PBVR: Physically Based Rendering in Virtual Reality

Yavuz Selim Tozlu Huiyang Zhou

North Carolina State University

Raleigh, NC, USA

{ystozlu, hzhou}@ncsu.edu

Abstract

Virtual Reality (VR) is a rapidly growing domain that requires high-fidelity graphics for immersion. To understand and improve the VR architecture, an open-source, end-to-end platform for VR research was recently proposed. However, studying the stereo rendering aspect of VR applications remains challenging due to the lack of infrastructure.

In this work, we augment the aforementioned open-source platform, ILLIXR, by integrating a high-end, physically based 3D rendering engine, Filament. This upgrade, named PBVR, enables developers to render high-quality graphics in a completely open-source VR platform.

As a case study, we leverage our proposed PBVR to look into gaze-tracked foveated rendering and profile three different scenes. We show that a handful of renderpasses consume the most time and that readily available foveated rendering solutions, such as Variable Rate Shading, might not provide significant advantages. Moreover, our results reveal that eye tracking can incur a significant overhead on the graphics processing unit (GPU).

1. Introduction

Virtual Reality (VR) is deemed as the next generation of entertainment platforms and has grown into a multi-billion dollar industry [13] [14]. Establishing immersive and high-quality VR entails a unique set of challenges. One of the biggest challenges is low-latency, high resolution, high frame rate stereo rendering. State-of-the-art VR headsets feature resolutions that are upwards of 1440x1600 per eye, with refresh rates ranging from 90Hz to 120Hz [5] [9] [12]. This problem is further exacerbated on mobile platforms due the power constraints.

Although VR imposes significant research challenges, there is limited research on VR system architecture from academia, and the primary reason is the lack of infrastructure. Recently, an initiative has been formed to enable and democratize VR research [3]. As part of this effort, an open-source VR testbed, named ILLIXR, has been built [21]. ILLIXR enables end-to-end analysis and study of VR systems by integrating the fundamental components of a VR system, including visual-inertial odometry, pose prediction, asynchronous reprojection, eye tracking, scene reconstruction, and more. Researchers can now investigate and experiment with these components and evaluate the overall performance of the system.

Nevertheless, stereo rendering, which is one of the biggest bottlenecks in the VR pipeline [21], remains difficult to analyze and improve with the existing ILLIXR infrastructure. This is mainly because ILLIXR provides the components of a VR system, but it is up to the researchers to develop applications and benchmarks that will be run on ILLIXR. In other words, researchers have to spend ample time developing 3D applications that can interface with ILLIXR's components. To address this problem, ILLIXR enables support for OpenXR [34] applications through Monado [6]. In practice, a game engine such as Godot [2] or Unity [11] is used to develop OpenXR applications, which can be run on ILLIXR. The catch, however, is that game engines are extremely complex tools, making them impractical to modify and extend for research purposes.

To support high-performance 3D rendering, in this work, we integrate Filament [1] into ILLIXR. Filament is an open-source, real-time, physically based rendering engine from Google. It exposes a C++ programming interface that can be effortlessly hooked up with ILLIXR. More importantly, the software architecture of Filament is flexible and allows for modifications and quick prototyping. To demonstrate this, we use Filament to render three scenes with varying complexities and breakdown frames for detailed analysis. We also extend the existing eye tracking component in ILLIXR to enable gaze estimation, which we use for foveated rendering in Filament. We employ Nvidia's Variable Rate Shading [8] technology to implement foveated rendering. Implementing foveated rendering in a game engine would require substantially more effort and expertise, whereas, with Filament, it is nearly trivial. Moreover, we avoid the additional complexity of OpenXR, by using ILLIXR's native interface. Our experimental results show that eye tracking consumes substantial GPU memory and computing power, affecting the overall benefits of foveated rendering. Finally, we open-source our augmented tool to foster VR graphics research.

The remainder of the paper is organized as follows. Section II provides the background and motivates our work. Section III elaborates on Filament's capabilities and how we couple Filament with ILLIXR. Section IV demonstrates an example use case of this augmented tool by analyzing

eye-tracked foveated rendering. Section V discusses the related work. Section VI concludes.

2. Background and Motivation

2.1. The Canonical VR Pipeline

Contemporary VR systems consist of several fundamental building blocks, along with a number of optional ones. Typically, the dataflow starts with the headset sensors, which include inertial measurement units, cameras, or laser emitters [4] [27]. The data from these sensors are processed to determine the headset's position and orientation using visual-inertial odometry (VIO) algorithms. The position and orientation data are then passed to the user application's graphics pipeline to render 3D scenes. Once a scene is rendered, it is sent to the display device.

Most VR systems also feature hand controllers, which are tracked in a similar fashion and used by the application to interact with the virtual world.

Some of the most recent VR headsets also feature eye tracking for foveated rendering and avatar expressions [5] [9]. These headsets are equipped with integrated eye cameras that image the user's eyes at a high frequency. These images are typically fed into a neural network to perform eye segmentation [16] [40]. Once the images are segmented, gaze estimation is carried out, which involves ellipse fitting and other algebraic operations [40].

Pose estimation, eye tracking, hand tracking, and other similar tasks are usually executed in parallel, as they are independent. Fig. 1 shows a generic VR system that features eye tracking.

In practice, VIO and stereo rendering may consume too much time and become the primary contributors to motion-to-photon (MTP) latency. Low latency is key to a pleasant VR experience, and high MTP latency is known to cause sickness and fatigue to the user [28] [32]. To mitigate these problems, VR headsets implement pose prediction and asynchronous reprojection [22] [35]. Asynchronous reprojection, also known as timewarp, works hand in hand with pose prediction. Pose prediction uses the pose calculated by the VIO algorithm, and updates it based on the latest sensor data. The updated pose is then used by timewarp to transform the rendered frame to match the latest headset pose. This whole update and transformation process happens just before the display refreshes. Ultimately, this technique reduces the MTP latency of the system, thus improving the user experience [35].

