memory intended for output to a display.

Term	Definition
Instances	Repetitions of a 3D object in a scene that share the same mesh data.
Node	A basic element in a visual programming system, representing a specific operation or function.
Node Parameters	Settings and options that can be modified on a node to change its behavior.
Normals	Vectors perpendicular to the surface of a polygon used for shading and lighting calculations.
Polygon Count or Polycount	The number of polygons used to construct a 3D model.
Prefabs	A pre-made object or collection of objects in Unity that can be reused and shared across multiple scenes.
Profiler	A tool in Unity used to measure and optimize the performance of a game or application.
Render time	The amount of time it takes to render a 3D scene into a final image or animation.
Shader	A program used to define the visual appearance of a 3D object.
Topology	The overall organization of the faces and vertices the mesh is comprised of.
Tris or Tricount	The number of triangles used to construct a 3D model. A Triangle is a type of Face, formed with 3 edges.
UV or UV map	Object data for the mesh that represents the 3D mesh as a 2D projection for texturing purposes.
Vertex	A point in 3D space. Connecting at least three vertices with edges allow a face to be made.

1 Introduction

The video game industry has experienced significant growth in recent years, with the global market expected to generate over $200 billion in revenue by 2023 (Newzoo, 2021). As games become increasingly complex and immersive, the demand for high-quality 3D assets has grown. However, creating optimized 3D models that meet the demands of modern games (Wolf, 2017) while maintaining optimal performance can be a challenging task.

This is particularly true for game developers on a tight time/cost budget who must balance artistic vision with the practical constraints of real-time rendering.

One tool that has maintained popularity for 3D modelling is Blender (Blender Foundation, n.d.), an open-source software used for creating high-quality 3D models, animations, and interactive applications.

Blender offers several workflows for building models, from traditional techniques like sculpting, mesh editing (or box modelling), cutting geometry away with other objects using the Boolean technique, to mathematically describing the procedural generation of meshes using the comparatively newer Geometry nodes.

Geometry nodes facilitate the development of procedural mesh-construction systems capable of rapidly generating a multitude of mesh variations (Blender Foundation, n.d.), potentially accelerating the game development workflow if the technique is a suitable time investment for the given project.

In contrast, Blender's typical workflow resembles that of making a model physically out of clay. Pushing, pulling, carving, adding, and cutting away the geometry until it resembles the desired shape (Simonds, 2013).

This study investigates the distinctions between these two workflows, examining their implications for polygon count (measured in triangles), file size, and performance of models when imported to Unity. The models produced are compared for their game-readiness/efficiency. The purpose of this is to discover

the best practices when including models into a game/application to increase performance when utilizing techniques such as geometry nodes, which can potentially decrease development time significantly.

Efficiency in this instance referring to how the model in question strained the system in milliseconds per frame, which is influenced by a variety of factors, this includes polycount (polygon count), file size, materials etc. (Gregory, 2018).

System strain in this instance referrers to the stress the model places on the user's CPU/GPU. This describes how intensive the device finds rendering the model to be. (Unity Technologies, n.d.) Every model will stress the system to an extent, and optimally models should aim to be as simplified as possible without sacrificing detail.

Using the open-source 3D modelling tool Blender, an evaluation of its different model making workflows is explored here, in regards particularly to game development in Unity (Unity Technologies, n.d.). Blender provides many different techniques for model generation, (Simonds, 2013) from the more typical 3D modelling techniques like sculpting, box modelling, boolean modelling, to describing the procedural generation of meshes using geometry nodes, both of which are examined here.

While Blender is an excellent tool for creating 3D assets, the objective is to integrate these assets into a game engine, where their efficiency and performance become critical factors.

Unity is one of, the more popular game engines, released in 2005 offering a powerful set of tools for building 2D and 3D games across multiple platforms (Unity Technologies, 2021).

Users will typically build their game worlds/characters using anything up to, or exceeding, hundreds of game assets, (Totten, 2019) all requiring a portion of computer memory time per frame to display. A model of incredible detail (polygon-wise and/or material-wise) is likely to cause a disruption to

performance (Gregory, 2018). Ideally game developers will strive to keep a cautious balance between detailed meshes and system performance.

As such, evaluating the efficiency and performance of 3D models created in Blender is essential to understand how to create optimized assets for game development in the Unity game engine.

Within this report the impact of using Blender's Geometry nodes on the polygon count, file size, and performance of 3D models when exported to Unity is examined, as compared to traditional modelling techniques is examined.

The polycount is mostly a colloquialism, as this term is usually used to refer to the number of triangular surfaces an object consists of. For example, a cube has 6 square faces, which are further broken down into 2 triangles each, or ‘tris’ for short.

Therefore, a typical cube has a polygon count (polycount) of 12. The more complex the shape, typically the more tris required to display it.

To compare efficiency, ten models were created - using both methods and ranging in complexity. These models were then imported into the Unity game engine, and duplicated one hundred times, where their file size and polygon count were recorded. Using Unity's built-in profiler, a debugging tool, the CPU/GPU usage of each tested mesh was monitored in milliseconds per frame to gauge how efficiently it could be rendered in a game environment (Unity Technologies, n.d.). The process was repeated on two different machines.

Based on this analysis, an understanding of how efficient each model is as an in-game object was formed. This process was repeated three times for each of the ten models and for each machine for a more accurate average time, due to ambiguity in the Unity profiler. Finally, the results were discussed to evaluate the overall benefits of each method for game development in Unity.

By evaluating and comparing multiple models created using both workflows, this study provides insights into best practices for optimizing models for game development in Unity. The following chapters provide a detailed examination of

the methodologies used in this study, the results obtained, and finally their implications for game development using Blender and Unity.

2 Literature Review

Despite the growing popularity of Blender and Unity in the game development industry, there is a surprising lack of literature on the use of Blender's Geometry Nodes in combination with Unity. While there are numerous studies examining the use of traditional 3D modelling techniques in game development, few studies have explored the effectiveness of procedural modelling tools like Geometry Nodes in this context.

As such, there is a need for a more comprehensive analysis of the impact of Geometry Nodes on model performance and file size, as well as the development workflow itself. This thesis aims to address this gap in the literature by providing a thorough investigation into the use of Geometry Nodes in the context of game development, with a particular focus on Unity development.

The use of 3D models in game development has become increasingly prevalent in recent years, and there is a growing need for efficient and optimized models. This section provides a review of the literature related to the use of 3D models in game development, with a focus on polygon count, file size, and performance optimization.

By expanding the scope of the literature search to game development optimisation in general, several research papers can be found. For example, “A survey of real-time rendering techniques for virtual reality.” (Anderson et al., 2017) discusses Various optimization strategies, including Level of Detail (LOD), occlusion culling, and data compression, can be applied to improve the performance of game engines like Unity, albeit this research does not directly reference geometry nodes.

2.1 Polygon Count:

The polygon count of a 3D model refers to the number of triangles used to create its surfaces. The more complex a model is, the higher its polygon count will be. A high polygon count can negatively impact performance, leading to issues such as slow frame rates and long loading times. Previous research has shown that reducing the polygon count of 3D models can result in significant performance improvements without sacrificing visual quality (Wang et al., 2019; Zhang et al., 2018).

While the research conducted by Wang et al. (2019) and Zhang et al. (2018) focuses on reducing the polygon count of 3D models through mesh simplification techniques, their work does not directly address the impact of using geometry nodes in Blender on the polygon count, file size, and performance of 3D models when exported to Unity, which is the central topic of the present study.

2.2 File Size:

The file size of a 3D model is another important factor to consider when optimizing for game development. Large file sizes can impact loading times and take up valuable memory, potentially leading to performance issues. Researchers have explored various methods for reducing file size, including compression algorithms and LOD (level of detail) techniques (Suykens et al., 2016; Gao et al., 2020). LOD in this case refers to an option available in Unity that allows the designer to swap models out for less ‘expensive’ versions with lower detail and fewer polygons, the further the player camera is from the object. There’s a lower level of detail necessary for distant objects since they will appear very small on screen, and therefore more difficult to discern details from the player’s viewpoint in any case.

2.3 Performance Optimization:

Efficient performance is crucial in game development, as it can significantly impact the user experience. In addition to reducing polygon count and file size, other optimization techniques have been explored in the literature, such as occlusion culling and dynamic batching (Liu et al., 2020; Lee et al., 2017). These techniques aim to reduce the workload of the CPU and GPU, resulting in smoother gameplay and improved frame rates. Occlusion culling in this instance refers to a game engine feature that automatically removes geometry hidden from player view to save on memory load. Dynamic batching refers to reducing the number of individual draw calls sent to the GPU by combining several similar ones into a single batch. In Unity, for instance, dynamic batching is used for smaller objects that share the same material properties and have fewer vertices. The engine automatically batches these together and sends them to the GPU in a single draw call. This process is 'dynamic' because the batched objects can still move independently and retain their unique properties, unlike static batching where the batched objects are expected to remain stationary.

In summary, the literature suggests that optimizing polygon count, file size, and performance is crucial for the successful development of 3D models for gaming applications. The next section of this thesis will explore the use of Geometry nodes in Blender and how they can be optimized for efficient game development in Unity.

3 Methodology

3.1 Methodology Introduction

To achieve the objectives of this study, a quantitative research method was employed. The research was conducted in two phases:
In the first phase, eleven models were created using two different techniques in Blender: four models were created using traditional 3D modelling techniques, and six models were created using Blender's Geometry Nodes tool. An extra model was made using this method to explore two different techniques to create the same model, to see if that made a difference to the end file size. The models were designed to have varying levels of complexity and detail, ranging from simple geometric shapes to more intricate designs. The models were then exported as .fbx files for use in Unity. The export options remained consistent across all models, ignoring UV maps (which were removed if present) animation data, materials etc. Only the mesh, was exported. The file size and polygon count (in terms of triangles) was then noted for evaluation.

In the second phase of the study, the models were imported into the Unity game engine, and their polygon count and file size were recorded. Using Unity's built-in profiler tool, the CPU/GPU usage of each model was monitored to gauge how efficiently it could be rendered in a game environment. The models were also evaluated for their visual quality and overall game-readiness and tested on two different machines.

To ensure that the results were reliable and consistent, each model was tested three times, and the average values were recorded, furthermore the tests were run on two different machines. The version of Blender used was 3.3.1 and the version of Unity used to test on both machines was 2021.3.4 f1.

The first device being tested with is a desktop PC with a 12th Gen Intel® Core i5-12400F 2.5 GHz, with 16GB installed RAM. The GPU is an older ASUS NVIDIA GeForce GTX 970 4GB model. Referred to as ‘Home PC’ during testing.

The second testing device is a desktop PC with an Intel® Core i9-9900 CPU 3.10GHz with 64GB of installed RAM. The GPU is an NVIDIA GeForce RTX 2080 Ti 11GB model. Referred to as ‘University PC’ during testing.

3.2 Methodology – Model Making Process

Initially, three cube models were created: a default cube, a geo node cube, and an alternative geo node cube. To examine different techniques, with each cube featuring 2-meter edge lengths on all sides.

Next, a default torus and a geo node torus were modelled, with each possessing 576 faces and a major radius of 1 meter.

Also, a plane with 20 instances of a cube scattered randomly across its surface, to imitate a particle system, alongside a geo node version of the same random scattering. In the geo node version, both the plane and the 20 cube instances were created using geo nodes.

Finally, a helmet was crafted using both traditional and geometry nodes. The helmet was designed using a complex node tree of 104 nodes for increased customizability. Similarly, to the cube mesh, 3 versions were made. One mesh exported from the geometry node network directly, one exported after applying the geometry node modifier beforehand, and an edited version using box modelling techniques to clean up the geometry of the mesh.

Making models ‘traditionally’ in Blender can be done in a number of ways, but for the scope of the research focus was placed on the ‘box modelling’ technique of modelling, sometimes referred to as ‘hard surface’ modelling. This typically involves starting with a primitive shape or single vertex, and extruding, cutting, pushing, and pulling the model into the desire shape (Figure 1)

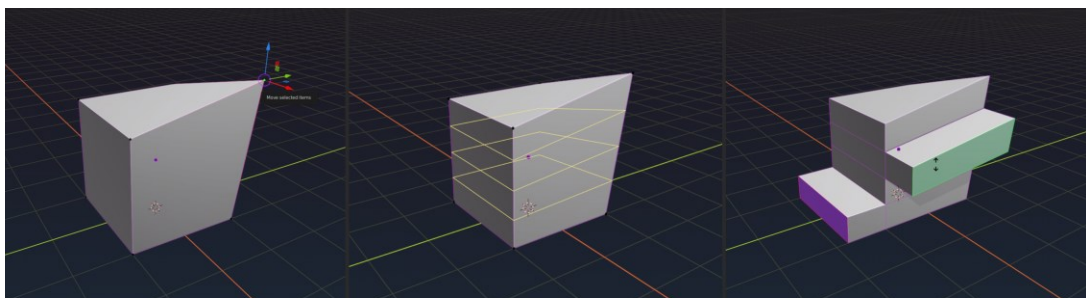

Figure 1. A default cube first having a vertex moved independently, 3 loop cuts being made around its surface, and 2 faces being extruded. Using this box modelling technique alone it’s possible to create practically any shape.

