VOXELISATION AND VOXEL MANAGEMENT OPTIONS IN UNITY3D

M. Aleksandrov[1*], S. Zlatanova[1], D. J. Heslop[2]

[1] UNSW Built Environment, Red Centre Building, Kensington NSW 2052, Sydney, Australia
(mitko.aleksandrov, s.zlatanova)@unsw.edu.au
[2] UNSW Public Health and Community Medicine, Kensington NSW 2052, Sydney, Australia
d.heslop@unsw.edu.au

Commission IV, WG IV/9

KEY WORDS: Voxels, Voxelisation, Voxel visualisation, Voxel data structures, Mesh construction.

ABSTRACT:

Voxels have been used in various application domains successfully for the last several decades. Their main advantage is the underlying discrete data structure allowing to reliably work with surrounding voxels all the time. In this paper, capabilities of the Unity game engine for voxels management and geometry voxelisation are assessed, where 4 native solutions and 7 open-source projects written for Unity are investigated. Although many voxel-based options exist in Unity, they only deal with one part related to voxels. Therefore, the available capabilities to voxelise, visualise, structure and export voxels are combined and extended with the goal of successfully processing large 3D models and geometries. Many voxel visualisation techniques are investigated including mesh, VFX, point clouds and SVO, which have distinctive benefits in various aspects. Possibilities to structure voxels for effective management in simulations and other tasks are shown. Also, it is enabled to export voxels as point clouds and to the Postgres database for further processing, spatial analysis and distribution. One of the main conclusions is the lack of support for state-of-the-art voxel data structures, where the presented platform can easily be extended to support any. This platform can be used by people who deal with 3D discrete data, require voxelising 3D data, visualise voxels in different ways and technologies, as well as manage more efficiently sparsely occupied voxels.

1. INTRODUCTION

Voxels are commonly being investigated in many application domains including computer graphics (Amanatides & Woo, 1987), city modelling (Nießner et al., 2013), geology (Jørgensen et al., 2013), spatial analysis (Beckhaus et al., 2002), point clouds processing (Vo et al., 2015), machine learning (Poux & Billen, 2019), collision detections (Silver & Gagvani, 2000), shadow calculations (Gorte et al., 2018), visibility analysis (Aleksandrov et al., 2019), egress modelling (Gorte et al., 2019), pedestrian navigation (Bernardus Gorte et al., 2019) and so on. As we can see voxels are highly applicable and further investigations are required to better understand their characteristics and advantages.

Voxelisation is a process of transferring vector data into a structure that discretely represents geometry and semantics. Many researchers considered for voxelisation different geometric primitives such as points (Nourian et al., 2016), lines (Laine, 2013), triangles (IX & Kaufman, 2000), polygons (Kaufman & Shimony, 1987), surfaces and solid models (Schwarz & Seidel, 2010). At the same time, many voxelisation algorithms achieve different voxelisations targeting properties such as connectivity and separability. The first step in understanding what kind of voxelisation is possible to achieve is important to assess the available implemented algorithms. One of the main issues as suggested by Aleksandrov et al., (2021) is the availability of these approaches in common software and platforms for usage and testing. Storing additional information for each voxel goes in the direction of non-binary voxelisation, which is useful in preserving some properties needed for voxel rendering, management, and interaction. Therefore, it is important to assess the current capabilities of voxelisation algorithms implemented in Unity in this regard.

Many times voxels are visualised as points using different libraries such as UnityPointCloudViewer(1) and Open3D (Zhou et al., 2018). Aleksandrov et al. (2019) indicated that voxels can share common faces which can be omitted for rendering purposes. Thus, it is important to understand if better performance is possible to achieve by visualising simply all occupied voxels as cubes, by rendering only the visible voxel's faces or some other voxel-based techniques could be used allowing us to reduce the number of voxels' triangles to render a voxel model in a more light-weighted way. Thus, assessing different voxel visualisation techniques is needed.

To deal with large voxel data sets, efficient compression and decompression methods are suggested since many 3D scenes have less than 1% of occupied voxels. Thus, the most prominent approaches for visualisation of large voxel scenes rely on sparse voxel octrees (SVO) (Laine & Karras, 2010), sparse voxel directed acyclic graphs (SVDAG) (Kämpe et al., 2013) and symmetry-aware sparse voxel directed acyclic graphs (SSVDAG) (Villanueva et al., 2017, 2016). Apart from voxels visualisation, 3D spatial queries and simulations can be performed over voxels. Aleksandrov et al. (2021) indicated that for modelling more dynamic phenomena such as smoke propagations and fluid simulations the use of voxel data structures such as SPGrid (Setaluri et al., 2014) and VDB (Museth, 2013) are suggested. Therefore, it is crucial to understand if these data structures are easily available or some other data structures are more commonly utilised.

To be able to work with voxels, it is key to identify what the options are in one software, where the Unity3D game engine is selected as a testing platform. The paper consists of 4 main parts. In section 2, the most relevant voxel-based technology and projects are assessed in aspects related to the acquisition, storage, interaction, and visualisation of voxels. Section 3

* Corresponding author

(1) UnityPointCloudViewer. https://github.com/unitycoder/UnityPointCloudViewer (accessed on 27 April 2022)

explores how to combine and extend the available capabilities to achieve voxelisation and voxel visualisation of large 3D models, as well as shows how to structure and export voxels for further processing and usage. Section 4 presents the discussion and resulting conclusions, whereas section 5 proposes future directions to deal with voxels more effectively.

