Efficient Approximate Computation of Scene Visibility Based on Navigation Meshes and Applications for Navigation and Scene Analysis

Sebastian Freitag* Benjamin Weyers* Torsten W. Kuhlen*

Visual Computing Institute, RWTH Aachen University, Germany — JARA-HPC, Aachen, Germany

ABSTRACT

Scene visibility—the information of which parts of the scene are visible from a certain location—can be used to derive various properties of a virtual environment. For example, it enables the computation of viewpoint quality to determine the informativeness of a viewpoint, helps in constructing virtual tours, and allows to keep track of the objects a user may already have seen. However, computing visibility at runtime may be too computationally expensive for many applications, while sampling the entire scene beforehand introduces a costly precomputation step and may include many samples not needed later on.

Therefore, in this paper, we propose a novel approach to precompute visibility information based on navigation meshes, a polygonal representation of a scene's navigable areas. We show that with only limited precomputation, high accuracy can be achieved in these areas. Furthermore, we demonstrate the usefulness of the approach by means of several applications, including viewpoint quality computation, landmark and room detection, and exploration assistance. In addition, we present a travel interface based on common visibility that we found to result in less cybersickness in a user study.

Index Terms: H.5.1 [Information Interfaces and Presentation]: Multimedia Information Systems—Artificial, augmented, and virtual realities; I.3.6 [Computer Graphics]: Methodology and Techniques—Interaction techniques

1 INTRODUCTION

Supporting users of virtual environments (VEs) to reach their goals efficiently is an important task that often requires a certain understanding of the VE and the user's objectives. For example, salient parts of the environment such as landmarks can be highlighted in the VE or on maps to facilitate navigation [27] only if they are known to the system. However, virtual scenes are often not equipped with the additional information necessary for such supporting interfaces, and supplying it for each scene requires significant manual effort. Instead, this task can to some extent be fulfilled by automatic scene analysis that extracts the necessary information.

An example for this is the automatic adjustment of travel speed in large or multi-scale environments based on the viewer's distance to the scene geometry (e.g., [23, 34, 37, 42]). Furthermore, the usefulness of World-in-Miniatures (WIMs) can be improved, for example, by automatically identifying rooms [38] or stories [25]. Another possibility is the automatic generation of virtual tours based on an understanding of the structure of the scene (e.g., [2, 11]).

A significant amount of information about the scene can be derived from *visibility*, i.e., the knowledge about which parts of the scene are visible from which locations. While such information is often used for occlusion culling to accelerate rendering (e.g., [1, 22, 35, 36]), it can also provide some understanding of the structure of the environment, e.g., to automatically create guided tours guaranteeing that certain parts of the scene are visually accessible [11]. Furthermore, visibility can be used to compute the viewpoint quality of a location [16, 30, 40], which in turn allows to determine the best (i.e., most interesting or informative) positions in the virtual scene [40], adjust travel speed [17], create and improve camera paths through the scene [2, 19], and can be used in image-based modeling [40]. In addition, visibility information can be used to find regions of relative homogeneity—regions where the visible parts of the scene are similar or share a set of visible geometry [11, 22, 35]—or for identifying landmarks that are visible throughout a large area.

However, determining visibility is computationally expensive. Often, it is done by rendering an image or a cubemap from the desired position using an *item buffer*, where each entity—e.g., triangle or object—is drawn in a different color [3, 16, 40]. If this has to be performed in each frame, the additional render pass often causes significant computational overhead. On the other hand, while visibilities usually differ only slightly between spatially close points—except as a result of discontinuities like walls—and can therefore be approximated effectively by interpolating between points computed in advance, sampling an entire 3D scene with sufficient resolution requires an expensive precomputation step and significant memory consumption at runtime. However, for most applications, visibility information only has to be available at locations accessible to the user, which in many realistic applications that use ground-based navigation interfaces is typically in walkable areas.

Therefore, in this paper, we present an approach to computing and storing visibility information for virtual scenes above navigable surfaces that usually requires only inexpensive precomputation, and manageable memory consumption at runtime. Furthermore, we evaluate its performance and accuracy using different virtual scenes. Moreover, we demonstrate its viability for several applications, including automatic landmark and room detection and an exploration assistance interface, in addition to a novel travel interface that we evaluated in a user study.

The rest of the paper is structured as follows. In section 2, we give an overview of related work regarding visibility computation. We describe our method in section 3 and evaluate its performance in section 4. Section 5 describes the proposed assisted travel interface, followed by further applications in section 6. In section 7, we then discuss limitations of the approach, before we conclude the paper in section 8 and give a short outlook on future work.

2 RELATED WORK

Scene visibility computation has various important applications. One example is *occlusion culling*, the process of discarding occluded surfaces to speed up the rendering process. Pre-computing a conservative estimation of visibility—a *potentially visible set* (PVS) [1, 36]—can often significantly reduce the amount of geometry to consider when rendering. Exploiting properties of the virtual environment can lead to further optimizations. For example, the structure of many architectural scenes is mapped in cell-and-portal graphs that represent cells that share a PVS (rooms) linked by portals (doors or windows) [22, 35], while the structure of terrains allows optimizations such as the computation of horizons [10]. An overview over methods to compute visibility information for occlusion culling and terrains can be found in [9] and [14]. However, while PVS-based approaches

*e-mail: {freitag | weyers | kuhlen}@vr.rwth-aachen.de

allow to discard geometry that is certainly invisible due to their conservative approximation, they do not capture *how well* certain parts of the environment are visible, nor can reliably indicate *if* they are visible at all. In contrast, exact visibility sets have been used, e.g., in a voxel-based approach for guided tours that divides the space regarding the visibility of landmarks [11]. However, this is only efficient for very few landmarks due to the potentially exponential number of sets and does not reflect how well each landmark is visible.

In contrast, the computation of viewpoint quality requires an accurate approximation of the visibility of all scene entities. The visibility of polygons and objects can, for example, be computed on the CPU by sampling the scene using raycasting [19]. Furthermore, vertex visibility for a set of viewpoints arranged in a regular grid can be approximated using a reverse formulation, and projecting the geometry as seen from the vertex onto the grid [32]. However, due to its good performance, the most common approach to approximate visibility in geometry-based scenes uses the GPU, rendering each entity in the scene (e.g., triangles [2, 3, 12, 16, 28, 30, 39], objects [16, 24], vertices [16] or domain-specific entities like atoms and bonds in molecular visualization [41] or residue features in protein structures [18]) into an item buffer and summing up each contribution. For volume rendering, the visibility of voxels has been computed by raycasting due to the massive transparency used [5, 7]. An alternative approach uses the aforementioned geometry-based GPU approach on isosurfaces extracted from a scalar field [33].

In all of these approaches, the visibility information is not stored, but only used to compute, e.g., a scalar viewpoint quality value for each position of a regular sampling within or around the scene, which requires new computations when, e.g., the importance of entities changes. In contrast, we aim at efficiently computing and storing visibility information even for large scenes, to use them for different applications such as viewpoint quality, landmark detection, and different travel interfaces.

3 Visibility Computation

The visibility of the scene from a position can be represented by a *visibility histogram* $a = (a_1,...,a_n)$ with $\sum_i a_i = 1$, where the bin a_i represents the *visual size* of the i-th entity (e.g., triangle or object) in the scene, i.e., its area when projected onto a sphere around that position, relative to the surface of that sphere [2, 16]. Visibility histograms can be approximated well by rendering a cubemap of the scene from the viewpoint, drawing each entity in a different color (using an item buffer), before counting the pixel area of each entity. To correct for perspective distortion, the pixel areas are weighted by their subtended solid angle (similar to [2, 16]).

Although this is computed efficiently on the GPU and a limited render resolution is usually sufficient, the process often takes several milliseconds, which can be too long for many applications. However, between close points, the visibility usually differs only slightly (unless there is a discontinuity, such as an obstacle, between them). Therefore, good approximations can be achieved by sampling the space and storing a visibility histogram at each sample point, then obtain the visibility for new points by interpolation. As most of a scene is usually invisible from most viewpoints, most visibility histograms are sparse and can be stored efficiently (cf. section 4.4).