2.2. The ILLIXR VR System

ILLIXR adopts a similar organization as in Fig. 1. It consists of "plugins", i.e. components that run in parallel and communicate with each other through a well-defined interface. Users can easily add new plugins, modify existing ones or remove them completely. By default, ILLIXR provides plugins for pose estimation, pose prediction, timewarp, scene reconstruction, camera and inertial-measurement unit(IMU) sensors, audio and more. A rudimentary plugin for eye tracking is also provided, but it merely runs a neural network for eye image segmentation and does not perform gaze estimation. We discuss how we added gaze estimation in Section IV. Fig. 2 depicts the plugins that we use and how they are connected.

In its essence, ILLIXR is a C++ program that glues together an array of plugins that are implemented as C++ classes. Most plugins run in a dedicated thread, and communicate with other plugins through callbacks and thread-safe queues. The ILLIXR runtime, i.e., the entry point of the program, dynamically loads the plugins and initializes them.Plugins either run in a tight loop in a dedicated thread, or they are invoked by these looping plugins. For example, the offline IMU/Camera plugin runs in a loop that reads stereo camera images and IMU sensor data from disk at a rate of 30Hz, packs them into a structure, and pushes a pointer to that structure into a thread-safe queue. The VIO plugin is executed upon a callback from the IMU/Camera plugin when a new data is pushed to the queue. The plugin then reads this pointer from the queue and accesses the underlying data. The plugins that run in a loop are derived from the threadloop class that ILLIXR infrastructure provides. This base class provides two functions that are overridden by the derived class: *_p_thread_setup*() and *_p_one_iteration*(). As the names imply, the first function is executed just once as part of setup the phase before the thread goes into a loop, where it executes the second function in each iteration. Our custom Filament plugin is also derived from this threadloop class.

Therefore, the only requirement for a plugin to hook up with ILLIXR is to match this producer/consumer interface. This allows ILLIXR to be highly modular and extensible.

Currently, there are two ways to run a VR application on ILLIXR. The first way is to use the native interface of ILLIXR, which involves writing an OpenGL application using a C++ base class that ILLIXR provides, and hooking up this plugin with other plugins via the switchboard system of ILLIXR. The second way is to provide ILLIXR with an OpenXR application binary. In this case, the OpenXR application runs using Monado, an open-source implementation of the OpenXR standard. Monado, then, communicates with ILLIXR and uses its plugins to read headset pose, submit frames, and more. In other words, ILLIXR imitates a VR headset in this case. Both of these approaches have pros and cons. Using ILLIXR's native interface is convenient because, the whole VR pipeline is managed by ILLIXR, and the additional complexity of OpenXR and Monado is avoided. Moreover, in this way, ILLIXR is not limited by what the OpenXR spec does and does not allow. The downside is that users have to rewrite their applications just to interface with ILLIXR. The main advantage of using the OpenXR interface is that existing applications can be run directly on ILLIXR, without any modifications. However, this means the users now have to work with an even more complex system consisting of ILLIXR and Monado. In this work, we use the native interface of ILLIXR to develop our applications.

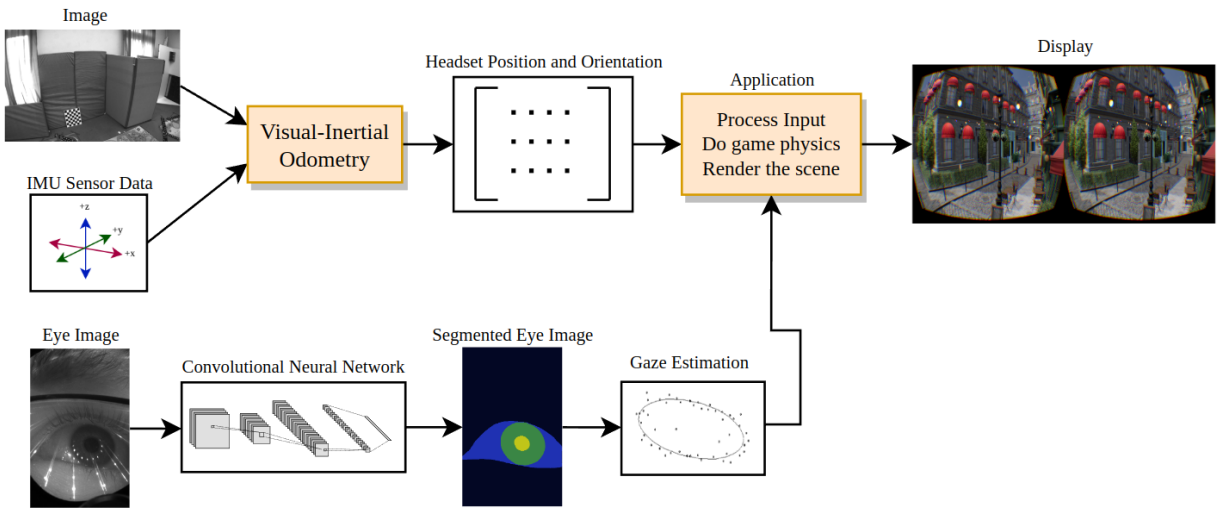

Figure 1: Organization of a generic VR system with eye tracking.

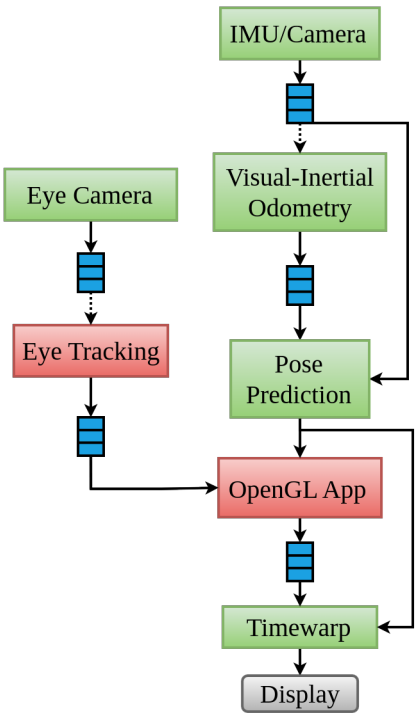

Figure 2: Simplified diagram of ILLIXR plugins used in this work. Red rectangles represent heavily modified plugins. Blue blocks represent thread-safe queues. Producer plugins insert data to concurrent queues, which consumer plugins read. Dotted arrows indicate that the plugin is invoked through a callback when new data is pushed to the queue.

