The Unit Pipe: A memory-efficient representation for real-time visualization of massive MEP models

SPRINGER NATURE

WESTERN SYDNEY UNIVERSITY

CONVR 2024

24th International Conference on Construction Applications of Virtual Reality (CONVR 2024)
4-6 November 2024, Western Sydney University, NSW, Australia

The Unit Pipe: A memory-efficient representation for real-time visualization of massive MEP models

Mikael Johansson* and Mattias Roupé

Chalmers University of Technology, Gothenburg, Sweden
* jomi@chalmers.se

Abstract. Real-time visualization of Mechanical, Electrical and Plumbing (MEP) models is a common task in many applications areas, including Virtual Reality (VR) design review, 4D planning, and Digital Twins (DT). Still, ever increasing complexity of MEP models makes this a challenging problem, both in terms of memory consumption as well as rendering performance. When considering Building Information Models (BIM) in general, the concept of geometry instancing is often utilized, where the same geometry is reused at several different locations, such as in the case of identical windows or doors. For some of the components in MEP models, like valves and radiators, it is possible to also take advantage of the instancing approach. However, a significant part of the geometry in a MEP model consists of straight pipes of varying dimensions and lengths, which – in contrast – are far from identical. As such, the common strategy to solve this problem in previous research has been to take advantage of various decimation techniques in order to simplify individual pipe geometries to reduce overall memory consumption and rendering complexity. In this paper we instead introduce the Unit Pipe, which – together with a non-uniform scaling transformation – makes it possible to represent all the different straight pipes with a single geometry reference. The technique is fully compatible with openBIM and the IFC file format, and in practice it can reduce the memory consumption for straight pipes by more than 90% without sacrificing the visual quality.

Keywords: Building Information Modeling, BIM, MEP, Real-Time Rendering

1 Introduction and Background

Building Information Modeling (BIM) is transforming the way we create, share, and exchange information within the architectural, engineering, and construction (AEC) industries. Collaborative design review, mobile AR applications, VR walkthroughs, 4D planning, and more recently, various digital twin technologies are all examples where real-time visualization of BIMs has become a crucial component [1]. Nevertheless, this development has also introduced a number of challenges with respect to memory consumption and rendering performance. In particular, we see the size, complexity, and detail of federated Building Information Models (BIM) increase at the same time as many different applications require these models to be visualized in real-time on a wide range of hardware – from desktop gaming computers with dedicated graphics hardware, to mobile devices, and even stand-alone VR

headsets. For the purpose of architectural visualization, the challenge and complexity of rendering BIMs taken from real-world projects has been highlighted and addressed [2]. These models typically have two main features that can be taken advantage of for the purpose of efficiency: high level of occlusion as well as high level of component reuse. By utilizing occlusion culling only visible geometry needs to be rendered, and by using geometry instancing a single geometry reference can be re-used at several different locations, such as for identical furniture or doors. In addition to significantly reduce the memory footprint, geometry instancing can also lead to increased rendering performance [3]. This is especially true for web-based rendering where the relative performance cost for draw calls are high [4]. Geometry instancing can mitigate this cost. Still, what is found suitable and efficient for one discipline or category of models might not always be applicable for another discipline.

More recently, the specific task of visualizing Mechanical, Electrical and Plumbing (MEP) models have gained interest [5, 6]. In comparison to architectural models they have different features, especially when considering visualization in isolation. Although it is true that MEP models also features many identical components – like radiators, valves, and lighting fixtures – and therefore can take advantage of geometry instancing, a significant part of the geometry consists of straight pipes of varying dimensions and lengths. In contrast, these are far from identical. As such, it makes sense that recent approaches have instead focused on simplification and compression. Hu et al. [5] employed a Quadric-Error-Metric (QEM) mesh simplification algorithm together with additional data compression techniques to significantly reduce the memory footprint for large MEP models. Reducing the number of triangles has a direct and linear effect on memory. A pipe with 50% less triangles will essentially consume 50% less memory. Still, there is always a limit on how much that can be gained, as beyond a certain threshold geometry can start looking faceted and the visual fidelity be degraded when viewed up-close. Optimally, the simplified version should be viewed at a distance, while the original geometry be used up-close, but this would then actually increase the memory footprint. In a somewhat similar approach, but without mesh decimation, Guo et al. [6] performed data compression and shape matching to reduce redundancy. Even if the variation among pipe dimensions and – especially – lengths is typically high, there might be redundancy and unused instancing potential in the form of similar layout on different floors in a multi-storey building, for instance. Yet, only identical geometry can be used for "redundancy-removal".

In the approaches presented so far it has been assumed that geometry instancing requires identical (i.e. "the same") geometry in order to be functional. However, this doesn't necessarily has to be true. In fact, for certain types of geometric shapes, it is possible to reuse a single geometry reference together with a non-uniform scaling transformation to represent many different variants. One such geometric shape is the cylinder, which is commonly used to represent straight pipe segments in MEP BIMs.

In this paper we take advantage of these two features and present The Unit Pipe – a memory-efficient representation for straight pipe segments in MEP models. By using a single reference geometry for all straight pipe segment in a MEP model, we can significantly reduce the memory footprint. In addition, this representation is fully compatible with openBIM and the IFC file format, and also allows for easy integration of LOD-mechanisms (Level-of-detail), where far-away objects are rendered in less detail.