2. EVALUATION OF VOXEL-BASED APPROACHES IN UNITY3D

In the paper, the Unity game engine is selected as a testing platform for several reasons. Unity is one of the best 3D game engines along with the Unreal game engine. Although Unity is not a voxel-based engine, it can support the use of different data structures and provide user-specific rendering. Also, it allows application deployment to many platforms including Windows, Mac OS, Linux, web and so on. The community using the software is huge and many voxel-related projects are known in the software. The evaluation of several native solutions and open-source projects built in Unity are presented in the following subsections.

2.1 Native Unity solutions

It is possible to work with voxels in several ways based on the technology that Unity provides. 4 different approaches that can be utilised for the successful management of voxels are identified including GameObjects, DOTS, VFX graph and compute shaders (Table 1).

GameObjects are fundamental blocks in Unity to which components can be attached to hold properties that users can manipulate defining their behaviours. This approach supports standard C#-based programming, which can be highly advantageous for people not familiar with the other available approaches, allowing them to reuse many available C# libraries and code. By using GameObjects, users would work with triangulated meshes (e.g., quads, cubes and triangles) to work with voxels. One of the main issues with this approach is performance, having difficulty dealing with large scenes, many objects and computationally expensive operations.

Data-Oriented Technology Stack (DOTS) has several advantages in dealing with the issues associated with the GameObject approach. This allows users to create richer user experiences and execute faster iterative operations. To work with each object (i.e., known as an entity) users can assign components which are represented as data containers and be processed by systems holding behaviours and logic. Unity allows the effective transition and compatibility between GameObject and DOTS, as well as a partial usage of these two. This method is particularly useful to offload GPU to balance the usage of both processing units in terms of rendering and computations. To use voxels, users would need to work with meshes as well as subscenes which allows subdividing large scenes into smaller areas for effectively loading/unloading as needed. Also, all voxel-related data would be stored in components and processed by systems in specified ways. Although DOTS can handle a large number of objects it might not the best for rendering and simulations of millions of objects.

VFX graph is a powerful tool that directly uses GPU to create particle systems which can be manipulated in user-defined ways. The main advantage of this system is the ability to work with even millions of particles. In this case, each voxel would be most likely considered a cube which can be manipulated by a graph-based logic. It should be pointed out that other visual representation options for voxels such as quads and meshes are possible to use. The transition between other approaches is possible, but it can result in an additional overhead in case of requiring a frequent transfer of data to the CPU and vice versa. Therefore, it might not be the best option if the system requires loading very often data from a database or files. Apart from the option to calculate a signed distance field, other data structures like octrees are not available, which is the current main issue to manage voxels efficiently. Due to the use of specific shaders and rendering pipelines like the high definition rendering pipeline (HDRP) some platforms cannot be targeted for application export like WebGL, so the universal rendering pipeline (URP) is more recommended.

Compute shaders are shader programs that use the GPU without the need to operate on meshes or texture data, and their main usage is for a large set of calculations involving mathematics and parallelisation. Voxels are usually represented in discrete space via arrays over which some mathematical operations are performed. The main difference with the VFX graph approach is the low-level programming control. However, the main issue can be the programming knowledge required as well as the incompatibility with even more platforms.

Approach	Characteristics			
	Processing unit	Pros	Cons	Platform level compatibility
GameObjects	CPU	Full control of the scene; standard C# object-oriented programming	Not suitable for physical simulations and dealing with a lot of objects	High
DOTS	Multicore CPU	Multicore processing; simplified usage; dealing with large scenes; reduced power usage; smaller building applications	Not suitable for extremely large simulations and rendering, and compatibility with different Unity versions	High
VFX graph	GPU	Particle-based modelling; node-based visual logic; dealing with a lot of objects; suitable for creating visual effects and physical simulations	Limited programming control; availability of voxel data structures; heavily shaders usage; adjustable occlusion culling; not suitable to combine with CPU processes	Medium[(2)]
Compute shaders	GPGPU	Usage of massively parallel functions; accelerated game rendering	In-depth knowledge of GPU architecture and algorithms parallelisation	Low[(3)]

Table 1. Voxelisation approaches and voxel data structures supported in Unity

[(2)] https://docs.unity3d.com/Packages/com.unity.visualeffectgraph@14.0/manual/System-Requirements.html (accessed on 27 4 2022)
[(3)] https://docs.unity3d.com/Manual/class-ComputeShader.html (accessed on 27 April 2022)

2.2 Open-source projects in Unity

The first step to working with voxels requires their generation. As presented in the introduction section, several geometric primitives can be used for voxelisation. While voxelisation of points is relatively easy to achieve, other primitives require the use of specific algorithms for voxelisation. In the case of having sparsely represented voxels, the use of plane arrays can result in a high memory footprint. Therefore, more efficient voxel data structures are suggested (Aleksandrov et al., 2021).