3.1 Storing and Accessing Visibility Information

As fundamental entity, we use objects instead of geometric primitives for two reasons. First, there are usually significantly fewer objects than geometric primitives in a scene, leading to a considerably smaller memory footprint to store each histogram. Second, objects are semantically more expressive than vertices or triangles—they are typically the entities a user perceives, such that the question of whether a certain object is visible from some location is probably more meaningful than for a certain triangle. As it is usually infeasible to specify the scene's objects manually, we determine objects automatically, using

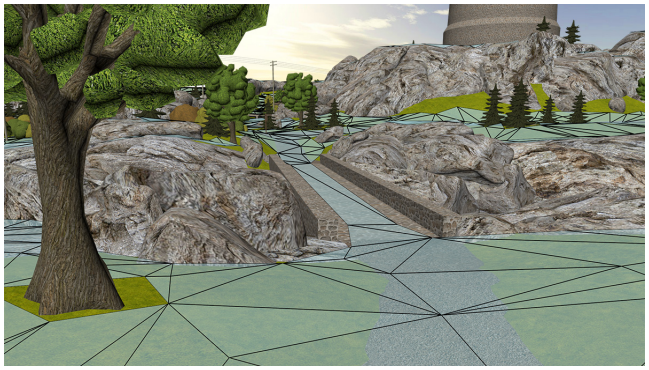

Figure 1: A navigation mesh produced by the open-source toolset *Recast*, overlayed on a countryside scene.

the object definition from [16], where each lowest-level geometry of the scene model is interpreted as a separate object. However, we extend this definition by additionally splitting object meshes into their connected components and considering each a separate object. The reason for this is that mesh parts with the same material across the scene are often stored as a single geometry to improve storage and rendering efficiency, and would otherwise be treated as a single large object.

Note that this definition does not necessarily correspond to a human observer's perception of an object—in fact, many of the resulting objects would probably be considered parts of objects. However, grouping in human perception is not unambiguous and highly subjective, and we found this definition to provide reasonable results. Furthermore, our method only requires that there is *any* object definition—all methods described in the following can be used equally well with any other object or entity definition, manually defined objects, or polygons, although storing polygon visibility would require more memory.

The simplest way to sample and store visibility information is probably in a regular 3D grid throughout the bounding box of the scene. However, this will include many points at locations inaccessible to users, such as inside walls, below the ground, or, if ground-based navigation is used, way above the surface. Furthermore, this method is insensitive to discontinuities, resulting in high errors when interpolating between points on opposite sides of walls or obstacles. If the height of the ground is uniform and known, it is often possible to use a 2D grid at a fixed height instead (e.g., [16, 17]). However, walls and obstacles will still lead to interpolation errors.

Therefore, in our approach, we use *navigation meshes* (also called *navmeshes*) as underlying data structure to sample the scene. Navigation meshes [31] represent an approximation of the navigable surface of a virtual scene as a polygonal mesh (cf. Fig. 1), and are commonly used for automated pathfinding of autonomous agents through navigable areas of a 3D scene. A convenient property of navmeshes for our purpose is that, in addition to representing the regions most accessible to users in many applications, they implicitly encode obstacles. Although navmeshes can be created manually, they can also be efficiently generated automatically, and are widely used as the standard solution for automated pathfinding in many game engines and authoring platforms such as Unreal Engine or Unity3D. Most approaches allow to define the *agent height* and *radius* as a convenient measure of the user's space requirements and minimum distance from obstacles, as well as the *maximum slope* considered navigable. For this work, we used the open-source toolset *Recast*[1] to generate navmeshes (which is also included in the Unreal Engine), as it is freely available and provides good results.

[1] https://github.com/recastnavigation/recastnavigation

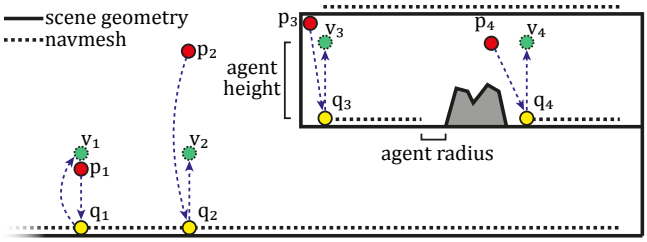

Figure 2: Determination of a visibility histogram. For a query position p_x, a point q_x on the navmesh is found, and its associated visibility histogram (computed for viewer position v_x) is returned.

Our approach uses the vertices of the navmesh—more precisely, points at agent height above the navmesh—as sample points for computing the visibility information. This is advantageous, as the vertices of the navmesh are placed at locations where the navigable surface changes direction or shape, for example, at corners or obstacles. These locations often coincide with places where visibility changes, for example due to reaching the end of a wall or obstacle, making them good positions for sample points. In addition, there are never any obstacles between any point on a polygon of the navmesh and its vertices. Therefore, an interpolation on the polygon face will never go through walls, avoiding large interpolation errors in most cases.

To determine a visibility histogram for an observer position p, the following steps are performed:

1. Try to project p vertically down onto point q on the navmesh.
2. If q exists:
   (a) If distance$(p,q) \leq$ agent height (Fig. 2, p_1):
      i. p is in a navigable area. Compute the visibility histogram for q by linearly interpolating the visibility histograms associated with the three vertices of the navmesh triangle containing q, using its barycentric coordinates to weight each contribution.
   (b) Else: perform an intersection test with the scene geometry to check for obstacles between p and q.
      i. No obstacles (Fig. 2, p_2): p is a tall observer or flying above the scene. Return interpolated histogram at q.
      ii. Obstacles: p is close to an obstacle (Fig. 2, p_3), above a non-navigable obstacle (Fig. 2, p_4) or outside the scene. Find the point q on the navmesh closest to, but below p and return interpolated histogram at q.

Note that in 2(b)(ii), we have to find q below p, as navmesh parts above p represent navigable surfaces above (e.g., the next floor above, cf. Fig. 2), and therefore typically very different visibility. However, this does not ensure that interpolations through obstacles are avoided, and can lead to similar interpolation errors as when sampling, e.g., along a regular 3D grid. Alternatively, if visibility information is requested for a user moving through the scene who left the navmesh only temporarily, the last valid histogram can simply be retained. However, if the information has to be reliable, no histogram should be computed at all to reflect that no reliable approximation can be made.

3.2 Navmesh Refinement

While navmesh vertices are often placed at positions where the visibility changes, they may be far apart, reducing the accuracy of the interpolation. Furthermore, visibilities may differ significantly between adjacent vertices, especially between positions close to an obstacle or wall (that takes up much of the view) and positions a little farther away.

Therefore, our approach refines the navmesh whenever visibility information at neighboring vertices differs too much. For each edge of the navmesh, we compute the *histogram intersection* $HI(a,b) = \sum_i \min(a_i,b_i) \in [0,1]$ of the (individually normalized) visibility histograms a and b of its vertices. If the refinement criterion $HI(a,b) < \theta$ for some threshold $\theta \in [0,1)$ is met, an edge split is performed by inserting a new vertex at the edge's midpoint (see Fig. 3), and computing the visibility histogram for the new position. This process is repeated iteratively until no edge fulfilling the refinement criterion is left.

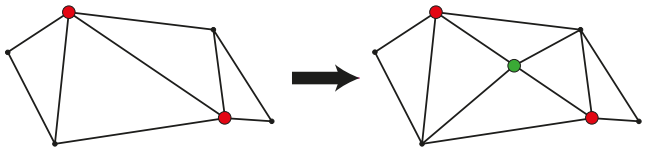

Figure 3: When the visibility histograms at the vertices of an edge differ too much, the navmesh is refined by performing an edge split.

An advantage of using the histogram intersection as refinement criterion is its clear interpretability. For example, if $HI(a,b) = 0$, then the positions with the visibility histograms a and b see completely different objects. If $HI(a,b) = 0.5$, half of each field of view is taken up by the same objects, while the other half is different, etc. For measurements regarding the effects of different thresholds θ, see section 4.