2 IFC and MEP Models

Following the IFC object model, straight pipes are defined by the *IfcFlowSegment*, whereas non-straight segments, like an elbow or tee, are defined as *IfcFlowFitting*. We use the 2x3 specification here mainly as our real-world sample models for the experimental evaluation are delivered in this version. Later versions of the IFC specification (i.e. IFC4 and IFC4x3) has extended the object hierarchy for MEP models and introduced new, more specific object definitions, such as *IfcPipeSegment* and *IfcDuctSegment*, however, the concept of distinguishing between straight and non-straight pipes still occurs. Although IFC offers great flexibility in terms of geometric representation, straight pipes created using standard settings in BIM authoring software, like Revit or MagiCAD, is typically exported as either solids (e.g. *IfcExtrudedAreaSolid*) or a shell-based representation (e.g. *IfcShellBasedSurfaceModel*). In Fig. 1 a simplified object hierarchy illustrates how this is represented

according to the IFC model (2x3), highlighting the positioning, geometry, and coloring. Except for the obvious effect on appearance, what mainly differentiates these two approaches is that while the *IfcExtrudedAreaSolid* representation is parametric (i.e. extrusion of a circular profile) and therefore allow both length and diameter to be easily extracted from the geometric definition, the shell-based surface model does not. Instead it can be seen more as a "triangle soup".

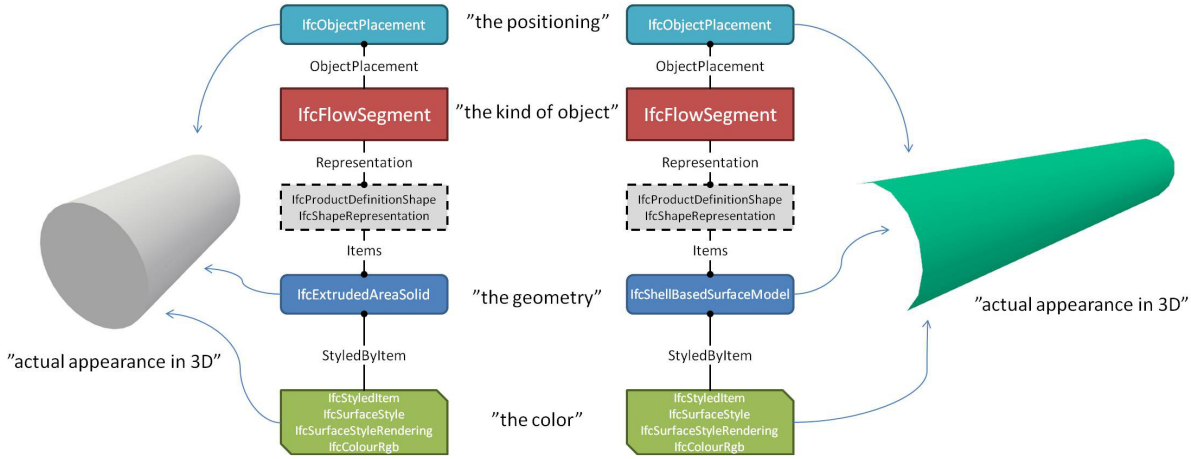

Fig. 1. The IFC object hierarchy for typical pipe representations.

3 The Unit Pipe

As stated, the idea behind our memory-efficient representation for all straight pipes in a MEP model is conceptually simple – by using a non-uniform scaling operator we can form all these pipes from a single unit pipe geometry. Similar to the concept of a unit cube (i.e. a cube centered around origo whose sides are one unit) we create a pipe centered around origo with both length and diameter set to one (1). For the purpose of illustration we will represent the pipe using a solid cylinder which is the default geometry created in Revit for pipe segments. However, a shell-based representation, as typically received from MagiCAD would work equally well and will be detailed in Section 3.1. With the unit pipe aligned with the X-axis, we control the length of the pipe by scaling in X, and the diameter by a uniform scale in Y and Z. Furthermore, by adding and combining this with a translation and rotation matrix we can place and orient the pipe anywhere in 3D-space. The concept is illustrated in Fig. 2.