Table 2 shows different voxel-ralated open-source projects available to download from GitHub. 7 projects are identified dealing with some aspects related to voxels, helping us to understand the current status and capabilities for effective management of voxels in Unity. When it comes to voxelisation options, 2 projects are identified where surface voxelisation is possible to achieve either on CPU or GPU, while solid voxelisation is possible to obtain only on GPU. 26-connected and conservative voxelisation is being targeted in these projects showing interest in different voxelisations. The first project is allowing to voxelise an object's texture, while the second project targets exclusively binary voxelisation of a 3D object.

Regarding voxel data structures, 3 of them use regular octrees to keep reference type objects storing them either as points or minimal bounding boxes (MBB). In general, octrees do not take a huge memory of a computer and they are useful if the intention is to conduct frequently some spatial queries like proximity analysis over objects which are spread around in a larger area. The Unity-SVO project uses a sparse version of an octree reducing the memory footprint even more and enabling effective rendering of large voxel scenes. For each voxel, colour can be stored as well as some integer parameters. Until recently, the OpenVDB library was also possible to use in Unity, but the project seems abandoned. This is unfortunate since the VDB data structure allows to manage effectively voxels in scenarios when voxels often update positions like in simulations.

Approach	Characteristics				
	Processing unit	Purpose	Queries	Unity version compatibility	GitHub stars
Unity-voxel[2]	CPU-based GPU-based	Conservative surface voxelisation Conservative surface and solid voxelisation	Array-based queries	Unity 2018.3.0f2	1031
Mesh-voxelization[3]	CPU-based	26-connected voxelisation	Array-based queries	Unity 2020.1.14f1	162
UnityOctree[4]	CPU-based octree	A dynamic and loose octree; allowing to store point and MBB referenced objects	MBB intersections, points within distance	all Unity versions	727
ESC-Octree[5]	DOTS-based octree	A dynamic and loose octree; allowing to store point and MBB referenced objects	MBB intersections, points within distance, raycasting	Unity 2020.1a	126
NativeOctree[6]	DOTS-based octree	allowing to store point referenced objects	Points inside a box	Unity 2019.3.0b10	89
Unity-SVO[7]	GPU-based SVO	Voxel storage and rendering	Raycasting	Unity 2019.4.21f1	74
OpenVDBForUnity[8]	CPU-based VDB	Manipulation of sparse, time-varying, volumetric data	Topological and morphological operations, etc.	Abandoned	209

Table 2. Voxelisation approaches and voxel data structures supported in Unity

3. VOXELS MANAGEMENT PLATFORM

Aleksandrov et al., (2021) presented that many geometrical primitives exist including points, lines, triangles, polygons, surfaces and volumes, where for each one of them different voxelisations can be achieved considering the application of interest. Apart from the geometrical transition to discretised shape, it is important to consider attributes preservation during the process. For the fast insert, retrieval, rendering and update of voxels and their properties, it is important to use voxel data structure. At the same time, visualisation of voxels is possible to perform in many ways considering different techniques, data structures and needs. Therefore, we identify 3 main components starting with input geometry, voxelisation, followed by data storage, visualisation and export (Figure 1). The main idea is to use the available technology that Unity provides, and the projects evaluated above to establish the support for voxelisation of various geometrical primitives, many data structures and export options. Therefore, a platform is created to deal with each of these aspects separately.

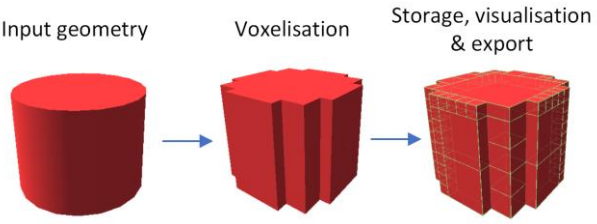

Figure 1. Processing workflow

[2] Unity-voxel. https://github.com/mattatz/unity-voxel (accessed on 27 April 2022)
[3] Mesh-voxelisation. https://github.com/Scrawk/Mesh-Voxelization (accessed on 27 April 2022)
[4] UnityOctree. https://github.com/Nition/UnityOctree (accessed on 27 April 2022)
[5] ESC-Octree. https://github.com/Antypodish/ECS-Octree (accessed on 27 April 2022)
[6] NativeOctree. https://github.com/marijnz/NativeOctree (accessed on 27 April 2022)
[7] Unity-SVO. https://github.com/BudgetToaster/unity-sparse-voxel-octrees (accessed on 27 April 2022)
[8] OpenVDBForUnity. https://github.com/karasusan/OpenVDBForUnity (accessed on 27 April 2022)

This contribution has been peer-reviewed. The double-blind peer-review was conducted on the basis of the full paper.
https://doi.org/10.5194/isprs-annals-X-4-W2-2022-13-2022 | © Author(s) 2022. CC BY 4.0 License.

3.1 Voxelisation of 3D models

As a first step, it is important to understand how to voxelise quickly not only one object, but an entire scene. At the same time, enabling voxelisation of any size voxels without having constraints in terms of available computer memory is needed.

