Toolset for Run-Time Dataset Collection of Deep-Scene Information

Gustav Aaro, Daniel Roos, and Niklas Carlsson[(✉)]

Linköping University, Linköping, Sweden
`niklas.carlsson@liu.se`

Abstract. Virtual reality (VR) provides many exciting new application opportunities, but also present new challenges. In contrast to 360° videos that only allow a user to select its viewing direction, in fully immersive VR, users can also move around and interact with objects in the virtual world. To most effectively deliver such services it is therefore important to understand how users move around in relation to such objects. In this paper, we present a methodology and software tool for generating run-time datasets capturing a user's interactions with such 3D environments, evaluate and compare different object identification methods that we implement within the tool, and use datasets collected with the tool to demonstrate example uses. The tool was developed in Unity, easily integrates with existing Unity applications through the use of periodic calls that extracts information about the environment using different ray-casting methods. The software tool and example datasets are made available with this paper.

Keywords: Deep-scene data collection · Virtual reality · Unity · Light-weight ray-casting

1 Introduction

Virtual reality (VR) provides a safe way to explore environments that otherwise may not be easily accessible or safe for users. For example, VR can be used to train and prepare people for rare but mission critical situations, explore disaster areas, or simply visit places that otherwise would not be accessible to the user. With current technological advancements, the potential applications of VR appear limited only by our imagination. However, to most effectively deliver such services (e.g., over resource constrained networks and/or by offloading computing to edge-cloud servers) and to make the most appropriate system optimizations, it is important to understand how users interact with these environments.

In this paper, we present a methodology and software tool for generating run-time datasets capturing a user's interactions with such 3D environments. In particular, the tool collects time-series information about the *user's movements* (head position and rotation in the 3D environment) together with *deep-scene information* about the objects visible to the user, including each objects' identifier, the user's distance to the object, the angle offset between the center of the user's field

of view to the closest recorded point on the object, the volume of the object, and a ray-based metric that estimates how visible the object is to the user.

The tool was developed in Unity, easily integrates with existing Unity applications through the use of periodic calls that extracts information about the environment. To collect deep scene information in a game engine, we implement different *ray-casting methods*. First, to provide a close to constant overhead while prioritizing objects close to the center of the user's field of view, we implement an algorithm that uses a Gaussian distribution to carefully distribute the rays over the user's field of view. We then implement lightweight refinement methods to further improve the recall rates of this baseline (at a limited additional overhead) and provide an extension that allows rays to be cast also in other directions. The methods are evaluated against a gold standard that uses a dense grid to capture all objects in static example scenes.

While the tool is developed for the VR context, we note that the data-driven approach enabled by the tool (and VR) potentially also can be used for both qualitative and quantitative studies aiming to better understand how people explore and interact within new environments in general.

To demonstrate how the tool can be used to help understand the user's interactions in the environment, we use the tool to collect a few use-case-driven datasets and use relatively simple methods to illustrate example uses of these datasets[1]. While other papers have produced head-movement datasets for 360° videos [3,4,7–9,12,13], these works typically only focus on the viewing direction within a video, do not consider users movement within 3D environments, and do not collect deep-scene information about the objects visible to the user. This paper addresses this void and presents a novel toolset to capture how users explore these exciting new environments.

Outline: Section 2 presents our general framework, Sect. 3 presents the ray-casting methods used, and Sect. 4 evaluates these methods. We then use datasets to demonstrate example uses (Sect. 5). Finally, we present related works (Sect. 6) and our conclusions (Sect. 7).
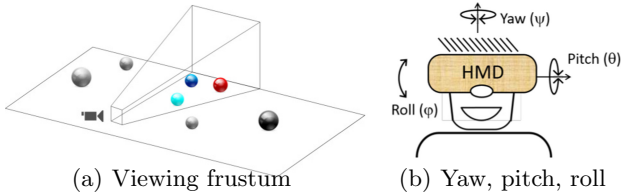
2 Methodology Framework

2.1 Environment and Tools

We use the Oculus Rift[2] Head Mounted Display (HMD), consumer version 1, together with the Unity[3] 3D game engine that renders the scenes to the user. Oculus Rift provides a rich open-source SDK, including a specific SDK for Unity. For the experiments and collection of the datasets, we used a Windows 10 workstation (Intel Xeon CPU E5-1620 V4 3.50 GHz) with NVIDIA GeForce GTX 1080 graphics card with a dedicated HDMI video output, and 32.0 GB RAM.