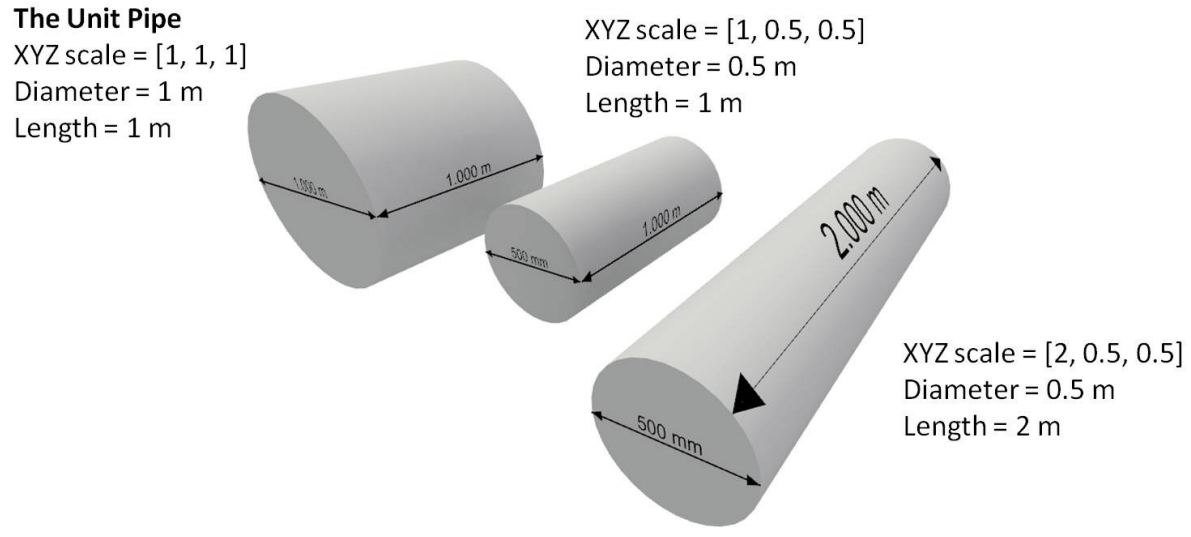

Fig. 2. Effects of non-uniform scaling on The Unit Pipe.

Thus, to replace a non-instanced pipe we only need to extract three (3) things from it:

- the center of the pipe
- the direction (i.e. orientation) of the pipe
- the length of the pipe
- the diameter of the pipe

Depending on geometry representation type, there are multiple different ways to extract this information from the IFC-file. For instance, in the case of a parametric geometric representation like an *IfcExtrudedAreaSolid* with a circular profile the required information can be extracted without even generating a 3D representation. However, for BREP or shell-based representations, the parametric information is not available. Therefore, we choose to present a more general solution: the creation of an oriented bounding box (OBB) that encloses the pipe geometry. Although this means we always have to derive a mesh-based representation it allows us to support all representation variants. There are many different ways to find a tight-fitting OBB from a triangular mesh, we choose the method proposed by Larsson & Källberg [7], which is both fast and produces good results. From the OBB computation we get the center as well as three vectors representing the extent and direction of the major axis. With the assumption that the pipes always have a circular cross-section we treat the vectors with equal length as the diameter and the final vector then gives us the length and direction of the pipe. In Fig. 3 we illustrate a subset of the resulting OBBs generated from non-instanced pipe geometry in the case of solid geometry (left) as well as shell-based geometry (right). Once the OBBs are found, we can calculate the unique, per-instance transformation matrices and then replace all the non-instanced straight pipes with a single unit pipe geometry.

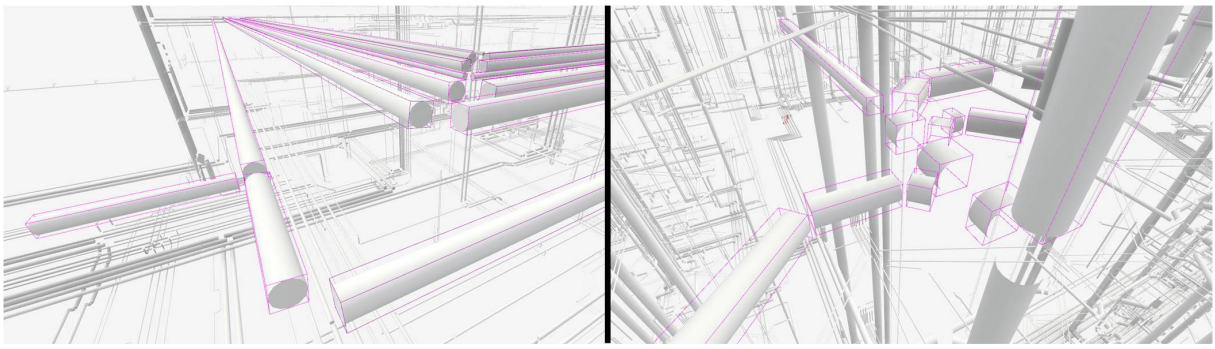

Fig. 3. Oriented bounding boxes (OBBs) generated from different pipe geometries.

3.1 Pipe Types

A crucial part of our approach is to be able to detect the specific pipe type to be replaced with a unit pipe. By analyzing several MEP IFC models received from real-world projects, three (3) distinct types of pipe representations where identified (Fig. 4, left); solid cylinder, solid hollow cylinder, and shell-based cylinder (non-solid). These were found to be represented both parametrically and non-parametrically (i.e. BREP or mesh-based). Consequently, without being fully certain to always receive a parametric pipe representation (i.e. profile with extrusion), some additional logic is needed in order to detect the actual pipe type. Furthermore, care also has to be taken in order to separate ducts with a round profile from those with a rectangular profile, as the *IfcFlowSegment* element type is used for both. Although the *Pset_DuctSegmentTypeCommon* has a definition for the shape of the duct it is often not present in files received from real-world projects. The same concept is of course equally possible to use together with rectangular profiles (i.e. using a unit cube), but we must still be able to distinguish between rectangular and round profiles. Thus, we use a number of heuristics based on ray casting and topological features to help us detect the correct pipe type from a mesh-based representation (i.e. triangles). This process also guarantees that the correct longitudinal direction (i.e. the flow direction) is found. In almost all of the cases this is the longest side of the OBB. Still, it is possible to encounter a pipe segment that has the same length as the diameter, and in that case we simply cannot rely only on the OBB characteristics.