To establish such a flexible approach, we consider several steps. The first step involves the storage of bounding boxes of objects representing a 3D model in an octree to be able to query areas which should be voxelised. Afterwards, a model is divided into 256^3 blocks which can fit in the computer's memory for voxelisation. The following step involves the identification of components belonging to each block using the bounding boxes stored in the octree. Once all objects are identified, objects' meshes are combined and sent to the voxeliser. For now, only the conservative surface voxelisation is considered using compute shaders to process voxels on the GPU. After that, voxels can be processed for visualisation and stored in an octree. The process is repeated for each block until the whole area of interest or 3D model is voxelised.

Figure 2 shows the voxelisation process of a 3D model using a voxel size of 10 cm. It can be observed that 3 blocks are needed to voxelise the model successfully. Users can voxelise only part of the building if needed at any time since bounding boxes of all objects are stored in an octree and can be queried using a 3D area. Also, to preserve semantic information during voxelisation objects are voxelised individually.

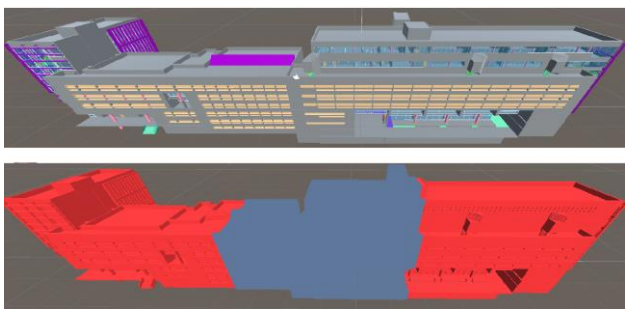

Figure 2. Binary voxelisation of a 3D model using 10 cm size voxels; upper image shows the 3D model in Unity; lower image shows 3 blocks used for voxelisation, where the middle one is highlighted.

3.2 Voxels visualisation options

Many ways exist to render voxels, where each approach has a specific aim for its usage. Reviewing the presented open-source projects and Unity technologies 4 approaches are identified as suitable for voxel visualisation, but there can be even more. These approaches include mesh, VFX, point clouds and SVO. As assessed above these approaches' purpose is not just visualisation. They can be utilised for different applications requiring simulations, physical interactions and shadow analysis.

3.2.1 Mesh-based voxels visualisation

Working with meshes represents a common way to render voxels in software dealing with 3D computer graphics. There are several ways to utilise meshed for voxels visualisation. The most obvious way is to create cubes representing the occupied voxel, where each cube will be a separate GameObject in Unity. However, this process is expensive and not many voxels can be rendered this way. A slightly better approach is to create a mesh where only one face is created if two occupied voxels are sharing it, but such faces can be even omitted since they are not visible from outside a voxelised model. Thus, such faces are omitted during the mesh construction, having fewer quads and correspondingly triangles to render.

Small voxel sizes can result in large meshes where smaller blocks/chunks can be used to construct voxel-looking meshes. The same approach can be easily extended to visualise colour-based voxelised meshes (Figure 3).

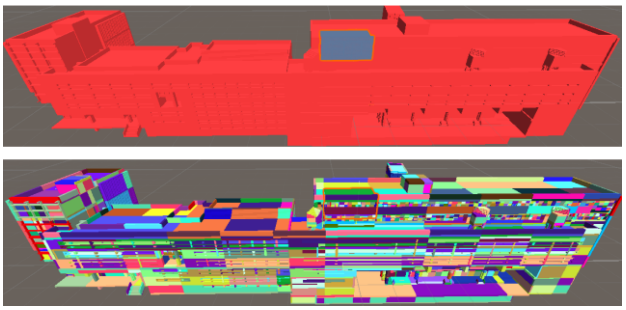

Figure 3. Voxels visualisation using blocks of meshes; upper image shows voxelised model created from smaller mesh chunks; lower image shows mesh chunks colouring each voxel based on object's id to which it belongs.

At the same time, such blocks can be utilised for occlusion culling that Unity supports. It is required to define all GameObjects (i.e., meshes) as static allowing an octree to be generated and automatically cull objects that are not visible from a camera point of view (Figure 4).

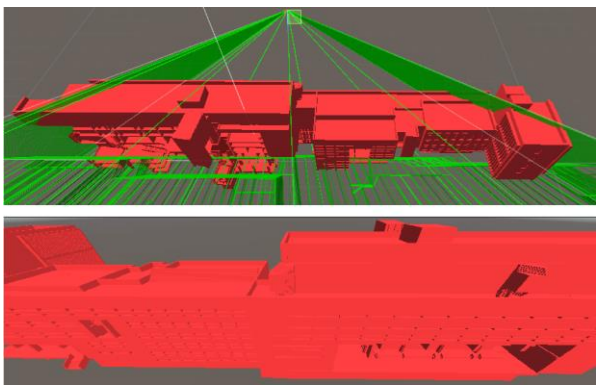

Figure 4. Occlusion culling of voxel blocks; upper image shows the rays send from the camera to see which components to turn off; lower image shows the corresponding camera view.

Once a voxel-looking mesh is constructed, the first impression is that triangles belonging to one surface can be merged to reduce their number and improve the visualisation performance. After a thorough search of a suitable method, a library[(9)] utilising quadric mesh simplification is identified (Garland & Heckbert, 1998). However, the implemented method works well (i.e., keeps the voxel-looking structure intact) only when using large voxel sizes (e.g., >50 cm) and it requires fully enclosed meshes (Figure 5).