2.3. Physically Based Rendering

With the advent of powerful graphics processing units (GPUs), more advanced and realistic graphics rendering techniques are now possible. Physically Based Rendering(PBR) is one such technique, and the idea is to model the materials of objects as physically accurately as possible. Materials are characterized by parameters such as color, metalness, roughness, reflectance, anisotropy, etc. Collectively, these parameters determine the behavior of light as it interacts with the material, as physically accurate as possible. Therefore, when rendered with PBR, the materials in a 3D scene can look incredibly realistic under any lighting scenario. Fig. 3 showcases different materials rendered in a PBR fashion using the Filament engine.

Figure 3: A range of materials from metals to rock is rendered in the Filament engine. Objects appear highly realistic thanks to accurate modeling of their material properties. Image attributed to Filament developers [1].

Nevertheless, PBR's visual fidelity comes at a price. The physically based calculations of how light interacts with the material are typically carried out during the fragment shading stage of the graphics pipeline. These calculations can involve multiple expensive texture fetches for every material, evaluating mathematical operations such as dot products, trigonometric functions, exponents, roots, etc [17] [31] [36]. Even though these operations are not complex

individually, they are executed in fragment shaders, which are carried out multiple times for every pixel on the screen. Thus, they can introduce a substantial overhead.

Nowadays, most 3D engines implement some form of PBR to achieve the best possible visual fidelity [2] [7] [11]. Consequently, the state-of-the-art VR applications such as games also benefit from this technology. The use of PBR can be considered essential in achieving a high level of realism and creating a captivating virtual world.

2.4. The Need For a Rendering Engine in ILLIXR

While the ILLIXR infrastructure provides most of the necessary components of a VR system, it is up to the user to develop the VR application itself. Currently, ILLIXR only supports OpenGL applications. Therefore, the user has to either write an OpenGL application from scratch and use ILLIXR's native interface, or write an OpenXR application that uses OpenGL as its renderer. One possibility is to employ game engines to develop OpenXR applications. The problem with using game engines is that, they are usually extremely complex pieces of software, and while possible, the codebase is not designed to be modified or extended by the user. This makes it difficult to add new features or modify the rendering pipeline. Developing an OpenGL application from scratch is another option, albeit not practical, especially if high-quality graphics is desired. The reason is that implementing PBR, shadows, and post-processing effects in an optimized manner is no trivial task. Therefore, we resort to using an existing rendering engine, Filament, that has the best of both worlds. Filament is basically an open-source C++ library for rendering high-quality graphics. It supports OpenGL and has all the features needed to render realistic graphics. With Filament, users simply provide the 3D model, light sources, and graphics settings, such as ambient occlusion, reflections, and anti-aliasing, and Filament does the heavy lifting.

3. Augmenting ILLIXR

3.1. Coupling Filament with ILLIXR

Making ILLIXR and Filament work together involves three main challenges as discussed below.

The first challenge is OpenGL context sharing. ILLIXR creates and shares its own OpenGL context across the entire application. This OpenGL context is used by the main OpenGL plugin and the timewarp plugin. On the other hand, Filament also requires an OpenGL context. For the whole system to work, Filament and ILLIXR must share the same OpenGL context. Fortunately, Filament makes it simple to share an OpenGL context with ILLIXR, as it has an option of passing an external OpenGL context during initialization. It then proceeds to initialize its own OpenGL backend using this context. Therefore, we can simply forward the OpenGL context that ILLIXR creates to Filament.

The second challenge is stereo frame sharing between Filament and timewarp. In ILLIXR, the main OpenGL application renders the stereo frames into a pair of OpenGL textures. Once the rendering is finished, the handles to these textures are then broadcast through ILLIXR's interface. The timewarp plugin reads these handles, and accesses the textures to apply the transform. To prevent data races between the timewarp and the OpenGL application, the latter uses a double-buffering technique where two pairs of textures are maintained. The OpenGL application renders to one pair of textures, while timewarp reads from the other pair. Indeed, we observed visual artifacts when a single pair of textures was used. By default, Filament renders the frames into an internal framebuffer, which is not accessible from outside. However, Filament allows the user to import OpenGL textures from outside, and use them as the rendertarget. Thus, we create four (two pairs) color textures and a depth texture, and import them to Filament. As Filament renders the scene into these textures, we broadcast their handles for the timewarp plugin.

The third challenge is run-time environment setup. To build and link ILLIXR and Filament in the same environment, we resort to a Docker container. The reason is that the ILLIXR infrastructure alone has a large number of dependencies ranging from specific versions of OpenCV, GTSAM to OpenGL headers and libraries. In addition, it also requires a specific version of the clang++ compiler to be built. As a result, it is nearly impossible to meet all these requirements in every machine. The Filament project also has its strict requirements before it can be built. In particular, Filament expects clang++-7 compiler, whereas ILLIXR expects clang++-10. Since these two versions of clang are not fully compatible, we have to introduce minor modifications to Filament's codebase, such as explicitly including standard library headers, and adding namespaces to variable declarations. Moreover, by default, Filament uses libc++, which is LLVM's implementation of the standard library, instead of libstdc++, which is GNU's implementation of the standard library. We observed that using libc++ instead of libstdc++ in ILLIXR causes deadlocks. Thus, we forced Filament to use libstdc++. This is an unfortunate by-product of the fragmented software ecosystem.

3.2. Writing an ILLIXR plugin using Filament

With our augmented ILLIXR tool, it becomes nearly trivial to write the main application plugin. Algorithm 1. shows the important pieces.