When beginning a geometry node set up, the Geometry Node workspace is used. This displays a blank canvas with a ‘New’ button above. Clicking this, a new empty geometry node system can be created.

The geo node system is added to the mesh as a modifier. Blender modifiers, which can be used for various purposes, are a form of non-destructive modelling instructions applied to a mesh to change its shape. They can be removed or altered at any time to return the mesh to its original shape. A common example of this is the Simple Deform modifier, that can bend or twist the mesh into a curved shape and removed any time. (Figure 2).

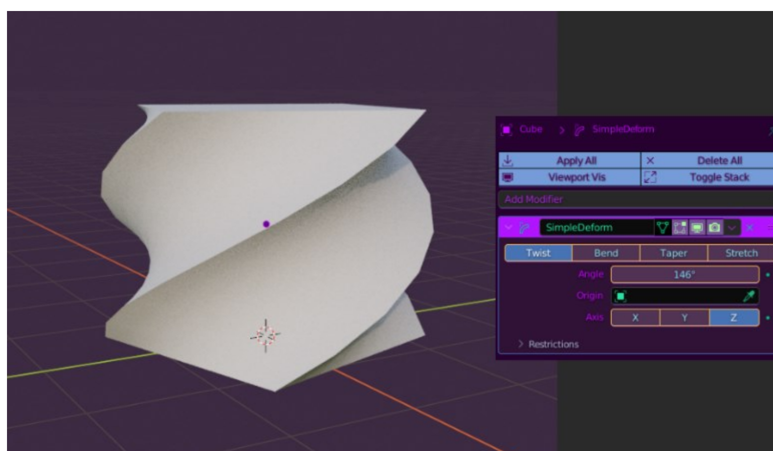

Figure 2. A Simple Deform modifier non-destructively altering the shape of the mesh.

A geometry node modifier operates within the geometry node workspace. Once a new system is created, the original mesh is represented by a 'Group input' node (Figure 3), and the 'Group Output' node functions as the final step in the mesh building process. It can be seen that the two nodes are connected via a wire, showing how the information from the original mesh is ‘sent’ to the output, rendering it to the screen.

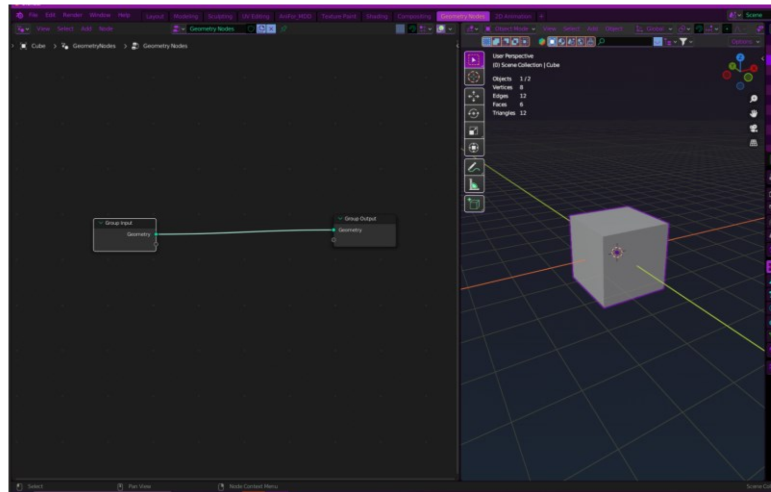

Figure 3. A new geometry node system.

Only elements connected, at least indirectly, to the Group Output node are displayed. The original mesh can be restored by reconnecting the Group input to the Group Output at any time. Clicking on the output or input sockets on nodes and ‘dragging’ a wire out allow the user to string the nodes together in this way. Other operations can be used for the process such as holding CTRL and right-clicking, which allows the user to ‘cut’ the wire, disconnecting the nodes. There are many other such functions beyond the scope of the research here, needless to say adding, connecting, and disconnecting nodes together is simple and intuitive in itself, even if the mathematics of the entire node tree can reach staggering complexity (Figure 4).

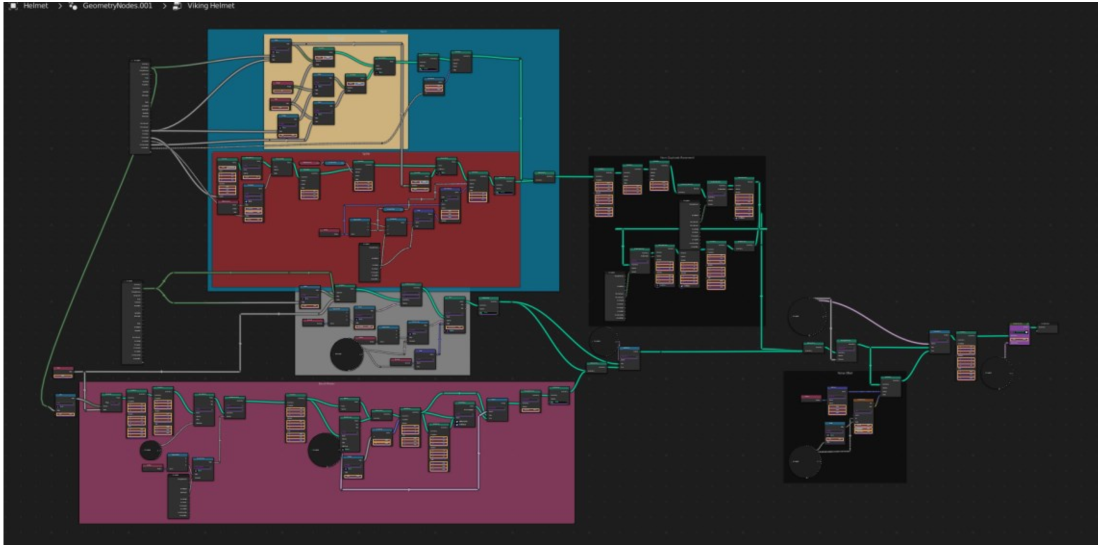

Figure 4. A node tree containing 104 individual nodes. Generates a variety of helmets.

3.2.1 Default Cube

Here, the model making process using both techniques will be explained in greater detail, starting with the default cube.

Typically, a default .blend file in Blender starts with a default cube, making the first model relatively simple. In case the cube is absent however, a new cube can be generated by holding down SHIFT + A and selecting Mesh > Cube, or by navigating to the Add menu and choosing Cube under the Mesh category. The resulting cube has the default dimensions of (2m x 2m x 2m) on the X, Y and Z axis.
It has no animation data but does include a UV map. To isolate the impact of the mesh on file size, the UV map was removed before exporting the cube. The mesh alone, exported as an .fbx file, resulted in a file size of 11.4 KB (11,708 bytes).

3.2.2 Geometry Node Cubes

For testing variety, two methods were used to create two different geo node cubes. This was done to see if the complexity of the node tree effects the file size/system strain of the final mesh. Also, the first method was so straightforward it lacks a certain degree of demonstration.

Method 1 was simply adding a cube node into the geometry tree. This is done by pressing SHIFT + A in the geo node workspace, selecting 'mesh primitives' from the menu, and 'cube' create a Cube node. A node can be seen hovering around the cursor (Figure 5), and the user may place it anywhere in the geometry node workspace.

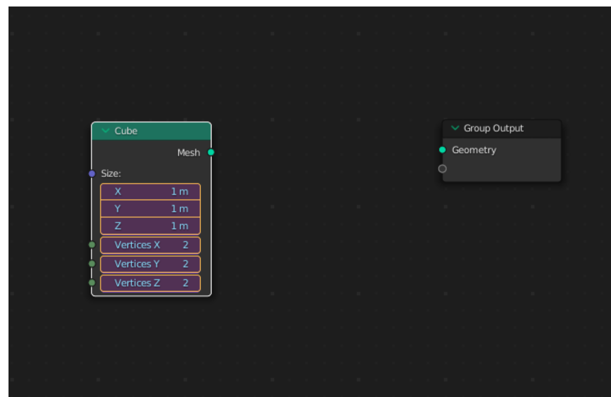

Figure 5. A cube node being added.

The node can then be connected simply to the group output, the group input can either be ignored or removed entirely by pressing X on the keyboard or right-clicking and selecting 'delete'.

It's worth mentioning that the dimensions of the cube generated with the cube node are (1m x 1m x 1m) by default, which were altered to 2 metres to match the default cube generated previously (Figure 6).

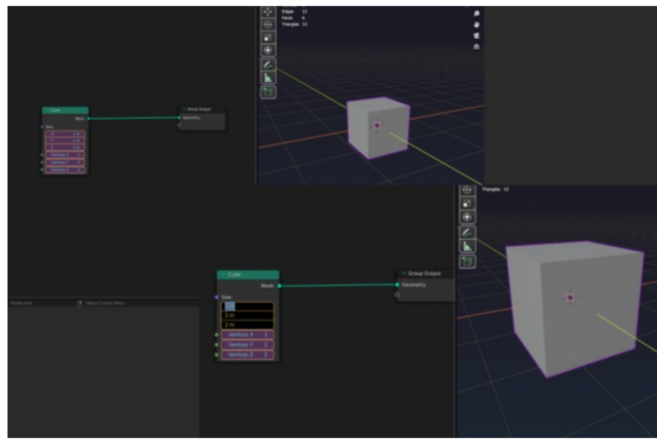

Figure 6. The cube node has its dimensions changed to match the default cube.

This cube was exported using the same export settings as the previous default cube (these settings can be saved as a preset for identical options every time).

As with each model, the .fbx file was filed away appropriately and its size noted. Interestingly this node system results in a marginally smaller file size (4.5%) than the original default cube coming in at 11,180 bytes. The reasoning for this is unknown currently, both models possessed no additional data, such as a UV map, or animation data for example. It may be that Blender includes with the default cube additional data that cannot easily be removed by the user.

In Method 2, a more intricate approach was taken to actually construct a cube using geo nodes. The process began with the addition of a Grid node connected to the Group Output (Figure 7). Next, the vertex count was reduced to minimize intersecting loop cuts that were not present in the default cube. The grid size was then increased to match the dimensions of the original default cube (2m x 2m).

To transform the flat square surface into a cube, an 'Extrude' node was added to the geo node system.

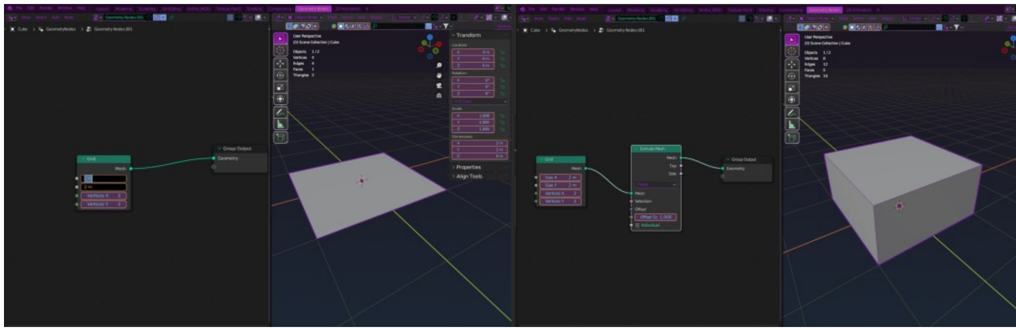

Figure 7. Extruding the grid.

This node extrudes the original grid-face upwards along the Z-axis by 1 meter, generating new edges and faces. Adjusting the offset value to 2, produces the default cube dimensions.
It can be seen that while the cube-to-be is looking slightly more presentable, there is no bottom face when using this set up (Figure 8).