[(9)] UnityMeshSimplifier.https://github.com/Whinarn/UnityMeshSimplifier (accessed on 27 April 2022)

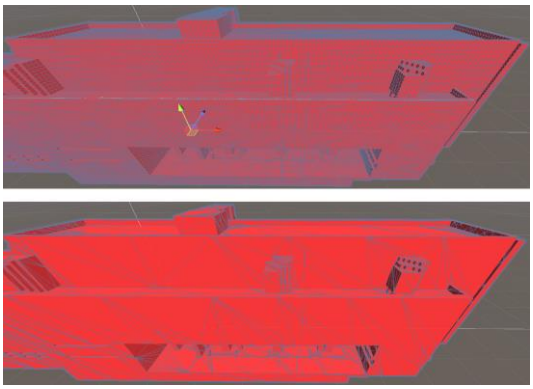

Figure 5. Mesh simplification of a voxelised model; upper image shows all triangles representing voxel visible faces; lower image shows simplified mesh.

3.2.2 VFX voxels visualisation

As explained previously VFX works with particles allowing users to directly work with the GPU performing simulations or other computations on them. Two approaches are assessed, where voxels are visualised as cubes and quads (i.e., faces) that are visible. To create voxels at specific locations their coordinates and all other information (e.g., colour, voxel size, etc.) should be transferred from CPU to GPU where the desirable objects will be instantiated. Figure 6 shows different representations of the voxels. The top image represents voxels visualised as cubes, but since there is no option to enable receiving shadow only the outline of the building voxelised model is visible. On the other hand, if quads are used visible voxels' faces can receive and cast a shadow. A graph used to achieve the quad visualisation is available in the appendix.

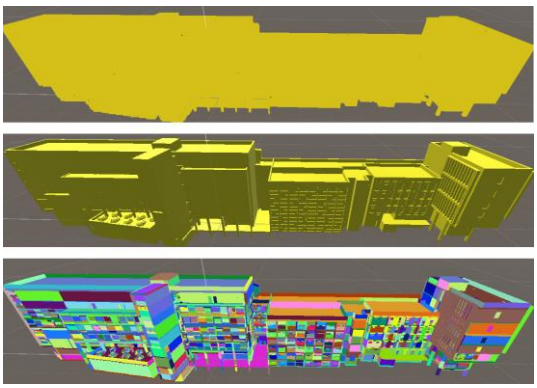

Figure 6. Voxel visualisation using VFX graph; top image shows voxel visualisation using cubes for each voxel; the middle image shows voxel visualisation using only visible quads; bottom image shows voxels considering each object to which they belong.

3.2.3 Point clouds voxel visualisation

Another way of visualising voxels is to use point clouds techniques where every voxel is displayed as a point, which is very similar to visualising them as particles. To test this, the previously mentioned UnityPointCloudViewer plugin is used. Voxels can be grouped into blocks for quick load/unload. It allows random culling of voxels based on their distance from the camera. Although this might work nicely for point clouds allowing to render larger scenes, culling randomly voxels creates a discontinuity effect (Figure 7).

Figure 7. Rendering voxels as point clouds; upper image shows all voxels as points; lower image shows some randomly culled voxels based on the distance to a camera

3.2.4 SVO visualisation

The implemented SVO in Unity is the one from Laine & Karras (2010). It allows visualising large voxel models without requiring a huge memory (Figure 8). Users can add voxels at different octree depth levels, and assign a colour and other attributes. On our computer, it is possible to render successfully a $4K^3$ size model. Out of all presented solutions, this one is the only one that is officially designed for voxels rendering and to some extent for management.

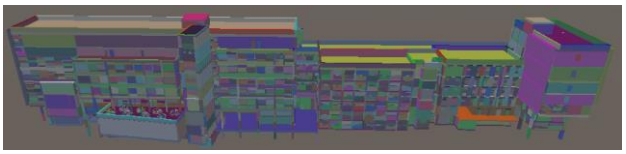

Figure 8. Voxels rendering using SVO

3.2.5 Methods comparison

Several techniques are presented for voxels visualisation in the previous sections. They can be evaluated in a few aspects to better understand their strengths and weaknesses. 10 different scenarios are selected for testing (Table 3). The first 4 are related to mesh visualisation. After that, the next 4 are considering VFX graphs to visualise voxels. Scenario 9 is testing the usage of the point clouds visualiser, while the final scenario is related to SVO usage.

As mentioned, URP is used for better performance reasons and compatibility with more platforms. Experiments are run on a computer with an Intel(R) Core(TM) i7-7600 CPU @ 2.80GHz using 16GB of RAM, Intel(R) HD Graphics 620, and running Windows 10. The project is available for download and contributions on GitHub[(10)].

As a case study 3D model of Red Centre building at the UNSW is selected. The size of 3D model is 150x28x32 meters and contains 11457 objects, being a very complex model. Update rate and creation time to rendering are measured for 20 cm (i.e., 2179434 cubes or 3032906 quads) and 50 cm (i.e., 267615 cubes or 315244 quads) voxel sizes. The selection of these sizes is arbitrary, but they perfectly show the differences between the selected methods. SVO outperformed other methods in terms of achieved frame rate, while VFX approaches accomplished the lowest frame rate. Among mesh-based methods, the one reducing the number of triangles performed the best. The method that uses the occlusion culling method achieved a smaller frequency than without using it. However, this technique might perform better if the user is inside a building,

[(10)] https://github.com/grid-unsw/Voxel-Management

and many objects ahead are easily occluded by walls. For the mesh using many colours, we create the same number of materials, which might not be the best strategy and the achieved frame rate is smaller than expected.

