Newcastle's Digital Twin – Visualising Data in the Urban Sciences Building

E. J. Williams, Student Number: 170532613

***Abstract* — This paper describes a software engineering project that aims to recreate Newcastle University's Urban Sciences Building in the digital realm. By making existing architectural mesh data and textures "game engine-ready", a real-time, fully-navigable virtual-reality (VR) simulation has been constructed using the popular game engine, Unity. Via a Web-based API, sensor data from the building is streamed into the simulation, allowing for near-real-time visualisation of parameters such as room temperature, CO2 level, humidity, light power and window status. In order to minimize the issue of motion sickness commonly associated with the use of a head-mounted display (HMD), it has previously been deemed necessary to maintain a consistent framerate of at least 90fps. In an attempt to achieve this target, various optimization techniques such as occlusion culling and mesh decimation have been implemented. However, only a consistent 45fps has been achieved in VR, but this has proven to be perfectly adequate. A sustained 60fps has been attained in an integrated non-VR mode.**

***Index Terms* — Level of Detail, Mesh Decimation, Occlusion Culling, Oculus Rift, Real-Time Simulation, Unity, Virtual Reality, Visualising Data.**

I. Introduction

NEWCASTLE University's £59m Urban Sciences Building (USB) was conceived as a "building as a lab" [1], with a vast array of sensors deployed throughout its rooms in order to constantly monitor and record physical parameters such as room temperature, CO2 level, relative humidity, light power, whether a window is open and the status of the heating and cooling system. Additionally, a weather station situated on the roof provides information on wind speed and direction, precipitation and visibility. All this data is available via a Web-based portal [2], and a WebGL-based 3D model already exists to allow users to observe the distribution of sensors throughout the building [3]. While *Virtual Reality (VR)* has a history dating back over 50 years, it is only in recent years that small, lightweight and high-DPI displays, in addition to sufficiently powerful GPU's capable of rendering simultaneously on two such displays at a consistently fast framerate, have come into existence at an affordable price point, making the technology finally viable for the mass market [4]. There is motivation to create a more detailed and immersive incarnation of the existing USB 3D simulation, ideally for use in *VR* with a *head-mounted display (HMD)*, such as the *Oculus Rift* or *HTC Vive Pro*, to enable the user to traverse to any accessible point within the building and visualise in real-time the full gamut of sensor data at that location. A non-VR incarnation is also desirable, preferably within the same application.

Industry Foundation Classes (IFC) data files are designed to promote and facilitate interoperability between the architectural, engineering and construction industries, and are one of the main types of *Building Information Modelling (BIM)* file [5]. As such, despite containing 3D mesh data and corresponding material information, they are not intended for use in a real-time 3D simulation, and are not directly compatible with popular game engines such as *Unity* and *Unreal*. Since they incorporate every building component required for construction, there is considerable geometry included that would never be visible to an occupant of the finished building, and the number of contained vertices for a fully-furnished, large-scale building can be anticipated to total in the order of tens of millions, spanning separate files from the various collaborating companies.

Aside from omitting as many of the extraneous objects as possible, it can be envisaged that some of the techniques used in the preparation and deployment of 3D meshes for large-scale, open-world games will be applicable to efficiently rendering IFC data in a commercial game engine. Such techniques include *mesh decimation* (reducing the number of vertices and triangles in a mesh, while maintaining an acceptable loss of detail), *occlusion culling* (discarding vertices that are calculated by the CPU to be obscured from view – *occluded* – by an object – an *occluder* – avoiding wasted GPU processing, analogous to how *view frustum culling* and *back-face culling* reduce *overdraw* by discarding vertices outside the view volume and hidden at the rear of an object, respectively) and *level of detail (LoD)* (using meshes with progressively fewer vertices as distance from the camera increases).

The use of HMD's is notoriously associated with a form of motion sickness, termed *cybersickness*, which is thought to be principally caused by the mismatch between visual and vestibular stimuli in a VR simulation, though this can be substantially mitigated by maintaining a consistent framerate of at least 90fps [6]. Coupled with the increased display resolution of a typical modern HMD - 2160 x 1200 for the *Oculus Rift* and 2880 x 1600 for the *HTC Vive Pro* [7] – it can be asserted that a virtual reality simulation requires roughly 90 to 150% more pixels to be rendered per second in comparison with a traditional 2D game at a standard 1080p/60fps, placing a significantly heavier burden on graphics hardware, and a

greater impetus on rendering efficiency by utilizing the aforementioned techniques.

II. BACKGROUND AND RELATED WORK

A. Previous Examples of Large Meshes in VR Projects

The challenge of converting BIM and IFC data into a form suitable for real-time rendering with VR has been reported in the literature, and a notable example caters for changes in the source data being reflected in real-time in the VR simulation using Unity, and even vice versa [8]. The authors note that this initial process of conversion, even for a simplistic model, typically takes several days to complete, and involves professional skills in modelling and programming; they lament that this delay is what is impeding widespread adoption of VR in the construction industry. Another paper echoes this sentiment, arguing that it may require a year to accurately duplicate a real room in virtual reality, even with a team of programmers [9]. Going back to the previous paper, while their system still requires some of this laborious process, it subsequently uses a Cloud server to organise, process and transfer IFC metadata between the VR model and its source, in both directions.

The authors advise that materials are generally not ported across accurately, leading to incorrect colours, so this is likely to require overhauling if accuracy is important. Also, it has previously been demonstrated that global illumination and shadows significantly improve the subjective feeling of immersion in VR [10], therefore it is declared to be worth the effort of manually including them if the negative effect on framerate can be mitigated, where baked shadows are advised for performance.

Reusing textures to maintain use of the same material across different objects is recommended, which will reduce the expensiveness of draw calls and permit batching on static geometry, and this hints at the use of texture atlases (i.e. combining all textures into one, and providing each object with suitable texture coordinates to sample from the appropriate part). Also, occlusion culling has been found in their tests to improve framerate by around 50%, making it more effective in terms of performance improvement than any other technique they have implemented.

Another study describes using meshes (albeit not from IFC files) incorporating over 5 million vertices, and how these were prepared for use in VR, given that initial framerates were lower than 10fps [4]. This was for exhibition, not using a HMD, but instead an "Immersive 3D Visualising Lab", in which real-time images were projected onto all 6 inner faces of a cubical *CAVE (Cave Automatic Virtual Environment)*, which presumably requires a substantially higher resolution than what is required for any existing VR headset. The geometry was simplified to reduce the number of vertices by a factor of 10, which involved replacing a complex mesh with simply a texture (going from 670k vertices to 25, with no noticeable loss of visual quality), and employing architects to rebuild the models from scratch. In tandem with the deployment of occlusion culling, LoD and lightmapping, the framerate increased to around 120fps, though planned future work to include crowds, snow and rain is expected to require the additional computational power of a supercomputing cluster.

A further example allowed members of the public to experience an architectural exhibition before the actual building was fully built [11]. The author emphasises the importance of maintaining a stable and high framerate for VR, and how limiting the number of draw calls is the most influential aspect to this. Nevertheless, light reflections, shadows and realistic surface materials are still considered essential enough to the VR experience that they are included. Starting from a *Rhino* model (a computer-aided design application, hence not dissimilar to IFC data), the mesh consisted of over 2 million triangles even after pruning extraneous details, and standard mesh simplification techniques invariably altered the appearance dissatisfactorily. The only solution found was to completely remodel parts of the architecture, in order for it to retain its structure but with fewer vertices. This reiterates the point from the previously-described paper that existing decimation algorithms are not always useful in scenarios where quality of appearance is paramount, and is corroborated by the observations of other authors who state that human perception is based on high-frequency changes, so perceived visual quality of a mesh is hinged upon the preservation of intricate geometric features, which many decimation algorithms destroy [12].

The author neglects to mention the use of occlusion culling and LoD, or any framerates, but admits that the Unity simulation does not permit locomotion, and only allows spinning on the spot. This implies that these optimisation techniques were not implemented, and therefore the framerate would have been prohibitively low had complete freedom of movement been granted. However, mention is made of baked global illumination being used, in order to leverage multi-sample anti-aliasing for improved image quality, and how the Unity Profiler proved indispensable for ensuring stability [11].

B. Occlusion Culling

Whilst a wide variation of different occlusion culling algorithms exists, they all revolve around determining a *Potentially Visible Set (PVS)* of objects, so that any object not part of this set can be considered occluded and immediately discarded (i.e. not sent to the GPU for rendering) for the remainder of that frame [13]. Figure 1 depicts how this compares with frustum culling and back-face culling.

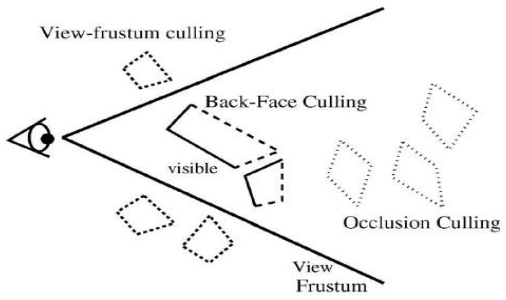


Fig. 1. Diagram showing how occlusion culling discards objects that are hidden behind other objects [14]

Four main types of algorithm exist:

(1) *Exact*, *conservative* or *approximate*
(2) Point-based or *region-based*
(3) Online or *offline* processing
(4) *Object space* or *image space*

The *exact approach* attempts to determine the *PVS* by solving for all visibility events in the scene for all possible viewpoints (yielding no false results), but these are too impractical for large scenes. The high expense can be averted by overestimating the *PVS*, which is the *conservative* approach, which definitely includes all visible geometry, but may also include some invisible geometry (i.e. some false positives) [15]. Conversely, the *approximate* approach trades accuracy for speed, and can be subdivided into *sampling* and *aggressive* strategies. The former utilises random or structured sampling (via ray-casting or sample views) and merely estimates the *PVS*, with the risk that visible objects may be missed. The latter is more akin to the *conservative* method, but to eschew false positives and work more efficiently, it declares some objects as invisible based on their very low probability of contributing to the image, therefore risking a slight chance of false negatives [14].

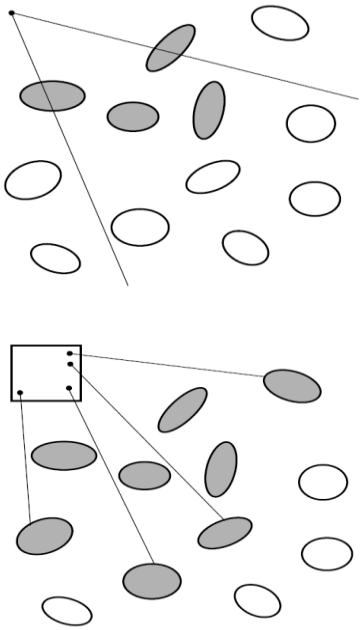


Fig. 2. Diagram showing the *point-based* (top) and *region-based* (bottom) approaches to occlusion culling, where the grey objects are part of the *PVS*, and the white objects are occluded [15].