---

[1] Scripts and example datasets are made available here: https://www.ida.liu.se/nikca89/papers/deep-scene-2020.html.

[2] https://www.oculus.com.

[3] https://unity3d.com/.

(a) Viewing frustum   (b) Yaw, pitch, roll

Fig. 1. Viewing frustum and direction (relative headset).

2.2 High-Level Framework

Our methodology provides light-weight data collection of time-series traces that for each time instance captures (i) the user's head position, (ii) the user's viewing direction, and (iii) information about the visible objects within the user's field of view. For this purpose, we place a Unity *player camera*[4] at the position of the user's head location and point it in the user's viewing direction. From this location we then use *ray-casting*[5] to determine objects that are visible to the user (and count how many rays hit each visible object). As the name implies, ray-casting is a method used in the Unity game engine that casts a ray in a predetermined direction from a point of origin, and if an object is hit by the ray it returns detailed information about the object that was hit. While this is a very useful method to discover object, it comes at a high computational cost. It is therefore important to limit the number of rays used. In this paper we implement and test different methods to determine the directions to cast rays. Of particular interest are the objects within the *view frustum*, defined as the region of the 3D world that appears on the screen for the user (Fig. 1(a)). For the identified objects, we then collect deep-scene information and record them into a trace file.

Sampling Frequency: While the Oculus headset updates its data at a frequency of 1,000 Hz and has a refresh-rate 90 Hz, through our experiments, we have determined that a reasonable sweet spot to collect sample data about in-engine generated content (such as objects visible to the user) is 50 Hz, which we use for our default settings here. Due to high computational complexity, higher sample frequency can easily result in performance problems (which slow down rendering and can result in user sickness [10]) and lower sample frequency may not be sufficient to track all the objects that a client interacts with.

To trigger all physics-related operations needed for the data collection we use the hook-in method `FixedUpdate`[6] associated with Unity's C#-scripting interface. In particular, we set the *fixed timestep* variable in Unity's `TimeManager` to 0.02 s; providing us 50 Hz "heartbeat".

---

[4] https://docs.unity3d.com/ScriptReference/Camera.html.
[5] https://docs.unity3d.com/ScriptReference/Physics.Raycast.html.
[6] https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html.

Positional and Rotational Information: We import the *Oculus Platform SDK* assets into Unity and use an `OVRPlayerController` object to gather information about the headset's position (in world coordinates provided by Unity) and rotation (in *yaw*, *pitch*, and *roll*). Figure 1(b) illustrates these rotations.

2.3   Dataset Structure

Our application saves all sample points in *JSON*-format [5]. Each such JSON entry contains the following:

- The `timestamp` of the sample point.
- Player `position`, using Unity's world coordinates.
- Player `rotation`, using yaw, pitch and roll.
- The collection speed in frames-per-second (`fps`) possible at that specific sample point.
- An array (`visibleObjects`) of visible objects, where each entry contains detailed object information, including (i) a `name` descriptor for the object, (ii) a unique object identifier `id`, (iii) the `distance` between the center point of the player and the object, (iv) the smallest `angle` between the closest observed point of the object and the player, (v) the number of rays that hit the object during the sample point, and (vi) the `volume` of the object's bounding box, as given by the `size` property of the renderer's bounds[7].
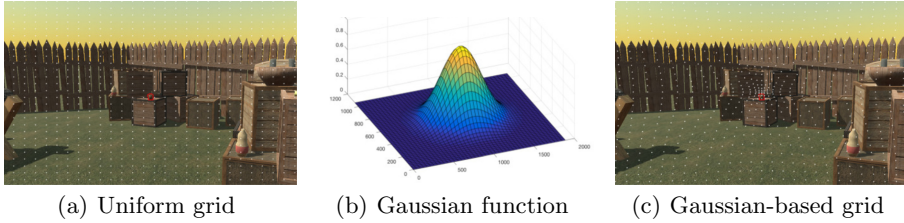
3   Object Detection Methods