We use the xBIM toolkit [8] to parse IFC files and generate a triangular mesh for each *IfcFlowSegment* element. xBIM already provides meshes that are decomposed into faces (i.e. segmented), as illustrated in Fig. 4 (left). Additionally, for each face, we calculate a vector that is the sum of all the normals of the triangles that belongs to that face. For the caps the length of this vector will be significantly larger than one (1), but for the lateral surfaces the length will instead be between zero (0) and one (1). The reason for this behavior is because in the first case we're adding several normalized vectors that all point in the same direction, but in the second case we're adding multiple normalized vectors that all points in different radial directions and therefore "cancels out" each other. This metric essentially allow us to classify surfaces as lateral or caps (Fig. 4, right).

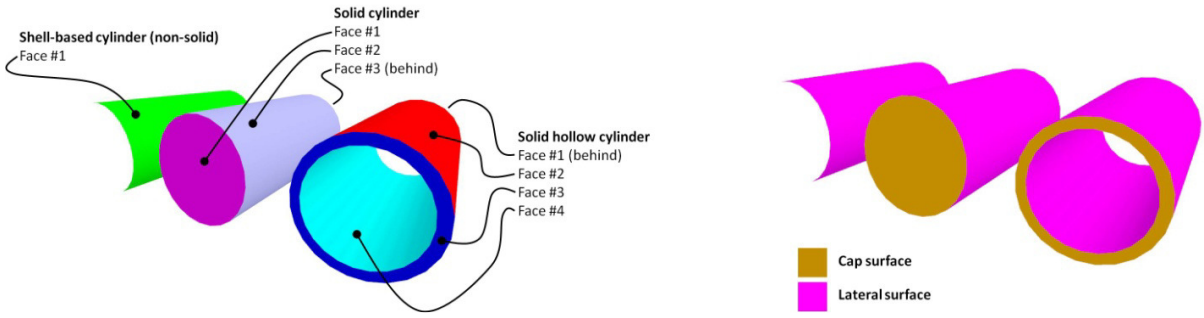

Fig. 4. Different pipe representations with faces and surface classifications.

Number of faces together with their classification (i.e. cap or lateral surface) is typically enough information to find the pipe type. However, we also perform a number of ray casts on the geometry in order to extract the longitudinal direction and thickness, but also as an extra verification regarding pipe type. Based on the center and orientation of the OBB, we perform ray casts in the major directions as illustrated in Fig. 5 (left). The intersection results are then used together with face count and face classification to accurately determine the pipe type as well as dimensions and direction. For instance, the shell-based cylinder should only have a single lateral face which is intersected by four of the six ray casts. The other two rays will intersect nothing, and then denote the longitudinal direction. Similarly, the solid cylinder will have a single lateral face, intersected by four ray casts, but then also two cap faces intersected by two ray casts that define the longitudinal direction. Still, in either case the corresponding size of the OBB (i.e. the extent in the specific direction) is used as the diameter and length of the pipe. In comparison, the solid hollow cylinder is a bit more involved, in that the two intersections per ray cast against the lateral faces – in addition to confirming the pipe type – are used to calculate the thickness of the pipe (Fig. 5, right). However, due to the fact that a specific thickness cannot be maintained while also scaling the outer diameter, the solid hollow cylinder pipe type is not compatible with the Unit Pipe approach, but instead has to be treated as a Semi-Unit Pipe (as further described in the following section).

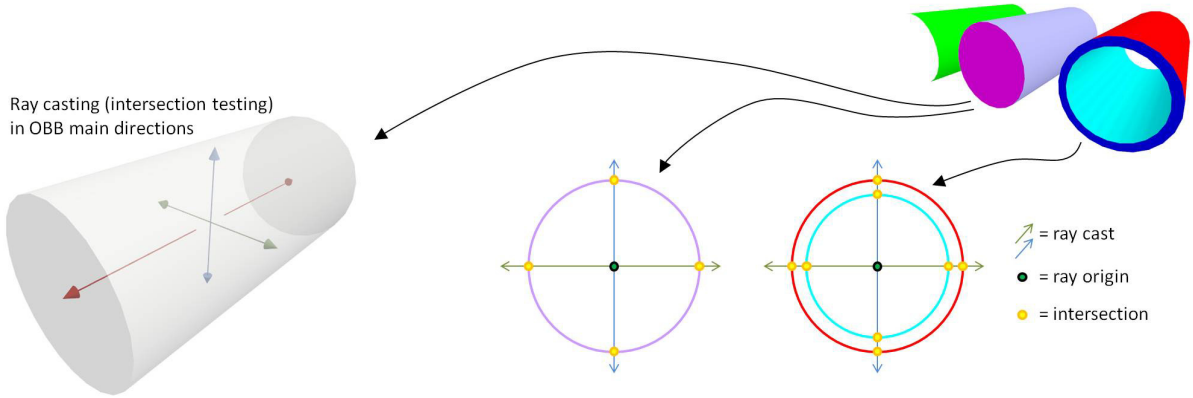

Fig. 5. Ray casting is used to aid pipe type detection and to identify the longitudinal direction and thickness.