The plugin is derived from the threadloop class in ILLIXR, which encapsulates a dedicated thread running in a tight loop, as discussed in Section II. We add several variables to this class for essential objects such as Engine, View, SwapChain, and others that Filament requires. These structures are initialized during the thread setup phase, which is executed just once before the thread goes into the loop phase. In the setup phase, the Filament engine and other vital parts are initialized. The GLTF loader of Filament starts importing the 3D model and its textures asynchronously. We also create the OpenGL textures, import them into Filament, and set them as rendertargets.

Algorithm 1 ILLIXR plugin using Filament. _p_thread_setup() and _p_one_iteration() are inherited from the threadloop base class.

```
1:  procedure FILAMENT PLUGIN
2:  _p_thread_setup():
3:      filament :: Engine : Create(glContext)
4:      Create Swapchain
5:      Create Renderer
6:      Create Camera
7:      Create View
8:      Create Scene
9:      Load 3D asset
10:     Create GL textures
11:     Import textures to Filament
12: _p_one_iteration():
13:     Add entities to scene
14:     Get headset pose
15:     for i ← 1, 2 do
16:         Update camera
17:         Update rendertarget
18:         Render scene
19:     end for
20:     Publish texture handles
21: end procedure
```

Within the thread loop, the scene is populated with asynchronously loaded objects. Next, the headset pose is read through ILLIXR's interface. This headset pose, which consists of a position vector and a quaternion that describes the head's orientation in space, is transformed into a matrix and used as the Filament's camera model matrix. As the last step before rendering the scene, the appropriate pair of OpenGL textures are set as the rendertarget of Filament. Finally, two render calls are issued to produce a stereo image.

At any point within the plugin, graphics settings can be adjusted. Filament features point, spot and directional light sources that can cast shadows, image-based lighting, anti-aliasing, screen-space ambient occlusion(SSAO), screen-space reflections(SSR), screen-space refractions and more [1].

Our augmented tool is open sourced and available at github.com/yavuz650/ILLIXR and github.com/yavuz650/filament, which include the custom plugin along with the modified Filament source code.

4. Case Study: Gaze-tracked foveated rendering

We now demonstrate a use case of our augmented tool. First, we further extend ILLIXR by implementing gaze estimation. Then, we add Nvidia Variable Rate Shading (VRS) to Filament, and use the estimated gaze to enable foveated rendering in ILLIXR [8]. To conduct this study without this augmented tool, we would have to spend a great amount of time writing an OpenGL renderer and the major tasks would include GLTF loading, where we would need to parse a GLTF file and manage the vertices, textures, and other attributes. We would also have to write our own shaders, which would demand expertise and time if PBR is desired. Moreover, any modern 3D application would require lighting and shadows, which are not trivial to implement. Thanks to our augmented tool, we do not have to worry about any of these daunting tasks, as Filament handles all that. Instead, we only need to invoke a few API calls to Filament, as summarized in Algorithm 1, where we provide the settings of the scene. We benchmark three popular scenes and present the frametimes, motion-to-photon latencies, and frame breakdowns, and analyze the overhead of eye tracking.

4.1. Gaze Estimation in ILLIXR

In the context of VR, gaze estimation refers to the task of determining where on the screen the user's eyes are looking at. Currently, the ILLIXR infrastructure features a plugin that runs a convolutional neural network to perform eye segmentation [16]. Eye segmentation is a computer vision task that involves the identification and labeling of different regions within an image of the human eye, such as the iris, pupil, and sclera. However, additional steps beyond eye segmentation are required to perform gaze estimation. In this work, we adopt the open-source implementation that DeepVOG [40] uses, which is based on ellipse fitting, and unprojection algorithms [30] [33]. We used OpenCV library's [10] ellipse fitting functions, and implemented the unprojection algorithm in C++. Ultimately, the eye tracking plugin computes a pair of numbers that corresponds to the screen coordinates where the user's eyes are looking at.

We use the OpenEDS 2019 [19] dataset to stimulate the gaze estimation system. This dataset consists of over 350,000 images collected from over 150 participants. The images are grayscale and have a resolution of 400x640. We feed the input images at a rate of 20 per second. Each input consists of one image, which we duplicate in the plugin to imitate a stereo-eye camera.

4.2. Foveated Rendering with Filament in ILLIXR

Foveated rendering is an optimization that exploits the decrease in human visual acuity between the eye's center and the periphery. In this scheme, the frames are rendered at the highest quality where the eyes are focused, and at a gradually lower quality in the peripheral regions. This curtails computation without sacrificing any visual quality that is perceptible to the user [20] [29].

Foveated rendering has been the subject of ample research, and there is a multitude of possible implementations [18] [20] [26] [29]. While most of the existing methods require significant changes to the rendering pipeline, Nvidia's VRS technology offers a simple and effective way of implementing foveated rendering [8]. VRS exposes a set of API calls that allow developers to specify a shading rate for every 16x16 pixel tile on the screen. The shading rate corresponds to how many fragment shaders are invoked for the associated tile. For example, the shading rate could be

1 invocation per 1 pixel in the eye fixation region; whereas it could be reduced to 1 invocation per 2x2 pixels in the periphery region, and 1 invocation per 4x4 pixels in the outer periphery regions. The developer is responsible for supplying a palette, which is a map that pairs numbers with shading rates, and a 2D texture, which specifies the shading rates for every 16x16 tile using the numbers in the palette.

In this work, we modify the OpenGL driver of Filament and incorporate the necessary API calls to enable VRS. We only enable VRS for the color renderpass, and disable it for other renderpasses such as shadowmaps, as the color pass is where the scene is actually rendered from the user's perspective. All other passes can be considered as either preprocessing or postprocessing.

Inside the ILLIXR plugin, we use the estimated gaze position to construct a 2D array that contains the shading rates of the 16x16 tiles. This 2D array is passed to Filament's OpenGL driver and is uploaded to the GPU as a 2D texture. Algorithm 2 summarizes the important changes we made to the existing code.