In contrast, VFX achieved the quickest rendering creation time. SVO requires slightly more time for construction especially if the model is large. To create the occlusion culling octree several minutes are required. The mesh simplification approach is relatively quick, achieving ~10x reduction of triangles for 50 cm voxel size (Figure 4). However, as mentioned previously it cannot handle models when the voxel size is small. The point clouds approach can be positioned somewhere in the middle regarding the creation time and update rate. It should be mentioned that the time to perform the binary voxelisation is not included in the calculations, where the voxelisation time for the 20 and 50 cm size voxels is 3.8 and 0.4 seconds, respectively.

	Voxel size 50 cm		Voxel size 20 cm	
	Update rate [fps]	Creation time [s]	Update rate [fps]	Creation time [s]
Mesh	102*	0.7	22.3**	7.3
Mesh with colours	47*	7.8	17.8**	20
Occlusion culled mesh	80*	198	22.4**	889
Simplified mesh	115	36	/	/
VFX-cube	60	1.1	12.2	1.7
VFX-cube with colour	59.2	1.4	12.1	4
VFX-quad	68	0.2	13.1	1.6
VFX-quad with colour	67.8	0.7	13	3.8
Point clouds	88	2.5	16.5	29
SVO	128	0.9	53.1	9

Table 3. Comparison of voxel visualisation approaches; * 32^3 blocks used; ** 128^3 blocks used.

3.3 Voxels export

There are several ways in which voxels can be exported. A common way is to export them as point clouds. Another option enabled is to distribute them to a spatial database over which various spatial analyses can be run. For the point clouds, only PTS is considered as an exporting option for now, whereas PostgreSQL with several PostGIS extensions is considered for exporting voxels to a database. Since Postgres does not provide support for any voxel data structure like octrees, users can export them as POINTs, pgPointcloud or Arrays (Li et al., 2020). To export voxels, at the right location, an offset and coordinate reference system should be specified in Unity. Once the voxels are stored in PostgreSQL, they can be visualised in QGIS even in 3D (Figure 9) and perform many spatial analyses.

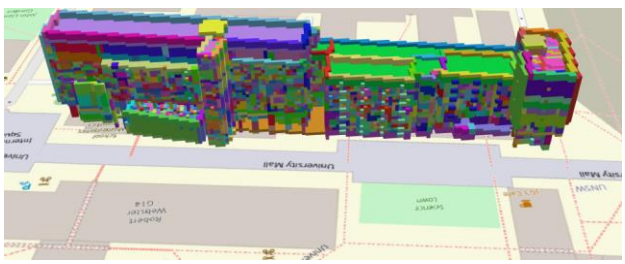

Figure 9. Visualisation of 1m voxels in QGIS via Qgis2threejs plugin

3.4 Voxels storage

As presented in section 2.2 several options exist to store and manage voxels. Surprisingly all of them rely on octrees targeting different technologies including the use of single-threaded CPUs, multi-core CPUs based on the DOTS technology as well as the GPU by using SVO.

Once the voxels are created, users can store them in any of these octree-based data structures for any further voxel management purposes or simulations. Regarding semantics, the CPU-based approaches are created to support the storage of reference type objects which are perfect for OOP, while the SVO can store colour and integer type attributes.

4. DISCUSSION AND CONCLUSION

Several native and open-source solutions are explored in the first section showing the interest in voxelisation, voxel visualisation, management and export. The most interesting fact is that many different options exist to work with voxels. The native solutions mainly concentrate on bringing the technology to programmers allowing them to explore different options for the same problem. On the other hand, the open-source projects proceed further to create context-specific solutions. Various voxelisation strategies and voxel data structures are explored. Surface and solid voxelisation are mainly explored targeting different voxelisation properties such as connectivity and coverage. The main area that is not explored in the process is attention to the preservation of semantic information. In terms of voxel data structures, octree-related structures are predominantly used. However, the latest research has not been implemented, where many great data structures such as SPGrid, VDB, SVDAG and SSVDAG are yet to be explored how they perform for different scenarios.

In section 3, voxelisation of large 3D models, 4 voxel visualisation techniques, as well as voxel storage and export are presented. Voxelising large 3D voxel models is possible to achieve in a matter of seconds. At the same time, the approach is extended to support the usage of any voxel size if the purpose is to deal with large scenes or small voxels. When it comes to voxel visualisation, several different techniques are investigated, where all techniques performed distinctively. Several methods are presented including chunking, mesh simplification and occlusion culling trying to improve the performance of using meshes in Unity. By splitting the mesh into smaller blocks, the possibility to construct large voxel-based mesh models is achieved. Mesh simplification is partially explored, but the results seem promising since the number of triangles can be reduced significantly especially if not many materials are used and there are a lot of flat areas in the model. In contrast, the occlusion culling did not bring any performance improvement for the given scenario if the full model is to be visualised, and it can be expensive to construct, but it might be interesting to consider for indoor navigation where some parts will be culled out from the camera forward view. VFX performed the best in terms of the time needed to create the rendering of voxels. Voxels can be rendered as cubes or quads. For quads shadow casting and receiving is possible to enable, which is not the case with the cubes. We should mention that the creation of mesh on VFX is not explored which can increase the frame rate performance due to the reduction of mesh draw calls. Although VFX allows direct usage of GPU, the main issue is the lack of support for voxel data structures which will allow storage and management of voxels. The use of point clouds techniques to render voxels can be used, but most likely