3.3 Online Computation and Parallelization

To avoid a precomputation step, the visibility information can also be computed online. We do this in a separate thread, independent of the main application thread, using the *glfw* library[2] for OpenGL. By prioritizing positions close to the user, visibility information can always be provided for the user's location, as long as they do not move too fast. Note that computing visibilities for the complete scene online usually takes significantly longer, depending on the GPU load of the main application.

Furthermore, as all visibility computations are independent of each other, the computation process can be parallelized trivially for multi-GPU and cluster setups, such as many powerwall or CAVE installations. Measurements of the speedup we achieved are summarized in section 4.4.

4 Evaluation

In this section, we show and evaluate the results of our method described in section 3, measuring its accuracy and performance on different scenes.

4.1 Scenes

We tested our method on 8 different realistic scenes that cover different types of environments (e.g., indoor/outdoor, single-floor/multistory) and different complexities:

- **countryside**: A large, sparse outdoor scene
- **city**: A low-detail city generated automatically using the Esri CityEngine[3]
- **house1**: A highly detailed house with small garden and patio
- **house2**: A highly detailed small house
- **office**: A large office floor
- **bookstore**: A large, single-floor bookstore with a small café
- **office buildings**: Two large, low-detail multi-story office buildings in a small, park-like outdoor area
- **campus**: A university campus, with one of the lecture buildings modeled in detail and walkable

Characteristic information about each scene is summarized in Table 1, screenshots of some of the scenes are shown in Figure 4. All navmeshes were generated using *Recast* with the following settings: agent height=1.5 m, agent radius=0.1 m, maximum slope=45°, cell size=0.1 m (*city* and *countryside*: cell size=0.5 m, *campus*: cell size=0.2 m, due to larger scene size). For all visibility computations, we used an eye height of 1.5 m above the navmesh, and a resolution of 256×256 pixels for each side of the cube map.

[2] www.glfw.org
[3] www.esri.com/software/cityengine

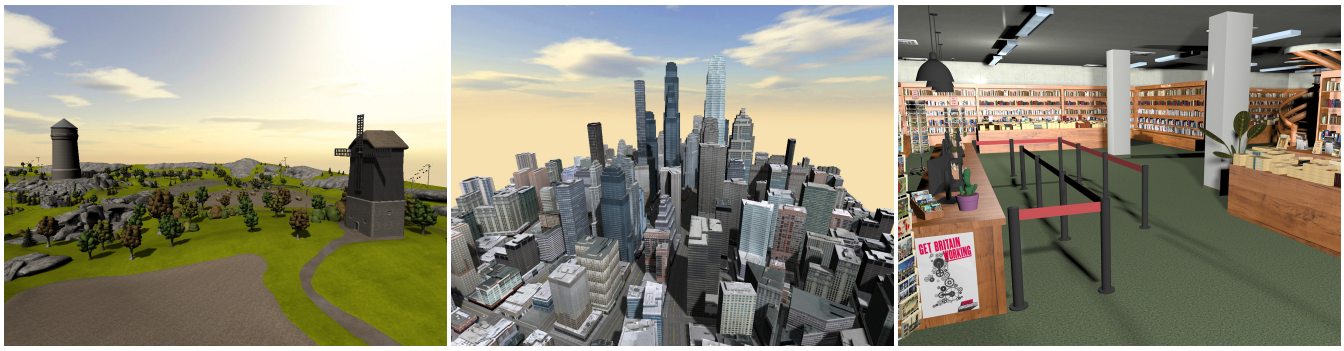

Figure 4: Screenshots of some of the scenes used in our evaluation. Left: *countryside*, Center: *city*, Right: *bookstore*.

Table 1: Characteristic information of the scenes used in the evaluation.

		Scene				Navmesh	
Name	Type	Bounding Box	#Triangles	#Objects	Layered	#Vertices	#Triangles
countryside	outdoor	$498\times500\times90$	1,874,237	34,420	No	13,730	20,502
city	outdoor	$901\times910\times322$	73,259	4,728	Yes*	25,590	20,606
house1	indoor	$20\times15\times5$	2,377,158	1,121	Yes*	301	383
house2	indoor	$11\times14\times5$	429,557	3,123	Yes*	103	120
office	indoor	$46\times31\times6$	3,147,964	3,296	No	778	771
bookstore	indoor	$42\times49\times7$	1,268,120	22,250	Yes*	628	727
office buildings	mixed	$125\times75\times21$	188,262	1,231	Yes	3,123	3,597
campus	mixed	$390\times392\times45$	1,135,843	63,373	Yes	6,840	9,362

* Scene includes building tops representing navigable areas

4.2 Accuracy and Refinement

We measured the average approximation accuracy of our method on all scenes as a function of the refinement threshold θ. The approximation accuracy for a certain position is given as the histogram intersection of the actual and the interpolated visibility histograms at that position. For each scene and refinement threshold, we randomly sampled 10,000 positions in the navigable area of the scene, measured their accuracy, and averaged over all values.

The accuracy results, along with the increase in the number of visibility histograms to be computed, are summarized in Figure 5. While the necessary refinement threshold to reach a certain approximation accuracy varies across scenes, most already provide an arguably high accuracy without refinement due to the structure of the navmesh (except for the *city* scene). Furthermore, on all scenes but the *city*, an accuracy of more than 0.9 is reached at $\theta = 0.65$, while requiring visibility computations for less than 50% additional positions. At $\theta = 0.8$, the average accuracy is arguably very high, at about 0.95 or above (except *city* with 0.894), so we use this threshold throughout the rest of the paper. However, in real applications, the threshold can also be raised iteratively, stopping after a certain time or number of computed positions.

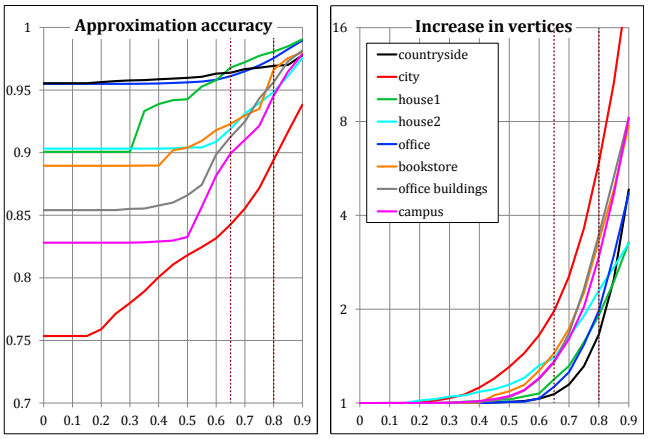

Figure 5: The average approximation accuracy (left) and the factor of increase of the total number of computed visibility histograms (right) for each tested scene as a function of the refinement threshold θ.

4.3 Comparison

We compared the average approximation accuracy of our approach to that of a regular, uniform 3D grid with a similar number of sample points. We always chose $\theta = 0.8$ as refinement threshold, and selected the 3D grid resolution in a way that the resulting number of sample points was about the same. For each scene, we randomly sampled both 10,000 positions in the navigable area of the scene and 10,000 positions within its bounding box, and averaged the approximation accuracy for each.

The results are summarized in Table 2. It is evident that our approach is superior when providing visibility information in navigable areas, while the regular 3D grid provides better interpolation results for arbitrary positions on average. However, in large scenes, the average is dominated by "flying positions" high above the ground, where the visibility changes with low frequency, making low sampling feasible. For example, in the *campus* scene, from the vast majority of viewpoints, only a few very large low-detail buildings are visible, while the volume of building interior modeled in detail makes up only 0.2% of the scene. In addition, note that although the 3D sampling sometimes incorrectly assumes objects to be visible that are on the other side of a wall due to interpolation, the effect of this on the average accuracy is limited, as the majority of query positions are in open space.

A simple method to improve accuracy across the whole scene is to combine both methods, using our approach for (often highly detailed) navigable areas, while a low-resolution 3D sampling of the scene might suffice in areas way above the surface.