Figure 2 shows the distinction between the point-based and region-based approaches, which are related to online and offline processing. Offline processing necessitates the region-based approach, due to the impossibility of computing the *PVS* for the infinite number of points in a particular cell. Determining the PVS dynamically at runtime is usually point-based, and is commonly more effective in its ability to cull more of the scene. Methods which use offline pre-processing divide the scene into cells, and for each one of these a *PVS* (in this case, a list of all other *cells* that are potentially visible) is produced [14]. The cells are often divided in a hierarchical and recursive form, such as an octree or binary space partition (BSP), accelerating occlusion queries since once a particular cell is culled, all of its children are simultaneously culled and removed from further queries.

This method allows for rapid rendering, since objects only need to be rendered from the pre-computed *PVS* (after frustum culling) from the cell in which the current viewpoint is located, with no additional occlusion culling overhead, making it ideal for walkthroughs of large-scale environments [14, 16]. The only disadvantages are that this is only valid for static objects, the *PVS* can take several minutes to calculate, even when hardware-accelerated with the GPU, and there is a modest memory requirement to store it. This is the approach adopted by Unity, with the depth of the tree being related to its *smallest occluder* parameter [17].

Object space approaches perform visibility calculations on raw objects, whereas image space methods consider fragments of an object during the rasterization process, meaning they are generally synonymous with online, on-the-fly processing. A distinct advantage of the latter approach can be explained by referring to Figure 3, where it can be seen that objects A and B cannot occlude object C in isolation, but their combined presence does occlude it. This *occluder fusion* principle is inherently supported by occlusion that is point sampled in image space because no distinction is drawn between different occluders, leading to them being automatically combined, whereas inefficient culling is a general consequence of failing to account for this [14]. Algorithms that use object and image spaces are not mutually exclusive, however, and some approaches use both [18].

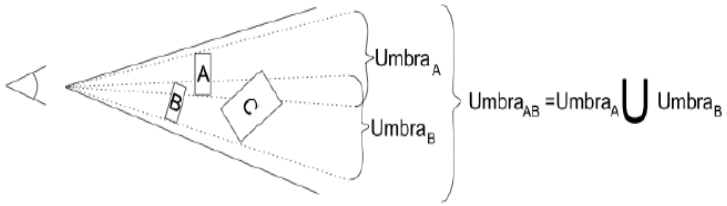


Fig. 3. Diagram showing how A and B don't occlude C in isolation, even though their combined presence does occlude it. Occluder fusion is the term given to this phenomenon being accounted for. [14]

The principle of occlusion culling can be extended to cull more than just invisible objects, and it has been successfully used to cull shadow casters when rendering from a light's point of view, accelerating the process of shadow map generation [19].

C. Mesh Simplification and Level of Detail

Mesh-based methods for *Level of Detail (LoD)* have enjoyed an extensive history in the field of computer graphics. There exist two major types – *subdivision* methods and *simplification* algorithms. The former involves low-resolution 3D models that gain higher *LoD's* via *tessellation* [20]. The latter method seeks to reduce the complexity of the mesh, whilst preserving its

salient features and topology to the fullest extent possible, and many algorithms have been devised to this end, including *vertex removal, vertex clustering, regional collapse and edge collapse* [21, 22].

A common scenario is that a mesh consists of triangles that have a size which is independent of the object's geometry, so that flat surfaces and intricate features share the same number of vertices and faces, being conducive to a waste of storage space and processing power when stored and viewed on a computer [22].

A means of quantitatively comparing the performance of the algorithms is to compare the mean geometric errors between the simplified and original meshes. Doing so demonstrates that the quadratic error metric generally gives the best and fastest results, except in the case of an extreme reduction in resolution (i.e. to 5% of original), where triangle decimation is usually best. The results demonstrate that there is no perfect algorithm to simplify all meshes, but that quadratic error metric should be chosen whenever possible, unless the mesh is to be used for the most distant LoD, in which case triangle decimation is typically superior [22].

Despite the prowess of occlusion culling, complicated scenes incorporating fine detail may still give rise to performance issues. In these scenarios, such complex meshes, when distant to the camera, often only cover a tiny proportion of the screen. This is where *LoD* techniques can profoundly boost performance, by dynamically adapting the resolution of meshes to maintain a set number of model samples per pixel [16].

The two incarnations of LoD are discrete and continuous. Discrete has the advantage of relying purely on offline production of geometry, avoiding all real-time rendering constraints, and the issue of *popping* (the abrupt transition from one LoD to an adjacent level) can be mitigated with *transition zones* that fade one level out while simultaneously fading the other one in [23].

III. DESIGN AND IMPLEMENTATION

A. Computer Hardware Requirements

The simulation has been made in Unity version 2019.1.10f1; recent Unity updates have been eschewed, lest they lead to breakages in the code. It has been tested on two different sets of hardware with fairly commensurate specifications – both systems were equipped with an nVidia GeForce GTX 1070 Ti GPU, 32GB of RAM and an Intel CPU - one with an i5-8600k and the other with an i7-4790k. Performance figures quoted are relevant to both systems.

B. IFC File Conversion Issues and Selection of Game Engine

Six IFC files were made available, as described in *Appendix A*, containing the entirety of the construction components used to build the Urban Sciences Building. However, due to their incompatibility with Unity and Unreal, these first had to be converted to either .obj or .dae (which will be referred to as OBJ and DAE files, respectively, from now on), and a free command line utility, *IfcConvert*, was used for this purpose. A typical command to extract all entities into an OBJ is of the form:

```
IfcConvert --use-element-names --sew-shells --center- model --
generate-uvs --verbose --yes USB- RYD-00-XX-M3-A-0001.ifc
Inside.obj --include entities IfcSpace -- exclude entities
```

While DAE files possess distinct advantages (e.g. such as when it's necessary to rotate an object about a particular axis) by defining an object in local coordinates and providing a transform, the geometry was found to be misaligned and distorted when viewed in any 3D mesh application, an example of which is shown in Figure 4.

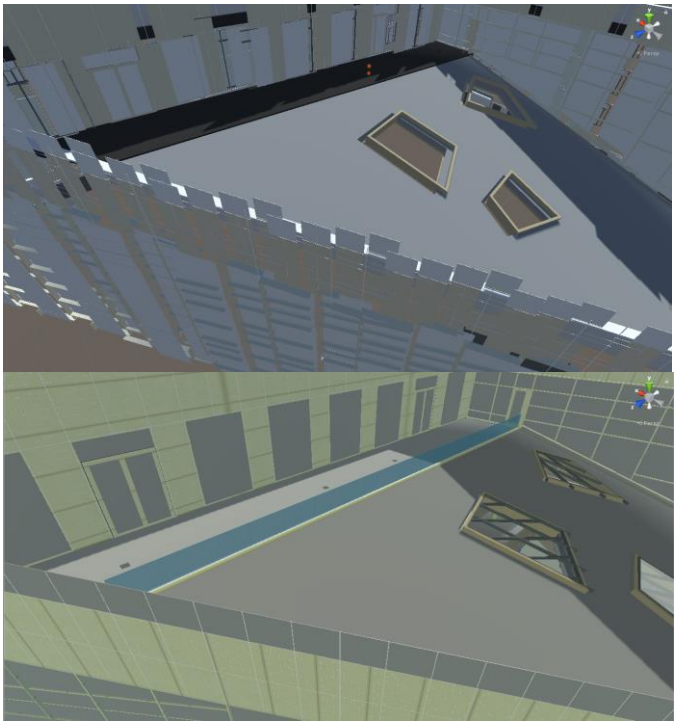


Fig. 4. Screenshots from Unity of misaligned geometry when the IFC files are converted to DAE files (top), in comparison with correct geometry when converted to OBJ files (bottom).

When converted to OBJ, it was found that the largest two files (containing the *Inside* and *Pipework* geometry) crashed both Unity and Unreal. Also, the associated .mtl (materials) file is compatible with neither Unity nor Unreal, so a further conversion to an FBX file was determined to be the only means of importing the meshes into a game engine whilst retaining materials information. Autodesk's *FBX Converter 2013.3* was used for this purpose, but this also displayed an error of "Not enough parameters" when attempting to convert the aforementioned largest OBJ files. *FBX Converter* scaled all meshes down by a factor of 100 compared to the OBJ source files. However, it was found that scaling back up to 100 in all axes of the Unity transform did not bestow the building with a feeling of being "life size", especially in VR-mode, so a scale of 110 was used instead.

It was observed that the process of importing meshes into Unreal took far longer than with Unity, possibly due to its advanced lighting calculation stage. Crashes and frozen states were also not uncommon with the former. Since it became apparent in the earliest stages that the nature of the project was going to entail considerable editing of the meshes outside of the

engine, making importing FBX files a frequent occurrence, the decision was made to focus exclusively on Unity as the engine.

C. Matching Up the Meshes

If the "center-model" option was not declared in the IfcConvert command, the resulting OBJ file was found to contain x- and z-coordinates in the region of 424000 and 564000, which was later learnt to be a consequence of Ordnance Survey (OS) grid reference numbers being used as the coordinate system (which is presumably the cause of the misaligned geometry in the DAE files). When imported into Unity, an error message warning of floating-point precision limitations would arise, and attempts to alter the position of the object in the transform would malfunction, meaning it was not possible to bring the object anywhere near to the world-space origin. This would cause all manner of problems further in the development process, not least preventing the model from being correctly lit.

While using the "center-model" option in IfcConvert allowed the resulting OBJ files to be opened in Unity without error, an undesirable side effect was that each file had its coordinate system origin set as the geometric centre of its contained objects, so when all the OBJ files were opened in the same space, they did not align with one another. Due to the absence of any obvious matching reference points between most of the separate meshes, and the fact that the building is not aligned with the x- and z-axes, it became an arduous and painstaking process to translate all the geometry to match up seamlessly.

Additionally, Unity uses a left-handed coordinate system, and defines the y-axis as "up", meaning that all imported meshes were rotated 90 degrees about the x-axis. While this could be fixed with a simple rotation in the transform in Unity, it was going to prove laborious having to do this with every import.

D. Texture Coordinate Issues

Another issue was that the included texture coordinates were based on the vertex coordinates, so these too were principally circa 424000 and 564000. This resulted in distorted and fuzzy textures, as shown in Figure 5, that noticeably juddered when the camera or object in question moved. Several pdf files containing vector-drawings (permitting resizing without pixelation) of images used on walls and doors throughout the building have been sourced. Since it was hoped to apply such textures to several of the building's prominent features, not least to the distinctive atrium windows, fixing this problem was deemed essential. However, no free 3D mesh editor appears to provide this feature. While a custom shader could be used to translate the texture coordinates on-the-fly, this would harm performance, so it's obviously preferable to solve the issue in the source OBJ files.

Fig. 5. Screenshots from Unity showing how textures appear when rendered using the existing texture coordinates (left), and with remedied texture coordinates (right).

E. OBJ-Editor in C++

In order to remedy all the issues mentioned hitherto, a decision was made to write an OBJ Editor application in C++, with initial functionality to rotate and translate vertices, to divide geometry up (by both location and object type) into smaller OBJ files and to fix the texture coordinate problem. The translation and rotation process also rotates the normal vectors, which is necessary, and decreases the number of decimal places from as many as 15 down to 6, reducing OBJ and subsequent FBX file sizes considerably, with no discernible loss of accuracy in the rendered geometry.