it is not the best strategy to consider. The one that is explored here renders voxels as individual points, which might not be the best way if voxels are condensed in one place as in our example. Also, point clouds can use octrees to store and render many points, becoming similar to the SVO approach. Using SVO users can reliably render huge voxel models without taking a large portion of the computer's memory. However, the construction time starts increasing with large models, which users should bear in mind if the intention is to consider dynamic scenarios where many voxels need position update within the octree.

Regarding the storage and management of voxels in Unity, the underlying data structure needs to support the storage of reference type objects which can be updated via other scripts out of the data structure but still be reflected in the data structure itself.

It is possible to export voxels as point clouds and to a PostgreSQL database, allowing to further process and work with voxels on other platforms. However, in both cases, voxels can be exported predominantly as points, which are not native containers for voxels storage.

5. FUTURE RESEARCH

At the moment conservative surface and solid voxelisation of 3D models are explored, but it would be nice to provide support for all geometrical primitives including points, lines, curves, triangles, polygons, and solids with the intention of attribute preservation during the voxelisation process. Also, different voxelisations including 26- & 6-connected should be possible to achieve using not only GPU but also CPU.

As mentioned previously, the available voxel data structures are not suitable if voxels move their positions as in simulations. Therefore, bringing the OpenVDB library for the processing and storage of voxels can bring another dimension to the platform. In the future, it would be nice to implement the other mentioned data structures to speed up the voxelisation to render process and consider the storage of voxels into different data structures directly on GPU (Schwarz & Seidel, 2010).

Since the mesh simplification approach is not working with meshes constructed of small voxels and even does not simplify all possible triangles for some scenarios, we should explore using other methods which are based on quadtrees and rectangular decomposition to find per slice in 2D the minimal number of triangles (Suk et al., 2012). Of course, this can work well if the scene has many flat areas, not many colours and when small voxels are used. By having lighter models, applications can be more performant and models can be easily loaded on other software.

For easier georeferencing of 3D models in Unity, without defining manually model's offset, integration of the platform with some of the available GIS systems in Unity like ArcGIS or Mapbox can be suggested. Also, voxelisation of geospatial data stored in a database should be enabled to provide two directional integration with PostgreSQL.

Although more testing is needed, the project uses URP and not many shaders are used which should be compatible with many platforms. Therefore, it should be determined exactly for each approach the compatibility with WebGL, Linux, Android, and so on.

REFERENCES

Aleksandrov, M., Zlatanova, S., Kimmel, L., Barton, J., & Gorte, B. (2019). Voxel-based visibility analysis for safety assessment of urban environments. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *4*(4/W8). https://doi.org/10.5194/isprs-annals-IV-4-W8-11-2019

Aleksandrov, Mitko, Zlatanova, S., & Heslop, D. J. (2021). Voxelisation Algorithms and Data Structures: A Review. *Sensors*, *21*(24), 8241.

Amanatides, J., & Woo, A. (1987). A fast voxel traversal algorithm for ray tracing. *Eurographics*, *87*(3), 3–10.

Beckhaus, S., Wind, J., & Strothotte, T. (2002). Hardware-based voxelization for 3d spatial analysis. *Proceedings of the 5th International Conference on Computer Graphics and Imaging*, *20*.

Garland, M., & Heckbert, P. S. (1998). Simplifying surfaces with color and texture using quadric error metrics. *Proceedings Visualization'98 (Cat. No. 98CB36276)*, 263–269.

Gorte, B., Aleksandrov, M., & Zlatanova, S. (2019). Towards egress modelling in voxel building models. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *4*(4/W9). https://doi.org/10.5194/isprs-annals-IV-4-W9-43-2019

Gorte, B. G. H., Zhou, K., Van Der Sande, C. J., & Valk, C. (2018). A computationally cheap trick to determine shadow in a voxel model. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, *4*(4).

Gorte, Bernardus, Zlatanova, S., & Fadli, F. (2019). Navigation in Indoor Voxel Models. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *4*, 279–283. https://doi.org/10.5194/isprs-annals-IV-2-W5-279-2019

IX, F. D., & Kaufman, A. (2000). Incremental triangle voxelization. *Proceedings of Graphics Interface*, 205–212.

Jørgensen, F., Møller, R. R., Nebel, L., Jensen, N.-P., Christiansen, A. V., & Sandersen, P. B. E. (2013). A method for cognitive 3D geological voxel modelling of AEM data. *Bulletin of Engineering Geology and the Environment*, *72*(3), 421–432.

Kämpe, V., Sintorn, E., & Assarsson, U. (2013). High resolution sparse voxel dags. *ACM Transactions on Graphics (TOG)*, *32*(4), 1–13.