Algorithm 2 Important steps in foveated rendering with ILLIXR augmented with Filament. Only changes to the original code are shown.

procedure COMPUTE THE 2D VRS MASK IN ILLIXR
_p_thread_setup():
  $sri_w \leftarrow display\_width/16$
  $sri_h \leftarrow display\_height/16$
  $mask.resize(sri_w * sri_h)$
_p_one_iteration():
  $x, y \leftarrow$ *Latest gaze coordinates*
  for $i = 0$ to sri_h **do**
    for $j = 0$ to sri_w **do**
      $dist \leftarrow sqrt(|j-x|^2 + |i-y|^2)$
      if $dist > 19$ **then**
        $mask[i \times sri_w + j] \leftarrow 2$
      else if $dist > 12$ **then**
        $mask[i \times sri_w + j] \leftarrow 1$
      else
        $mask[i \times sri_w + j] \leftarrow 0$
      end if
    end for
  end for
  renderer.setVRSMask(mask)
end procedure
procedure ENABLE VRS IN FILAMENT'S OPENGL DRIVER
beginRenderPass():
  if *if(enableVRS)* **then**
    glBindTexture(VRS_mask_texture)
    glTexSubImage2D(VRS_mask)
    glEnable(GL_SHADING_RATE_IMAGE_NV)
  else
    glDisable(GL_SHADING_RATE_IMAGE_NV)
  end if
end procedure

Similar to [29], we use three eccentric layers of different shading rates. Fig. 4 depicts the sizes of each layer.

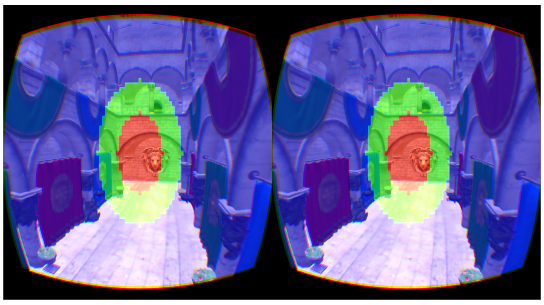

Figure 4: Three eccentric layers used in this work. The eyes are assumed to be looking at the center of the screen. The red region is rendered with 1 fragment shader invocation per 1x1 pixel, the green region is 1 invocation per 2x2 pixels, and the blue region is 1 invocation per 4x4 pixels. The regions are not perfectly circular due to lens distortion.

We experimented with different shading rate configurations but did not notice remarkable differences in terms of performance. As the goal of this study is workload analysis, the configuration shown in Fig. 4 is adequate.

4.3. Methodology

We conduct benchmarks lasting 60 seconds each, using the three scenes described in Table 1. Intel Sponza [25] is a new, PBR based model of the Sponza palace. This scene features very high geometric complexity along with 4k textures, and is the most intense scene that we use. Due to GPU memory limitations, we downscaled the textures to 2k resolution. San Miguel [24] is another popular scene in the graphics community with a large number of textures with varying resolutions. Finally, Amazon Bistro [23] is also widely used for research, and it consists of over 2 million triangles and high-resolution textures. All scenes are illuminated with image-based lighting, and we added multiple point-light sources to Sponza and San Miguel. All light sources cast shadows. We also enabled SSR, SSAO, and 4x multisample anti-aliasing.

TABLE 1: SCENE STATISTICS. TEXTURES INCLUDE BASE COLORS, SPECULAR, NORMAL, AND ROUGHNESS MAPS.

Scenes	# of triangles	# of texture files	Total size of textures
Intel Sponza	5,744,002	148	903MB
San Miguel	5,600,782	323	139MB
Bistro Exterior	2,828,266	231	679MB

We use the EuRoC MAV dataset [15] to emulate a real camera and an IMU sensor. An offline camera, i.e., a dataset, is also used to ensure that the benchmarks are consistent across different runs, as the camera movement is identical across different runs.

We run the benchmarks in four different configurations. We start with no eye tracking or foveated rendering, which

is the baseline. Then, we enable eye tracking without foveated rendering to determine its overhead. We then enable both eye tracking and foveated rendering to see how much performance, if any, is gained. Finally, we also test foveated rendering without eye tracking by setting a constant gaze at the center of the screen. We gather the frametime, motion-to-photon latency, and GPU memory usage. To provide more insightful analysis, we also measure how much time each renderpass within a frame takes in the Sponza scene. We incorporate OpenGL queries into the OpenGL driver of Filament to measure the frametime of every frame in the benchmarks. In a similar fashion, we measure renderpasses of every 10th frame. We use the existing logging infrastructure that ILLIXR has to gather motion-to-photon latency. Finally, we use the nvidia-smi tool to measure GPU memory usage. We run these benchmarks on a workstation desktop machine with an Intel Xeon E5-1650 CPU, 32GB of memory, and an Nvidia RTX 2080 GPU.

4.4. Results and Discussion

We first present the frametimes and MTP latency in Fig. 5 and 6 respectively. We clearly see the impact of eye tracking on performance. Frametimes increase noticeably in all scenes, and almost by 2x in Sponza. This level of performance impact is understandable given that the GPU is now burdened with running a large neural network in addition to rendering graphics. Enabling foveated rendering alleviates this overhead, though not completely. We also note that foveated rendering alone does not provide any meaningful performance gain. Finally, there is hardly any change in MTP latency. This is mainly because MTP does not directly depend on frametime, but rather on timewarp's performance. In ILLIXR, frames are sent to the display by timewarp, regardless of whether the main OpenGL application rendered a new frame or not. Timewarp will always read the latest frame, predict the headset pose, apply the transformation, and send it to the display. The only way foveated rendering can improve MTP is by virtue of releasing more GPU resources to timewarp.