Table 2: Average accuracy of our approach compared to sampling along a regular 3D grid. Acc_N specifies accuracy results for a random sampling in the navigable areas of the scene, Acc_{3D} within the scene's bounding box. For both samplings, the higher value is marked in bold.

	Our approach			Regular 3D grid			
Name	#Points	Acc_N	Acc_{3D}	Grid	#Points	Acc_N	Acc_{3D}
countryside	22,774	**0.975**	0.691	9.81	22,950	0.752	**0.878**
city	151,244	**0.894**	0.361	11.94	153,900	0.171	**0.629**
house1	576	**0.981**	0.667	1.27	576	0.649	**0.814**
house2	237	**0.969**	0.511	1.39	240	0.463	**0.681**
office	1,541	**0.948**	0.519	1.66	1,596	0.919	**0.854**
bookstore	2,085	**0.966**	0.765	1.69	2,088	0.688	**0.775**
office buildings	10,847	**0.956**	0.782	2.66	10,904	0.804	**0.891**
campus	20,157	**0.946**	0.477	26.68	20,230	0.462	**0.978**

Table 3: Average time to complete the precomputation for our method ($\theta = 0.8$) on a single-GPU workstation, and memory required to store the resulting visibility information and navmesh structure.

Scene	#Points	Per point	Complete	Memory
countryside	22,774	8.8 ms	201.1 s	183.6 MB
city	151,244	8.4 ms	1,276.6 s	125.7 MB
house1	576	10.8 ms	6.2 s	0.6 MB
house2	237	7.3 ms	1.7 s	0.7 MB
office	1,541	10.7 ms	16.6 s	1.2 MB
bookstore	2,085	9.0 ms	18.8 s	29.1 MB
office buildings	10,847	7.0 ms	76.9 s	11.3 MB
campus	20,157	9.5 ms	191.4 s	199.9 MB

Table 4: Performance and speedup achieved when computing visibilities using typical VR cluster configurations (*countryside* scene).

Setup	Single node	Cluster	Speedup
Powerwall, 2 nodes*	299.1 s	159.5 s	1.9
Powerwall, 6 nodes†	184.5 s	38.6 s	4.8
CAVE, 49 nodes*	300.2 s	19.1 s	15.7

* Intel Xeon X7550 @ 2 GHz, NVIDIA Quadro 6000
† Intel Xeon E5-1620 @ 3.7 GHz, NVIDIA GeForce GTX 780 Ti

4.4 Performance

We measured the time to complete the precomputation necessary for our method (using $\theta = 0.8$) on a typical workstation machine equipped with an Intel Xeon E3-1225 3.2 GHz Dual-Core processor and an NVIDIA GeForce GTX 1070 GPU, running Windows 7, by averaging over 10 identical runs. The results, including memory consumption, are summarized in Table 3. For most scenes—with the exception of the *city* scene that requires many sample points due to its large size and high occlusion—the complete computation takes less than 3.5 minutes, and has to be performed only once. Note that although the scenes vary significantly in geometric complexity, the average time to compute the visibility at each point is similar, mainly due to optimizations such as frustum culling, and constant contributions, such as reading the frame buffer. The memory requirements consist mainly of the sparsely stored visibility histograms, each requiring 8 bytes for each visible object (32 bit for each visible object's index and 32 bit for its visibility—note that to save memory, in many cases, 16 bit for the index and a 16 bit half-precision floating-point number for the visibility would suffice).

Furthermore, we measured the performance in three multi-GPU cluster setups: a 2-node and a 6-node powerwall, and a 49-node CAVE setup, all running CentOS 7. The times to complete the necessary precomputation for each setup (simply distributing the workload between nodes evenly), as well as the speedup achieved, are listed in Table 4. For conciseness, we only report results on the *countryside* scene, as it is large both regarding geometric complexity and actual size.

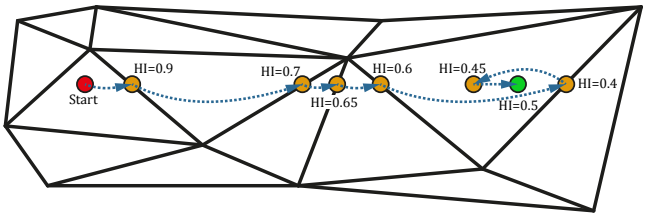

Figure 6: Starting from the red dot, the end of the region of common visibility (green dot) is found by sampling points at navmesh edges, then doing binary search on the final navmesh triangle.

5 Assisted Travel based on Common Visibility

Especially in larger virtual environments, detail is often not distributed evenly throughout the scene, such that users often have to travel through less interesting areas. If a steering-based travel technique is used, this either leads to longer travel times, or the necessity to increase the travel speed in these areas. However, manual speed changes require the user to control an additional parameter (e.g., via a menu), potentially increasing cognitive load and error rates [17]. Therefore, several methods to adapt the travel speed automatically have been proposed, often based on the distance to the environment [23, 34, 37, 42]. However, for ground-based navigation, the distance to the scene geometry is largely constant, such that these methods cannot be used. As an alternative, choosing the speed inversely proportional to the local viewpoint quality has been suggested [17], but may lead to speed changes intransparent to the user. Furthermore, faster speeds are in general harder for the user to control, and may lead to errors (e.g., traveling too far) or cybersickness due to frequent direction changes caused by inaccurate steering or swaying.

Therefore, we propose an alternative travel approach, based on the observation that segments of a path where the environment changes only slightly (e.g., in low-detail regions, or in open areas far away from most objects) are often the ones that users want to pass quickly. These can be determined by finding regions of *common visibility*, i.e., areas throughout which the visible parts of the scene are similar. When such a region is encountered, the proposed interface suggests to quickly move through it along a straight line, to avoid errors and changes in direction and acceleration that may cause cybersickness [21].

5.1 Method

We define the *region of common visibility* for a position p with visibility histogram v_p to be comprised of all positions q where half of the visible parts of the scene is identical to p, i.e., all points with visibility histogram v_q where $HI(v_p, v_q) \geq 0.5$.

The proposed interface is based on a simple ground-based steering-by-pointing travel technique, where the user points in the desired travel direction using an ART Flystick 2 or a similar device. The joystick on top of the Flystick is moved forward to seamlessly control the movement speed in that direction along the ground surface. Whenever the user is traveling with the maximum speed, the system tries to find a travel target candidate by computing the boundary of the current position's region of common visibility in movement direction. To do this, the visibility histograms are sampled in that direction at every navmesh edge until the histogram intersection with the current position drops below 0.5 (cf. Fig. 6). The target position is subsequently refined to find the exact position where the intersection is 0.5, by doing a binary search on the corresponding navmesh triangle. Alternatively, when the outer edge of the navmesh is reached, sampling is also stopped and the position 1 m before the edge is selected as target position.

Next, the system calculates the minimum speed necessary to reach the target within a time of t_{max} (we use $t_{max} = 2$ s), out of a set of discrete speed levels. We chose discrete levels to make differences between speeds noticeable, to give the user a sense of the distance

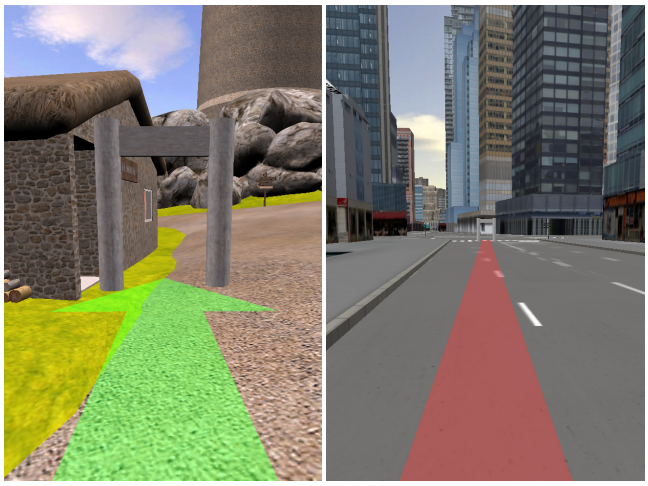

Figure 7: Visual feedback for a target suggestion in the *countryside* (left) and *city* scene (right), indicating low (left) and high (right) speed and distance to the target.

covered during automated movement. If traveling directly with this speed saves time compared to normal steering (our implementation requires saving at least 2 s), a suggestion is made, visually illustrating the path to the proposed target using an arrow on the ground leading to a gate at the end (see Fig. 7). The arrow is colored depending on the speed level, to give the user an indication of the speed and the distance to the target.