3.2 Semi-Unit Pipe

The solid hollow cylinder shape does not support scaling in the radial direction with respect to maintaining a specific thickness of the pipe; as we scale to increase or decrease the outer diameter, we also increase or decrease the thickness. However, even with constant radial dimensions, it is still possible to control the length by scaling in the longitudinal direction only. As we will later see, there is typically only a limited number of unique solid hollow cross sections found in files from real-world projects, which allows us to greatly reduce the memory footprint also with the semi-unit pipe. The main difference compared to the unit pipe implementation is that with the semi-unit approach we cannot pre-generate the prototype geometry, but instead have to do it on-the-fly during the IFC import process. Once the pipe type has been determined and the outer diameter and thickness has been calculated using ray casting and OBB computation (Section 3.1 and Fig. 5), we store a copy of every pipe with a unique profile in a look-up table and use it as a prototype for instancing. That is, once a hollow cylinder with a unique diameter and thickness is found during import, its geometry is added to the look-up table for use as a prototype. Subsequently found hollow cylinders with the same diameter and thickness will then use the already collected geometry as prototype geometry for instancing. As we no longer can assure a normalized length of the prototype geometry, we also have to take that into account during the longitudinal scaling operator (i.e. to form a 3 m pipe from a 1.5 m prototype, the scaling operator would have to be 2 instead of 3).

3.3 Rendering

The concept of instancing can be utilized in several different ways, depending on graphics hardware as well as the platform and graphics API. The most basic way is simply to repeatedly draw the unit pipe with a different transformation matrix. However, although this takes advantage of instancing at the memory level (i.e. we only ever need the geometry of a single unit pipe), the sheer number of individual draw-calls may lead to non-optimal performance [2]. A better approach is often to use an instanced draw call, which allows all the individual pipes to be drawn with a single function call that takes the number of requested instances to draw as an argument. This concept typically comes in two flavors; either the transformation matrices are submitted as an instanced attribute or they are fetched from a data texture or buffer object in the vertex shader based on an "instance counter" (i.e. gl_InstanceID) provided by the API [9]. Regardless, it allows all the different pipes to be drawn with a unique transformation using a single API function call. Other per-instance attributes, like material color, are handled in the same way, meaning that, not only position and size, but also appearance can be unique per-instance. However, in order to be able to mix instanced and non-instance drawing, and also be able to take advantage of additional acceleration techniques, we have chosen to implement Multi-Draw Indirect (MDI), which, in many ways, share similarities with instancing, but allows for much more flexibility. Essentially, an array of draw commands is uploaded to GPU-memory and then all of the objects in a whole model (wall, pipes, radiators, floors, etc.) can be drawn with a single API draw command (i.e. *glMultiDrawArraysIndirect*), see Fig. 6. So, whereas a conventional instancing draw call allows a single geometry to be drawn multiple times, an MDI draw call allows multiple different geometries to be drawn multiple times. In addition, this functionality allows more advanced acceleration techniques, like frustum and occlusion culling, to be performed entirely on the GPU. This concept is often referred to as GPU-driven rendering [10].

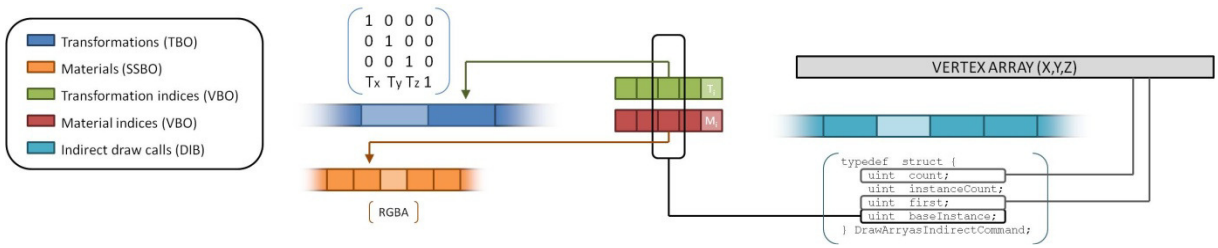

Fig. 6. Geometry instancing with Multi-Draw Indirect rendering.

3.4 Representation According to IFC

The concept of geometry instancing is supported within IFC using something called *IfcMappedItem* which – instead of a dedicated geometric shape – holds a reference to another re-usable shape through an *IfcRepresentationMap*. Together with a transformation operator this allows re-placement of a shape or geometry anywhere in the model. This is a very common concept that is used when multiple instance of the same component exists, such as identical furniture, windows, doors, etc. Thankfully, the IFC schema also allows for a non-uniform transform operator, as well as material color override at the instance level – essentially the only additional things we need in order for it to support the concept of a Unit Pipe. As illustrated in Fig. 7 we use the *IfcCartesianTransformationOperator3DnonUniform* to allow for both placement and separate scaling in X, Y, and Z, meaning that we can use a single unit pipe shape representation. Furthermore, as it is possible to assign a dedicated surface style directly to the *IfcMappedItem* we can also support per-instance coloring. Now, all of this doesn't necessarily reduce the actual size of the IFC file. Parametric geometry, such as *IfcExtrudedAreaSolid*, is already a very "slim" representation in the file and replacing it with a transform operator and a reference to a mapping source might actually increase the file size slightly. However, instead, this representation "forces" any software or BIM-viewer to treat the pipes as instanced geometry and thereby reducing the memory footprint, even if the software itself is unaware of the Unit Pipe concept.