Determining all objects visible to a user is highly computationally expensive. We therefore design and use relatively light-weight sampling methods that may miss some objects within the user's field of view, but that allows us to sample at a much higher frequency. In this section we describe some key tradeoffs considered (Sect. 3.1), a constant-ray baseline heuristic that prioritizes finding objects closer to the center of the users' viewing direction (Sect. 3.2), and a few refinement methods (Sect. 3.3) that help improve the results.

3.1   Per-Object Vs Constant-Ray Approaches

Two fundamental approaches to identify visible objects are (i) to go through the list of all objects and check whether they appear visible, and (ii) to cast rays within the viewer's viewing frustum and record the objects hit by such rays.

Basic Implementations: Our simplest implementation of the first approach (called *per-object ray-casting*) first performs frustum culling, followed by ray-casting in the center direction of each of the remaining objects. If an object is hit by at least one such a ray, the object is included in the list of visible objects; otherwise it is assumed hidden. We have also considered extensions that cast

---

[7] https://docs.unity3d.com/ScriptReference/Bounds.html.

| (a) Uniform grid | (b) Gaussian function | (c) Gaussian-based grid |

Fig. 2. Per-view ray grid examples using 1700 rays.

additional rays towards different locations within the object's bounding box. In contrast, our implementation of the second approach (called *per-view ray grid*) uses the `ScreenPointToRay` method built into Unity to map rays within the user's view field according to some distribution. Through experiments, we have found that the second approach typically performs and scales better.

Scaling Problem with Per-Object Rays: Clearly, even a small object can obstruct a ray cast towards a much bigger object, and there may be many objects hidden behind a big object. With the first approach, an important tradeoff therefore arises when determining how many rays should be cast before considering an object hidden. However, in scenes with many objects, the time casting just one ray per object can be substantial and have typically been all that the system can handle 50 Hz for the basic scenes considered here.

In general, this method is not scalable to scenes with many objects (many of which may be hidden). Clearly, in the ideal case, the sampling method should not depend on the number objects in the scene. Therefore, in the following, we focus on the second baseline approach (that uses a constant number of rays, regardless of the number of objects) and optional refinement methods that limit the number of additional objects to consider based on the findings from this baseline approach.

3.2   Baseline Approach: Per-View Ray Grid

To ensure that the number of rays that are cast is independent of the number of objects in the scene, we distribute rays across the user's current view. Figure 2(a) presents a simple example in which we have used a uniform grid of 1,700 rays.

While the uniform distribution is good at detecting objects within the field of view, it does not account for users being more likely to see (and interact with) objects closer to their viewing center. To increase the likelihood that objects closer to the center of the field of view (e.g., the mug in the center of Fig. 2(a)) are not missed, while keeping the same budget of rays fixed, we therefore selected to use a two-dimensional Gaussian distribution function to adjust the grid points. In particular, we use the general expression of a multivariate Gaussian distribution function:

$$f(x,y) = A \cdot e^{-\left(\frac{(x - x_0)^2}{2 \cdot \sigma_x^2} + \frac{(y - y_0)^2}{2 \cdot \sigma_y^2}\right)}, \tag{1}$$