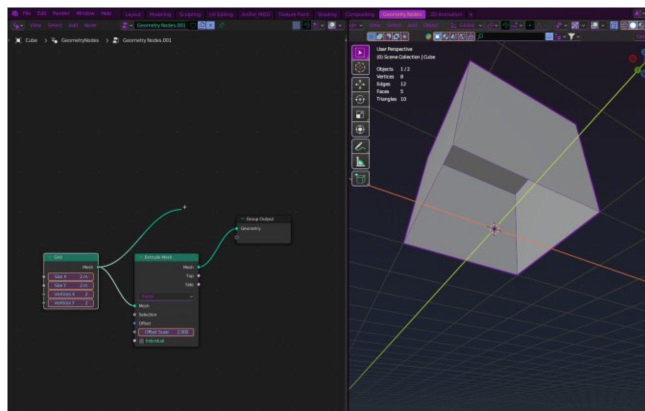

Figure 8.The vacancy left behind after extrusion.

The extrude node extends the original grid-face upwards along the z axis but leaves a vacancy behind.
It's worth reminding there's still the original grid-face node in the workspace (Figure 9), which can be viewed at any time by reconnecting its mesh output wire into the Group Output node. Notably this surface is at Z axis position 0, meaning the perfect placement for the cube's bottom face.

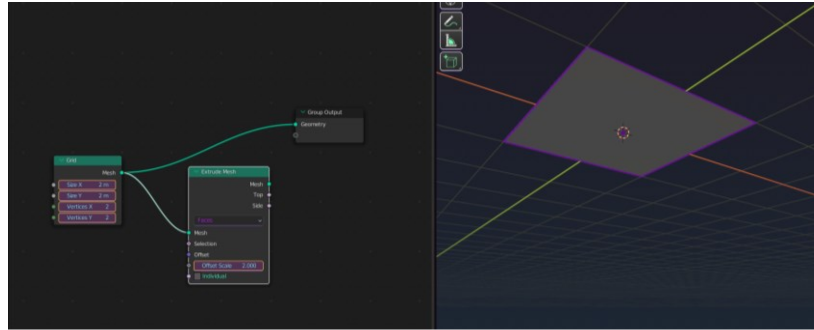

Figure 9. The original grid.

The grid face was combined with the extruded mesh by way of a Join Geometry node. The node has a wide socket (Figure 10), meaning it can take multiple inputs. Adding both meshes, and by connecting the join geometry node to the group output a cube was displayed.

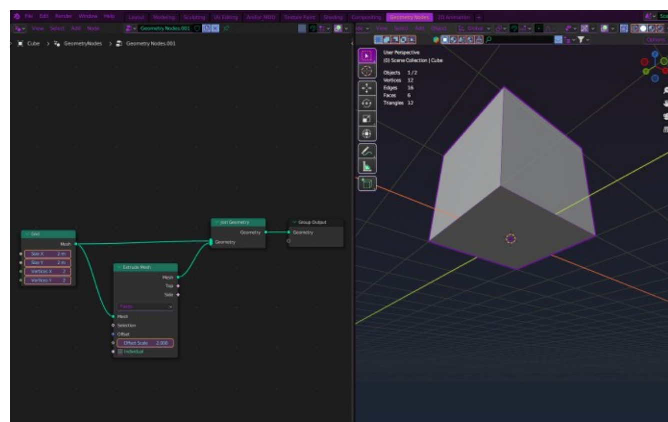

Figure 10. Join geometry node added.

However, there is yet another issue. In the top left of the 3D viewport in figure 10, it’s shown that the vertex count of this new geometry is 12, an inappropriate amount for any self-respecting cube. This is due to the grid node face and extruded mesh both have 4 vertices at z-axis 0, overlapping one another perfectly in 3D space but not physically connected to one another. This can be remedied by adding a ‘Merge by Distance’ node between the join geometry and group output node (Figure 11).

Figure 11. The merge by distance node combines vertices that are close to one another, giving the resulting mesh the correct 8 vertices needed.

The merge by distance node looks for vertices close together within an adjustable distance value and replaces them with a single vertex in the exact centre between them, while maintaining both vertices edge connections. In Figure 11 the vertex count shows the vertices overlapping at the base of the cube were merged together seamlessly. There was no concern for the position of the vertices altering, as the original vertices are exactly overlapping mathematically.

The model was exported to a .fbx, again with identical export settings. The file size came to 11,164 bytes. This is interesting, as the node tree is more complex than method 1 using the cube node, yet the file size came out with a (admittedly insignificant) 0.1% file size decrease over the geo cube method 1. It may be expected typically that a more complex geometry node tree would lead to a larger file size when exported to .fbx, as more complex geo node systems create larger .blend files. This suggests further research is necessary into possible hidden mesh data being added/removed from geometry node meshes during export. The two cubes are geometrically identical regardless. The entire geo cube node group creation process takes less than a minute.

3.2.3 Default Torus

A default torus was created exactly the same way as the previous default cube, SHIFT+A, clicking ‘Mesh’, then ‘Torus’. This resulted in a torus with a major radius of 1m and a minor radius of 0.25m, the torus has 576 vertices and faces, or 1,152 polygons in terms of tris. Note the ‘minor radius’ in this case refers to the radius of the edge loop radius (Figure 12).

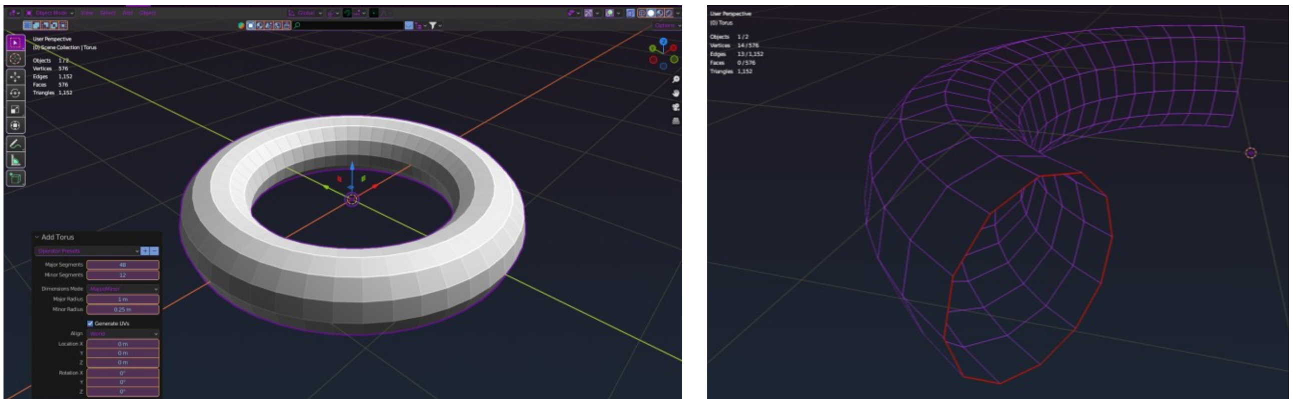

Figure 12. A cross-section of the torus showing the inner edge loop.

The default torus, when exported had a file size of 25,516 bytes or 25.5kb. Approximately double that of the default cube.

3.2.4 Geometry Node Torus

As before, all geometry node setups were created by clicking new in the geometry node workspace. Creating a torus with geometry nodes is fairly simple. There’s no ‘torus node’ as with the method1 geo cube earlier, but one can be constructed using a Curve Circle node. This node has settings for resolution (number of vertices) as well as radius (Figure 13).

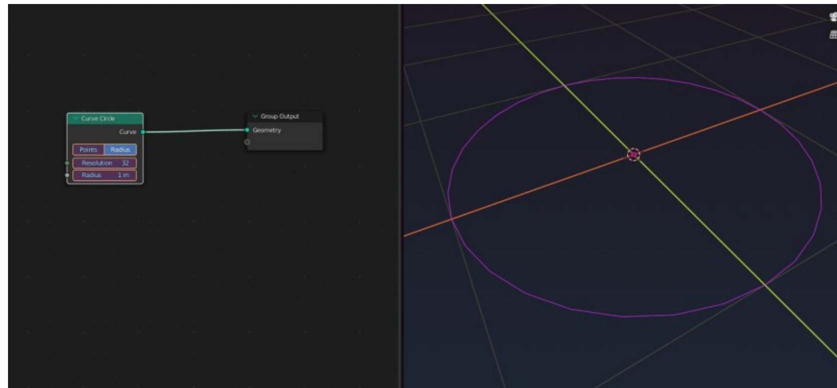

Figure 13. The curve circle node added.

Connecting the curve circle node to the Curve to Mesh node, provides the option of adding a Profile Curve to the geometry. The profile curve has a socket for another curve object node and controls the shape of the loop in (Figure 14). The resolution and radius of this profile curve was adjusted to match the dimensions, and poly count, of the default torus.

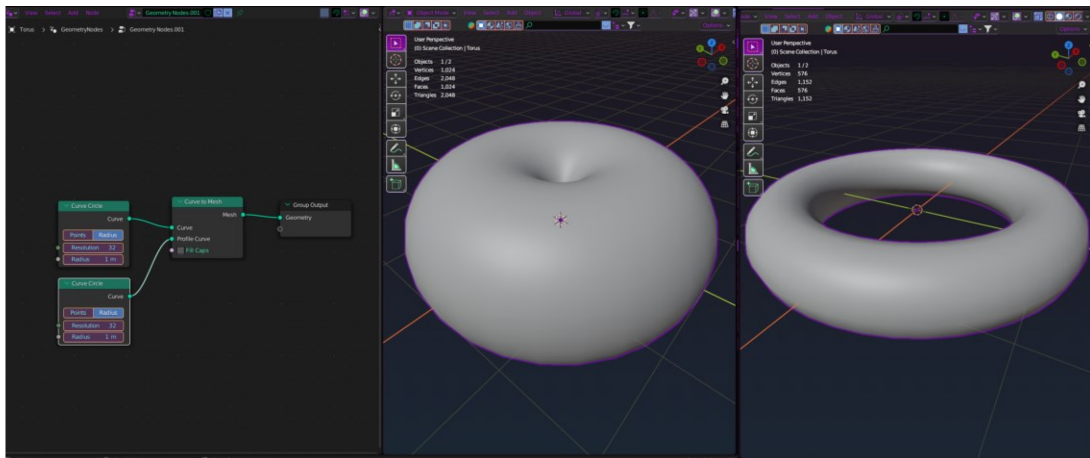

Figure 14. The profile curve being set to another curve circle node.

It’s worth noting that the torus generated with this method looked considerably different than the default torus. This was due to the ‘smooth shading’ setting being active by default in geometry node systems. This setting is disabled in the default mesh primitive objects such as the default torus.

This can be disabled by (counterintuitively) by adding a ‘Set Shade Smooth’ node to the system and unchecking the tick box on the node (Figure 15). This produces a geometrically identical torus to the default torus in both vertices and tris. The file size upon exporting was 26,764 bytes, or 26.7KB, making it 4.9% larger than the default torus.

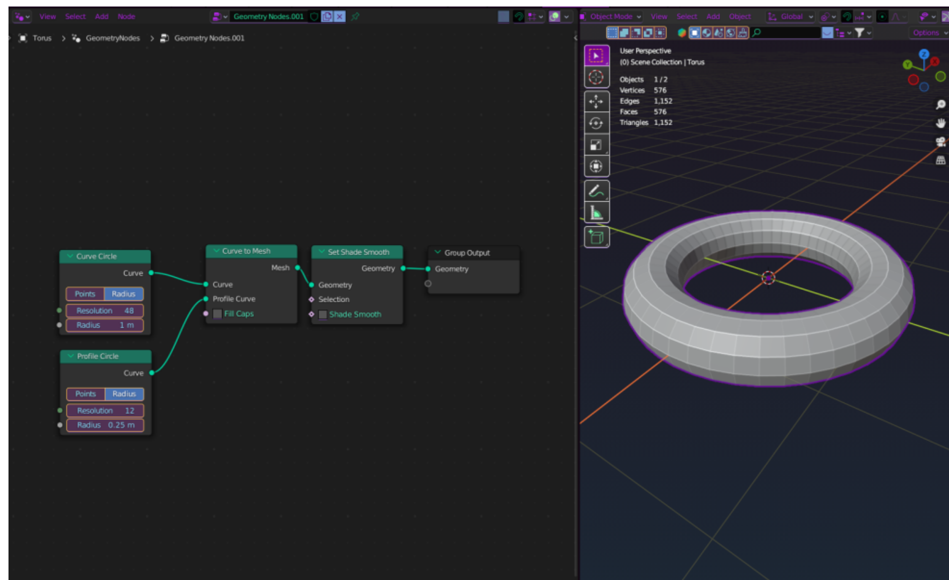

Figure 15. Using the Set Shade Smooth node to shade ‘flat’.