This OBJ Editor now totals nearly 9500 lines of code, and includes 65 functions, which have the ability to, for example, copy a particular object to its own OBJ file, delete a specific object or set of objects (such as geometry that will never be seen), combine OBJ files, translate a particular object, remove normal and texture coordinates entirely, split a mesh up by vertex indices, replace materials and re-order objects by material. The code for this is available on GitHub [24]. All functions that are referred to in the remainder of this document, unless otherwise stated, are in the OBJ Editor application.

F. Removing g Lines in the OBJ Files

Removing the "g" lines from an OBJ altogether merges all objects into one, and performing this operation on the *Inside* mesh was initially the only way to convert it to FBX and successfully import it into Unity. Removal of "g" also has the advantage of drastically reducing the number of *batches* (grouped draw calls), with the disadvantages being that access to individual objects in Unity is lost, and if only a single triangle of the mesh is not outside the frustum or occluded by another object, then the entire mesh will be drawn. In the early stages of the project, it was observed that ordering the objects in an OBJ by type, by using the SplitFiles() function to group objects with a similar "g" line together in their own OBJ file, then using the Reconcile() function to merge the OBJ's back into a single OBJ (which is a crude way of grouping objects of similar materials together), and subsequently removing all the "g" lines from the OBJ, allowed it to be rendered simultaneously with all the other meshes (apart from *Lighting* and *Pipework*) of the USB (that had also undergone the same processing) at 30 to 40fps with the Oculus Rift. When removing the normals and texture coordinates from the OBJ files, the framerate was

observed to approximately double (diffuse lighting remains fully functioning, hence Unity must generate its own normals in such cases, which are presumably optimised in some fashion). Without occlusion culling, such framerates may never have been exceeded, but performance using the "no g" technique would only have deteriorated as more vertices were added in the form of *Lighting* geometry, and there was motivation to apply textures to certain objects, so removing texture coordinates en masse was not an option.

G. Reducing Materials to Reduce Draw Calls

Throughout the entire project, it has been observed that rendering bottlenecks are usually caused by the CPU, the main cause of which is excessive draw calls through rendering too many objects. Draw calls are more expensive when there are state changes in between them, which would arise from changes in material, so there was motivation to reduce the number of materials in the meshes as much as possible, and then order objects by material.

To achieve the first goal, the 215 different materials of the *Inside* mesh were carefully compared. Since most were subtle variations on shades of grey, a few shades were retained and the rest reassigned to the closest match. This process gradually whittled the number of materials down to around 40, with no noticeable alteration in the colours of objects.

H. Ordering Materials to Reduce Draw Calls

After considerable experimentation with how to render in the most efficient manner, the best results were achieved by using the function called DivideMaterials(). This pushed all the definitions of triangles in an OBJ file to the end, after all the vertices, normals and texture coordinates had already been defined. It would then group all triangles with the same material together, and assign a new "g" line for each one, dispensing with the original object names. This way, in the absence of lighting with shadows, the resulting OBJ file should be drawn with only as many draw calls as there are materials in the file, which, in the case of the *Inside* mesh with reduced materials, should be no more than 40. Retaining these new "g" lines or dispending with the altogether appeared to make no difference all to rendering efficiency in the case of small objects. However, when scaled up, retaining "g" lines offers the great advantage of being able to cull individual materials when they are not within the frustum or are being occluded. A BatchDivideMaterials() function was written to facilitate the processing of a list of OBJ files in the same directory.

I. Setting Up Occlusion Culling

The basic idea was to use the SplitFiles() function to divide the *Inside* mesh into OBJ files containing objects of a similar type, viewing these in Microsoft's *3D Builder* to identify walls, floors and other large objects, and then grouping these files together in a folder. Once this process was complete, the files were merged back into one OBJ using the Reconcile() function, and this was designated the *occluders* OBJ. This could then be converted to FBX, imported into Unity, and marked as *Static* for everything. The remaining OBJ files that had not been deemed occluders could be similarly merged into one *occludees* OBJ and split up into "rooms" by the DivideGeometry() function (grouping geometry into between 30 and 35 boxes that approximate the location and extents of rooms in the building).

The BatchDivideMaterials() function could then be used to order materials in each "room", as previously described, before converting each one to FBX and finally importing into Unity, this time marking all meshes as *Static* for everything except *Static Occluder*. An occlusion volume was then created in Unity, with its volume modified to encompass anywhere the player avatar may conceivably roam, and the occlusion bake parameters assigned: the *smallest occluder* set to 0.5 (from the default 5), *smallest hole* set to 0.05 (from the default 0.25) and the *backface threshold* kept at the default 100. The occlusion bake could now commence, and this combination of reduced and ordered materials and occlusion culling consistently achieved framerates of 90fps in VR-mode.

However, this was before any of the lighting and pipework meshes had been incorporated, so there was motivation to begin reducing the number of vertices of the *Inside* mesh in preparation.

When a new occlusion bake was to be performed, the old one had to be deleted. However, doing this from within Unity did not remove the old files, which typically numbered in their thousands and were just several kilobytes in size each. Failure to delete these frequently from the Library/Occlusion folder, and allowing them to accumulate, could easily add hours to the time required to copy the project folder over to another directory, so it's good practice to clear the Occlusion folder after every discarded bake.

J. Decimating the Most Expensive Objects

BIM and IFC data is generally available in a range of levels of detail, but, unfortunately, the University was only granted access to the 3D meshes with the highest level of detail. In situations where many tens of millions of vertices are being drawn simultaneously, such as in the atrium of the building, or outside the building where the atrium is visible, rendering was occasionally observed to be GPU-limited. In order to ameliorate this, an attempt was made to reduce the number of vertices using mesh simplification and decimation techniques.

Using the SplitFiles() function, the occludees OBJ was again split into separate OBJ's for each object type. The resulting OBJ files were ordered by file size, with the ten largest files ranging in size from around 100MB to up to 300MB, collectively constituting over half the data from the 400+ files (which predominantly consisted of chairs and tables). *Blender* was found to intermittently alternate between triangles and quads in its output of OBJ files, which would break compatibility with the OBJ Editor. MeshLab had the disadvantage of removing "g" lines entirely, and it was unclear how to batch process with a Python script and retain material information. Nevertheless, its *quadric edge collapse decimation* algorithm, using settings to preserve the boundary of the mesh, preserve normals, preserve topology, optimise the position of simplified vertices and deploy planar and weighted simplification with post-simplification cleaning, was found to provide excellent results with certain meshes when coupled with a suitable target percentage reduction.

To carry out the decimation process, the ten most expensive OBJ files were grouped into a single OBJ file using the Reconciliation() function. The remaining OBJ files were similarly grouped into one OBJ, and this became the new occludees OBJ. As before, this was then divided by location using the DivideGeometry() function, and its materials grouped using the BatchDivideMaterials() function. Finally, each OBJ file was manually imported individually into MeshLab (as this was the only method known by which materials information would be retained), decimated with the percentage reduction set to 0.1, and saved without normals and texture coordinates. Since no textures were planned to be applied to these objects, it was hoped that some of the previously-observed performance boost to removing normals and texture coordinates would again be realized.

One disadvantage to doing this, however, is that distortion of some textures was observed on some objects after their containing OBJ file had been merged with an OBJ containing objects lacking normals and texture coordinates, hence it was decided that such OBJ files should never be combined. This is why the expensive objects had to first be divided by location and then materials, since it was known in advance that they could no longer be recombined with the rest of the occludees.

A function called AddgBackInFileList() was then written to reinstate object names for each material, to capitalize on the culling advantages that have previously been observed. After conversion to FBX, the files were then imported into Unity and marked as *Static* (but *Static Occluder* unticked). The new updated occluders file then had to undergo the divide by location/material process again before it could be imported into Unity.

K. Successful Decimation, but LOD Not Viable

The number of vertices in the resulting OBJ files was reduced to around 20% of the original value, and the objects appeared, to all intents and purposes, identical to the undecimated counterparts, even at close range. It was found that applying more aggressive decimation, with a view to creating different levels of detail, was only successful in reducing the number of vertices further by a modest amount, before the meshes would become noticeably corrupted, even when reduced substantially in size (i.e. viewed from a distance), hence it was decided that deploying different LOD was not a viable option unless a different means of mesh simplification could be found. Failing that, a better approach would simply be to decimate all the mesh geometry using *quadric edge collapse* to an extent that there was no noticeable deterioration when viewed at close range. However, time constraints and the inability to batch process without losing material information prevented decimation being performed on more objects from the *Inside* mesh beyond these ten most expensive object types.

L. Preparation of InsideSupports and Pipework Meshes

Most of the objects in *Pipework* consist of plumbing that is either concealed in boiler cupboards or has minimal value in being rendered, so can be discarded. However, certain ventilation units that fill many holes in ceilings and occupy a considerable amount of ceiling space in nearly every room must be included. Also, since the roof of the building was accessible, the solar panels that reside there should also be retained.

The InsideSupports mesh contained a lot of geometry that was hidden between floors or inside staircases, so this was also discarded.

Once all extraneous objects had been removed from both meshes, the materials left over were reduced, as with *Inside*, from 17 down (with 15 unique to its mesh) to 5 (with only 1 unique to its mesh) for *InsideSupports*, and remaining at 3 materials for *Pipework* (but originally all unique to its mesh, down to just 1 unique).

All the objects in Pipework were deemed to be occludees, so were simply then added to the occludees OBJ. The InsideSupports mesh had to be divided by type using the usual method, and the resulting files assessed for their ability to occlude, with the occluders added to the existing occluders OBJ, and the same for the occludees. Every time the occludees file was updated in any way, it had to again undergo the laborious process of being split by location and material before it could be incorporated into Unity, with a new occlusion bake instigated.

The *Pathways* mesh was so inexpensive that it seemed futile to process it in the same manner, so it was largely left intact, aside from fixing a levitating litter bin issue with the TranslateObject() function.

M. Atrium Windows Texture

The texture used for the large atrium window was made available through two pdf's from the design phase of the building. The entire texture was split over three images, which were in a vector graphics format, permitting a potentially unlimited resolution. The free vector graphics editor, *Inkscape*, was used to painstakingly remove all the extraneous labels and gridlines from the images, before exporting in PNG format. Another free raster graphics editor called *GIMP* was then used to stitch the three images together, make the white background transparent, and change the black circles to white. When imported into Unity, made into a material (with *Rendering Mode* set to transparent) and applied to the atrium windows, the texture was blurred and low definition; however, increasing the maximum texture size from the default of 2048 to 8192 remedied this.

The upper tier atrium windows had no texture available for it, so another material was made with the same texture, and the tiling and offset adjusted so that the texture looked appropriate when applied.