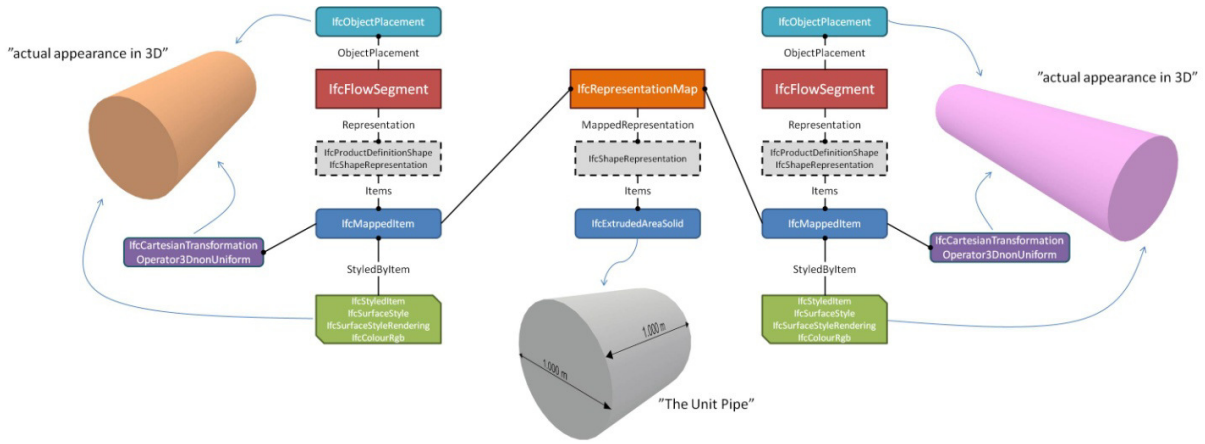

Fig. 7. Object hierarchy to represent the Unit Pipe according to IFC.

3.5 Level-of-Detail (LOD)

In addition to reduce memory footprint and number of draw calls – which will improve performance in CPU-bound scenarios – the Unit Pipe approach also makes it easy to take advantage of Level-of-detail to improve performance by reducing the geometric complexity. In its simplest form, it allows an application to select geometric complexity based on target hardware, such as choosing a less detailed version of the unit pipe for visualization on mobile devices. However, "traditional" use of LOD, such as rendering far away objects with less detail at the same time as objects close up are rendered in full detail also becomes much easier. As we can have ready-made versions of the unit pipe, there is no need for a dedicated simplification step when importing an MEP discipline model (i.e. file). Assuming three levels of detail, each one with approximately half the number of triangles as the previous level, there are multiple ways to implement this in practice. Using a basic geometry instancing approach this only requires three instanced draw calls, one per level. This does require per-instance attributes to be frequently updated, but even with per-frame updates this is a viable strategy from a performance point of view [3]. In Fig. 8 we illustrate the implementation of LOD and show that it is possible to significantly reduce rendering complexity without affecting the visual appearance negatively.

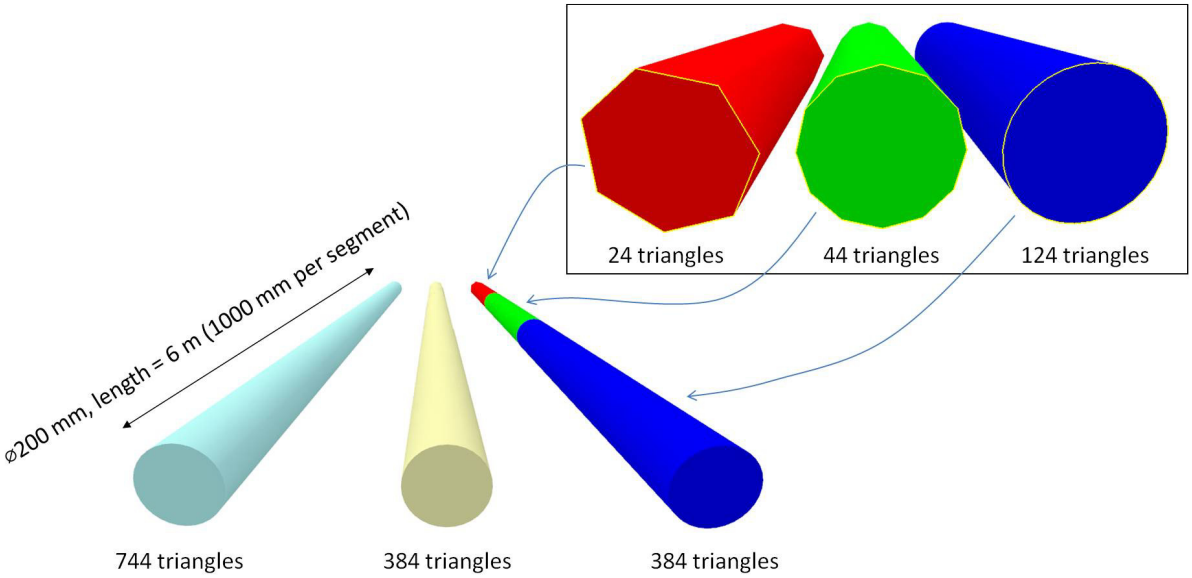

Fig. 8. Adding LOD to the Unit Pipe concept.