The entire process was simple, taking less than a minute, resulting in a torus much easier to edit than the default torus, that would have to be remade with different settings each time, or painstakingly edited by hand.

3.2.5 Particle System Comparison

For this model, 20 instances of a default cube were scattered across a plane. Note, that is instances, not duplicates. The difference being that an instance shares the identical mesh data to the original. Adjusting the original will adjust the instance they are linked by mesh data.

A duplicate on the other hand will produce an entirely separate object with its own characteristics that resembles the object only at the time of creation. A duplicate can be edited further independently of the original, into something entirely different. An instance is useful when there's a need for several of the same object to occupy the scene (such as trees or rocks).

A duplicate is useful for a number of reasons, one being to save a copy of the object before attempting a complex modelling operation that may ruin the mesh, or to create a slightly altered version of the original for some other purpose.

In any case, scattering instances of the default cube across a plane is as simple as creating a plane, creating a default cube, and holding ALT + D to create an instance of that cube (Figure 16). Repeat the last step for the desired number of instances.

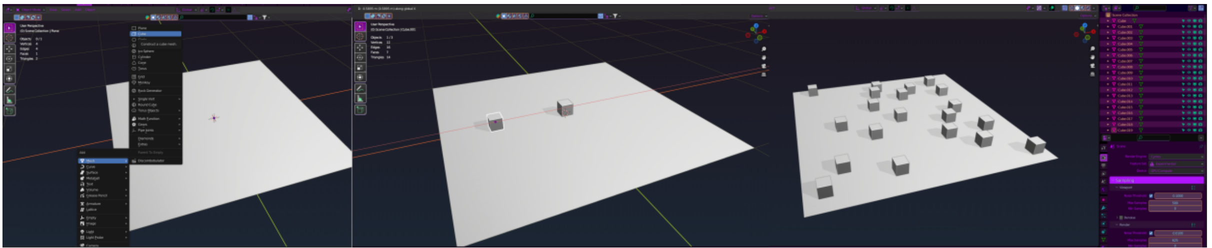

Figure 16. Manually scattering instances (or particles) across an arbitrary plane.

Holding SHIFT + Z while holding an instance can lock the Z axis position, making it easier to place on the plane at the same heigh as the original. This was done 19 times, creating 20 cubes, and one plane object. The original default cube was scaled to 0.05 its original size to fit, and the scattering placement as random as any human author placing them can be considered random. The resulting file size was 26,204 bytes or 26.2KB. As a brief side note, if the default cubes were duplicates as opposed to instances, the file size would be 44.2KB approaching twice the size.

3.2.6 Geometry Node Particle System

For the geometry node version of this same model a slightly different technique was used. Firstly, a grid node was used to generate a plane, and adjusted to the same size as the previous example. Then, a ‘Distribute Points on Faces’ node and ‘Instances on Points’ were added in tandem to the chain. The first scatters a series of points across the surface of the plane. While not inherently useful, the position of these points can be used as reference for where to scatter instanced objects. The second, does just that, and has a socket for ‘Instance’. A Cube node is connected to this (Figure 17).

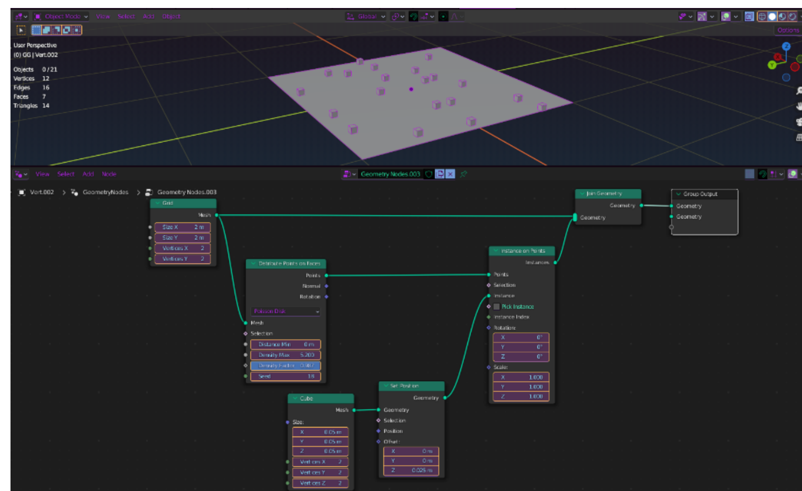

Figure 17. Initial scattering set up and plane generation.

Using a ‘Set Position’ node allows the user to alter the position of the cube before it is instanced, raising it by exactly 0.025 on the Z axis (half its dimensions) so the cubes sit perfectly on the plane. Which they did, once connected by a join geometry node as seen before in the geo cube example. It wasn’t discovered until later however, that this wouldn’t work in Unity as expected, importing only the plane with scattered cubes.
After going back and examining the node tree the mistake was clear. The instances cannot be imported to Unity the way typical geometry can be, they first need to be made ‘real’ as in actual geometry as opposed to referenced

instances of geometry. This is done easily with a ‘Realise Instance’ node added either before or after the join geometry node (Figure 18).

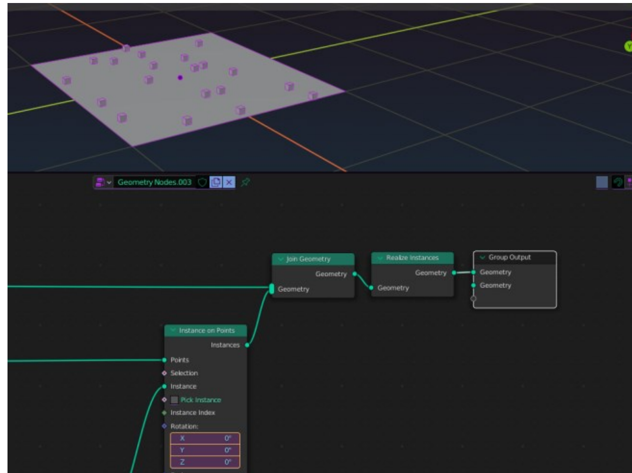

Figure 18. Realise instances node.

This ‘confirms’ the geometry of the instances as something the user desires to export. The file size of the .fbx was 13,548 byes or 13.5KB, an extraordinary 48.3% decrease from the traditional version. The reason for this is difficult to ascertain, it may be due to the way Blender manages objects. Every instance made manually, shares the same mesh data, but is still a separate *object*. Objects in blender are comparable to folders, containers that store mesh data, animation data, material data etc, within them. Despite this an object contains its own inherent data, storing its position, rotation, scale etc. Instances in this way can share mesh data, but have different positions stored as object-data.

The way geometry nodes stores instances, means they are generated as a part of the original object. Due to this only one object container is necessary with only one set of position, rotation, and scale data. When given the realise instances node, it is exported as a singular mesh, containing both the plane and the scattered cubes. Unity imports this the same way, as a singular object.

3.2.7 Geometry Node Helmet

The node tree necessary to develop a more complex object, such as a helmet, is significantly more labyrinthine than the previous used geometry systems. Such a system was created with a series of features, that were unnecessary strictly towards the basic shape, but provide useful for demonstration purposes. The system created required 90 unique nodes, although 14 group input nodes were duplicated for visual clarity (Figure 19).

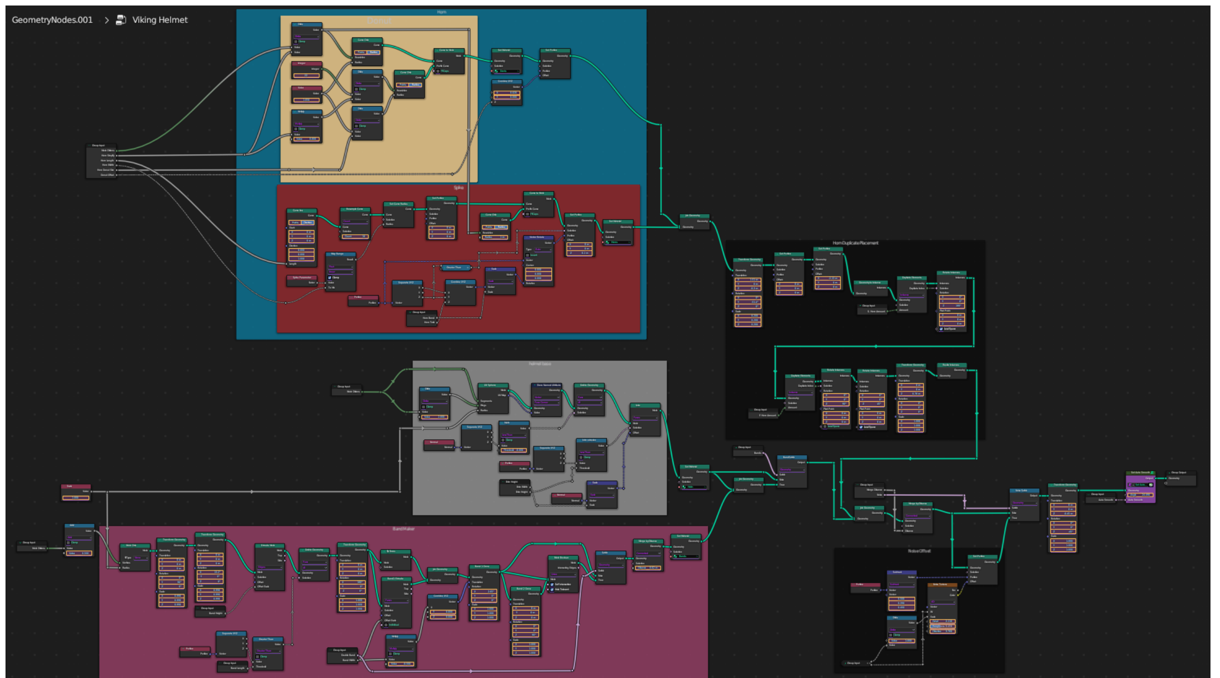

Figure 19. While flawed, this geo node system can generate a practically infinite number of different helmets.

Although discussing the (rather lengthy) creation of the entire node tree may be beyond the scope of this research, needless to say several of the previously mentioned technique were employed to create a simple horned helmet.

As seen in the grey section of the node tree (Figure 20), the general domed shape of the helmet was developed, and a thicker brim added. This was done by adding a UV sphere node and deleting the lower half of it with a delete geometry node, resulting in a half-sphere. The deletion is chosen by examining

the normal of each point on the sphere and filtering them out by whether the Z component of that normal vector is less than zero.

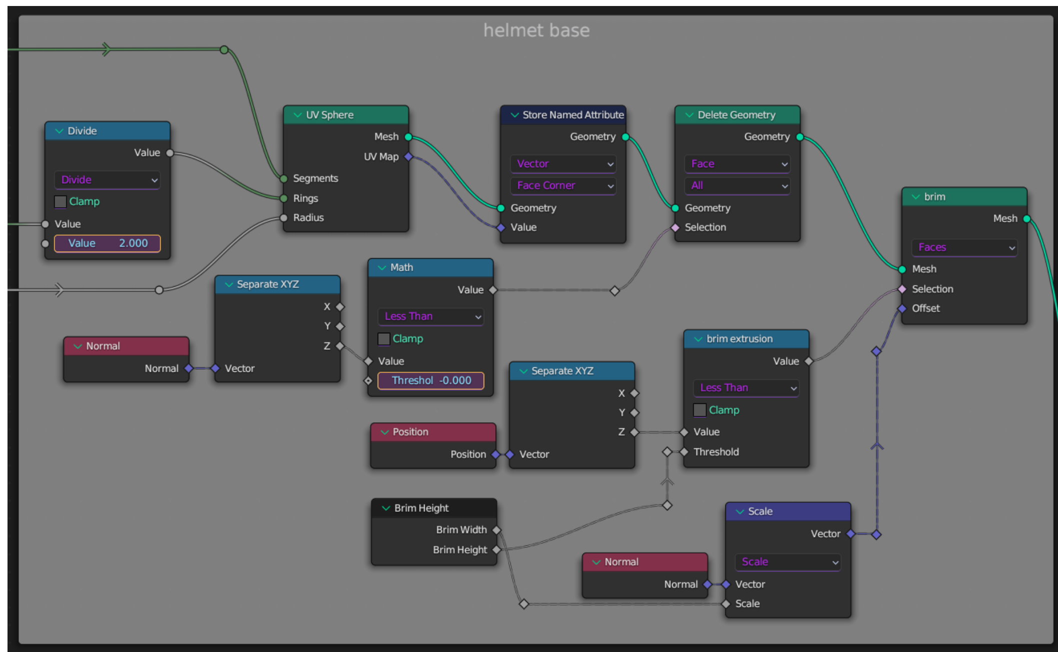

Figure 20. A closer look at the basic dome shape generation.