N. Atrium Windows Transparency Problem

When the window panes were combined into a single OBJ file and the "g" lines removed, the transparency effect was shown to be malfunctioning, as transparent objects behind the panes were observed to be drawn on top (i.e. the usual ordering of rendering transparent objects from back to front was not being applied). This same malfunction was observed even in a single pane, defined as an individual object, if it were scaled up to be beyond a certain size. Experimentation with altering the render queue for the atrium windows was partially successful in solving this problem, but resulted in the HUD being drawn

on top of by the windows. Ultimately, the problem was completely solved simply by having each pane defined as a separate object, and marking them as *Static* to mitigate the increased number of draw calls this would cause (while unticking the *Occluder Static* option, as transparent objects should obviously not be designated as occluders). This involved first having to painstakingly remove all panes from the *Outside* OBJ, in order for them to be defined as separate entities. Figure 6 depicts this transparency problem before and after it was solved.

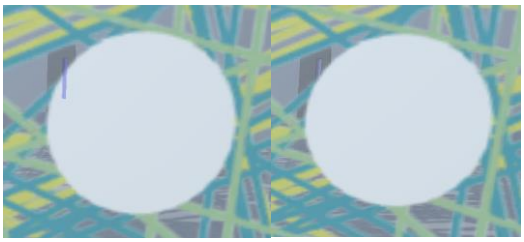


Fig. 6. Screenshots from Unity showing how transparency malfunctioned in the atrium windows while they were grouped as a single object (left), but how this was remedied once they were given their own "g" line in the OBJ file.

O. Outside Mesh

Given the above transparency problem, it was decided that all glass on the outside of the building should be drawn as separate objects. For rendering efficiency, it was deemed that the frame of both tiers of the atrium windows should be drawn as one object. After removing all the components from the source file and grouping them into one OBJ, the "g" lines were thus removed.

The remainder of the mesh was divided by location and material, and after conversion to FBX and importing into Unity, the parts were marked at *Static* for everything, since the outside walls were certainly occluders

P. Building Textures

The "Welcome" texture used in the entrance of the building, the texture on the wall opposite the main elevators on the ground floor and the texture used for one of the dining room walls were similarly derived from vector graphics in a provided pdf document, and processed in the same way. The texture coordinates for the two triangles forming the inside face of the entrance wall, however, had to be located in the OBJ file and modified to suitable values of either 0 or 1. The background "brown" colour in the texture also had to be made to exactly match the colour assigned to the rest of the entrance structure.

Q. Doors and Windows

Many of the doors and windows had their frames integrated into the same object, with no difference of material in the OBJ file to allow for easy isolation of the two components, so manual analysis of vertices was required to separate them. After extracting all the doors and windows from their source files (meaning they no longer existed in the source OBJ), the geometry could be processed further. This involved splitting an OBJ file containing a group of doors/windows into separate OBJ files per object, opening one of them in *Sublime Text* (an advanced text editor), then utilising a "trial and error" approach of deleting triangles in the OBJ file, and observing the rendered result in *3D Builder*. Once the triangles (and therefore vertices, normals and texture coordinates) of a frame were known, a function, BatchSplitMesh(), was written and deployed to batch process splitting all the other OBJ files from the same group into two OBJ files each – one containing the isolated door/window and the other containing its associated frame. In an effort to keep the file sizes as small as possible, care was taken to only include in each OBJ file the required vertices, normals and texture coordinates for the object contained, which required an elaborate self-devised algorithm. The separated doors/windows were then available for further processing, and the frames were reconciled into one OBJ to be reintegrated into the occludees mesh.

Rotating the doors about the y-axis and the windows about the z-axis while in their original forms caused them to be rotated about the origin of the world coordinate system, rather than about an axis located at one side (for the doors) or the top edge (for the windows). Had the doors and windows been axis-aligned, a simple translation of the vertices to the world space origin, a translation of half the width in the x-direction (for the doors) or half the height in the y-direction (for the windows), and inserting them back into Unity while applying the reverse of the transforms would return them to their original locations, but now with the ability to rotate them correctly about an appropriate edge. Unfortunately, none of the doors or windows was axis-aligned in the x- and z-axes, but a saving grace was unanimous alignment to the y-axis. This permitted an angle of rotation about the y-axis to be ascertained that would render the door or window axis-aligned in local space, allowing it to be translated by half its width or height, respectively, in order to define the pivot point. The following algorithm was devised and implemented in the function called FindTransformAngle().

Using the rotation matrix for rotations about the y-axis, each vertex of the door or window (after being translated to the origin) was rotated in 1-degree increments through 180 degrees. With each increment, the extents of the door or window along the x-axis were determined. The angle of rotation that yielded the smallest extents was then rotated about by 1 degree either side, in increments of one-thousandth of a degree, and the process of finding the smallest extents along the x-axis repeated. This results in the angle required to rotate the door or window about the y-axis in order for it to be aligned with the z-axis. A function called BatchFindTransform() was written to read in a list of OBJ files from a folder, translate the object inside each one to the origin and write the result out to another OBJ file; subsequently, it will then find the angle of rotation about the y-axis required to make the object in each of the new OBJ's aligned to the z-axis, and perform this rotation, outputting each result to a further OBJ file, with a text file written each time of the transform required to return each object to its original position and orientation.

For each object in the final set of OBJ files, a translation in the z-direction of half the door's width results in one edge of the door being at the origin of the coordinate system, and this defines the pivot point (i.e. the location of the hinges); similarly, a translation of half a window's height down in the y-direction effectively places the hinges at the top edge. A function,

FixDoorHinges(), was written to perform this process, outputting a final OBJ file per object, with an updated transform text file for each, where the reverse of the final translation for doors must take into consideration the sine or cosine of the transform angle and the scaling factor of 110% used when importing meshes into Unity; for the windows, only the scaling factor need be reflected. Information regarding where the hinges are placed on each particular door was not available without making direct observations of every door in the building, so a "best guess" approach was deployed for the majority of doors.

R. Making Door and Window Prefabs

Now that the meshes for all the doors and windows were in an axis-aligned form, with the axis of rotation at the origin, and the transforms known, all that was left to do was import all 300+ doors and 100+ windows back into Unity and manually apply the associated transform to each. To circumvent this, a method was concocted that allowed a single door or window from a group of (hopefully) identical entities to be inserted and transformed back into Unity, and a prefab of it made. Since prefabs are, to all intents and purposes, simply text documents that contain, crucially, transform information, this prefab could be read into a C++ function and copies made, but altering the transform each time to match that of all the other doors or windows in the same group. This idea was highly successful, except that groups of doors and windows that appeared to contain identical entities often had subtle variations in size, which somewhat hampered the process of automation, as considerable intervention was often required to ensure that all the doors and windows fitted correctly into their respective frames. Nevertheless, a procedure that, from start to finish, would arguably have taken weeks to perform manually was compressed into the space of a week.

S. Non-Standard Elevators

The USB contains six elevator shafts, four of which contain non-standard elevators that open at a different side on the upper floors than they do on the ground floor. Unfortunately, no elevator packages were available from Unity's asset store that accommodated for this unusual property, but one of the highest rated, *QA Elevator*, was purchased for around £10. The elevator itself does not actually move, but rather has a copy of itself placed on each floor, and the occupant is simply teleported between floors by changing its transform. The mesh of the elevator was in FBX format, which was converted to an OBJ file using a free utility called *Open 3D Model Viewer*. The triangles forming the rear of the elevator could then be located, through trial and error, and deleted. Once the triangles of the front of the elevator were ascertained, a copy of them were transferred to another OBJ file using the SplitMesh() function. The associated vertices could then be rotated through 180 degrees using the Rotate180() function, and translated to where the rear of the elevator was situated, using the TranslateObject() function.

These two OBJ files were now in a form ready to be converted back to FBX using *FBX Converter*. Whilst these OBJ files could have been merged back into one OBJ, it was preferable to have them as separate entities to allow fine adjustment of their relative positions within Unity.

A copy was made of the Unity prefab of the animated front doors and placed at the rear of the elevator for the ground floor; for the upper floors, the original doors and the copy were switched places due to the difficulty in making the copy animate correctly. This part of the process could not be performed whilst manipulating the OBJ file data, as such files cannot contain child-parent information, which was a requirement for the animation to work correctly.

The internal elevator controls unfortunately now appeared twice, but this was remedied by making a copy of the associated texture atlas files (diffuse, normal and ambient occlusion), and editing the relevant areas to remove all trace of the controls; the files were in TGA format, and a free raster graphics editor named *Krita* was used. A Unity material was then created to use these modified textures, which could then be applied to the newly-constructed front of the elevator. Finally, for the upper-floor elevators, the up and down arrow game objects of the front controls, each with a box collider, had to be rotated and translated to the other side of the elevator, as did the game object externally displaying the floor number.

Two of the USB's elevators go from floor 0 to 4, another two go from 0 to 6, a single one goes from 0 to 7 and a final one goes from 0 to -1. Unfortunately, time constraints prevented editing of the internal controls of each elevator to contain only the actual permitted floors, so they all remain with the default texture showing floors 0 to 9, in addition to -1 and some miscellaneous controls; however, only the game objects containing the box colliders for the buttons for allowed floors remain in each installed elevator, hence the extraneous buttons are deactivated.

In the actual building, there are two pairs of elevators where, on the intermediate floors, the external controls consist of only two buttons (up and down) shared between the pair, and on the ground and top floors, only one shared button is used for up and down controls, respectively. Again, time constraints prevented this being simulated accurately, so each elevator in the simulation has its own independent up and down controls, irrespective of floor, where both buttons have exactly the same effect of bringing the elevator to the current floor, at which point the avatar is free to go up or down to any allowed floor.

T. Existing Elevator Scripts and Elevator Controls

The asset had two scripts included, but the *ElevatorManager* was found to cause a null reference error, and failed to make the elevators begin on a randomly generated floor; however, this was quickly fixed. Also, the *Elevator* script was modified to improve the efficiency of the raycasting method (since RaycastAll() was being used, and only the nearest intersection point was necessary) to detect when the player avatar is looking directly at the elevator controls, internally and externally. The original scripts are included in the *Assets > QA Elevator > Scripts* folder for comparison.

It was hoped to make use of *Oculus Rift's* virtual hands to activate the controls, but the spherical collider on the hands inexplicably would not trigger the box colliders on the elevator. A forced *Oculus* update late in the development of the project removed the virtual hands from the simulation altogether, and

reimporting the assets associated with the *LeftHandAnchor* and *RightHandAnchor* did not remedy the problem. Time constraints prevented a full investigation of how to restore this functionality, so a good compromise (which would also aid in reconciling the VR and non-VR incarnations) was to, when in close proximity to an elevator, place an object (in this case, a rotating cube) at the location of the raycast intersection point, with the cube textured with a message to instruct which button or key to press in order to activate the controls when the cube is placed over them.

U. Reflection Probes on Elevators

It was hoped to place a reflection probe outside each elevator, or pair of elevators, on each floor to have realistic reflections on the animated doors. However, the doors were assigned only one material, and it was not possible, while in OBJ format, to assign an additional material to the triangles on the exterior of the doors since, as previously mentioned, the OBJ format has no functionality to retain parent-child relationships, so the animated doors that had gone through the conversion from FBX => OBJ => FBX were rendered useless. A consequence of a single material was that the reflection probe would unavoidably cause a reflection on the inside of the doors that depicted the outside scenery.