Kaufman, A., & Shimony, E. (1987). 3D scan-conversion algorithms for voxel-based graphics. *Proceedings of the 1986 Workshop on Interactive 3D Graphics*, 45–75.

Laine, S. (2013). A topological approach to voxelization. *Computer Graphics Forum*, *32*(4), 77–86.

Laine, S., & Karras, T. (2010). Efficient sparse voxel octrees–analysis, extensions, and implementation. *NVIDIA Corporation*, *2*.

Li, W., Zlatanova, S., & Gorte, B. (2020). Voxel data management and analysis in PostgreSQL/PostGIS under different data layouts. *ISPRS Annals of Photogrammetry,*

Remote Sensing & Spatial Information Sciences, *6*.

Museth, K. (2013). VDB: High-resolution sparse volumes with dynamic topology. *ACM Transactions on Graphics (TOG)*, *32*(3), 1–22.

Nießner, M., Zollhöfer, M., Izadi, S., & Stamminger, M. (2013). Real-time 3D reconstruction at scale using voxel hashing. *ACM Transactions on Graphics (ToG)*, *32*(6), 1–11.

Nourian, P., Gonçalves, R., Zlatanova, S., Ohori, K. A., & Vu Vo, A. (2016). Voxelization Algorithms for Geospatial Applications: Computational Methods for Voxelating Spatial Datasets of 3D City Models Containing 3D Surface, Curve and Point Data Models. *MethodsX*, *3*, 69–86. https://doi.org/10.1016/j.mex.2016.01.001

Poux, F., & Billen, R. (2019). Voxel-based 3D point cloud semantic segmentation: unsupervised geometric and relationship featuring vs deep learning methods. *ISPRS International Journal of Geo-Information*, *8*(5), 213.

Schwarz, M., & Seidel, H.-P. (2010). Fast parallel surface and solid voxelization on GPUs. *ACM Transactions on Graphics (TOG)*, *29*(6), 1–10.

Setaluri, R., Aanjaneya, M., Bauer, S., & Sifakis, E. (2014). SPGrid: A sparse paged grid structure applied to adaptive smoke simulation. *ACM Transactions on Graphics (TOG)*, *33*(6), 1–12.

Silver, D., & Gagvani, N. (2000). Shape-based volumetric collision detection. *2000 IEEE Symposium on Volume Visualization (VV 2000)*, 57–61.

Suk, T., Höschl, C., & Flusser, J. (2012). Rectangular decomposition of binary images. *International Conference on Advanced Concepts for Intelligent Vision Systems*, 213–224.

Villanueva, A. J., Marton, F., & Gobbetti, E. (2017). Symmetry-aware Sparse Voxel DAGs (SSVDAGs) for compression-domain tracing of high-resolution geometric scenes. *Journal of Computer Graphics Techniques Vol*, *6*(2).

Villanueva, A. J., Marton, F., & Gobbetti, E. (2016). SSVDAGs: Symmetry-aware sparse voxel DAGs. *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 7–14.

Vo, A.-V., Truong-Hong, L., Laefer, D. F., & Bertolotto, M. (2015). Octree-based region growing for point cloud segmentation. *ISPRS Journal of Photogrammetry and Remote Sensing*, *104*, 88–100.

Zhou, Q.-Y., Park, J., & Koltun, V. (2018). Open3D: A modern library for 3D data processing. *ArXiv Preprint ArXiv:1801.09847*.

6. APPENDIX

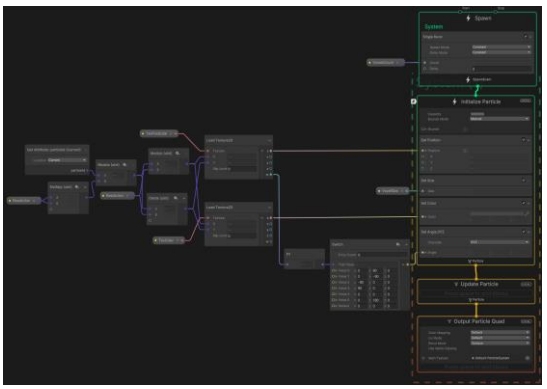

Figure 10. VFX graph using quads for voxel visualisation

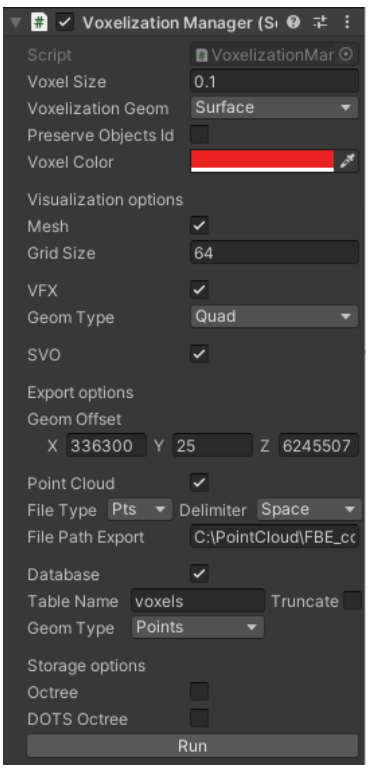

Figure 11. Voxel management options