When the user presses a button, the suggestion is accepted and the user is transported to the target along a straight line (but restricted to the ground). The motion can also be aborted by pressing the button again. Note that a button already reserved for other interaction (e.g., selection or menus) can be used, as most interaction is usually not performed while traveling.

The suggestion is updated regularly, using a rather high update rate of 4 Hz determined in pilot tests. When the new suggested target position differs by a visual angle of less than 5° and a distance of at most 10 m, the target is continuously moved there in a straight line to reduce user errors, while otherwise jumping to the new position instantly to avoid delays.

Furthermore, some steps are performed to improve the suggested targets. Very large or close objects have a large visual size, and correspondingly large entries in the visibility histogram. A consequence of this is that the histogram intersection between visibility histograms of, for example, a point close to a wall and one a little farther away is relatively low, even though the visual size of almost all visible objects—except for the wall—changes only very little. To prevent these objects from dominating the determination of common visibility regions, we cap the visual size at 0.025 (corresponding to about the size of a door at a distance of 2.5 m), distributing the excess on all other objects proportionally to their respective visual size. The effect on the resulting common visibility region is illustrated in Figure 8. Note that this procedure is relatively robust with respect to the cap used—in our tests, values between 0.01 and 0.1 led to similar results.

Furthermore, the target position is corrected along the movement direction based on a viewpoint quality measure. As the runtime of viewpoint quality estimation methods is usually clearly dominated by the determination of visibility, it can be computed very efficiently from a visibility histogram if the method is based on the same entities (e.g., objects). We used *object area entropy* [16] to measure quality and select the position with the highest viewpoint quality within ±40% of the initially computed distance along the indicated direction as target. This is done for two reasons. First, target locations with a higher viewpoint quality are typically more favorable targets, both for

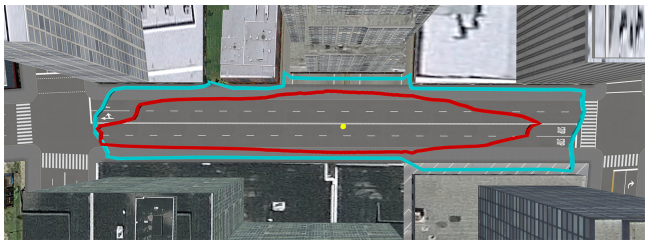

Figure 8: Regions of common visibility in the *city* scene, computed for the central yellow position, with (cyan) and without (red) capping of the visual size of each object.

viewing the environment (more informative view) and for subsequent travel (e.g., most points very close to objects have a lower quality due to one object dominating the view). Second, this correction improves the stability of the target position, which is moved less frequently while the user travels forward.

Especially in hilly terrain, the direction of movement is changed often when using ground-based travel due to changes in slope, which we found to contribute to cybersickness. Therefore, we simplify the path to the target by allowing slight deviations from the ground. To this end, we apply the Ramer-Douglas-Peucker curve simplification algorithm [29] to the (piecewise linear) path, removing vertices that change the path only slightly, allowing for a maximum deviation of the user's feet of 2 m above or 0.5 m below the ground. We found in pilot tests, and were confirmed of this in our user study (section 5.2), that most users do not notice that their feet leave the ground within this range when traveling to the target.

5.2 User Study

We tested the proposed travel interface (**A**) in a user study against a ground-based point-and-fly method (also using the Flystick) where the maximum travel speed could be changed manually using a menu (**M**). We used this manual approach as comparison as its interaction is very similar to **A**, except that the speed changes performed automatically through suggestions are replaced with manual changes. Similar to manual steering in **A**, when a faster speed is chosen in **M**, the user can still control the speed continuously through the joystick on the Flystick. After pilot testing, we chose the speed levels of 2 m/s, 8 m/s, 24 m/s and 48 m/s for both travel techniques, where in **A**, 2 m/s was used for manual steering, and the faster speed levels for automated travel. To compute viewpoint quality, we used the *object area entropy* method [16].

For the study task, we tried to imitate a realistic scenario, where users had to travel medium distances in order to perform some local task at the target. We were mainly interested in whether using **A** makes travel easier or harder, if it can reduce cybersickness, and whether the automated travel has an impact on spatial orientation.

The study was conducted in a 5-sided CAVE system (no ceiling, 5.25 m×5.25 m floor area, 60 Hz optical tracking, 60 Hz refresh rate). It consisted of six phases, one training phase for each travel technique, and one trial for each technique on each of the *city* and *countryside* scenes. In the beginning, participants were thoroughly informed about both travel techniques and the study procedure, and filled out Kennedy's simulator sickness questionnaire (SSQ) [20]. Then, they entered the CAVE and performed the first training, practicing the travel technique in a training scene until they felt comfortable using it, before starting the first trial with the same technique. Afterwards, the second training phase with the second travel technique began, followed by the second trial on the same scene, and the third and fourth trial on the second scene. Then, participants left the CAVE and filled out Kennedy's SSQ again, in addition to a questionnaire comparing both techniques regarding efficiency, precision, cybersickness, ease of use and overall preference.

During each trial, participants had to travel to 5 houses distributed throughout the scene by following signposts, and perform a simple search task in each house. Both trials on the same scene used the same travel path, except that one of them was reversed. In the house, they had to find a certain object (e.g., a fire extinguisher or soccer ball) and "touch" it with the Flystick, before leaving the house again to receive instructions which house and object to find next. The search task was kept intentionally easy, as it was mainly meant to be a break from medium distance travel. All instructions were always displayed on large billboards placed in the virtual scene.

Whenever participants reached a house, they would be asked to point in the direction of the last house they visited, and the house they started their journey at. Then, they indicated how they were feeling, using a scale from 0 (indicating how they felt before the experiment) to 7 (indicating that they wanted to abort the experiment), similar to Fernandes et al.'s *discomfort score* [13]. We used this simple indication of cybersickness instead of, e.g., Kennedy's SSQ, as we wanted to avoid long interruptions during the study. However, we only evaluated the last of these scores in each trial, as it is least affected by previous trials, and used the others only to check on the participants' well-being.

A trial ended when participants had found the object in the fifth house, after which they could take a short break if they wanted (8 participants made use of this after the second trial, and 2 after each trial). The total procedure took an average of 73 minutes, of which 43 minutes were spent in the CAVE (33 min. during trials, 5 min. during training and 5 min. for loading).

In total, 35 unpaid individuals participated in the experiment (8 female, 27 male, aged 20 to 45, mean age 27.3). 9 were VR professionals, 13 had used a CAVE or HMD at least once before, and 13 were first-time users. 3 participants had to abort the experiment due to cybersickness, leaving 32 sets of data to evaluate. We counterbalanced the order of techniques, scenes, and whether the path was reversed between participants (resulting in 8 different setups with 4 participants each). Furthermore, participants were balanced by gender and previous experience (first-time user, repeated user, VR professional) concerning the order of techniques and scenes.

5.2.1 Results

We analyzed the effects of the travel technique on different measurements using independent-samples t-tests at the .05 level of significance.

We found no significant effect on the mean time spent traveling between houses (**M**: 176.5 s per trial, SD=94.4 s; **A**: 199.5 s, SD=58.8 s, p=.101), or during the search task (**M**: 110.5 s per trial, SD=42.7 s; **A**: 104.3 s, SD=57.3 s, p=.488). Furthermore, there was no significant difference in the distance covered between houses (**M**: 1819.4 m per trial, SD=135.8 m; **A**: 1836.4 m, SD=206.8 m, p=.584) or during the search task (**M**: 100.9 m per trial, SD=30.9 m; **A**: 96.8 m, SD=36.2 m, p=.491). Moreover, there was no effect on the average error pointing to the last house (**M**: 19.5°, SD=11.3°; **A**: 20.8°, SD=13.2°, p=.542) or to the first house (**M**: 43.4°, SD=21.3°; **A**: 44.5°, SD=25.8°, p=.798). However, we found a significant effect on the discomfort score (**M**: 1.92, SD=1.8; **A**: 1.30, SD=1.4, p=.031). The mean SSQ score was 11.2 (SD=14.8) before and 42.5 (SD=33.4) after the experiment. We found a strong significant correlation between the discomfort score and the increase in SSQ score (r=.82, $p<.001$), supporting the validity of this way of measuring discomfort.