However, following our chosen approach with indirect draw calls, we instead add geometry for less detailed version of the unit pipe to our global vertex array, and then, depending on distance or screen size, we modulate the *first* and *count* parameters to make the indirect draw call fetch the lower or higher detailed version of the unit pipe.

Unfortunately, there is not yet any possibility to use the concept of LOD within an IFC file. Although the specification supports more than one shape representation per element, there is currently only one for 'Body', which refers to the actual 3D representation. As such, the use of (multi-)LODs becomes application dependent. Also, while LOD can be supported for shell-based and solid cylinders simply by a collection of pre-generated unit pipes of varying tessellation, this is not equally simple for solid hollow cylinders (i.e. Semi-Unit Pipes). Still, replacing solid hollow cylinders by non-hollow ones at a distance might be considered a viable approach to reduce geometric complexity.

4 Evaluation

We have evaluated the unit pipe concept on six different MEP BIMs received from real-world projects (Fig. 9). Models A to D are different disciplines from the same project, while E and F are from two separate projects. Statistics for each one of the models is presented in Table 1. Regarding triangle statistics (# tris), *view* refers to the number of triangles that are visible and needs to be rendered, whereas *mem* refers to the number of triangles that needs to be stored in-memory. That is, if a certain component, such as a lighting fixture, has 500 triangles, but is (re-)used as an instance 5 times, the # tris (view) would be 2500, but the # tris (mem) would only be 500. Furthermore, pipes refer to the straight pipes that we are replacing with unit pipes (for these pipes view and mem is the same as they all have their own, unique geometry in the IFC files). Although we process the geometry as triangulated meshes, all the pipe geometries originally have parametric representations in all six test files. As such, the triangle count is directly affected by the tessellation settings. The xBIM toolkit uses Open Cascade as the geometry engine, which exposes both linear and angular deflection parameters as a means to control tessellation behavior. In all of the tests we have used linear deflection 5 mm, and angular deflection 1 rad.

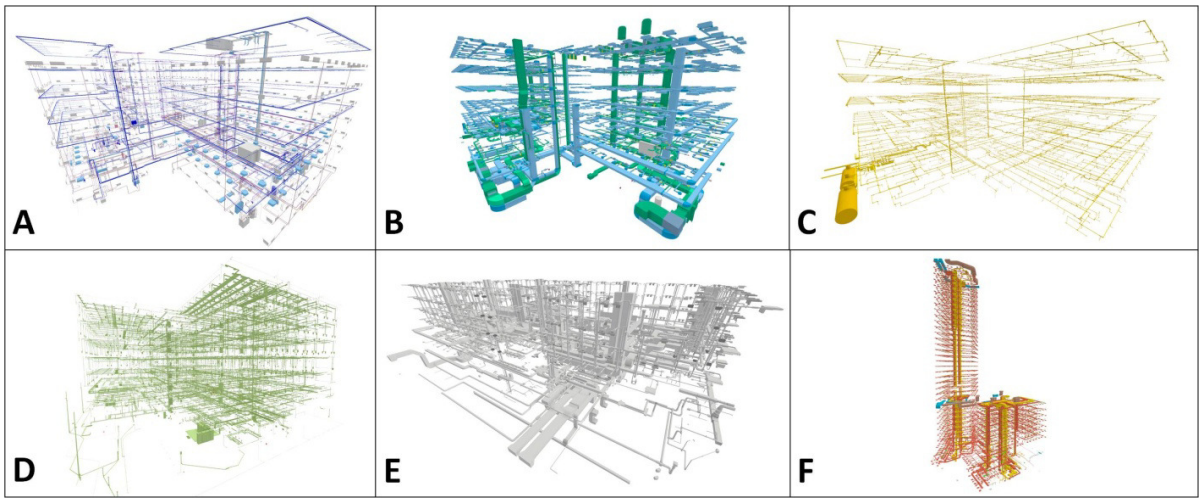

Fig. 9. Real-world IFC-models used for evaluation of the Unit Pipe concept.

Table 1. Model statistics

Project	A	B	C	D	E	F
Discipline	Plumbing	Ventilation	Sprinkler	Electrical	Ventilation	Ventilation
Pipe geom	Solid	Solid	Solid	Solid	Shell	Solid hollow
# objects	73,067	18,819	40,449	57,347	36,130	45,117
# tris (view)	10,791,835	4,676,449	11,452,896	9,089,343	13,144,757	10,811,303
# tris (mem)	2,626,557	922,113	1,400,358	1,141,820	1,654,249	2,789,958
# pipes	32,762	6,673	13,920	10,903	13,533	18,440
# pipe tris (view)	1,581,360	343,924	767,760	440,984	650,612	2,183,287
# pipe tris (mem)	1,581,360	343,924	767,760	440,984	650,612	2,183,287
# unit pipes	1	1	1	1	1	
# semi-unit pipes						12