where A is the amplitude of the function, (x_0, y_0) are the median coordinates, and σ_x and σ_y represent the deviations in the x and y dimensions, respectively. First, we empirically searched for good parameters choices, resulting in the following parameter choices: $A = 0.8$, $x_0 = W/2$, $y_0 = H/2$, $\sigma_x = W/8$, and $\sigma_y = H/8$, where W and H represent the width and height of the screen. Second, we used this perturbation function to move all points in the grid towards the center of the screen by a distance proportional to the Gaussian value (Fig. 2(b)) of each coordinate. Figure 2(c) shows the result using the same scene and number of rays as before. We note that the mug that earlier was missed is now hit by multiple rays, illustrating the higher weight given to trying to find objects close to the user's viewing center. Of course, this comes at the cost of potentially missing some objects in the periphery of the viewing field.

3.3   Ray-Casting Refinement Methods

While we have found the *per-view ray grid* method desirable over the *per-object ray* method, in some instances, it clearly misses some objects that a per-object method would find. To reduce the chance of (mis)labeling visible objects as hidden, we developed and tested different enhancement algorithms. Here, we describe two light-weight algorithms and a more computationally expensive algorithm to select a set of objects to perform additional *per-object* ray castings on.

All three methods are designed to enhance the results of the *per-view ray grid* method and the combination can be seen as a hybrid approach. In particular, they all identify a small subset of additional objects, which has not yet been found (using the *per-view ray grid* method), and then (similar to the *per-object ray* method) check the objects in this subset one by one, to see if their center points are visible to the user. We also briefly discuss how the low-cost algorithms can be modified to require a constant number of ray castings.

Delaunay Surface (DS): This method leverages the 3D point cloud defined by the coordinates where each of the original rays cast by the *per-view ray grid* method first hit an object to delimit the set of candidate objects to consider. In particular, the method only considers objects that (i) have not yet been deemed visible, (ii) are within the viewing frustum, and (iii) have a center point closer to the camera than the Delaunay surface defined by these points. The Delaunay 3D surface (exemplified in Fig. 3(a)) is made up by triangles from a Delaunay triangulation with the points in the point cloud making up the vertices on the surface. A Delaunay triangulation maximizes the minimum angle in each triangle and allows us to loop trough all objects meeting the first two criteria above to check whether the intersection between the line passing through the camera and the object's center point would intersect this surface before or after passing through the objects center point. To do so, we first determine the intersection triangle, and then calculate the point where the projection line intersects the plane defined by this triangle. If this point is further away from the user then the candidate object's center point (i.e. the object is between the 3D surface and the user), the object is included in the candidate set of objects to check

<table>
<tr><td>(a) 3D Delaunay surface</td><td>(b) RoI using DT, $P = 0.5$</td></tr>
</table>
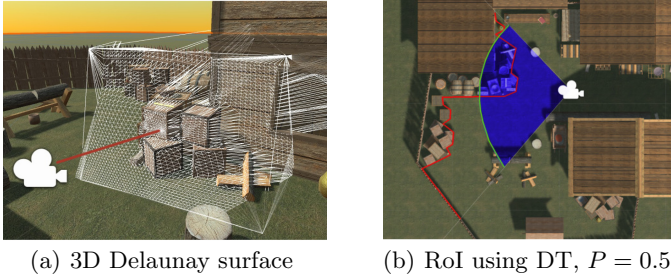
Fig. 3. Example boundaries for region of interest (RoI). (Color figure online)

further. While this method nicely can prune the search space of objects, it is computationally expensive.

Percentile-Based Distance Threshold (DT): To reduce the complexity and allow users to tune the number of rays that need to be cast, we implemented a *percentile-based distance threshold* (DT) method. First, all distances between the camera and the points in the point cloud are sorted. Second, a distance threshold D_P is determined that corresponds to a specific percentile P of these distances. Third, we only include the objects that (i) have not yet been deemed visible, (ii) are within the viewing frustum, and (iii) have a distance $d < D_P$ to its center point. Figure 3(b) shows the point cloud (red line) and region of interest (blue area) when using $P = 0.5$, for a simple 2D example. It is important to note that in comparison to ray-casting, determining which objects fall within the viewing frustum is inexpensive.