Furthermore, we obtained results from the questionnaire where participants compared both techniques regarding different properties on a 7-point scale, choosing a value closer to 1 the more a statement was true for **M**, and closer to 7 the more it was true for **A**. We analyzed whether the median was different from the neutral value of 4 using one-sample Wilcoxon signed-rank tests.

Participants perceived that they could reach their target faster using **M** (3, IQR=[2,5], p=.014) and could also travel more precisely with **M** (2, IQR=[1,3], $p<.001$). Furthermore, they found **M** easier to use (3, IQR=[2,4], p=.019). The rating of which technique caused more dizziness barely missed significance (3, IQR=[2,5], p=.057). Overall, participants preferred **M** (2, IQR=[1,5], $p<.001$).

When examining the data based on VR experience or whether participants were VR professionals, we did not find significantly different tendencies, such that details are omitted here for brevity.

5.2.2 Discussion

The proposed method had a positive impact on cybersickness, as indicated by the discomfort score results (questionnaire results pointed in the same direction, but barely missed significance at $p = .057$). We attribute this to the fact that users mainly moved in straight lines with the method, did not do any curves, did not change the speed while moving, and were less affected by the hilly terrain in the *countryside* scene. However, some participants told us or wrote in comments that the sudden stop upon reaching the target with **A** made them dizzy. Unfortunately, there is no easy way to avoid this without introducing deceleration (which could again lead to sensory conflicts and cybersickness) or discontinuities like teleportation.

Furthermore, we found no significant differences between **A** and **M** regarding completion times, covered distance, or pointing errors, which suggests that objectively, the proposed method performs similarly well and does not introduce negative effects on spatial orientation due to automated travel. This supports our assumption that the placement of the target suggestions is generally good.

Nevertheless, participants preferred **M**, finding it easier to use and more precise. This is probably due to participants feeling limited by the choice made through the target suggestions, wanting to go farther or not as far, or with a different speed. Furthermore, we observed anecdotally that participants would sometimes ignore a well-placed suggestion while traveling, waiting for one that suited them better, especially when the suggested target was not as far away as they seemed to expect. We suspect that participants construct mental waypoints that they try to reach with the method, and are irritated when the system suggests different waypoints. This might be alleviated by giving users a better understanding of the method and more choice, e.g., by explicitly showing the edge of the region of common visibility as a semi-transparent wall and possibly allowing them to choose between different targets resulting from different thresholds of common visibility.

Finally, in the study, the participants' decisions on where to go next were not based on the environment and its structure, but mainly on the inscriptions on signposts. However, an underlying assumption of the proposed method is that users will usually change direction when the visible parts of the scene change significantly, e.g., when reaching a corner. While the method stopped reliably, e.g., at intersections in the *city* scene (cf. Fig. 8), it largely ignored the signposts, failing to guide users just close enough that they would be able to read it. Therefore, we suspect the performance to be better in scenarios where users can decide more freely where to go.

6 Further Applications

The availability of approximate visibility information throughout the scene enables a wide range of applications. This section illustrates its broad applicability by means of several further applications based on visibility computation: the efficient calculation of a best set of viewpoints (section 6.1), an exploration assistance interface (section 6.2), landmark detection (section 6.3) and room detection (section 6.4).

6.1 Efficient Best Set of Views

While viewpoint quality can directly be used to determine the best (most informative) location in a virtual environment, one position is usually not enough to represent an entire scene. However, computing an optimal set of n views [19, 24, 30, 40] with limited redundancy between them, that best represents the scene, is not trivial—in fact, it is NP-hard (related to the art gallery problem [26]). The most common and simple method to approximate an optimal set is a greedy approach,

Figure 9: Assisted exploration of the *office* scene. Top: viewpoint qualities computed using the *object uniqueness* method before (left) and after (right) the user has moved along the indicated path through the scene. Bottom: upon command, three portals to suggested target locations (marked with a green dot in the map on the upper right) are opened, allowing the user to step through to go there.

where iteratively the best viewpoint is added to the set and all parts of the scene visible from there are subsequently ignored, until the set has size n (e.g., [19, 40]). However, this requires the viewpoint quality of all candidate points in the scene to be computed in each iteration. If visibility information is available, both determining which parts of the scene to subsequently ignore and recomputing viewpoint quality can be performed efficiently at runtime. This allows best sets of views to be computed interactively, which we used as a basis for an exploration assistance interface (section 6.2).

6.2 Interactive Exploration Assistance

When visiting an unknown virtual environment, one of the first and most important tasks is the *exploration* of the scene [6]. However, during free exploration, important parts of the scene are often missed, which can have a negative impact on spatial knowledge and subsequent tasks. One approach that has been proposed for this are virtual tours through the scene that ensure that no essential regions are missed [2, 11]. However, the interactivity of virtual tours is limited, and they usually do not allow the manual exploration of parts of the environment.

As an alternative, we propose an interactive, "Show me what I've missed" approach that keeps track of the parts of the scene the user has already seen and suggests worthwhile travel targets on command.

In the beginning, each object is assigned a weight of 1.0, which is reduced whenever the object is visible to the user. When the user asks for support, a current best set of views (see section 6.1; we use $n=3$ positions) is computed, taking into account the weight of each object. Then, a virtual portal is opened to each of the target locations (similar to [15] or [8]) that the user can step through to go there, but also use to inspect the target in advance (see Fig 9, bottom). The portals are placed in a half-circle in front of the user and, at the target location, rotated in the direction with most free space. Note that although virtual portals enable the user to look at each target location before making their choice and reach it without inducing cybersickness through virtual movement, the user does not learn the way there. As an alternative, better spatial knowledge of the scene may be obtained by letting the user choose one of the portals before just displaying a route there for the user to follow.

In our prototype, the object weight is reduced linearly with time and with how well it is visible from the user's position (cf. Fig 9, top), following the rationale that smaller or far-away objects should warrant closer inspection. We use a weight reduction rate (per second) of $r=\max\left(\frac{1}{t_{max}}, \frac{\min(vis,vis_{max})}{vis_{max}\cdot t_{min}}\right)$, where $vis\in[0,1]$ is the *visual size* of the object, and t_{min} and t_{max} correspond to the minimum/maximum time it should take to reduce the weight of a visible object to 0 (we use $t_{min}=3$ s, $t_{max}=15$ s). Furthermore, we consider an object "well visible" at a visual size of $vis_{max}=0.001$ (corresponding to about the size of a postcard at a distance of 1 m), and reduce the weight of objects of that size (or larger) to 0 in the minimum time of t_{min}.

Note that our method to determine visibility is non-directional, while the user's field of view is limited. Therefore, we tested an approach where only objects within a cone around the user's view direction are considered visible, which we approximate based on the user's head position and orientation, the object's center and its visual size. However, pilot tests indicate that this tends to push users to closely examine objects, suggesting locations visited before shortly but not closely inspected, so we retained the non-directional approach. Nevertheless, this should be studied more thoroughly in a formal user study.

Furthermore, to avoid unhelpful suggestions after a scene has been completely explored, only viewpoints where at least one visible object retains a weight of at least 0.5 are considered for suggestion.

6.3 Automatic Landmark Detection

A user's orientation in certain virtual environments can be improved if landmarks are known to the system, for example by integrating them in virtual maps or navigation interfaces [27]. Landmarks are often objects or structures that can be seen from a large part of the navigable areas of the scene, such as churches, towers, or mountains. Furthermore, they are usually spatially compact relative to the area they are visible from—e.g., while a large pasture is visible from a large area, it is not normally regarded as a landmark, as this area is typically the pasture itself and a small stretch around it.