A brim is added similarly using an extrusion node (labelled above as ‘brim’) and the selection made using again, by comparing the z component of a vector, this time the position of each point. The result is as follows (Figure 21):

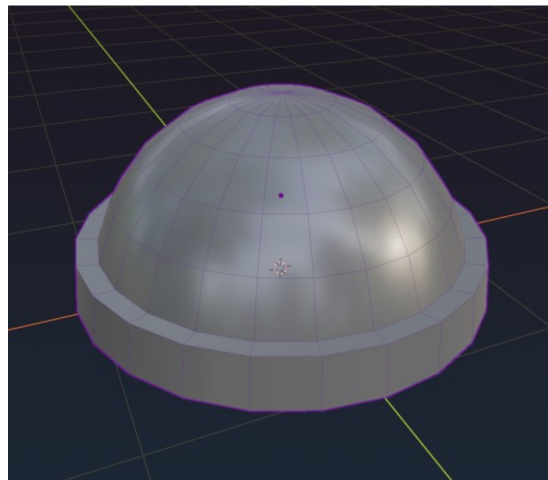

Figure 21. A basic helmet.

Note that in some figures, the helmet model has a metallic-like material assigned to it, this was for visual clarity during the demonstration process. All materials were removed before importing to Unity so as not to effect testing results.

Applying many of the same techniques, extra decorative elements were added to the geometry, bands across the top of the helmet and horns. It's possible to 'expose' geometry node input fields to the geometry node modifier tab section, by connecting any input value to a group input node (Figures 22,23).

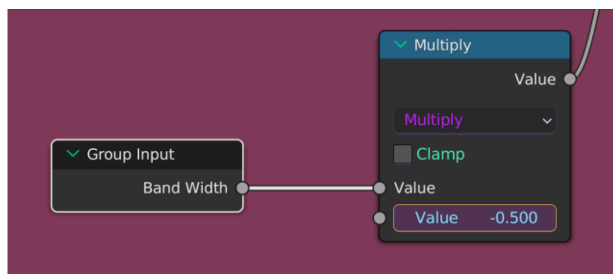

Figure 22. The Group Input node.

Here, the first input of this multiply node has been 'exposed' to the modifiers tab for easier editing. It can also be renamed to something more easily recognisable, in this case Band Width.

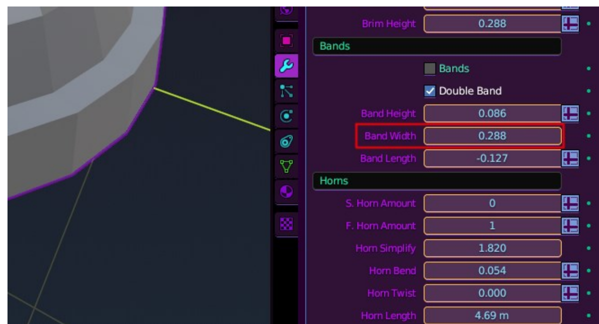

Figure 23. The exposed parameters available for faster editing.

In any case, the entire geometry node tree once connected to the group output produces a variety of different helmets (Figure 24).

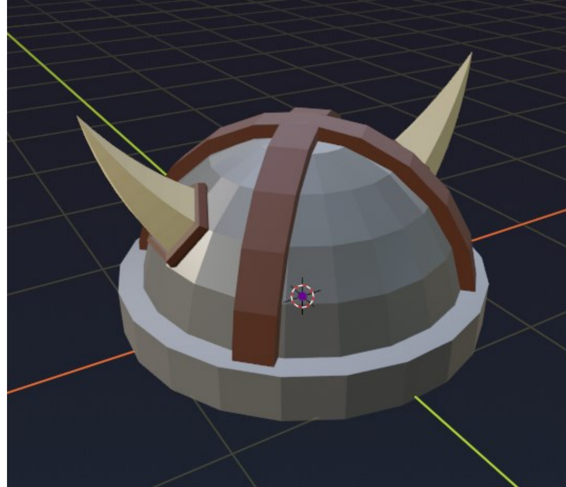

Figure 24. A helmet variant.

Once exported using the usual settings, and having stripped all materials, the resulting .fbx size was 56,476 bytes or 56.5KB. All subsequent helmet models were exported using the same helmet shape and polycount for testing uniformity.

3.2.8 ‘Applied’ Geometry Node Helmet

Simply by applying the geometry nodes modifier before exporting a new mesh is created and exported to a .fbx file. This realises the geometry into typical geometry that can be manipulated in edit mode alone. Interestingly, gave a slightly different result of 56,492 byes, a minute 0.03% increase in size.

This change in size, however slight, could be due to a number of reasons. The export options being used for all model automatically applies any modifiers, including the geometry nodes system. (Figure 25)

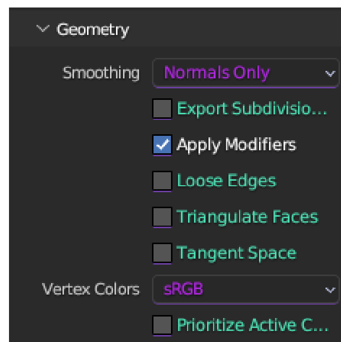

Figure 25. An excerpt from the export settings.

The results were replicated multiple times and the minor file increase persisted. On simpler geometry node systems such as a simple cube, this was not the case. This is an interesting difference, but a difference this slight could be due to a few factors, meta data for example, it’s possible that a geometry node mesh contains less of it, or it could be the automatic modifier application at export is marginally more efficient than a manual application. However, the difference is fairly negligible.

3.2.9 Modelling Adjustment Helmet

A problem with the geometry node helmet, while visually appealing, it does have topology issues. The bands intersecting at the top of the helmet, for example, do not accurately merge together, instead leaving overlapping faces and intersecting edges (Figure 26).

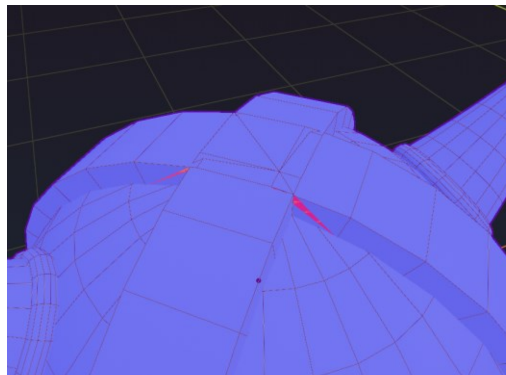

Figure 26. Intersecting edges are a mark of poor topology.

Once again by duplicating the geo node helmet and applying the modifier, a new mesh was created. This time, instead of beginning the modelling process from scratch or a mesh primitive such as a sphere or cube, the geo node helmet was used as a base. An argument could be made this was more retopology than modelling, but this technique was used to ensure an almost identical

polycount to the original geo node mesh, which was 2040 tris. After completion the topology of the mesh was far cleaner, and a distinct effort was made to match the tri count back to that of the original, subdividing some innocuous inner faces (Figure 27).

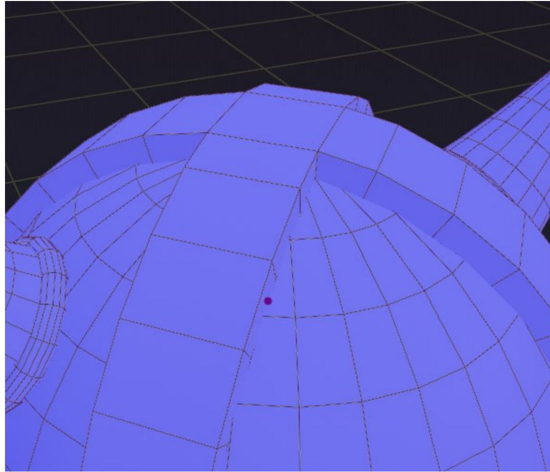

Figure 27. The same mesh, but the topology has been cleaned up to avoid overlapping edges.

Generally speaking, cleaner topology is desirable, as it helps the distortion of meshes that require animating. Furthermore, lighting errors can occur when the model has poor topology, especially with regards to flipped normals and overlapping faces or edges (Gregory, 2018). Even within Blender, certain modifiers will fail to function correctly if the topology of the model is too abstract (Figure 28).

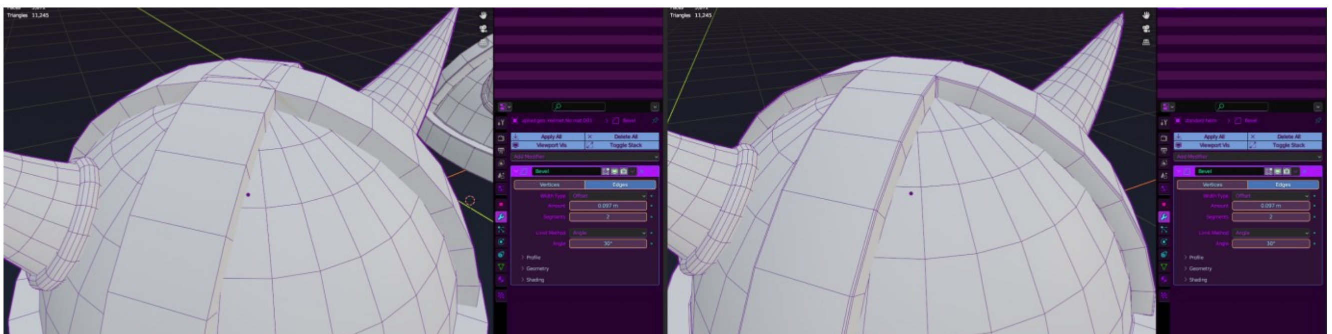

Figure 28. The same bevel modifier being applied to both models, only to fail with the poor topology of the geo node helmet (left).

√Note that commonly applied modifier is the ‘Subdivision Subsurface’ modifier, that helps to smooth the geometry, while rounding out sharp corners on a mesh. While versatile, this modifier can fail or provide unexpected/unwanted results if the geometry is topologically disjointed or contains stray vertices poorly connected to the rest of the mesh etc (Figure 29). The geo node helmet created has this problem, whereas the remodelled version functioned as expected:

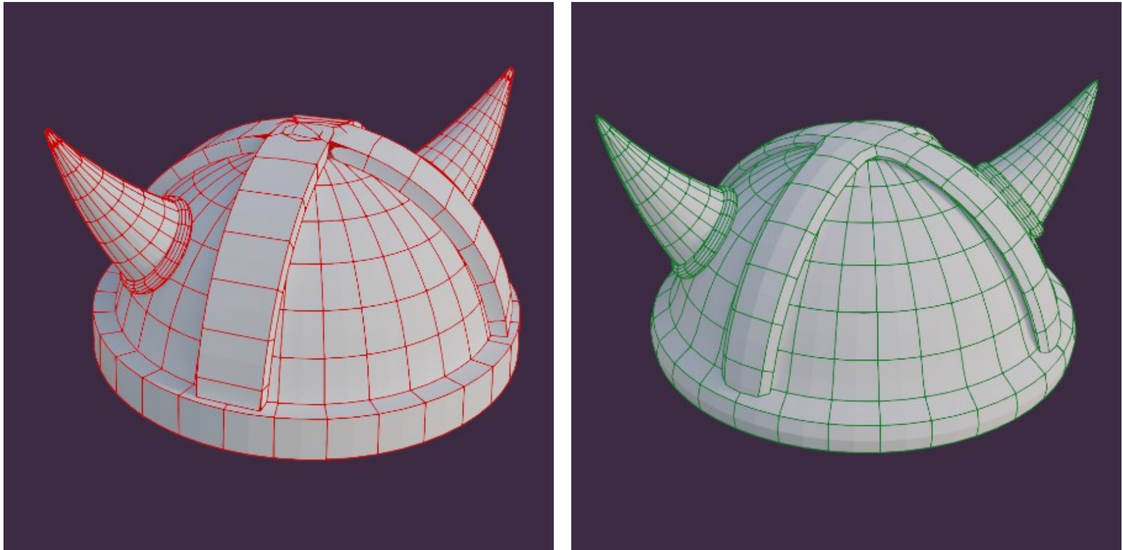

Figure 29. The topology on the right allows the subdivision modifier to smooth out the corners of the geometry for a pleasing aesthetic.