Finally, we note that DT easily can be modified to return no more than a constant number of objects. For example, simply sort the distances to the center points of all objects that satisfy the first two (or three) criteria and select the K closest such objects. In this paper, we focus on the hybrid version of DT, described more carefully above.

Angular Threshold (AT): This method is similar to DT, but uses a threshold α on the angle offset θ_i of object i to determine which objects to consider further. In particular, the method selects the set of objects that (i) have not yet been deemed visible, (ii) are within the viewing frustum, and (iii) have an angular offset $\theta_i < \alpha$. We note that many hidden objects may be included by this method, and that a maximum distance (as used with DT, for example) therefore ideally also should be used. As with DT, this method can also easily be modified to bound the number of objects to (at most) K, using the offset angle θ_i to rank candidates.

4   Methodology Evaluation

While our system is designed to capture the objects within the user's view also in dynamic environments with lots of movement, for our ground-truth-based
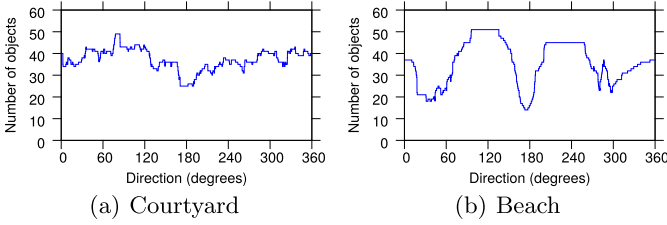
Fig. 4. Number of visible objects in each direction using the gold standard approach.
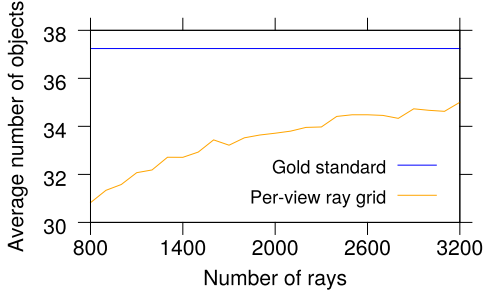
Fig. 5. Average number of objects detected as a function of the number of rays used by *per-view ray grid*. (Courtyard)

validation we used two scenes supplied by a third party[8]: *Courtyard* and *Beach*. The Courtyard scene (e.g., Fig. 3(b)) is densely populated with objects close to the user, whereas the objects in the Beach scene are significantly further away and more clustered. For each scene, we placed the player in the center of the scene, spun the player 360° at a constant speed, and recorded the objects identified under different parameter settings.

Gold Standard: To generate a type of ground-truth, called the *gold standard* here, we slowly moved the camera around its z-axis while casting vertically spaced rays. By slow rotation of these even spaced rays we created a 9,000 × 2,000 grid of rays, spanning the full 360° view, giving us higher than per-pixel granularity of the objects in the scene. Through post-processing we then recreated exactly which objects are visible within the field of view of each viewing direction. Figure 4 shows the number of objects possible to observe using this approach for each viewing direction.

4.1 Impact of Grid Density

Figure 5 shows the average number of objects detected over all (sample) directions as a function of the number of rays used with the *per-view ray grid*

---

[8] Mega Fantasy Props Pack: https://assetstore.unity.com/packages/3d/environments/fantasy/mega-fantasy-props-pack-87811.
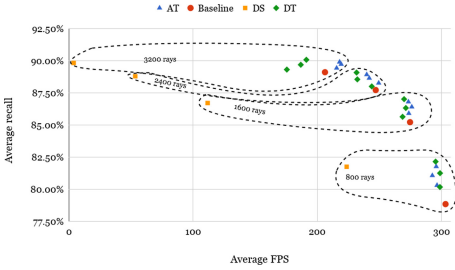
Fig. 6. FPS and recall scores for each method, with points grouped based on the number of rays by the original baseline.
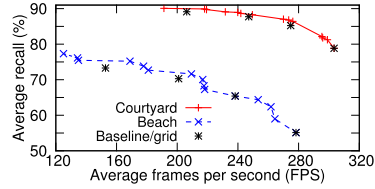
Fig. 7. Recall-FPS tradeoff frontiers.

approach. As a comparison, we also include the average number of visible objects as determined by the *gold standard*. We note that the *per-view ray grid* baseline performs well and achieve within 7% of the gold standard when using a grid with 3,200 rays. However, we also observe diminishing recall returns, motivating the use of refinement methods such as those considered here.