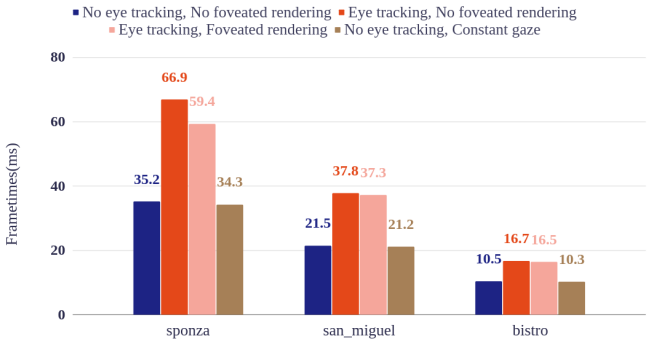

Figure 5: Frametimes measured in different scenes with different configurations.

We now present the average durations of renderpasses in Fig. 7 for a deeper understanding of the system. As we

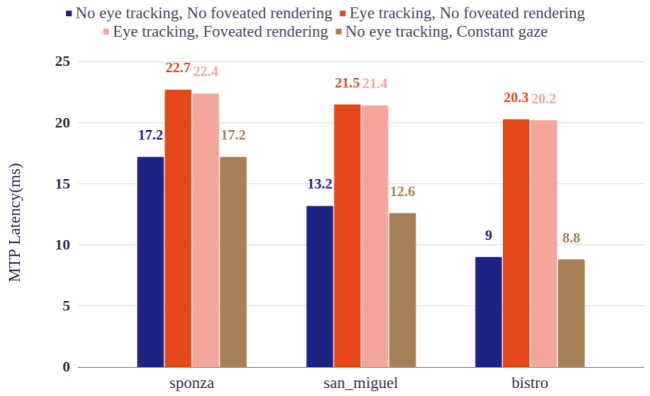

Figure 6: Motion-to-photon latency measured in different scenes with different configurations.

mentioned before, VRS is only enabled during the main view pass, i.e., the color pass. Therefore, it can only speed up the color pass, which already appears to take a small portion of the overall frametime. Moreover, VRS only reduces the number of fragment shader invocations, which further limits its potential. This implies that, with VRS, the performance gain heavily depends on the scene. We observe that SSR and shadowmap rendering takes the most time. One possible optimization would be to render shadowmaps once, and use them for both left and right eye views, as shadowmaps are independent of the user's view point.

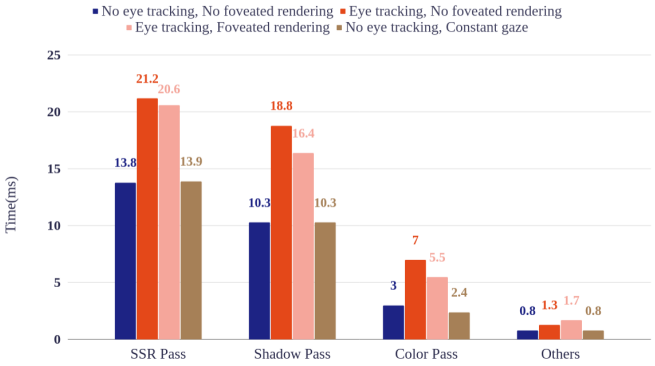

Figure 7: Renderpasses measured in the Sponza scene. Other passes include SSAO, color grading, and other trivial passes.

Fig. 8 shows the GPU memory consumption for all configurations. We note that eye tracking introduces a large memory footprint, approximately 3 GB, which increases the pressure on caches and contributes to the performance overhead.

At a first glance, it may appear as if foveated rendering is not beneficial and the extra overhead of eye tracking is unjustified. However, as discussed before, we use VRS to implement foveated rendering, but there may be other more efficient ways to implement it. Moreover, eye tracking can be used for realistic personal avatar rendering, and developers can incorporate special effects or features into their applications based on the user's gaze. In other words,

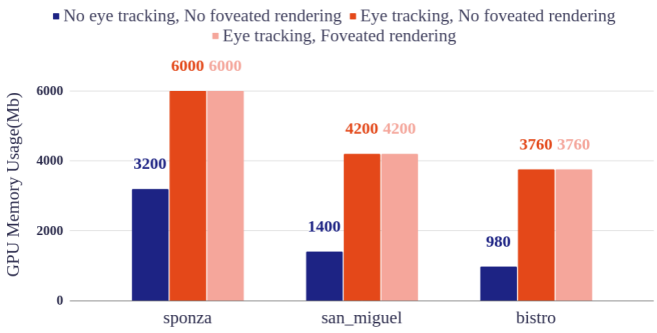

Figure 8: GPU memory usage.

the overhead of eye tracking can be amortized among multiple tasks.

5. Related Work

5.1. Virtual/Augmented Reality Systems

The original ILLIXR work [21] conducts a detailed performance and power analysis of the Virtual/Augmented Reality system on both desktop and mobile systems and reveals that almost all components fail to meet their targets on mobile platforms, and that system-wide optimizations are imperative. Another study that extends ILLIXR is conducted by Zhao et al. [41], where they develop a customized Augmented Reality(AR) system and use it to evaluate their power-efficient design.

5.2. VR Accelerators

Xie et al. propose domain-specific accelerators that utilize processing-in-memory to accelerate timewarp [39], and exploit the data locality between stereo frames to optimize stereo rendering [38]. Wen et al. profile 3D VR applications and present an accelerator that optimizes the post-processing stage of the graphics pipeline [37]. Our augmented tool can be an open and customizable alternative to the workloads used in these works.

6. Conclusion and Future Work

In this work, we extend an existing tool, ILLIXR, by integrating a high-end rendering engine, Filament. This augmentation allows researchers to render high-quality, physically-based graphics in ILLIXR, paving the way for more thorough research of VR graphics. We demonstrate a use case of this augmented tool by investigating gaze-tracked foveated rendering. We analyze the overhead of eye tracking and the performance gains from foveated rendering.

For future work, ILLIXR can be upgraded to use the Vulkan API instead of OpenGL, which can improve performance and consistency of the system. In addition, integrating faster and more efficient eye-tracking systems into ILLIXR can be beneficial.