From the model statistics alone, it is evident that MEP discipline sub-models are, indeed, geometrically complex. The total number of triangles that must be visualized per discipline ranges between 4 to 13 million. Still, we can see that the concept of geometry instancing is already present in all of the test models. The fact that the number of *view* triangles are much larger than the *mem* triangles tell us that identical components are, indeed, already re-used. As such, similarity analysis and mapping as suggested by previous research could be considered a redundant process, at least for these models. However, perhaps more important, is the fact that the straight pipe geometries represent between 30-70% of the actual memory footprint, which makes them a very suitable candidate for optimization. When replacing this geometry with a Unit Pipe reference, we essentially only require 16 floats (i.e. a 4x4 transformation matrix) per object. In comparison, a single triangle requires 18 floats (3D coordinates and normal vector for 3 vertices), and with around 50-100 triangles per pipe object we see a reduction of more than 90% for the straight pipe geometries. When considering the whole model this means between 30-70% reduction. Still, in practice the memory savings are typically even more as many real-time graphics engines allocate a 4x4 transformation matrix per object anyway. In that case, the Unit Pipe implementation becomes the same as just removing storage for all straight pipes.

Focusing instead at the F model and the Semi-Unit Pipe approach, it is also proven to be very efficient, at least for this model. In fact, only 12 unique pipe types where found which means that also in this case we are looking at a significant level of memory reduction.

When considering rendering performance there are only minor improvements from non-instanced to instanced rendering when the Multi-Draw Indirect (MDI) technique is used. With MDI having a very low draw call overhead "by design" (i.e. multiple, individual draw calls are processed as one), this is expected. Unfortunately, MDI is not available on all platforms (e.g. WebGL), and if we instead were to compare instanced rendering to conventional rendering using one draw call per object, the performance difference would be much higher – even on older hardware – simply because of lower driver overhead and CPU load [3, 9, 11].

5 Conclusions

In this paper we have presented the concept and implementation details of the Unit Pipe, which can significantly reduce the memory footprint when visualizing large MEP BIMs. As evaluated with several BIMs from real-world projects we see memory reductions of more than 90% for straight pipe geometries which means between 30-70% for complete MEP models. The Unit Pipe approach further supports openBIM and can be stored and transferred as a fully valid IFC file. In addition we have shown how it simplifies and supports the concept of LOD and can adapt geometry complexity depending on hardware.

The main limitation with our approach is that we only support a certain type of geometry, and not, for instance, spiral ducts that are represented in full detail (i.e. with the actual spiral profile modeled). However, even if we include state-of-the-art, fully model-based Total BIM projects [12] we have yet to encounter that type of geometry in practice. Also, when considering interoperability among authoring software it makes more sense to only use standard cylinders (shell, solid, or solid hollow) which can be represented parametrically.

For future work the Unit Duct would be an obvious extension using a unit cube instead of a pipe to support rectangular profiles. Another aspect would be to consider support for pipe insulation.

References

1. Park, C., Rahimian, F. P., Dawood, N., Pedro, A., Dongmin, L., Hussain, R., & Soltani, M. (Eds.). (2023). Digitalization in Construction: Recent trends and advances. 1st edn. Routledge, UK, London.
2. Johansson, M., Roupé, M., & Bosch-Sijtsema, P. (2015). Real-time visualization of building information models (BIM). Automation in construction, 54, 69-82.
3. Johansson, M. (2013). Integrating occlusion culling and hardware instancing for efficient real-time rendering of building information models. In International Conference on Computer Graphics Theory and Applications (Vol. 2, pp. 197-206). SCITEPRESS.
4. Klein, F., Spieldenner, T., Sons, K., & Slusallek, P. (2014). Configurable instances of 3D models for declarative 3D in the web. In Proceedings of the 19th International ACM Conference on 3D Web Technologies (pp. 71-79).
5. Hu, Z. Z., Yuan, S., Benghi, C., Zhang, J. P., Zhang, X. Y., Li, D., & Kassem, M. (2019). Geometric optimization of building information models in MEP projects: algorithms and techniques for improving storage, transmission and display. Automation in Construction, 107, 102941.
6. Guo, Z., Li, Q., Wang, Q., Qiao, S., Mei, T., & Zuo, W. (2024). Reducing redundancy in large MEP building information models for facility management using oriented bounding boxes and hash sets. Automation in Construction, 165, 105514.
7. Larsson, T., & Källberg, L. (2011). Fast computation of tight-fitting oriented bounding boxes. Game Engine Gems 2, 1.
8. Lockley, S., Benghi, C., & Cerny, M. (2017). Xbim. Essentials: a library for interoperable building information applications. The Journal of Open Source Software, 2(20), 473.
9. Hillaire, S. (2012). Improving Performance by Reducing Calls to the Driver. In OpenGL Insights, A K Peters/CRC Press, pp. 353-364.
10. Boudier, P., & Kubisch, C. (2015). GPU Driven Large Scene Rendering. Presentation at the GPU Technology Conference (GTC 2015), San Jose, CA, USA.
11. Stein, C., Limper, M., Thöner, M., & Behr, J. (2015). hare3d - rendering large models in the browser. In WebGL Insights, pp. 317-332.
12. Disney, O., Roupé, M., Johansson, M., Ris, J., & Höglin, P. (2023). Total BIM on the construction site: a dynamic single source of information. J. Inf. Technol. Constr., 28, 519-538.