It was found that blackening out parts of the texture atlas for the elevators allowed the inside of the doors to no longer show the reflection, without affecting the outside, but it was extremely time-consuming to precisely find all the relevant locations in the texture atlas for this to be a total solution. The manufacturer of the elevator asset was consulted for a guide to the exact layout of the texture atlas, but this information has hitherto been unforthcoming. Two reflection probes have been left in the simulation as a proof of concept, with the downside being that there is a graphical inconsistency on the inside doors of both sets of paired elevators at the point at which the desired floor is reached, when starting from the ground floor.

V. Newcastle FBX

An FBX file containing a large section of the geometry of Newcastle surrounding the USB was available. Unfortunately, it is several years old, and does not therefore contain geometry for the new buildings that exist in the immediate vicinity of the USB. A scheduled updated mesh has been delayed by several months, so cannot yet be included. Also, the *Pathways* mesh does not contain geometry to cleanly integrate the building with its surroundings, so invisible (i.e. no activated mesh renderer) inclined planes were set up, gradually bringing the existing geometry down to a level surface, allowing the avatar to seamlessly patrol some distance around the USB. Four invisible quads were set up to form a boundary around the entire site, preventing the avatar from falling off the edge of the ground planes into oblivion.

The FBX of Newcastle was converted to an OBJ file using *Open 3D Model Viewer*, and the FixNewcastleMesh() function was used first to remove all triangles from the far edge of the mesh that would never be seen, and then to cut out triangles from the area where the USB would be situated. Without this extra step, the *Newcastle* mesh could not be brought level vertically with the USB without intercepting the lower floors of the building. The unavoidably rough finish of this editing was patched up with more quads. The various elements of the mesh, such as buildings, trees and roads, were already grouped together as one object per element, preventing them (without considerable work) from being split by location and regrouped, for the purposes of occlusion culling. However, since there was minimal impact to the framerate from including the mesh, this wasn't deemed to pose a problem.

Many of the features of the *Newcastle* mesh, such as the grass, roads and pavements, however, were only visible from the underside, meaning they were invisible from the side where the buildings and trees were located. The ReverseTriangleWinding() function was written to address this issue, selectively swapping the order of the vertices around in the triangle definitions.

W. Glitching/Transparent Porcelain

A sink with prominent visibility in the "Eat @ Urban" dining room possessed a highly noticeable glitch that wasn't resolvable using the usual method of shifting certain vertices with respect to others, since the object consisted of over 22,000 vertices. An additional problem was that certain areas within the sink were invisible, even though only one non-transparent material was used for all the triangles. Thus, this was another instance of incorrect triangle winding, which was impractical to solve by locating the offending triangles and reversing their winding, due to the sheer quantity of them. Simply removing the sink, with an additional 82 sinks or toilets sharing the same issue, was considered a last resort, since the ethos behind the simulation was to retain as much of the original geometry as possible in order to maximise its authenticity.

A solution was found in the form of assigning a material to the porcelain objects that uses a *Particles/Standard Surface* shader, and selecting the *Two Sided* option. This would effectively switch off back-face culling for any object to which it was applied. To offset the increased rendering cost of this, in addition to the fact that around 2 million vertices were currently being extraneously required to render all of these relatively simply objects, it was decided that they were ideal candidates for decimation. The 83 objects were moved out of the occludees OBJ file using the MoveObjectToIndividualOBJ() function, separated into individual OBJ files using the SplitIntoOneOBJPerObject() function, and a script written in Python to batch process the OBJ files in MeshLab using its quadric edge-collapse decimation simplification method, reducing the number of vertices by a factor of over 6, with no discernible decrease in the resolution of the mesh. Since there was only one material assigned to these objects, losing material information this time was irrelevant, hence the ability to batch process.

After reinstating g with the AddgBackInFileList() function, the files could then be recombined into one OBJ file using the Reconcile() function, converted to FBX and imported into Unity. After applying the new material, both the glitching and transparency problems were solved, and the rendering efficiency was actually substantially increased over the original objects due to the decimation process. Had there been thousands of these porcelain objects, a better method would

have been to divide them up by location in order to group them together as one object per room, but this wasn't considered necessary due to their limited number, and it was sufficient simply to mark them as *Static* (but untick *Occluder Static*).

X. Fixing Glitches

Close to a hundred instances of glitching geometry, due to coplanar and coincident faces of merged meshes, have been fixed in the OBJ files by slightly translating one object with respect to the other. Due to the incessant importing of newly-edited meshes throughout the duration of the project, remedying the problem directly in the source files, as opposed to merely in Unity, is hugely advantageous, as the issues are then fixed permanently.

Y. Lights

There are 2178 lights in the lighting IFC file, with over ten different varieties, many of which possessed the unfortunate characteristic of having only one material. This would pose a problem if only part of the light object is intended to be emissive, which was true of all of them. Attaching an emissive material to an object with only one material assigned for the whole mesh will cause the entire object to appear to emit light when the emissive material responds accordingly to light data. Each emissive surface must also be a separate entity (i.e. it cannot be merged with other objects by removing its "g" line in the OBJ), otherwise it cannot be independently lit. There was also the issue of how to incorporate the nearly 13.6 million vertices of light geometry from the 1.6GB of data that remained once all the sensor geometry had been removed from the original lighting OBJ that was converted from IFC data. In order to address all of these problems, a number of steps were required.

Z. Reducing Materials in Lights

Firstly, the 7 materials in the lighting OBJ (6 of which were unique to the lighting geometry) ideally had to be replaced with equivalent materials from the other meshes, and this process finished with only one material being unique to the lights.

AA. Separating Emissive Geometry from Chassis Geometry

The next step was to use the SplitFiles() function to split the OBJ file containing all the light geometry into an OBJ file for each type of light. The types of light that assigned only one material per object all had the emissive surface pointing vertically downwards. A FixLights() function was written, which read in these OBJ files, ascertained the lowest vertex in y for each object, then assigned a different material to all the triangles in that object that contained vertices that were within a small range of height above that vertex, outputting all the data into a new OBJ file. The function called SplitObjectsIntoSingleMaterials() was then used to read in this new OBJ and split the objects (which now all had two materials each) into separate objects each with a single material, written to one further OBJ, after which the SplitIntoOneOBJPerObject() function was used to output OBJ's for each individual object. The resultant OBJ's were ordered by file size to easily identify the files containing the emissive surfaces (which contained the fewest vertices, hence were the smallest). With the Reconcile() function, the emissive surface OBJ's were merged into one OBJ for each light type. Due to the "brute force" method of effectively skimming off the emissive surfaces from each light object, the OBJ's containing the "left-over" part typically had triangles incorrectly drawn. Hence, it was decided to use the original OBJ's (i.e. those containing the "unskimmed" geometry) for the light enclosures, which from now on will be referred to as chassis geometry, and the emissive surface geometry could simply be layered on top as separate objects, but translated down in y slightly, so as not to be obscured.

Light objects that already contained a separate material for the emissive part underwent the above process, but missed out the "skimming" stage as it was not required. Since the emissive part was separated from the chassis in a completely precise fashion, the resulting chassis OBJ's were perfectly fine to use. Lights with separate materials for the emissive component that were identified as being "always on", such as those in corridors and stairwells, did not need their emissive component accessible as a separate object (as all of them would be lit with the same emissive material), so were able to miss out this procedure altogether and be deemed chassis geometry immediately.

The lights that were "always on", but only had one material (such as LED strips down staircases), were grouped together in one OBJ file and decimated, using the same procedure as before (removing normals and textures coordinates, in the process), reducing the number of vertices from over 2 million down to below 155,000, and grouping over 400 objects into one, with a single shared material. This could be immediately imported into Unity, assigned an "always on" emissive material and drawn potentially with a single draw call.

The remaining lights were now in a form where all the chassis geometry could be consolidated into a single OBJ, and the emissive parts could be immediately converted into FBX, imported into Unity and marked as *Static* (but with *Occluder Static* unticked).

BB. How to Include Light Chassis Geometry

The final problem was how to import the OBJ of the chassis geometry, which contained 1408 objects and just under 10 million vertices, without having a detrimental effect on the draw calls and framerate. It was quickly decided that decimation was crucial, making it ideal to incorporate the lights chassis geometry with the expensive objects from *Inside*. This involved merging the chassis geometry OBJ with the expensive furniture OBJ using the CombineOBJs() function, and going through the now-familiar procedure of splitting this file by location using the DivideGeometry() function, ordering the materials in the subsequent OBJ's using the BatchDivideMaterials() functions, decimating each OBJ using the same parameters as before in MeshLab, reinstating "g lines", before finally converting each OBJ to FBX, importing into Unity and again marking as *Static* (but unticking *Occluder Static*).

The "always on" lights were selected in the Editor, and the materials ascribed to their emissive mesh components were replaced with a suitable emissive material; once this was done for one light, all the other lights in the same object were lit with the same material. This method of incorporating all the lighting geometry into the simulation was highly successful, and was not observed to increase the draw calls or decrease the framerate in any significant way. It also provided all the lights for which light power data was available on the web API with an emissive mesh that could be accessed individually and given its own emissive material.

CC. Lighting a Room

The final step for lighting a room was to create an emissive material for each of the zones, and attach each material to the emissive components of each light in the zone to which the material was designated. A lighting script, which had to be slightly rewritten for the room in which it was to be used, could then be attached to one of the lights in each zone, and a spotlight attached as a child to each of these lights running the script. So long as the lights were named appropriately, such as "Room4015Zone1", the lights in each zone would mirror the light power level for that zone in the real building, once the simulation was running and the avatar navigated to that particular room. The real-world data can also be overridden with the PgUp and PgDn keys to increase and decrease the brightness, respectively, toggled on and off with the L key, and restored to the real-world data with the R key. The G key toggles global illumination for the lights on and off. When the lights are turned on and global illumination is active, the directional light representing the sun is automatically turned off.

DD. Transitioning Seamlessly Between VR and Non-VR

Switching automatically and seamlessly between VR and non-VR modes involved detecting for a connected HMD and disabling either the VR camera or non-VR camera accordingly. The script performing this step had to be set to run before all other scripts upon initialization, as the correct running of several other scripts relies upon correct knowledge of the VR status. This was performed in the *Script Execution Order* section of the *Project Settings*.

EE. Making Scripts Performant

Due to the many hundreds of instances of scripts running (from, for example, the doors and windows), it was imperative that these were as performant as possible. If a game object reference was required by a script, this was acquired in the Start() function, and the Update() function only contains inexpensive bool comparisons to determine changes of state as much as possible. Rather than calculate the distance between the player avatar and each door each frame, only the y-position is initially considered, so a difference in two floating-point values and a subsequent comparison with a constant is sufficient to determine whether the player is on the same floor as a particular door; if true, only then is the more complicated calculation performed to determine the square of the distance (avoiding an expensive square root) in the x-z plane, and this is calculated explicitly, rather than using a standard function, in order to avoid any potential overhead associated with a function call.