## 4.2   Refinement Method Comparison

To compare the different refinement algorithms, we started with using the *per-view ray grid* method with grids of 800, 1600, 2400 and 3200 rays. For DT we used percentile thresholds of $P = 0.5$, 0.7, and 0.9, and for AT we used the angular thresholds: $\alpha = 20°$, $30°$, and $40°$. Figure 6 shows a scatter plot of the average recall rate versus the average frames per seconds (FPS) across the samples of each of these methods and settings. Here, the recall rate (of each sample) is equal to the fraction of objects identified by the gold standard that the method tagged correctly, and we group the data points based on the number of rays used in the original grid.

The baseline and refinement methods did not discover any object that the gold standard did not discover. The precision is therefore always 100% and the F1-score is always $2r/(1 + r)$, where r is the recall. Of this reason, throughout, we only use recall (not precision or F1 scores).

Significant variations in the FPS metric caused some anomalies. For example, there are instances where the refinement methods have higher FPS than the corresponding baseline alone. These cases can be explained by unrelated background processes (running on the same Windows-PC). Some caution should hence be taken when interpreting the FPS values. However, in general we clearly see that additional rays tends to increase the recall and decrease the FPS. Therefore, it is perhaps not surprising that DS typically have the lowest FPS and among the highest recall. The simplicity of both DT and AT provide more attractive tradeoffs, with DT typically giving slightly better recall scores. This is most apparent with increased ray counts. Finally, we note that the gap in recall between the best methods in the 800 ray cluster and the baseline in the 1,600 ray cluster is
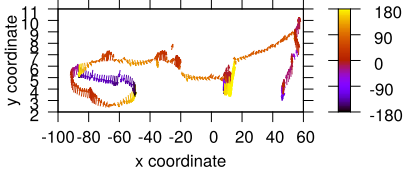
Fig. 8. Six-minute example path.
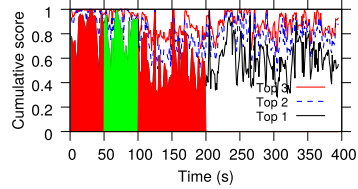
Fig. 9. Example scores of the top-3 objects over time.

due to many of the missed objects having hidden centers, showing that additional rays (e.g., with a denser grid or with more per-object rays) are needed to find these objects. Overall, we have found that the hybrid approaches analyzed here provides a nice tradeoff, pushing up the recall at a very limit overhead.
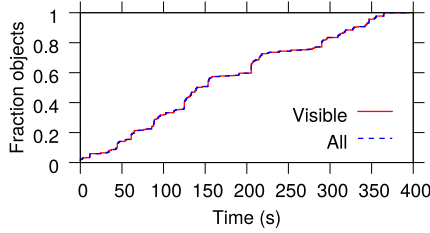
In general, the beach scene is much more challenging than the courtyard scene. Figure 7 plots the recall-FPS tradeoff frontiers (using the analogy of a Pareto frontier) for the two scenes. We note a substantial shift of the tradeoff curve, with the maximum recall differing by 14%. To put the frontiers in perspective, we also include markers for the four baseline experiments, again demonstrating the improvements provided by the lightweight refinement methods.
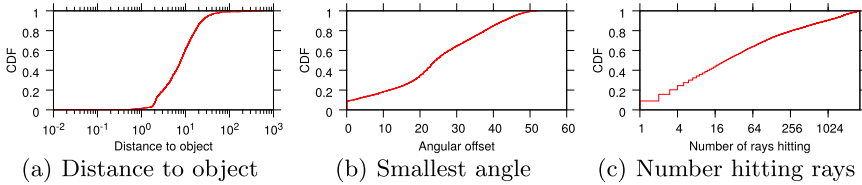
5   Example Use Case