Acknowledgements

We would like to thank the reviewers for their insightful comments. This work is partly funded by NSF grant 1908406 and an AMD gift fund.

References

[1] "Filament Rendering Engine." [Online]. Available: https://github.com/google/filament

[2] "Godot Engine." [Online]. Available: https://godotengine.org/

[3] "ILLIXR Consortium." [Online]. Available: https://illixr.org/

[4] "Inside-out tracking," Tech. Rep. [Online]. Available: https://learn.microsoft.com/en-us/windows/mixed-reality/enthusiast-guide/tracking-system

[5] "Meta Quest Pro." [Online]. Available: https://www.meta.com/quest/quest-pro/tech-specs/

[6] "Monado - open source XR platform." [Online]. Available: https://monado.dev/

[7] "Moving Frostbite to PBR." [Online]. Available: https://www.ea.com/frostbite/news/moving-frostbite-to-pb]

[8] "Nvidia Variable Rate Shading." [Online]. Available: https://developer.nvidia.com/vrworks/graphics/variablerateshading

[9] "PlayStation VR2." [Online]. Available: https://www.playstation.com/en-us/ps-vr2/ps-vr2-tech-specs/

[10] "The OpenCV Library." [Online]. Available: https://opencv.org/

[11] "Unity Engine." [Online]. Available: https://unity.com/

[12] "Valve Index Headset." [Online]. Available: https://www.valvesoftware.com/en/index/headset

[13] "Virtual Reality Market Size, Share & Trends Analysis Report." [Online]. Available: https://www.grandviewresearch.com/industry-analysis/virtual-reality-vr-market

[14] H. Bellini, W. Chen, M. Sugiyama, M. Shin, S. Alam, and D. Takayama, "Virtual & Augmented Reality: Understanding the Race for the Next Computing Platform ," 2016.

[15] M. Burri, J. Nikolic, P. Gohl, T. Schneider, J. Rehder, S. Omari, M. W. Achtelik, and R. Siegwart, "The EuRoC micro aerial vehicle datasets," *The International Journal of Robotics Research*, 2016. [Online]. Available: http://ijr.sagepub.com/content/early/2016/01/21/0278364915620033.abstract

[16] A. K. Chaudhary, R. Kothari, M. Acharya, S. Dangi, N. Nair, R. Bailey, C. Kanan, G. DIaz, and J. B. Pelz, "RITnet: Real-time semantic segmentation of the eye for gaze tracking," in *Proceedings - 2019 International Conference on Computer Vision Workshop, ICCVW 2019*, 2019.

[17] R. L. Cook and K. E. Torrance, "A Reflectance Model for Computer Graphics," *ACM Transactions on Graphics (TOG)*, vol. 1, no. 1, 1982.

[18] L. Franke, L. Fink, J. Martschinke, K. Selgrad, and M. Stamminger, "Time-Warped Foveated Rendering for Virtual Reality Headsets," *Computer Graphics Forum*, vol. 40, no. 1, 2021.

[19] S. J. Garbin, Y. Shen, I. Schuetz, R. Cavin, G. Hughes, and S. S. Talathi, "OpenEDS: Open Eye Dataset," 2019.

[20] B. Guenter, M. Finch, S. Drucker, D. Tan, and J. Snyder, "Foveated 3D graphics," in *ACM Transactions on Graphics*, vol. 31, no. 6, 2012.

[21] M. Huzaifa, R. Desai, S. Grayson, X. Jiang, Y. Jing, J. Lee, F. Lu, Y. Pang, J. Ravichandran, F. Sinclair, B. Tian, H. Yuan, J. Zhang, and S. V. Adve, "ILLIXR: Enabling End-to-End Extended Reality Research," in *Proceedings - 2021 IEEE International Symposium on Workload Characterization, IISWC 2021*, 2021.

[22] S. M. Lavalle, A. Yershova, M. Katsev, and M. Antonov, "Head tracking for the Oculus Rift," in *Proceedings - IEEE International Conference on Robotics and Automation*, 2014.

[23] A. Lumberyard, "Amazon Lumberyard Bistro, Open Research Content Archive (ORCA)," 7 2017. [Online]. Available: http://developer.nvidia.com/orca/amazon-lumberyard-bistro

[24] M. McGuire, "Computer Graphics Archive," 7 2017. [Online]. Available: https://casual-effects.com/data

[25] F. Meinl, K. Putica, C. Siqueria, T. Heath, J. Prazen, S. Herholz, B. Cherniak, and A. Kaplanyan, "Intel Sample Library," 2022.

[26] X. Meng, R. Du, M. Zwicker, and A. Varshney, "Kernel Foveated Rendering," *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 1, no. 1, 2018.

[27] D. C. Niehorster, L. Li, and M. Lappe, "The accuracy and precision of position and orientation tracking in the HTC vive virtual reality system for scientific research," *i-Perception*, vol. 8, no. 3, 2017.

[28] S. Palmisano, R. S. Allison, and J. Kim, "Cybersickness in Head-Mounted Displays Is Caused by Differences in the User's Virtual and Physical Head Pose," *Frontiers in Virtual Reality*, vol. 1, 2020.

[29] A. Patney, M. Salvi, J. Kim, A. Kaplanyan, C. Wyman, N. Benty, D. Luebke, and A. Lefohn, "Towards foveated rendering for gaze-tracked virtual reality," *ACM Transactions on Graphics*, vol. 35, no. 6, 2016.

[30] R. Safaee-Rad, I. Tchoukanov, K. C. Smith, and B. Benhabib, "Three-Dimensional Location Estimation of Circular Features for Machine Vision," *IEEE Transactions on Robotics and Automation*, vol. 8, no. 5, 1992.

[31] C. Schlick, "An Inexpensive BRDF Model for Physically-based Rendering," *Computer Graphics Forum*, vol. 13, no. 3, 1994.

[32] J. P. Stauffert, F. Niebling, and M. E. Latoschik, "Latency and Cybersickness: Impact, Causes, and Measures. A Review," 2020.