FF. Single Pass Instanced

In VR-mode, changing the *Stereo Rendering Mode* from the default setting of *Multi Pass* to *Single Pass* was not observed to reduce the number of batches, but did significantly reduce the CPU rendering time (by about 30%), with a concomitant increase in framerate. Swapping from there to *Single Pass Instanced* resulted in an approximate halving of the number of batches, but no observed decrease in CPU rendering time over *Single Pass*. In non-VR mode, these settings were observed to make no difference to rendering efficiency. The setting was left on *Single Pass Instanced*, since there were no perceived disadvantages to doing so.

GG. Visualising Data

In order to visualise room temperature, C02 level and relative humidity with colour in a room, it was decided to place a sphere at the location of each sensor in each room, which would equate to one sphere per zone; each sphere was given the name of the zone in which it was located, and all spheres were made children of an empty game object, which was given the name of the room. So long as these spheres were listed in the inspector in a clockwise or anticlockwise order, the script attached to the parent would automatically draw a polygon with a sphere at each node. It should be noted that at least 3 zones are required in a room to make the polygon feasible, and at least two zones to make visualisation with colour ascribed to spheres a worthwhile endeavour, hence Room 4.018 is not able to visualise data for room temperature, CO2 and relative humidity, other than numerically on the HUD.

The script for this "visualisation polygon" must be bespoke for each room in order to connect correctly to the script reading in data for that room (which is another bespoke script). Once received, the data is ready to be converted to a colour to be displayed at each vertex of the polygon and on each sphere. The colours of the vertices can then be linearly interpolated across the entire polygon. In order to not fully obscure the objects in the room, it was opted to make the polygon semi-transparent, and to also allow the option to move it around vertically, from floor to ceiling (using the *up* and *down* cursor keys in non-VR-mode, or the *left/right index trigger buttons* in VR-mode). The spheres can also be resized using the "," and "." keys. In order to make the data visualisation as clear and unobtrusive as possible, no shadows are cast or received by the polygon or spheres, and there is the ability to independently enable and disable the polygon (with the *V* key, or the *B* button on the *Oculus* controllers) and the spheres (with the *N* key, or the *Y* button).

HH. Converting Room Temp., CO2 and Humidity to Colour

The only standard method for converting temperature to colour relates to black-body radiation and colour temperature, but using such a scale to represent fluctuations in room

temperature would yield no discernible change in colour, even if there was an extreme temperature gradient across a room. Similarly, there is no established means of representing CO2 levels and relative humidity as colour. Therefore, it was decided to have a dynamic scale when representing the data for the zones in a room with colour. Since it was desired to linearly interpolate colours between zones for the polygon, the choice of colours was limited to the three primary colours of red, green and blue. To extend this a little further, it was decided to add white and black to this palette, assigning white to the largest current value of a parameter, black to the smallest, and red, green and blue at regular intervals in between (from large to small). The option of a static scale was also included, where the values were assigned as shown in Figure 7, but these can easily be reassigned in the code. For values of room temperature or the upper bound of CO2 level outside this range, white or black are set to be fully opaque (as opposed to semi-transparent).

	Black	Blue	Green	Red	White
Room temp. (degC)	< 16	19	22	25	> 28
CO2 level (ppm)	0	250	500	750	1000
Rel. humidity (%)	0	25	50	75	100

Fig. 7. Table showing how the static key assigns colours to values of certain physical parameters

II. Key

When either the polygon or spheres are active in a room, a key appears to show the parameter being represented with colour, and how the colours are being assigned to values. The key can be toggled on and off with the *K* key or *left thumbstick* button. The parameters can be cycled through with the *left* and *right* cursor keys or the *left/right middle-finger trigger* buttons.

JJ. HUD and API Problems

The HUD, toggled on and off with the F key or A button, shows the numerical data for each zone in the room, which is updated once every two seconds.

While it is possible, through the Web API, to read in data for all zones in a room by accessing a single Webpage, this method has been determined to be harmful to the framerate (sometimes causing a consistent 60fps+ to momentarily plummet to ~10fps). A better solution is to read the data per zone, resulting in a much smaller string to read, at the expense of having to perform this same process on a separate thread for every zone simultaneously. For a single zone, this Weblink shows the format of the data:

https://api.usb.urbanobservatory.ac.uk/api/v2/sensors/entity?meta:roomNumber=4.015&meta:roomZone=1

Omitting "&meta:roomZone=1" provides the data for the entire room. With 4 zones, streaming and processing the data, then displaying it on the HUD, only results in a modest framerate drop of ~3 fps. To remedy this further, there is a delay of 0.2 seconds in between readings for each zone.

Rooms have different numbers of zones (going up to as many as 17), and some zones have more data parameters available than others, meaning the scripts have to be tailor-made for each zone.

To make matters more difficult, it has been observed that the window position and window alarm data (which both mean the same thing) commonly go into an error state, which can alter the order in which the other data is arranged. The only way currently to correct for this is to observe the error and write code to counteract it, but it will not be practical to implement this solution for all zones for all 100+ rooms in the building.

In order to not waste CPU resources, the reading of data for a room is turned off when the player avatar leaves the room.

KK. Updating Lights and Window Status for All Rooms

Due to the time-consuming nature of having to write semi-bespoke scripts for reading in the data for each zone, and the common errors that have been encountered, only three rooms are currently able to display data – 4.015, 4.018 and 4.022. Upon initialization, these rooms take one reading from the API before being disabled, the purpose of which is to set up their lighting and window status correctly. It was hoped to have the status of all the windows and lights in the building update, say, once a minute, since the API allows access to data for a single parameter across all rooms and zones (albeit sometimes spread across several pages). C# scripts have been tentatively written (*WindowPosition.cs* and *LightPowerLevel.cs*, the latter being in a very primitive state) to read in such data, but it will be an extremely arduous process to connect up each piece of data correctly with their designated game object, given that there are around one thousand lights that are not "always on", and over one hundred windows. It is also anticipated that extensive use of delays in co-routines will be required to spread the load of this data acquisition and processing procedure over several seconds in order to mitigate CPU spikes and framerate drops.

For the windows that are not connected to data from the API, the *9* and *0* keys can be used to open and close them.

IV. RESULTS AND EVALUATION

A. Framerates

Up until close to the end of development, a consistent framerate of 90fps was being achieved in VR-mode when the player avatar was in most rooms. While integrating all the lighting geometry and the *Newcastle* mesh did not individually appear to significantly affect performance, it seems their combination has given rise to a negative effect, in that it is now a rare occurrence to observe the framerate reach 90fps in VR-mode, and typically hovers around 45fps. This is not to state that the performance has halved, as the *Oculus Rift* is set up in such a way that unless a solidly consistent 90fps can be reached, it will revert to 45fps.

B. Disabling the Directional Light

It was discovered right at the end of development that simply disabling the directional light that represents the sun resulted in

a reduction in draw calls by typically a factor of around five. While consistent framerates of 45fps were being achieved in VR-mode in most areas of the building, when the player avatar was viewing the atrium from inside and outside the building (the latter scenario being worse due to the additional cost of rendering the exterior, in tandem with the many extra rooms that then become visible), the framerate was often falling to as low as 20 to 30fps. In order to ensure a consistent framerate of at least 45fps throughout the entire simulation, it was decided to disable the directional light in VR-mode.

In non-VR-mode, however, it remains active, and the capped framerate of 60fps has been observed to be maintained throughout. The only exception to 45fps in VR-mode and 60fps in non-VR-mode is when the player avatar is inside a room with lights turned on and global illumination active, leading to the decision to allow the user to disable global illumination for all rooms for the duration of the session with the press of the *G* key. In non-VR-mode, since the influence of the directional light was minimal when inside a room with the lights switched on and global illumination active, in order to boost performance, it was decided to disable the game object representing the sun in such scenarios.

Activating deferred rendering did dramatically improve performance with multiple real-time lights, but the loss of anti-aliasing, with no adequate alternative, proved to be an unacceptable sacrifice, as jagged edges on polygons in VR are especially conspicuous.

C. No HTC Vive Pro Support

The original goal was to make the simulation functional for both the *Oculus Rift* and the *HTC Vive Pro*, but the latter's controllers lack analogue thumbsticks, opting instead for touchpads, making navigation in the simulation troublesome. Additionally, the controllers have far fewer buttons, which would necessitate the headset being continuously removed - which also happens to be a more time-consuming and troublesome process than for the *Oculus Rift* - in order to use a keyboard to control functionality. These two problems have led to the decision to abandon development for the *Vive*.

D. Distorted Textures

In the built/standalone version of the project, some of the physically-based rendering (PBR) textures can be seen to be distorted (such as in certain areas of the floor and on the wooden staircase in the atrium), but this is not observed when the project is run from within the editor of Unity. Using a different *Compression Method* does not solve the problem, and the cause remains unclear.

E. Minimum Specification Required

The standalone version uses less than 1GB of RAM when running, so as little as 4GB of RAM will likely be sufficient to run the simulation smoothly, assuming no other applications are running. However, a GTX 1070 Ti can be regarded as the minimum specification of GPU required, and a recent generation Intel i7 CPU is recommended.

F. Objectives Achieved

Aside from not attaining the difficult goal of maintaining a consistent 90fps in VR mode, coupled with the remaining performance issues when lights are switched on with global illumination active (but which is remedied by allowing the user to disable global illumination for the duration of the session with a single key press), and the fact that only three rooms currently allow visualisation of data, the simulation realizes all of its original aims and objectives.

V. CONCLUSIONS

A. Fully-Functional Simulation

Despite only currently working for a limited number of rooms, the data visualisation aspect of the simulation functions particularly well, and is a proof of concept for future work. In the absence of real-time global illumination, rendering is remarkably smooth and the player avatar satisfyingly agile. The best features are the moving doors and windows, the working elevators and lights that accurately dim according to light power data per zone. The atrium is particularly attractive, with its liberal use of PBR textures on the floor and staircase, coupled with the extremely faithful recreation of the textured atrium windows (a screenshot is shown in *Appendix C*). It is the assessment of the author that the project has been a resounding success, though additional time to add more features and to fix some of the remaining issues would have been ideal.

B. Future Work

A day/night system is partially set up, with functionality in place to read in the time of day, and by pressing certain keys (full controls are shown in *Appendix B*), swap the skybox around, alter the light intensity from the skybox and change the angle of the directional light (equivalent to the position of the sun in the sky, which is now only functional in the non-VR-version). The idea would be to control the skybox and sun position based on the time of day, and change the light intensity from the sun and the skybox based on data streamed from the weather station on the roof of the USB, from which real-time values for solar radiation and outside brightness are available. The major stumbling block to carrying this idea through to fruition was that the building in the simulation would be plunged into complete darkness during certain hours of the day, unless the player avatar was in a room with lights on and global illumination active. The best solution to this problem would be to bake lighting for all the "always on" lights in the corridors, preferably with area lights to accurately simulate lighting from all the various forms of lighting geometry. However, setting this up would involve considerable manual labour, in addition to a likely extremely lengthy computation process.