Based on this observation, we developed a simple algorithm to automatically detect n landmarks in a scene:

1. *Find well visible objects:* For each object in the scene, determine the area from which it is visible. This can be approximated by summing up the areas of all navmesh triangles where the object is visible from all three vertex positions.
2. *Restrict to spatially compact objects:* Select the m objects visible from the largest area (where m should not be too large to reduce computational cost) and determine their *spatial compactness*. Perform a principal components analysis (PCA) each on the vertices of the object, and on the vertices of the navmesh from where it is visible, ignoring the vertical dimension, to compute the two eigenvalues $\lambda_{a1},\lambda_{a2}$ for the area and $\lambda_{o1},\lambda_{o2}$ for the object. These represent the length of the eigenvectors of the corresponding covariance matrices and are greater the more distributed each set of vertices is in the direction of its eigenvectors. We estimate the compactness as $c=\frac{\lambda_{a1}\cdot\lambda_{a2}}{\lambda_{o1}\cdot\lambda_{o2}}$ and only keep objects with $c>\gamma$ for a threshold γ (we use $\gamma=10000$).
3. *Merge object parts:* As several objects can belong to the same landmark, spatially close landmark candidates are merged. For this, we intersect the 2D bounding boxes of candidate objects (ignoring the vertical dimension), and merge objects where the intersection area is larger than half of the smaller bounding box.
4. *Select final landmarks:* Finally, the n landmark candidates that are visible from the largest area are selected as landmarks.

The results of a run of the algorithm (with $n=3$, $m=20$) on the *countryside* scene are illustrated in Figure 10. As the top 3 landmarks,

Figure 10: Results of the automatic landmark detection in the *countryside* scene. The top 3 landmarks are marked in red, purple and cyan.

Figure 11: Results of the room detection on the *office* scene, each color corresponding to a different room.

the tower, the windmill, and a large tree are identified (in that order), where both tower and windmill consist of three objects each. This seems a fair assessment, as there are arguably only two landmarks in the scene (the tower and the windmill). The computation on our workstation (see section 4) took 4.1 s.

Note that an additional property of landmarks is usually that they are dissimilar to most other objects in the area. This could be integrated by computing a uniqueness measure for each object (e.g., [16]) and discarding objects that are not unique enough (such as the tree).

6.4 Automatic Room Detection

Automatically obtaining information of the room structure of an (indoor) environment is useful for different applications, such as constructing paths between rooms [2] or enhancing World-in-Miniatures [4, 38]. The room structure can be represented as a cell-and-portal graph [2, 4, 9, 38], where cells correspond to rooms and portals to the connections between them. A notable approach to deriving the structure automatically is based on a distance-field representation of the scene to identify cells with approximately constant visibility [38].

We suggest an alternative approach directly based on the visibility information obtained by our method (section 3). Similar to [38], we assume rooms to be regions of common visibility, i.e., from all positions within a room, a similar set of objects is visible. In addition, we use a navmesh as basic structure, using the information about navigable areas as starting point for the room detection.

We developed a simple prototype based on a region-growing process that assigns navmesh vertices to rooms based on the similarity of the visibility histograms at their positions. As in section 5, we cap the visual size of objects at 0.025 to reduce a dominating influence of large or close objects. Our prototype works as follows:

1. *Create a new room* by selecting a navmesh vertex not yet associated with a room and assign it a new room ID
2. *Perform region growing* from this vertex until convergence:
   2a. Compute the mean visibility histogram $\bar{h}$ of the room by averaging the visibility histograms of its vertices
   2b. Compute the histogram similarity s between $\bar{h}$ and the visibility histograms of the room's neighboring vertices
   2c. Assign all neighboring vertices to the room whose visibility is similar enough to $\bar{h}$ (we use $s > 0.5$), and, if they are currently assigned to a different room, are more similar to $\bar{h}$ than to the mean visibility histogram of that room
3. Repeat from 1 while there are unassigned vertices
4. *Merge rooms*: neighboring rooms whose mean visibility histograms are similar enough ($s > 0.5$) are combined
5. *Assign triangles to rooms*: for each navmesh triangle, assign it one of the rooms associated with its vertices, by choosing the one whose mean visibility histogram is most similar to the visibility at the triangle center

The result of a run of this algorithm on the *office* scene, choosing start vertices in step 1 randomly, is illustrated in Figure 11 (computed in 0.12 s). Note that different rooms are correctly detected, although the area in front of doors is often assigned to the room as well. Furthermore, the corridor is split into several "rooms", which may not be desired. The result can be further improved, e.g., by choosing the starting points of the region growing process more carefully. A limitation of the algorithm is that it assumes rooms to contain at least some objects, and may fail, e.g., on scenes with only empty rooms and all walls represented by a single object. However, this may be resolved by splitting large objects, again giving each room unique common visibility.

7 Limitations

Our approach focuses on the computation of scene visibility from navigable parts of the virtual environment. This means that, although the accuracy in these regions is high, non-navigable areas are not (or less accurately) represented. While this is probably less of a concern for locations within objects or below the ground, we cannot provide accurate visibility information for locations reached by flying, which is also a much-used category of travel metaphors. This could be alleviated by extending the navigation mesh in the vertical dimension in regions where flying is possible, e.g., by introducing "ghost planes" above the ground that are included in the computation of the navmesh. Depending on the application, a rough approximation achieved by combining our approach with a low-resolution regular 3D sampling of the environment may also suffice.

Furthermore, a general problem of interpolation approaches is the possibility that changes at positions between sample points are not correctly described by the interpolation. While the navmesh structure captures changes due to close obstacles, and the refinement avoids too much change between sample points, there are cases where these changes may be missed. An example for these may be a window with sample points to its left and right side, but not at a position directly in front of it, from where it is possible to look through. Unfortunately, there is no simple solution to reliably prevent such situations without increasing the number of sample points significantly (similar to aliasing problems when discretely sampling a continuous signal), although we did not notice them in our test scenes.

Finally, our current approach can only deal with static scenes. Although this is sufficient for a large number of use cases concerning scenes that are predominantly unchanging (with dynamic elements whose visibility is either less important or can be approximated otherwise), significant accuracy problems may occur if the structure of the navigable areas of the scene changes. While navigation meshes can be updated—and regularly are in many pathfinding use cases to account for dynamic obstacles—and the visibility for changed sample points can be recomputed quickly during runtime (see section 3.3), it is not trivial in the general case to decide which points have to be updated, as, for example, a newly introduced object may be visible from far away.

8 Conclusion

We have presented a new approach for computing and storing approximate scene visibility based on navigation meshes, providing fast results and good approximations in navigable areas of the environment. Furthermore, we demonstrated the usefulness of our approach by means of different applications, including the fast (re-)computation of viewpoint quality and best sets of views, landmark and room detection, an interface assisting exploration, and a travel technique shown in a user study to result in less cybersickness.

In future work, we will focus on extending the method to efficiently include regions reached by flying, in addition to ground-based travel. Furthermore, we want to explore dynamic scenes, updating visibility information especially in areas often changed through interaction, such as opening or closing doors.

References

[1] J. M. Airey. *Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations*. PhD thesis, UNC Chapel Hill, 1990.

[2] C. Andújar, P. Vázquez, and M. Fairén. Way-Finder: Guided Tours through Complex Walkthrough Models. In *Computer Graphics Forum*, volume 23, pages 499–508, 2004.

[3] P. Barral, G. Dorme, and D. Plemenos. Visual Understanding of a Scene by Automatic Movement of a Camera. In *Int. Conf. 3IA*, 2000.

[4] A. Bönsch, S. Freitag, and T. W. Kuhlen. Automatic Generation of World in Miniatures for Realistic Architectural Immersive Virtual Environments. In *IEEE Virtual Reality Conference (VR)*, pages 155–156, 2016.

[5] U. D. Bordoloi and H.-W. Shen. View Selection for Volume Rendering. In *Proc. IEEE Visualization (VIS 05)*, pages 487–494, 2005.