The file size for the modelled helmet without materials came to 56,668 bytes, or an almost imperceptible 0.3% increase from the geometry node mesh, the face count is marginally lower, the geo node helmet having a polycount of 2040, and the edited version just slightly fewer with 2035.

3.3 Methodology – Unity Testing Process

The performance of the models, cube, torus, helmet, and the plane with cubes distributed across its surface, were assessed via their file sizes, and by using the Unity profiler to measure CPU and GPU usage in milliseconds-per-frame (Figure 30).

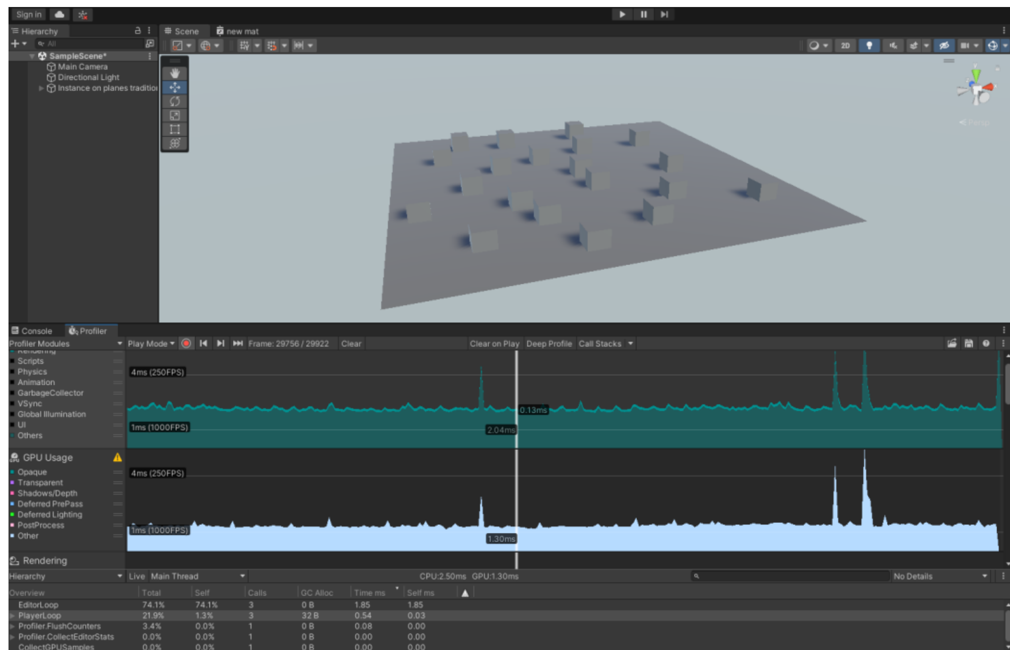

Figure 30. Unity’s built-in profiler.

The tests were performed three times for each model, and three times for a bare scene containing no meshes. This results in two devices with 33 tests each, for a total of 66.

The testing environment contained nothing beyond the default directional light and main camera. The mesh in question was placed in the scene and duplicated until there were 100 copies of the model in place.
Once running the scene uninterrupted, with no movement from the camera, objects, or even the Unity editor, for a few seconds, the program was paused in-editor. A note was made of the CPU/GPU approximate lowest point and approximate highest point over a 20 second period. These two values were

then averaged for a general result of the test, and the three tests were also averaged for a general result for each mesh. As an example, here (Figure 31) the baseline test for the empty environment on the Home PC can be seen:

	TEST 1	TEST 2	TEST 3	
(ms)	Baseline	Baseline	Baseline	TOTAL AVERAGE (ms)
CPU low	3.10	2.81	2.85	
CPU high	3.49	3.44	3.30	CPU
Av.	3.30	3.13	3.08	3.17
GPU low	1.06	1.05	1.05	
GPU high	1.55	1.52	1.54	GPU
Av.	1.31	1.29	1.30	1.30

Figure 31. Baseline CPU/GPU testing.

In test 1, the CPU high was 3.49 milliseconds, meaning how many milliseconds the CPU needed to spend on running the scene, while also taking a high of 1.55 milliseconds for the GPU to render the visuals during that same test.
The average across all three tests of an ‘empty’ scene causing a computational cost per frame of 3.17 milliseconds for the CPU and 1.3 milliseconds for the GPU. It’s desirable for this number to be as low as possible, to improve performance. The longer the device needs to compute the scene/mesh per each frame the slower performance will be overall. If the device is able to quickly render the frame and move on to the next, the framerate of the application will appear smoother while running.

4 Results

Displayed here are all results for each Unity profiler test for each mesh. The results are given in milliseconds-per-frame (ms), or how long the CPU/GPU takes per frame to calculate the particular scene. Table 1. The results from the home pc:

HOME PC				
	TEST 1	TEST 2	TEST 3	
(ms)	Baseline	Baseline	Baseline	TOTAL AVERAGE (ms)
CPU low	3.10	2.81	2.85	
CPU high	3.49	3.44	3.30	CPU
Av.	3.30	3.13	3.08	3.17
GPU low	1.06	1.05	1.05	
GPU high	1.55	1.52	1.54	GPU
Av.	1.31	1.29	1.30	1.30
	TEST 1	TEST 2	TEST 3	
(ms)	Def.Cube	Def.Cube	Def.Cube	TOTAL AVERAGE (ms)
CPU low	3.13	3.17	3.21	
CPU high	3.79	3.58	3.57	CPU
Av.	3.46	3.38	3.39	3.41
GPU low	1.08	1.11	0.99	
GPU high	1.36	1.40	1.22	GPU
Av.	1.22	1.26	1.11	1.19
	TEST 1	TEST 2	TEST 3	
(ms)	Geo.Cube	Geo.Cube	Geo.Cube	TOTAL AVERAGE (ms)
CPU low	3.17	3.16	3.18	
CPU high	3.27	3.44	3.75	CPU
Av.	3.22	3.30	3.47	3.33
GPU low	1.16	1.14	1.20	
GPU high	1.45	1.41	1.49	GPU
Av.	1.31	1.28	1.35	1.31
	TEST 1	TEST 2	TEST 3	
(ms)	Geo.Cube 2	Geo.Cube 2	Geo.Cube 2	TOTAL AVERAGE (ms)
CPU low	3.16	3.19	3.15	
CPU high	3.59	3.72	3.39	CPU
Av.	3.38	3.46	3.27	3.37
GPU low	1.01	1.01	1.19	
GPU high	1.26	1.25	1.49	GPU
Av.	1.14	1.13	1.34	1.20
	TEST 1	TEST 2	TEST 3	
(ms)	Def. Torus	Def. Torus	Def. Torus	TOTAL AVERAGE (ms)
CPU low	3.22	3.20	3.16	
CPU high	3.57	3.35	3.40	CPU
Av.	3.40	3.28	3.28	3.32
GPU low	1.16	1.13	1.14	
GPU high	1.43	1.38	1.38	GPU
Av.	1.30	1.26	1.26	1.27

	TEST 1	TEST 2	TEST 3	
(ms)	Geo. Torus	Geo. Torus	Geo. Torus	TOTAL AVERAGE (ms)
CPU low	3.22	3.14	3.14	
CPU high	3.61	3.51	3.41	CPU
Av.	3.42	3.33	3.28	3.34
GPU low	1.12	1.14	1.13	
GPU high	1.43	1.44	1.37	GPU
Av.	1.28	1.29	1.25	1.27

	TEST 1	TEST 2	TEST 3	
(ms)	Particle Sys.	Particle Sys.	Particle Sys.	TOTAL AVERAGE (ms)
CPU low	5.14	5.40	4.96	
CPU high	5.49	5.96	5.43	CPU
Av.	5.32	5.68	5.20	5.40
GPU low	1.96	2.00	1.96	
GPU high	2.29	2.38	2.25	GPU
Av.	2.13	2.19	2.11	2.14

	TEST 1	TEST 2	TEST 3	
(ms)	Geo. Particles	Geo. Particles	Geo. Particles	TOTAL AVERAGE (ms)
CPU low	3.19	3.23	3.18	
CPU high	3.45	3.46	3.47	CPU
Av.	3.32	3.35	3.33	3.33
GPU low	0.92	0.98	0.84	
GPU high	1.20	1.26	1.15	GPU
Av.	1.06	1.12	1.00	1.06

	TEST 1	TEST 2	TEST 3	
(ms)	Edited Helm	Edited Helm	Edited Helm	TOTAL AVERAGE (ms)
CPU low	3.16	3.15	3.15	
CPU high	3.55	3.59	3.52	CPU
Av.	3.36	3.37	3.34	3.35
GPU low	1.23	1.24	1.23	
GPU high	1.47	1.48	1.47	GPU
Av.	1.35	1.36	1.35	1.35

	TEST 1	TEST 2	TEST 3	
(ms)	Geo. Helm	Geo. Helm	Geo. Helm	TOTAL AVERAGE (ms)
CPU low	3.21	3.24	3.16	
CPU high	3.54	3.45	3.55	CPU
Av.	3.38	3.35	3.36	3.36
GPU low	1.24	1.21	1.19	
GPU high	1.48	1.45	1.47	GPU
Av.	1.36	1.33	1.33	1.34

	TEST 1	TEST 2	TEST 3	
(ms)	Applied Helm	Applied Helm	Applied Helm	TOTAL AVERAGE (ms)
CPU low	3.19	3.18	3.22	
CPU high	3.57	3.48	3.54	CPU
Av.	3.38	3.33	3.38	3.36
GPU low	1.24	1.21	1.23	
GPU high	1.46	1.46	1.46	GPU
Av.	1.35	1.34	1.35	1.34

Table 2. The results from the university device:

UNIVERSITY PC				
	TEST 1	TEST 2	TEST 3	
(ms)	Baseline	Baseline	Baseline	TOTAL AVERAGE (ms)
CPU low	3.26	3.32	3.34	
CPU high	3.76	3.69	3.76	CPU
Av.	3.76	4.14	4.04	3.98
GPU low	0.97	0.99	0.91	
GPU high	1.86	1.02	1.01	GPU
Av.	1.12	0.66	1.36	1.05
	TEST 1	TEST 2	TEST 3	
(ms)	Def.Cube	Def.Cube	Def.Cube	TOTAL AVERAGE (ms)
CPU low	3.67	3.74	3.65	
CPU high	3.84	4.54	4.42	CPU
Av.	3.76	4.14	4.04	3.98
GPU low	0.84	0.41	0.95	
GPU high	1.40	0.91	1.77	GPU
Av.	1.12	0.66	1.36	1.05
	TEST 1	TEST 2	TEST 3	
(ms)	Geo.Cube	Geo.Cube	Geo.Cube	TOTAL AVERAGE (ms)
CPU low	4.20	3.68	3.69	
CPU high	5.89	4.03	4.02	CPU
Av.	5.05	3.86	3.86	4.25
GPU low	0.45	1.01	1.02	
GPU high	1.45	1.64	1.79	GPU
Av.	0.95	1.33	1.41	1.23
	TEST 1	TEST 2	TEST 3	
(ms)	Geo.Cube 2	Geo.Cube 2	Geo.Cube 2	TOTAL AVERAGE (ms)
CPU low	3.65	3.65	3.67	
CPU high	3.99	3.99	4.05	CPU
Av.	3.82	3.82	3.86	3.83
GPU low	1.05	1.04	1.04	
GPU high	1.83	1.87	1.82	GPU
Av.	1.44	1.46	1.43	1.44
	TEST 1	TEST 2	TEST 3	
(ms)	Def. Torus	Def. Torus	Def. Torus	TOTAL AVERAGE (ms)
CPU low	3.68	3.65	3.64	
CPU high	4.31	4.25	4.44	CPU
Av.	4.00	3.95	4.04	4.00
GPU low	1.08	1.07	1.08	
GPU high	1.86	2.06	1.80	GPU
Av.	1.47	1.57	1.44	1.49
	TEST 1	TEST 2	TEST 3	
(ms)	Geo. Torus	Geo. Torus	Geo. Torus	TOTAL AVERAGE (ms)
CPU low	3.61	3.71	3.64	
CPU high	3.93	4.71	3.95	CPU
Av.	3.77	4.21	3.80	3.93
GPU low	1.08	1.06	1.06	
GPU high	1.85	1.88	1.79	GPU
Av.	1.47	1.47	1.43	1.45