A partial solution to restore the sun in VR-mode would be to bake lighting from the directional light, but this would probably then necessitate having the user select between VR-mode and non-VR-mode in a menu system. Since it would be ideal to have a version of the project that is capable of being run on lower-specification hardware at high framerates, in VR and non-VR, perhaps the best solution would be to make an entirely separate version with baked lighting for the sun (and possibly other simplifications to boost performance, such as removing some

non-essential geometry), and have both versions continue to automatically select between VR and non-VR modes.

Parallels can be drawn between the simulation and an "architectural walkthrough", although these are ordinarily made prior to construction work to provide an accurate visual depiction of how the finished building will appear. Unreal is arguably the engine of choice for such an endeavour, and is capable of delivering genuinely photorealistic results that are virtually indistinguishable from reality. Extensive use of ultra-high resolution physically-based textures on all objects and baked lighting are necessary to achieve such high-quality results, but such an endeavour is entirely feasible for the simulation given sufficient time, and it could then have use as a virtual tour of the building.

Unity defaults to using DirectX 11 as the graphics API - DirectX 12, deemed by Unity to be "Experimental", approximately halves the framerate, and switching to Vulkan crashes the project upon opening. As support for the newer API's matures, it can be anticipated that they will eventually lead to significantly enhanced performance. Rendering is still predominantly CPU-limited, yet the CPU usage has been observed to never exceed 30% at runtime, hence it is clear that there would be considerable gains to be had by multithreading. However, to enable this in the short-term, major modifications to the engine itself would be required.

Unreal is arguably an engine geared more towards higher performance for PC, with its use of C++ instead of C# in scripts to gain greater control of the hardware, so recreating the simulation in Unreal may be the quickest and most effective way of achieving a consistent framerate of 90fps in VR with real-time lighting active. Now that the meshes of the building are finalized, Unreal's lengthy import process that was detrimental to productivity in the early stages of development will no longer be such an issue.

It would be ideal to hear the sound of footsteps as the avatar traverses the building, and have these alter depending upon the surface underfoot (e.g. carpet, wood, tiles), and with reverberation effects related to the size of the room in which the avatar is located. Sliding doors and standard doors opening and closing should also be audible. General building noise, from whirring photocopiers and the fans in thousands of computers, would also bestow the simulation with enhanced immersion.

The USB could be more accurately divided up into rooms to optimise occlusion culling, though this would have to be carried out with extensive performance testing in between alterations, as making more "rooms" has the potential to precipitate more draw calls, which would obviously be counter-productive.

It was hoped to visualise weather, such as rain, wind and fog, given that the weather station on the roof provides data for rainfall accumulation, wind speed and direction, and visibility. Wind would be easily visualised by setting up and controlling a wind zone with the data, and replacing the existing trees with assets that respond to wind zones.

References

[1] http://www.constructionmanagermagazine.com. (2017). 'Building as a Lab' opens at Newcastle Uni [online] Available at: http://www.constructionmanagermagazine.com/onsite/building-lab-opens-newcastle-university/ [Accessed 25 June 2019].

[2] http://www.urbanobservatory.ac.uk. (2019). Urban Observatory. [online] Available at: http://www.urbanobservatory.ac.uk/#access-data/ [Accessed 26 June 2019].

[3] http://www.urbanobservatory.ac.uk. (2019). Urban Sciences Building. [online] Available at: https://3d.usb.urbanobservatory.ac.uk/ [Accessed 26 June 2019].

[4] J. Lebiedi, and M. Szwoch, "Virtual Sightseeing in Immersive 3D Visualization Lab," in *Proceedings of the Federated Conference on Computer Science and Information Systems 2016*, Gdansk, Poland, 2016, pp. 1641-1645.

[5] https://en.wikipedia.org. (2019). Industry Foundation Classes [online] Available at: https://en.wikipedia.org/wiki/Industry_Foundation_Classes [Accessed 21 June 2019].

[6] S. Jung, and T. Whangbo , "Study on Inspecting VR Motion Sickness-Inducing Factors," in *2017 4th International Conference on Computer Applications and Information Processing Technology (CAIPT)*, Kuta Bali, Indonesia, 2017.

[7] https://www.windowscentral.com. (2018). HTC Vive Pro versus Oculus Rift. [online] Available at: https://www.windowscentral.com/htc-vive-pro-vs-oculus-rift [Accessed 22 June 2019].

[8] J. Du, Z. Zou, Y. Shi, and D. Zhao, "Zero latency Real-time synchronization of BIM data in virtual reality for collaborative decision-making," *Automation in Construction 85 (2018)*, pp. 51–64, 2018.

[9] S. Mandal, "Brief Introduction of Virtual Reality and its Challenges," *International Journal of Scientific & Engineering Research*, Volume 4, Issue 4, April 2013.

[10] I. Yu, J. Mortensen, P. Khanna, B. Spanlang, and M. Slater, "Visual Realism Enhances Realistic Response in an Immersive Virtual Environment - Part 2," *IEEE Computer Graphics and Applications Volume 32 Issue 6*, pp. 36-45, December 2012.

[11] A. Kreutzberg, "High quality Virtual Reality for Architectural Exhibitions," *Virtual Reality Volume 2 eCAADa*, pp. 547-554, August 2016.

[12] J. Khalil, A. Munteanu, L. Denis, P. Lambert, and R. Walle, "Scalable Feature-Preserving Irregular-Mesh Coding," *COMPUTER GRAPHICS FORUM 36(6)*, pp. 275-290, 2017.

[13] M. Sunar, T. Sembok, and A. Zin, "Accelerating Virtual Walkthrough with Visual Culling Techniques," in *2006 International Conference on Computing & Informatics*, Kuala Lumpur, Malaysia, 2006.

[14] A. Mayer, and T. Wien, "Dynamic Occlusion Culling," 2007.

[15] H. Hey, and W. Purgathofer, "Occlusion Culling Methods," *EUROGRAPHICS*, 2001.

[16] S. Yoon, E. Gobbetti, D. Kasik, and D. Manocha, "Real-Time Massive Model Rendering," Morgan & Claypool, 2008, pp. 1-122.

[17] https://docs.unity3d.com. (2019). Occlusion Culling. [online] Available at: https://docs.unity3d.com/Manual/OcclusionCulling.html [Accessed 23 June 2019].

[18] B. Li, C. Wang, and L. Li, "Efficient Occlusion Culling with Occupancy Proportion," in *2008 International Conference on Computer Science and Software Engineering*, Hubei, China, 2008.

[19] A. Silvennoinen, T. Soininen, M. Maki, and O. Tervox, "Occlusion Culling in Alan Wake," in *SIGGRAPH 2011*, Vancouver, British Columbia, Canada, 2011.

[20] G. Loubet, and F. Neyret, "Hybrid mesh-volume LoDs for all-scale pre-filtering of complex 3D assets," *EUROGRAPHICS Volume 36 (2017)*, Number 2, 2017.

[21] K. Ng, and Z. Low, "Simplification of 3D Triangular Mesh for Level of Detail Computation," in *2014 11th International Conference on Computer Graphics, Imaging and Visualization*, Singapore, Singapore, 2014.

[22] B. Mocanu, R. Tapu, T. Petrescu, and E. Tapu, "An experimental evaluation of 3D mesh decimation techniques," in *ISSCS 2011 - International Symposium on Signals, Circuits and Systems*, Iasi, Romania, 2011.

[23] Z. Du, D. Zhou, and L. Zhang, "VR-Orientated 3D Modeling and Visualization of Pseudo-Classic Building Complex," in *16th International Conference on Artificial Reality and Telexistence--Workshops (ICAT'06)*, Hangzhou, China, 2006.

[24] https://github.com. (2019). peinchka/OBJEditor. [online] Available at: https://github.com/peinchka/OBJEditor [Accessed 23 August 2019].

Appendices

Appendix A

The file sizes quoted are for the resulting OBJ files, and all figures are prior to any processing.

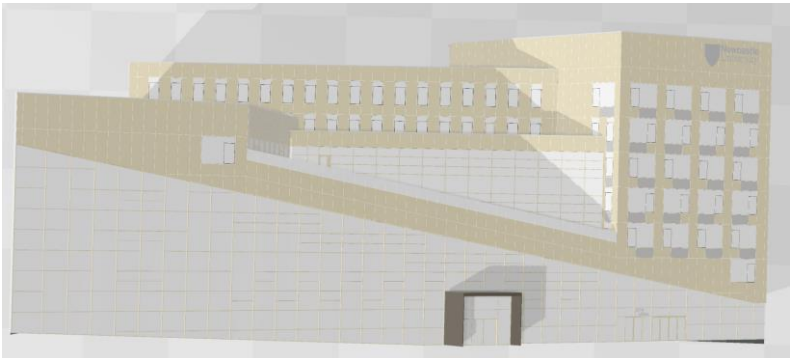
Source files:

USB-RYD-00-XX-M3-A-0001.ifc (OBJ referred to as “Inside”)

- The majority of the interior of the building, complete with furniture
- 4.3GB
- 23.8 million vertices
- 10.7 million triangles
- 16,174 objects
- 215 materials

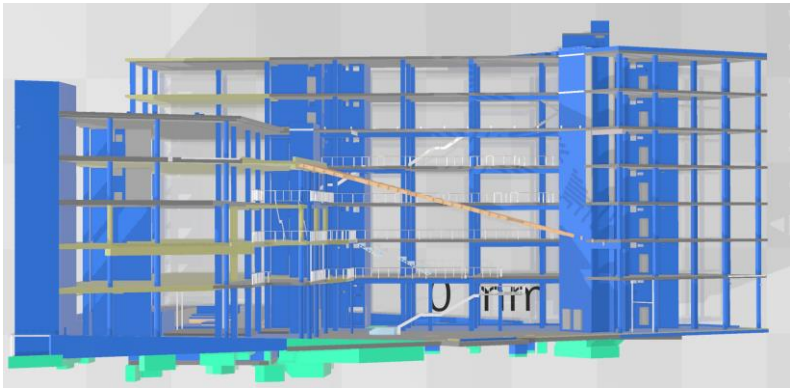
USB-RYD-00-XX-M3-A-0002.ifc (OBJ referred to as “Outside”)

- The exterior of the building
- 528MB
- 3.0 million vertices
- 1.6 million triangles
- 15,001 objects
- 39 materials