[6] D. Bowman, E. Kruijff, J. LaViola Jr, and I. Poupyrev. *3D User Interfaces: Theory and Practice*. Addison-Wesley, 2004.

[7] R. Bramon, M. Ruiz, A. Bardera, I. Boada, M. Feixas, and M. Sbert. An Information-Theoretic Observation Channel for Volume Visualization. In *Computer Graphics Forum*, volume 32, pages 411–420, 2013.

[8] G. Bruder, F. Steinicke, and K. H. Hinrichs. Arch-Explore: A Natural User Interface for Immersive Architectural Walkthroughs. In *IEEE Symposium on 3D User Interfaces (3DUI)*, pages 75–82, 2009.

[9] D. Cohen-Or, Y. L. Chrysanthou, C. T. Silva, and F. Durand. A Survey of Visibility for Walkthrough Applications. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):412–431, 2003.

[10] R. Cole and M. Sharir. Visibility Problems for Polyhedral Terrains. *Journal of Symbolic Computation*, 7(1):11–30, 1989.

[11] N. Elmqvist, M. E. Tudoreanu, and P. Tsigas. Tour Generation for Exploration of 3D Virtual Environments. In *Proc. ACM Symposium on Virtual Reality Software and Technology*, pages 207–210, 2007.

[12] M. Feixas, M. Sbert, and F. González. A Unified Information-Theoretic Framework for Viewpoint Selection and Mesh Saliency. *ACM Transactions on Applied Perception*, 6(1), 2009.

[13] A. S. Fernandes and S. K. Feiner. Combating VR Sickness through Subtle Dynamic Field-of-View Modification. In *IEEE Symposium on 3D User Interfaces (3DUI)*, pages 201–210, 2016.

[14] L. Floriani and P. Magillo. Algorithms for Visibility Computation on Terrains: A Survey. *Environment and Planning B: Planning and Design*, 30(5):709–728, 2003.

[15] S. Freitag, D. Rausch, and T. Kuhlen. Reorientation in Virtual Environments using Interactive Portals. In *IEEE Symposium on 3D User Interfaces (3DUI)*, pages 119–122, 2014.

[16] S. Freitag, B. Weyers, A. Bönsch, and T. W. Kuhlen. Comparison and Evaluation of Viewpoint Quality Estimation Algorithms for Immersive Virtual Environments. In *ICAT-EGVE 2015*, pages 53–60, 2015.

[17] S. Freitag, B. Weyers, and T. W. Kuhlen. Automatic Speed Adjustment for Travel through Immersive Virtual Environments based on Viewpoint Quality. In *IEEE Symposium on 3D User Interfaces (3DUI)*, pages 67–70, 2016.

[18] J. Heinrich, J. Vuong, C. Hammang, A. Wu, M. Rittenbruch, J. Hogan, M. Brereton, and S. ODonoghue. Evaluating Viewpoint Entropy for Ribbon Representation of Protein Structure. In *Proc. 37th Annual Conference of the European Association for Computer Graphics*, 2016.

[19] B. Jaubert, K. Tamine, and D. Plemenos. Techniques for Off-line Scene Exploration Using a Virtual Camera. In *Int. Conf. 3IA*, volume 6, 2006.

[20] R. Kennedy, N. Lane, K. Berbaum, and M. Lilienthal. Simulator Sickness Questionnaire: An Enhanced Method for Quantifying Simulator Sickness. *The Int. Journal of Aviation Psychology*, 3(3):203–220, 1993.

[21] J. J. LaViola Jr. A Discussion of Cybersickness in Virtual Environments. *ACM SIGCHI Bulletin*, 32(1):47–56, 2000.

[22] D. Luebke and C. Georges. Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, pages 105–106. ACM, 1995.

[23] J. McCrae, I. Mordatch, M. Glueck, and A. Khan. Multiscale 3D Navigation. In *Proc. of the ACM Symposium on Interactive 3D Graphics and Games*, pages 7–14, 2009.

[24] P. Moreira, L. Reis, and A. De Sousa. Best Multiple-View Selection for the Visualization of Urban Rescue Simulations. *International Journal of Simulation Modelling*, 5(4):167–173, 2006.

[25] C. Niederauer, M. Houston, M. Agrawala, and G. Humphreys. Non-Invasive Interactive Visualization of Dynamic Architectural Environments. In *Proceedings of the 2003 Symposium on Interactive 3D Graphics*, pages 55–58, 2003.

[26] J. O'Rourke. *Art Gallery Theorems and Algorithms*, volume 57. Oxford University Press, 1987.

[27] J. S. Pierce and R. Pausch. Navigation with Place Representations and Visible Landmarks. In *IEEE Virtual Reality Conference (VR)*, pages 173–288, 2004.

[28] D. Plemenos, M. Sbert, and M. Feixas. On Viewpoint Complexity of 3D Scenes. In *International Conference GraphiCon*, pages 24–31. GraphiCon, 2004.

[29] U. Ramer. An Iterative Procedure for the Polygonal Approximation of Plane Curves. *Computer Graphics and Image Processing*, 1(3):244–256, 1972.

[30] M. Sbert, D. Plemenos, M. Feixas, and F. González. Viewpoint Quality: Measures and Applications. In *Eurographics Workshop on Computational Aesthetics in Graphics, Visualization and Imaging*, pages 185–192, 2005.

[31] G. Snook. Simplified 3d movement and pathfinding using navigation meshes. *Game Programming Gems*, 1(1):288–304, 2000.

[32] D. Sokolov and D. Plemenos. Viewpoint Quality and Scene Understanding. In *Int. Conf. on Virtual Reality, Archaeology and Intelligent Cultural Heritage*, pages 67–73, 2005.

[33] S. Takahashi, I. Fujishiro, Y. Takeshima, and T. Nishita. A Feature-Driven Approach to Locating Optimal Viewpoints for Volume Visualization. In *Proc. IEEE Visualization (VIS 05)*, pages 495–502, 2005.

[34] H. Taunay, V. Rodrigues, R. Braga, P. Elias, L. Reis, and A. Raposo. A Spatial Partitioning Heuristic for Automatic Adjustment of the 3D Navigation Speed in Multiscale Virtual Environments. In *IEEE Symposium on 3D User Interfaces (3DUI)*, pages 51–58, 2015.

[35] S. J. Teller and C. H. Séquin. Visibility Preprocessing for Interactive Walkthroughs. In *ACM SIGGRAPH Computer Graphics*, volume 25, pages 61–70, 1991.

[36] S. J. Teller and C. H. Séquin. Visibility Computations in Polyhedral Three-Dimensional Environments. Technical Report UCB/CSD-92-680, EECS Department, University of California, Berkeley, 1992.

[37] D. Trindade and A. Raposo. Improving 3D Navigation in Multiscale Environments using Cubemap-Based Techniques. In *Proc. of the ACM Symposium on Applied Computing*, pages 1215–1221, 2011.

[38] R. Trueba, C. Andujar, and F. Argelaguet. Complexity and Occlusion Management for the World-in-Miniature Metaphor. In *International Symposium on Smart Graphics*, pages 155–166. Springer, 2009.

[39] P.-P. Vázquez, M. Feixas, M. Sbert, and W. Heidrich. Viewpoint Selection using Viewpoint Entropy. In *VMV*, volume 1, pages 273–280, 2001.

[40] P.-P. Vázquez, M. Feixas, M. Sbert, and W. Heidrich. Automatic View Selection Using Viewpoint Entropy and its Application to Image-Based Modelling. In *Computer Graphics Forum*, volume 22, pages 689–700, 2003.

[41] P.-P. Vázquez, M. Feixas, M. Sbert, and A. Llobet. Viewpoint Entropy: A New Tool for Obtaining Good Views of Molecules. In *Proc. Symposium on Data Visualisation*, volume 22, pages 183–188, 2002.

[42] C. Ware and D. Fleet. Context Sensitive Flying Interface. In *Proc. of the 1997 Symposium on Interactive 3D Graphics*, pages 127–ff., 1997.