The tool has many use cases and can be used for both qualitative and quantitative studies. For example, consider a crime scene analysis. Using a 3D map of a city, for example, an investigator can use test users to better understand what witnesses or victims are more or less likely to have seen. For example, a test user may be asked to walk the path of an actual witness or victim (potentially recreated using GPS, base station IDs, and/or other network operator related data). Qualitatively, example paths such as the one in Fig. 8 (in which we walked through a Viking village[9] looking for swords) of multiple test users can corroborate witness statements.

To aid in better understanding what objects a person is more likely to have seen, per-object ray counts and other scoring criteria can be used. Figure 9 shows the cumulative score associated with the top-three objects on a second-per-second basis. Here, we use a basic scoring function $\sum_{t \in T}(w_d \frac{1}{d_t} + w_\alpha \frac{1}{\alpha_t} + w_n n_t)$, where T is a time interval of interest, w_d, w_α, w_n are weights, d_t is the distance to the object at time t, α_t is the smallest angle to the object at time t, and n_t is the number of rays that hit the object at time t. For our example results, we set the weights so that the three factors are roughly equally weighted. First, without loss of generality, we chose $w_d = 1$. Second, we express the other weights relative to

---

[9] https://assetstore.unity.com/packages/essentials/tutorial-projects/viking-village-29140.

Fig. 10. Objects observed over time.

(a) Distance to object   (b) Smallest angle   (c) Number hitting rays

Fig. 11. Example CDFs.

this: $w_\alpha = \sum_{t,i} \frac{1}{d_t} / \sum_{t,i} \frac{1}{\alpha_t}$ and $w_n = \sum_{t,i} \frac{1}{d_t} / \sum_{t,i} n_t$, where the sums are over all observed samples (i.e., both over time t and visible objects i). Furthermore, for the first 200 s we use colors to indicate when the identity of the highest scored object changes.

Finally, note that various distribution statistics may be useful to better understand the rate that new objects appear (Fig. 10), the objects relative locations (Figs. 11(a) and 11(b)), or the number of rays that hits the objects (Fig. 11(c)). Comparing these statistics with the corresponding values (and scores) for specific objects may provide insight into the relative presence certain objects have. Of course, this does not account for color differences, contrast, and other aspects that may cause certain objects to stick out to an observer. Furthermore, the tool we use does not capture eye gaze [2], which could further improve the use of this type of tool.

6   Related Work

While there is a significant amount of use-case-driven research leveraging VR, there is a lack of generic tools to collect deep-scene information. In early work (2004), Chittaro and Ieronutti [6] present a tool for analyzing users' interactions with virtual environments through tracking of their position and orientation. However, they focus on the visualization aspects and do not collect any deep-scene information about the objects. Others have collected avatar traces and similar for virtual worlds [15] and games [11]. Since the official Oculus Rift release in March 2016, there have been many dataset papers and research studies using head movement [3,4,7–9,12,13] or eye gaze [2] data for 360° video. However, these works typically only focus on the viewing direction, as these videos do not

(a) View from above   (b) Side view

Fig. 12. Third-person example of 360 peripheral: forward-facing rays (magenta), backward-facing rays (green). (Color figure online)

allow users to move around within the environment. In contrast, in our context, users can move around and their interactions can change the scene itself (e.g., as both objects and users move). None of the prior works collect deep-scene information about the objects visible to the user.

Finally, we note that the data link between the HMD and the computer has extremely high data rate requirement and delay constraints. This complicates creating an untethered experience [1] and how to best deliver remote service [14]. Such important efforts to provide improved VR experiences are orthogonal to the work presented here.