	TEST 1	TEST 2	TEST 3	
(ms)	Particle Sys.	Particle Sys.	Particle Sys.	TOTAL AVERAGE (ms)
CPU low	5.97	6.04	5.90	
CPU high	6.97	7.22	6.52	CPU
Av.	6.47	6.63	6.21	6.44
GPU low	1.96	1.77	1.90	
GPU high	2.78	2.62	2.81	GPU
Av.	2.37	2.20	2.36	2.31

	TEST 1	TEST 2	TEST 3	
(ms)	Geo. Particles	Geo. Particles	Geo. Particles	TOTAL AVERAGE (ms)
CPU low	3.66	3.77	3.72	
CPU high	4.22	3.97	4.21	CPU
Av.	3.94	3.87	3.97	3.93
GPU low	0.79	0.76	0.83	
GPU high	1.57	1.29	1.56	GPU
Av.	1.18	1.03	1.20	1.13

	TEST 1	TEST 2	TEST 3	
(ms)	Edited Helm	Edited Helm	Edited Helm	TOTAL AVERAGE (ms)
CPU low	3.62	3.74	3.67	
CPU high	3.93	4.21	3.97	CPU
Av.	3.78	3.98	3.82	3.86
GPU low	1.14	1.19	1.14	
GPU high	1.82	1.89	1.90	GPU
Av.	1.48	1.54	1.52	1.51

	TEST 1	TEST 2	TEST 3	
(ms)	Geo. Helm	Geo. Helm	Geo. Helm	TOTAL AVERAGE (ms)
CPU low	3.86	3.75	3.72	
CPU high	4.53	4.12	4.19	CPU
Av.	4.20	3.94	3.96	4.03
GPU low	1.16	1.16	1.17	
GPU high	1.91	1.76	2.12	GPU
Av.	1.54	1.46	1.65	1.55

	TEST 1	TEST 2	TEST 3	
(ms)	Applied Helm	Applied Helm	Applied Helm	TOTAL AVERAGE (ms)
CPU low	3.65	3.73	3.65	
CPU high	4.14	4.30	4.18	CPU
Av.	3.90	4.02	3.92	3.94
GPU low	1.17	1.14	1.15	
GPU high	1.86	1.80	1.79	GPU
Av.	1.52	1.47	1.47	1.49

Table 3. An abbreviated version of the results only showing the average usage across all meshes and all tests in testing order:

Mesh	Test Target	Home PC (ms)	University PC (ms)
Baseline	CPU	3.17	3.98
	GPU	1.30	1.05
Def.Cube	CPU	3.41	3.98
	GPU	1.19	1.05
Geo.Cube	CPU	3.33	4.25
	GPU	1.31	1.23
Geo.Cube 2	CPU	3.37	3.83
	GPU	1.20	1.44
Def. Torus	CPU	3.32	4.00
	GPU	1.27	1.49
Geo. Torus	CPU	3.34	3.93
	GPU	1.27	1.45
Particle Sys.	CPU	5.40	6.44
	GPU	2.14	2.31
Geo. Particles	CPU	3.33	3.93
	GPU	1.06	1.13
Edited Helm	CPU	3.35	3.86
	GPU	1.35	1.51
Geo. Helm	CPU	3.36	4.03
	GPU	1.34	1.55
Appled Helm	CPU	3.36	3.94
	GPU	1.34	1.49

Table 4 and Table 5. The same table organised by greatest CPU processing time for both devices:

Mesh	Test Target	Home PC (ms)
Particle Sys.	CPU	5.40
Def.Cube	CPU	3.41
Geo.Cube 2	CPU	3.37
Geo. Helm	CPU	3.36
Appled Helm	CPU	3.36
Edited Helm	CPU	3.35
Geo. Torus	CPU	3.34
Geo.Cube	CPU	3.33
Geo. Particles	CPU	3.33
Def. Torus	CPU	3.32
Baseline	CPU	3.17

Mesh	Test Target	University PC (ms)
Particle Sys.	CPU	6.44
Geo.Cube	CPU	4.25
Geo. Helm	CPU	4.03
Def. Torus	CPU	4.00
Def.Cube	CPU	3.98
Baseline	CPU	3.98
Appled Helm	CPU	3.94
Geo. Particles	CPU	3.93
Geo. Torus	CPU	3.93
Edited Helm	CPU	3.86
Geo.Cube 2	CPU	3.83

Table 6 and Table 7. Once more, but sorted by greatest GPU processing time:

Mesh	Test Target	Home PC (ms)
Particle Sys.	GPU	2.14
Edited Helm	GPU	1.35
Appled Helm	GPU	1.34
Geo. Helm	GPU	1.34
Geo.Cube	GPU	1.31
Baseline	GPU	1.30
Geo. Torus	GPU	1.27
Def. Torus	GPU	1.27
Geo.Cube 2	GPU	1.20
Def.Cube	GPU	1.19
Geo. Particles	GPU	1.06

Mesh	Test Target	University PC (ms)
Particle Sys.	GPU	2.31
Geo. Helm	GPU	1.55
Edited Helm	GPU	1.51
Def. Torus	GPU	1.49
Appled Helm	GPU	1.49
Geo. Torus	GPU	1.45
Geo.Cube 2	GPU	1.44
Geo.Cube	GPU	1.23
Geo. Particles	GPU	1.13
Def.Cube	GPU	1.05
Baseline	GPU	1.05

Table 8. Here are the specifications for the file size of each mesh, including vertex and polycount (in terms of triangles) at the time of export using Blender's statistics viewer. Note that the traditional Particle System instance changes polycount when imported to Unity (discussed below).

Mesh	File Size (bytes)	Vertices	Polycount ()
Apply Helm	56492	1988	2040
Geo. Helm	56476	1988	2040
Edited Helm	56668	1961	2035
Geo. Particles	13548	164	242
Particle Sys.	26204	164	242
Geo. Torus	26764	576	1152
Def. Torus	25516	576	1152
Geo.Cube 2	11164	8	12
Geo.Cube	11180	8	12
Def.Cube	11708	8	12

5 Discussion – Geometry node performance

What follow is a brief examination into the CPU/GPU results, before an overall evaluation.

5.1 CPU

Overall, the results are quite typical. Solely in terms of CPU stress it indicates that the manual particle system model was the most computationally demanding, which aligns with expectations, since .fbx files also take note of the number of objects and stores their individual object data. When importing, Unity honours this style, and imports the objects as individuals in a prefab package.

This results in the geometry node version of the particle system being much less computationally demanding. There are no longer individual objects, all of the instanced cubes and the plane are combined to be one mesh.

Table 9. All CPU results across both devices, including an average across both.

Mesh	Test Target	Home PC (ms)	University PC (ms)	Cross Average
Particle Sys.	CPU	5.40	6.44	5.92
Geo.Cube	CPU	3.33	4.25	3.79
Geo. Helm	CPU	3.36	4.03	3.69
Def.Cube	CPU	3.41	3.98	3.69
Def. Torus	CPU	3.32	4.00	3.66
Appled Helm	CPU	3.36	3.94	3.65
Geo. Particles	CPU	3.33	3.93	3.63
Geo. Torus	CPU	3.34	3.93	3.63
Edited Helm	CPU	3.35	3.86	3.61
Geo.Cube 2	CPU	3.37	3.83	3.60
Baseline	CPU	3.17	3.98	3.57

From Table 9, it can be clearly seen that the particle system is the highest demanding asset, and upon examining the rest of the results it can be surmised that there isn't a tremendous disparity across all models in terms of CPU load. The standard deviation for the distribution (discounting the particle system entry) is 0.8 ms, a fairly insignificant time loss when comparing the polycount of the model.

There are a few unexpected occurrences, a simple object such as the default cube is shown to take more time-per-frame to render than the applied geometry nodes helmet, which has 16900% more polygons. This is most likely due to inaccuracies in the testing procedure, discussed below.

Another unexpected outcome is that the Home PC ran all scenes with slightly better performance, using less CPU/GPU processing time per frame per scene. This is unexpected as the Home PC has an older GPU with less memory available. This could be due to a number of reasons, but the hypothesis is that the university machine contains a greater number of background processes/network processes consuming memory resources.

As stated, the difference between all of the meshes is minimal in any case, but to further confound the issue, when profiling in Unity, the CPU load fluctuates naturally over time even in a completely empty scene, as can clearly be seen in the profiler graph (Figure 31).

Figure 31. Fluctuations in memory load even when rendering a blank scene.

In Figure 31, the profiler graph shows visually the milliseconds per frame, the top graph representing the CPU usage, and the lower the GPU usage. Various spikes in performance are natural for a number of reasons, Unity in itself has background process running for purposes such as database refreshing, and garbage collection. It’s not possible to run the profiler without fluctuations.

Selecting the ‘lowest’ and ‘highest’ CPU usage when this graph is paused poses a further challenge, as the two must be selected by eye from observing the dips and peaks of the graph and choosing what may appear to be the lowest/highest point. For this purpose, the tests were performed three times per model per device and averaged to help lower the inaccuracy, but the background fluctuations mean there will almost always be cause for some discrepancies. This background process discrepancy (and author-selection discrepancy) holds true for the GPU testing as well.

5.2 GPU

In terms of GPU the results followed a similar trend, but with an even lower standard deviation of 0.3 ms per frame, when averaging the two series of tests across both machines.

Table 10. All GPU results across both devices, including an average across both.

Mesh	Test Target	Home PC (ms)	University PC (ms)	Cross Average
Particle Sys.	GPU	2.14	2.31	2.22
Geo. Helm	GPU	1.34	1.55	1.44
Edited Helm	GPU	1.35	1.51	1.43
Appled Helm	GPU	1.34	1.49	1.41
Def. Torus	GPU	1.27	1.49	1.38
Geo. Torus	GPU	1.27	1.45	1.36
Geo.Cube 2	GPU	1.20	1.44	1.32
Geo.Cube	GPU	1.31	1.23	1.27
Baseline	GPU	1.30	1.05	1.17
Def.Cube	GPU	1.19	1.05	1.12
Geo. Particles	GPU	1.06	1.13	1.10

Again, this standard deviation was reached by discounting the particle system entries.

The particle system entries are so elevated due to the difference in how Unity imports combined objects as previously discussed, so to eliminate this factor it was discounted for the purposes of accurately examining the polycount. Discounted currently, but not dismissed entirely (see below).

As with the CPU table, the meshes are very close in terms of GPU usage, and these minor differences are most likely due to graph fluctuations.

5.3 Overall

Generally speaking, the results appeared to suggest that the meshes involved where not so radically different to cause any significant variance in the CPU load. Polygon count does effect CPU load of course, and this can be shown by 'stressing' the system with extremely high density meshes (Figure 32),

Figure 32. A scene containing 3 spheres with approximately 3 million polygons each, note the profiler graphs.

While the polycount's effect on performance is related, the particular scope of this research is to compare typical modelling to geometry node asset generation, so how well does the performance compare?

It would not be unreasonable to argue that there is practically no difference computationally between traditional modelling techniques and geometry node asset generation in terms of CPU and GPU load times, assuming a regular mesh is being generated. Simply put, a cube will have 8 vertices and 12 polygons regardless of the method of generating it, and Unity should render it identically regardless of where it came from.

The differences in render speeds unsurprisingly miniscule. The results show that the geo node cube method 1 takes a tenth of a millisecond longer to render than a default cube, a difference almost completely negligible when dealing with this minute of a time frame. As mentioned, this difference is also quite likely to be caused by the previously discussed discrepancy.

When a geometry node mesh is exported to Unity, it's modifier stack is applied, leaving it an ordinary mesh, and the results accurately reflect this.

6 Discussion – Geometry Node Instances

A more discernible difference between the two types is more apparent for particle systems i.e., the scattering of multiple instances of an object across some arbitrary space. Exporting a particle system generated through geo nodes with the ‘Instances on Points’ and ‘Realise Instances’ nodes, result in a singular mesh which imports to Unity as a singular object.

However, manually (or by other means) scattering the instances in Blender, then selecting all instances and performing an export to an .fbx file, will result in Unity importing a prefab with each instance included in the prefab as a series of child objects. This causes increased CPU/GPU load compared to the singular mesh, despite the polycount remaining identical, the manually instanced particle system, generates added data for each child object in the prefab (such as its transform component, position, rotation, scale etc) and so results in a more resource-expensive asset. Note that the child objects share the same mesh data however, decidedly less expensive than if each child object had its own unique mesh.

This could be troublesome, as Generating multiple instances of an object and scattering it across a plane is one of geometry nodes most useful and versatile features for 3D renders. However, having the ability to keep the instanced quality of Unity’s prefab method could drastically reduce file size over time, although, the rendering so many child objects in Unity increase CPU/GPU load per frame.