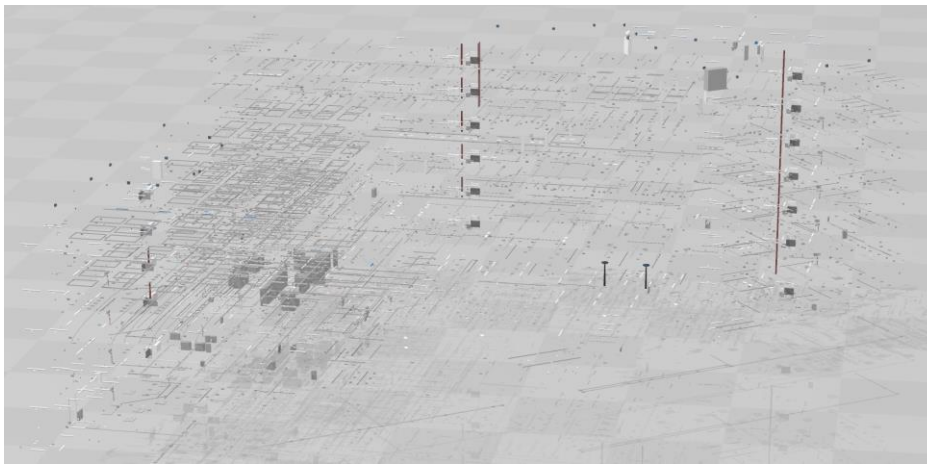
USB-MMD-00-ZZ-M3-S-0001.ifc (OBJ referred to as “InsideSupports”)

- Support structures inside the building, and some floors and walls
- 32MB
- 197,000 vertices
- 157,000 triangles
- 1,448 objects
- 33 materials



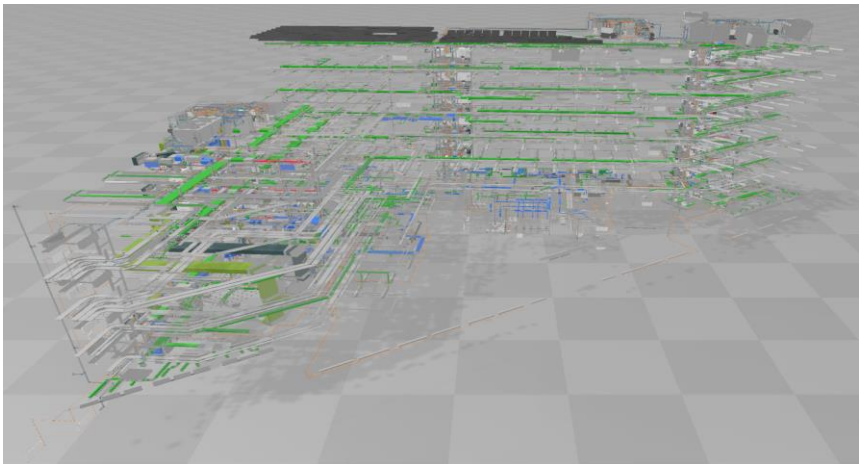
USB-NGB-00-ZZ-M3-N-0001-Revit2018-LightingElectrical.ifc (OBJ referred to as “Lighting”)

- All lighting and sensors, with electrical and network sockets
- 2.7GB
- 15.7 million vertices
- 7.0 million triangles
- 6,233 objects
- 46 materials



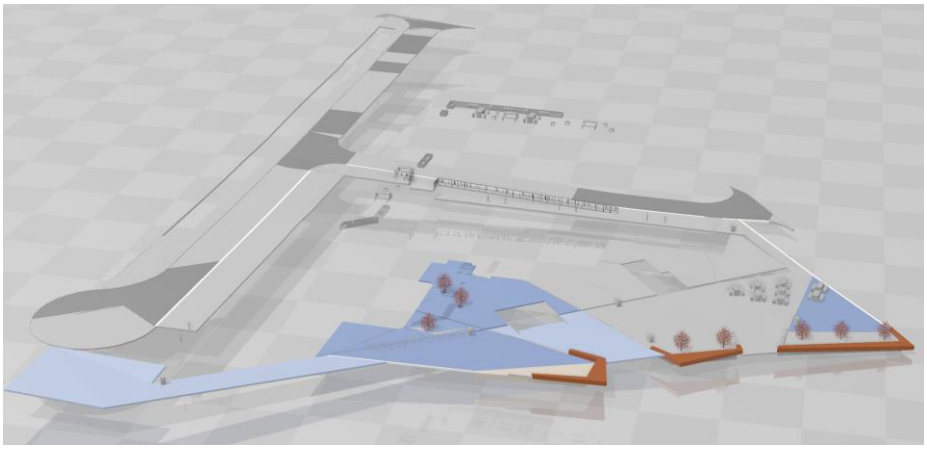
USB-NGB-00-ZZ-M3-N-0001.ifc (OBJ referred to as “Pipework”)

- All pipework and plumbing
- 14.5GB
- 78.5 million vertices
- 42.2 million triangles
- 103,029 objects
- 131 materials



USB-WAR-00-XX-M3-L-0001.ifc (OBJ referred to as “Pathways”)

- Outside pathways, street furniture and trees
- 58MB
- 344,000 vertices
- 134,000 triangles
- 173 objects
- 14 materials



Appendix B

VR Controls

Left thumbstick - forwards, backwards, slide-left and slide-right
Right thumbstick - rotate-left and rotate-right
Left thumbstick button - toggles data visualisation key on/off (only functions when inside a room)
Right thumbstick button - jump
A button - toggles HUD showing the framerate and whether room lighting has global illumination enabled. If inside a room, this also displays the room name and the real-time data for each zone.
B button - toggles visualisation polygon on/off (only functions when inside a room)
X button - cycles between a dynamic and static key for data visualisation (only functions when inside a room)
Y button - toggles sensor node spheres on/off (only functions when inside a room)
Left/right index trigger buttons - changes height of visualisation polygon down/up (only functions when inside a room)
Left/right middle-finger trigger buttons - cycles between parameter type (room temperature, CO2 level and relative humidity) (only functions when inside a room)

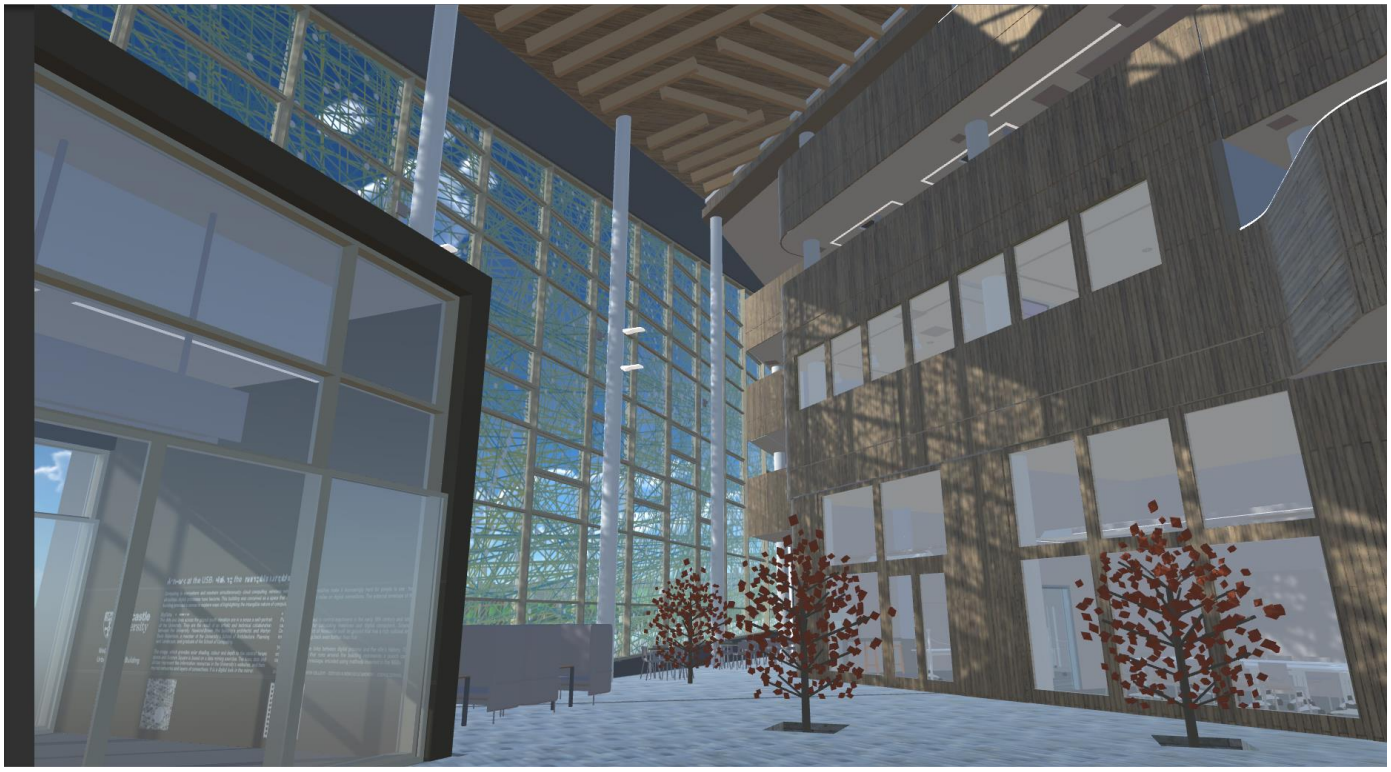
The following keys are also of use in VR-mode:
L - toggles lights in current room on/off (only functions when inside a room)
PgUp - increases brightness of lights in current room, turning lights on if they were previously off (only functions when inside a room)
PgDn - increases brightness of lights in current room, turning lights off when the brightness reaches zero (only functions when inside a room)
R - restores sensor data to lighting, after PgUp, PgDn or L have been pressed (only functions when inside a room)
G - toggles global illumination on/off
,/. - decrease and increase size of sensor node spheres (only functions when inside a room)
9/0 – open and close all windows that are not connected to sensor data

Non-VR Controls

A/D - slide-left and slide-right
W/S - moves forwards and backwards
Q/E - rotate-left and rotate-right by 45 degrees
F - toggles HUD showing the framerate and whether room lighting has global illumination enabled. If inside a room, this also displays the room name and the real-time data for each zone.
V - toggles visualisation polygon on/off (only functions when inside a room)
N - toggles sensor node spheres on/off (only functions when inside a room)
K - toggles data visualisation key on/off (only functions when inside a room)
L - toggles lights in current room on/off (only functions when inside a room)
PgUp - increases brightness of lights in current room, turning lights on if they were previously off (only functions when inside a room)
PgDn - increases brightness of lights in current room, turning lights off when the brightness reaches zero (only functions when inside a room)
R - restores sensor data to lighting, after PgUp, PgDn or L have been pressed (only functions when inside a room)
G - toggles global illumination on/off
,/. - decrease and increase size of sensor node spheres (only functions when inside a room)
9/0 – open and close all windows that are not connected to sensor data
H - enables night skybox (experimental)
J - enables original skybox (experimental)
T/Y – rotates the directional light, simulating movement of the sun, with Y advancing time (experimental)
U/I - decreases and increases skybox lighting (experimental)
Cursor left/right - cycles between parameter type (room temperature, CO2 level and relative humidity) (only functions when inside a room)
Cursor up/down - changes height of visualisation polygon down/up (only functions when inside a room)
Spacebar - jump
Hold Shift - run
Hold X - walk slowly
Left CTRL + Spacebar - secret super jump! (non-VR only)

Appendix C

A screenshot from the simulation, showing its recreation of the Urban Sciences Building's atrium.



Another screenshot, showing how real-time data is visualised in the simulation.