7 Conclusions

This paper presents a methodology and software tool for generating run-time datasets capturing user movements and visible objects in immersive 3D environments. Collecting deep-scene information in a game engine such as Unity is non-trivial and requires careful ray casting. Our tool implements baseline methods to carefully distribute the rays over the user's field of view and refinement methods that improves the recall rates at limited additional processing overhead. The methods are evaluated against a gold standard.

Our tool collects information about the users' movements (position, rotation) at a tunable time granularity together with information about the objects visible to the user, including each object's identifier, distance, angle offset, volume, and how many rays hit the object at each time instance. While our validation is done with two static environments, our tool also works for dynamic environments with moving objects.

Finally, we use example datasets and relatively simple methods to illustrate example use cases. Interesting future work include user studies (using the tool) to be better understand users' interactions with selected example objects, but also the development of methods (based on extensions of the tool) for improving user experiences and system performance. For example, while the algorithms used here focus on objects within the viewing frustum, our tool also includes an extension that collects information about peripheral objects. Figure 12 illustrates the rays cast when operating in this extended mode. Future work will evaluate the use of such extensions for predicting objects that are likely to appear for the user.

References

1. Abari, O.: Enabling high-quality untethered virtual reality. In: Proceedings ACM Workshop on Millimeter-Wave Networks and Sensing Systems (2017)
2. Agtzidis, I., Startsev, M., Dorr, M.: 360° video gaze behaviour: a ground-truth data set and a classification algorithm for eye movements. In: Proceedings ACM Multimedia (2019)
3. Almquist, M., Almquist, V., Krishnamoorthi, V., Carlsson, N., Eager, D.: The prefetch aggressiveness tradeoff in 360° video streaming. In: Proceedings ACM MMSys (2018)
4. Bao, Y., Wu, H., Zhang, T., Ramli, A., Liu, X.: Shooting a moving target: motion-prediction-based transmission for 360° videos. In: Proceedings IEEE Big Data (2016)
5. Bray, T.: The JavaScript object notation (JSON) data interchange format. RFC 8259, Internet standard, IETF (2017)
6. Chittaro, L., Ieronutti, L.: A visual tool for tracing users' behavior in virtual environments. In: Proceedings AVI (2004)
7. Corbillon, X., Simone, F.D., Simon, G.: 360° video head movement dataset. In: Proceedings ACM MMSys (2017)
8. David, E.J., Gutiérrez, J., Coutrot, A., Da Silva, M.P., Callet, P.L.: A dataset of head and eye movements for 360° videos. In: Proceedings ACM MMSys (2018)
9. Fremerey, S., Singla, A., Meseberg, K., Raake, A.: AVtrack360: an open dataset and software recording people's head rotations watching 360° videos on an HMD. In: Proceedings ACM MMSys (2018)
10. LaViola Jr, J.J.: A discussion of cyber sickness in virtual environments. ACM SIGCHI Bull. (2000)
11. Lee, Y.T., Chen, K.T., Cheng, Y.M., Lei, C.L.: World of Warcraft avatar history dataset. In: Proceedings ACM MMSys (2011)
12. Lo, W., Fan, C., Lee, J., Huang, C., Chen, K., Hsu, C.: 360° video viewing dataset in head-mounted virtual reality. In: Proceedings ACM MMSys (2017)
13. Qian, F., Han, B., Xiao, Q., Gopalakrishnan, V.: Flare: Practical viewport-adaptive 360° video streaming for mobile devices. In: Proceedings ACM MobiCom (2018)
14. Tan, Z., Li, Y., Li, Q., Zhang, Z., Li, Z., Lu, S.: Supporting mobile VR in LTE networks: How close are we? In: Proceedings of ACM SIGMETRICS (2018)
15. Varvello, M., Ferrari, S., Biersack, E., Diot, C.: Exploring second life. IEEE/ACM Trans. Networking (2010)