Of course, having individual child objects has its own benefits. It’s much more versatile for the application designer to be able to adjust the scattered objects wherever necessary in the game engine, simply for aesthetic reasons or for animating the objects etc. Ultimately it would seem the decision to choose performance over file size lies with the developer.

It is still simple enough to use geometry to randomly distribute the instances, then apply the geo node modifier, and link the instances mesh data after the

fact. This effectively turns the geometry node asset into the same type of particle system as the manually placed version, but can save an immense amount of time, especially when scattering hundreds if not thousands of instances. It involves a few extra steps but retains Unity's built-in prefab functionality, discussed below.

A few attempts were made to maintain the instancing quality from within a geometry node network but as of writing this functionality appears impossible.

This process was attempted with multiple geometry nodes and the instance quality was absent using the 'Instances on Points' or avoiding the node altogether (Figure 33).

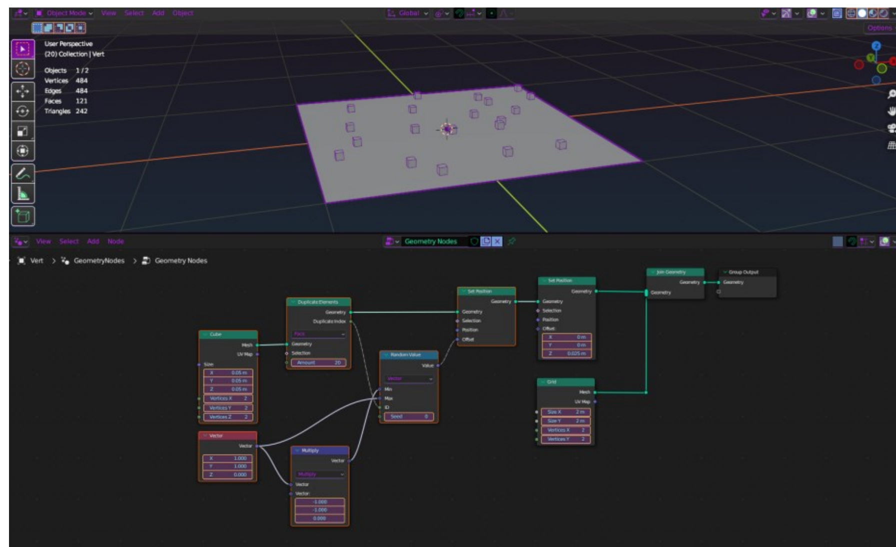

Figure 33. Cube scattering while avoiding instances, instead making uses of the 'Duplicate Geometry' node and some randomisation positioning.

While this may not translate into Unity directly, the option remains to apply the geometry node system, enter edit mode, and export all scattered objects to their own separate meshes, before lastly linking all object data to a singular mesh, and then reexport as usual to achieve the same effect. This is somewhat tedious (Figure 34), but effective at preserving Unity's prefab system.

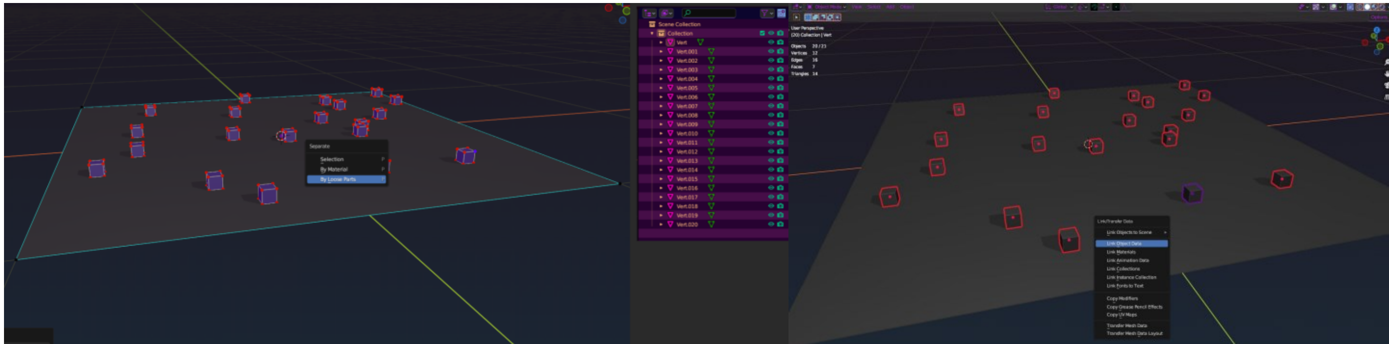

Figure 34. Using geometry nodes to make a Unity prefab-compatible file.

Making a temporary extrusion confirms all scattered objects are instances of one mesh (Figure 35).

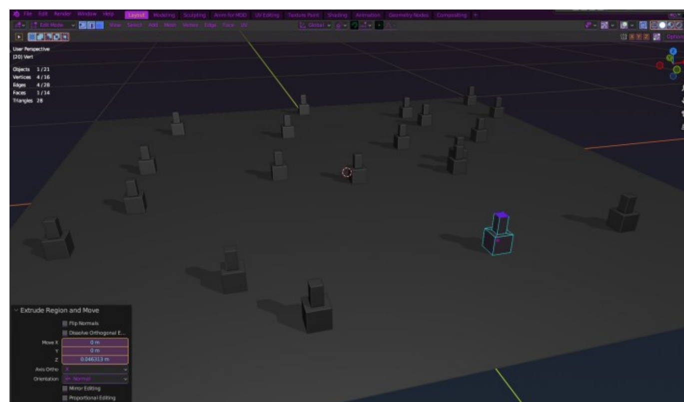

Figure 35. Extruded instances.

This particular point is rendered somewhat moot depending on how much instancing is necessary when scattering a particular object. Typically, a game designer can also instance and reposition many elements in Unity itself, however the geometry node randomisation of the position is an incredibly useful feature in its own right. This is something that can be a little awkward in Unity without instances (Harrison, 2017), (Goldberg, 2020).

Because this functionality still remains available to a geometry node-adept developer it could be seen as more advantageous to start instancing with

geometry nodes in initial scattering stage, for its procedural versatility, and if necessary, convert the system to applied instances at a later stage.

7 Evaluation

In terms of modelling game-ready assets geometry nodes can indeed be a powerful tool in a designer's toolkit. There's a cost-benefit analysis to be made however, depending on complexity, a geometry node system can take many hours to complete (Price, 2021).
A simple example is the geo node helmet system created for this research that was formed over the course of approximately 7 hours. This of course could be due a failing in the author's ability, but the result is a system that allows the generation of a practically infinite number of variations, each ready for exporting within seconds (Figure 36).

Figure 36. All of these helmets were generated in roughly 2 minutes, or approximately 10/15 seconds each.

Care should be taken to ensure topological integrity, something the helmet node tree lacks to a degree. Again, clean topology is important for certain lighting and material effects when imported to Unity although this can be ignored if graphical perfection is not required.
Once complete, the job of creating unique assets can even be given to a

developer completely lacking in modelling skills, as changing the settings is very straightforward if set up correctly, saving modeller time.

It can be surmised that using geometry nodes to create a single standard model (such as an item of detailed focus like the player character) may perhaps be a misplacement of developer resources, subject to model complexity, as the geometry node system to create such a model could easily take a hundred hours or more.

Despite this, the results indicate that using geometry nodes for generating multiple variations of simple models or background elements requiring less-than-perfect geometry is incredibly effective, generating dozens of different trees, for example. This technique could even be used to generate parts of another model, such as tire treads, to be imported into a larger model (Plush, 2018).

The results clearly show no extra load born by the system. Occasionally there can be shading errors formed from poor geometry, but this is more a case of human error.

One possible improvement for future research would be to increase the range of model complexity. In the present study, the complexity of the models was relatively low, with the simplest model consisting of only 12 triangle and the most complex having around 2450.
By expanding the range of complexity, clearer results could potentially be obtained, as this would enable a more comprehensive examination of the relationship between model complexity and performance in Unity.

Additionally, monitoring the amount of time used to create a particular geometry node tree. It could be a great boon to the developer decision making process to know exactly how advantages creating a geo node system might be for a series of meshes.

Further testing including UV mapping data, and even materials would also widen the scope of the research but provide useful information for modellers. It is simple enough to assign materials to geometry node models, the process of creating a UV map is also possible, if slightly more complex.

8 Conclusion and Recommendations

In this thesis, the impact of using Geometry Nodes in Blender on the polygon count, file size, and performance of 3D models when exported to Unity was investigated, in comparison to traditional modelling techniques. The study showed that the use of Geometry Nodes for modelling marginally increased the render time by roughly 0.2%, depending greatly on the model (excluding the particle instance model). As mentioned, this mainly appears to be an issue with the difficulties in testing, and at such time frames measured in milliseconds is most likely entirely negligible towards performance.

Furthermore, the results indicated that instancing objects in Geometry Nodes does not translate well into Unity. Additional steps must be taken for Unity to recognize that an object is meant to instance a mesh, preventing duplicate meshes from increasing the project size.

Despite the limitations, the Geometry Node system is an incredibly powerful tool with indispensable utility under the correct circumstances. It is crucial for developers to understand the limitations and assess whether Geometry Nodes is a suitable workflow for a particular series of meshes.

With the right conditions and a skilled developer, Geometry Nodes can save an extraordinary amount of time.
This research contributes to the understanding of the benefits and limitations of using Geometry Nodes in Blender for creating 3D models to be used in Unity. Further exploration could focus on optimizing the workflow between Blender and Unity for even more complicated meshes, as well as investigating the potential of Geometry Nodes in various other applications and industries.

References

[1] Anderson, C.A., Engel, G.L. and Comninos, P., 2017. A survey of real-time rendering techniques for virtual reality. Journal of WSCG, 25(2).

[2] Blender Foundation. (n.d.). Blender. Available at: https://www.blender.org/

[3] Blender Foundation. (n.d.). Geometry Nodes. Blender Manual. Available at: https://docs.blender.org/manual/en/latest/modeling/geometry_nodes/index.html (Accessed 15 April 2023).

[4] Gao, Y., Xu, J., Chen, Q., Liu, X., & Liu, L. (2020). A review on 3D mesh compression: Algorithms and evaluation metrics. Computers & Graphics, 87, 19-31.

[5] Goldberg, H. (2020). Unity in Action: Multiplatform Game Development in C#. 3rd ed. Manning Publications.

[6] Gregory, J. (2018). Game Engine Architecture. 3rd ed. Boca Raton: CRC Press.

[7] Harrison, K. (2017). Mastering Unity 2D Game Development. 2nd ed. Packt Publishing.

[8] Newzoo. (2021). Global Games Market Report. Available at: https://newzoo.com/resources/trend-reports/newzoo-global-games-market-report-2021-free-version/ (Accessed: 27 April).

[9] Plush, C. (2018). Modelling car tires in Blender. Blendernation. Available at: https://www.blendernation.com/2018/08/13/modeling-car-tires-in-blender/ (Accessed: 15 April 2023).

[10] Price, A. (2021). Blender 2.9x: The Comprehensive Handbook. Blender Guru.

[11] Simonds, B. (2013). Blender Master Class: A Hands-On Guide to Modelling, Sculpting, Materials, and Rendering. San Francisco: No Starch Press.

[12] Suykens, F., Vanaken, C., & Bekaert, P. (2016). Wavelet-based LOD for real-time rendering of large 3D models. Journal of WSCG, 24(1), 35-42.

[13] Totten, C. (2019). Level Up! The Guide to Great Video Game Design. 3rd ed. Wiley.

[14] Unity Technologies. (n.d.). Performance Optimization. Unity Manual. Available at: https://docs.unity3d.com/Manual/OptimizingGraphicsPerformance.Html (Accessed: 5 May 2023).

[15] Unity Technologies. (n.d.). Profiler: A deep-dive into Unity's performance analyzing tool. Available at: https://docs.unity3d.com/Manual/Profiler.html (Accessed: 27 April).

[16] Unity Technologies. (n.d.). Unity. Available at: https://unity.com/.

[17] Wang, H., Xu, K., Liu, L., Liu, X., Zhang, J., & Dong, J. (2019). A mesh simplification method based on vertex clustering and quadric error metrics. IEEE Access, 7, 32192-32204.

[18] Wolf, M.J.P. (2017). Video Games Around the World. MIT Press.

[19] Zhang, H., Liu, H., Wang, X., & Zhang, L. (2018). 3D mesh simplification based on improved quadric error metrics. Journal of Computer-Aided Design & Computer Graphics, 30(7), 1209-1217.