[33] L. Świrski and N. A. Dodgson, "A fully-automatic, temporal approach to single camera, glint-free 3D eye model fitting," in *Pervasive Eye Tracking and Mobile Eye-Based Interaction (PETMEI)*, 2013.

[34] The Khronos Group, "OpenXR Overview - The Khronos Group Inc," 2021.

[35] J. M. Van Waveren, "The asynchronous time warp for virtual reality on consumer hardware," in *Proceedings of the ACM Symposium on Virtual Reality Software and Technology, VRST*, vol. 02-04-November-2016, 2016.

[36] B. Walter, S. Marschner, H. Li, and K. Torrance, "Microfacet models for refraction through rough surfaces," *Eurographics*, 2007.

[37] Y. Wen, C. Xie, S. L. Song, and X. Fu, "Post0-vr: Enabling universal realistic rendering for modern vr via exploiting architectural similarity and data sharing," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 390–402.

[38] C. Xie, F. Xin, M. Chen, and S. L. Song, "Oo-vr: Numa friendly <u>o</u>bject-<u>o</u>riented <u>vr</u> rendering framework for future numa-based multi-gpu systems," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 53–65. [Online]. Available: https://doi.org/10.1145/3307650.3322247

[39] C. Xie, X. Zhang, A. Li, X. Fu, and S. Song, "PIM-VR: Erasing motion anomalies in highly-interactive virtual reality world with customized memory cube," in *Proceedings - 25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019*, 2019.

[40] Y. H. Yiu, M. Aboulatta, T. Raiser, L. Ophey, V. L. Flanagin, P. zu Eulenburg, and S. A. Ahmadi, "DeepVOG: Open-source pupil segmentation and gaze estimation in neuroscience using deep learning," *Journal of Neuroscience Methods*, vol. 324, 2019.

[41] S. Zhao, H. Zhang, C. S. Mishra, S. Bhuyan, Z. Ying, M. T. Kandemir, A. Sivasubramaniam, and C. Das, "Holoar: On-the-fly optimization of 3d holographic processing for augmented reality," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 494–506. [Online]. Available: https://doi-org.prox.lib.ncsu.edu/10.1145/3466752.3480056

Appendix

1. Abstract

This Artifact presents the source code of our augmented tool and instructions to run the benchmarks that are used in this work.

2. Artifact check-list (meta-information)

- **Program:** ILLIXR
- **Run-time environment:** Ubuntu 20.04 Docker Container
- **Hardware:** Nvidia GPU with Variable Rate Shading Support with at least 8GB device memory
- **Metrics:** Frametime, Motion-to-photon latency, frame breakdown
- **Output:** Metrics
- **Experiments:** Three scenes with four configurations each
- **How much disk space required (approximately)?:** 25GB for Docker image
- **How much time is needed to prepare workflow (approximately)?:** 1-2 hours in total to setup Nvidia Docker Toolkit and pull our image.
- **How much time is needed to complete experiments (approximately)?:** 15-20 minutes
- **Publicly available?:** Yes
- **Archived (provide DOI)?** Docker Image and the data used for graphs: https://doi.org/10.5281/zenodo.8260684

3. Description

3.1. How to access. We provide a Docker image hosted on Zenodo and Docker Hub that has the source code and all the dependencies installed in it.

3.2. Hardware dependencies. To run the benchmarks, an Nvidia GPU with Varible Rate Shading support is required. At least 8GB of device memory is also required to be able to run all benchmarks. We used an RTX 2080 GPU to run our benchmarks.

3.3. Software dependencies. A Linux system with X11 window system is required, Wayland is not tested. Docker engine and Nvidia Docker Toolkit must be installed in it. We used an Ubuntu 22.04 system to run our benchmarks. The augmented ILLIXR system and Filament have a large set of dependencies, but they are very difficult to install properly on every system, therefore a Docker container is mandatory.

4. Installation

Start with installing the Docker engine if you have not before, and then proceed to install the Nvidia Container Toolkit. The latter is required so that the Docker container can access your Nvidia GPU.

You can pull the Docker image from Docker hub using the command below,

```
docker pull ystozlu/illixr-docker:iiswc2023
```

Or, you can download the image from Zenodo, and run the following command.

```
docker load -i illixr-filament-docker.tar
```

You can check if the image is ready by running `docker image ls -a`. You should see an image that takes up roughly 25GB of space. Before starting a container from this image, you need to run the following command so that the Docker container can access your X windowing system.

```
xhost +local:root
```

Now you can start a container with the following command. Make sure you change the image name if you downloaded from Zenodo.

```
docker run -it --privileged --name iiswc-docker
-e "DISPLAY=${DISPLAY}" --hostname iiswc-docker
-v /tmp/.X11-unix:/tmp/.X11-unix
--gpus all ystozlu/illixr-docker:iiswc2023 /bin/bash
```

Once in the container, check if everything is properly setup by running `vkcube`. This should create a small window with a spinning cube in it.

5. Experiment workflow

While in the container, simply run `./iiswc2023.sh`. This will run the benchmarks one by one, and save the results in .csv files. The script should take about 15 minutes.

6. Evaluation and expected results

Once the benchmarks finish running, the generated metrics will be placed in three folders:

```
/opt/ILLIXR/iiswc_sponza_results
/opt/ILLIXR/iiswc_sanmiguel_results
/opt/ILLIXR/iiswc_bistro_results
```

You can use the scripts under `/opt/ILLIXR/metric_scripts` to summarize the .csv files. Note that the numbers you will get will not be same, maybe not even close, to the numbers we present in the paper. This is because ILLIXR is not a simulation, but a real-time workload. Therefore, the numbers will depend on your hardware. However, you should be able to observe the insights and trends that we point out in the paper. For example, when eye tracking is enabled for a given scene, the frametime and MTP should increase drastically. Likewise, when foveated rendering is enabled for a given scene, frametime should decrease, albeit not much. These observations should be particularly conspicuous for the Sponza scene.