applied sciences

MDPI

Article

VanityX: An Agile 3D Rendering Platform Supporting Mixed Reality

Ivan Zoraja [1], Mirjana Bonkovic [2,*], Vladan Papic [2] and Vaidy Sunderam [3]

[1] Zoraja Consulting, Spinčićeva 2E, 21000 Split, Croatia; zoraja@fesb.hr
[2] Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture, University of Split, Ruđera Boškovića 32, 21000 Split, Croatia; vpapic@fesb.hr
[3] Department of Mathematics and Computer Science, Emory University, 400 Dowman Dr., Atlanta, GA 30322, USA; vss@emory.edu
* Correspondence: mirjana@fesb.hr

Abstract: VanityX is a prototype, low-level, real-time 3D rendering and computing platform. Unlike most XR solutions, which integrate several commercial and/or open-source products, such as game engines, XR libraries, runtime, and services, VanityX is a platform ready to adapt to any business domain including anthropology and medicine. The design, architecture, and implementation are presented, which are based on CPU and GPU asymmetric multiprocessing with explicit synchronization and collaboration of parallel tasks and a predictable transfer of pipeline resources between processors. The VanityX API is based on DirectX 12 and native programming languages C++20 and HLSL 6, which, in conjunction with explicit parallel processing, the asynchronous loading and explicit managing of graphic resources, and effective algorithms, results in great performance and resource utilization close to metal. Surface-based rendering, direct volume rendering (DVR), and mixed reality (MR) on the HoloLens 2 immersive headset are currently supported. Our MR applications are directly compiled and deployed to HoloLens 2 allowing for better programming experiences and software engineering practices such as testing, debugging, and profiling. The VanityX server provides various computational and rendering services to its clients running on HoloLens 2. The use and test cases are in many business domains including anthropology and medicine. Our future research challenges will primarily, via the MetaverseMed project, focus on opening new opportunities for implementing innovative MR-based scenarios in medical procedures, especially in education, diagnostics, and surgical operations.

Keywords: 3D platforms; 3D engines; asymmetric parallel computing; surface-based rendering; direct volume rendering; mixed reality; DirectX 12; HoloLens 2

Citation: Zoraja, I.; Bonkovic, M.; Papic, V.; Sunderam, V. VanityX: An Agile 3D Rendering Platform Supporting Mixed Reality. *Appl. Sci.* **2023**, *13*, 5468. https://doi.org/10.3390/app13095468

Academic Editors: Zhihan Lv, Kai Xu and Zhigeng Pan

Received: 16 March 2023
Revised: 20 April 2023
Accepted: 25 April 2023
Published: 27 April 2023

1. Introduction

One of the biggest issues in real-time (online, interactive) 3D rendering is the lack of time. We must calculate at least 24 images (frames) per second (fps), produce high-quality images without tearing, and respond to user input with acceptable latency. In offline 3D rendering, single images can be calculated much longer and later combined into a movie; therefore, movies look better than interactive games, but the users cannot interact with movies as they can with games or with scientific and medical simulations.

Three-dimensional rendering [1] is a process of calculating (generating) two-dimensional images with depth (reliefs) from a scene. A scene is a collection of 3D objects, virtual cameras, lights, and other entities needed for creating images. The rendering process is based on a pipelined architecture, which, through various programmable and configurable stages, produces images. In surface-based rendering, models are represented via polygonal meshes, while in direct volume rendering (DVR), models are represented via volumes of scalars (clouds of points). DVR is typically used with medical imaging obtained from

scanning devices [2,3]. Using various triangulation techniques, clouds of points can be, via isosurfaces [4], converted into triangle meshes.

There are many sources where the performance of interactive 3D rendering can be accelerated, for example, in hardware (e.g., CPUs and GPUs), programming languages, 3D libraries, 3D algorithms and data structures, 3D engines [5–8], and 3D applications. The underlying hardware architecture for both online surface-based and volumetric rendering is based on old-school parallel (concurrent) processing [9,10]. The management and the sharing of graphics resources in this asymmetric parallel computing can be implicit and explicit. In the former approach, the management is left to the drivers and runtimes, while in the latter approach, fine-grain control and synchronization, e.g., via fences and resource barriers, are given to the programmer, which could lead to a performance boost [11]. In comparison to the former approach, the latter one also provides explicit multi-GPU rendering [8,12].

Three-dimensional libraries such as DirectX [13] and OpenGL [14] implement software layers around the underlying CPUs and GPUs, providing a programming model (API) mainly for tasks such as transferring data between CPUs and GPUs [15,16] and synchronizing their activities. They define rendering pipelines with configurable and programmable stages and expose them to the programmers while 3D real-time engines such as Unity [17] and Unreal Engine [18] typically provide object-oriented programming interfaces to the underlying libraries as well as various data structures, acceleration algorithms, editors, and tools for creating and rendering 3D scenes.

This paper presents the VanityX platform, which emerged as a prototype during work on our MetaverseMed project [19], the goal of which was to ensure efficient and constructive collaboration of the surgical team by providing interactive possibilities of mixed reality. VanityX is an integral XR solution (platform): it has its own game engine that can be configured to run on desktop and HoloLens 2 devices and a server that provides computational and rendering services. Using services, the 3D model can be created and/or prepared for rendering on more powerful desktop computers and transferred to a lighter SOC application processor (the Qualcomm Snapdragon 850 CPU with the Adreno 630 GPU) located in the back enclosure of the Hololens2, thus accelerating the 3D model blending process and meeting the client requirements running on HL2 more efficiently. Additionally, VanityX uses a *proprietary* API based on DirectX 12 and native programming languages C++20 and HLSL 6, resulting in performance and resource utilization close to metal. Similar systems could be realized using commercial/open-source components such as a (Unity) game engine with an appropriate (Microsoft) runtime, (OpenXR) APIs, services, etc. Integration of these components is usually time consuming and not an easy task, resulting in questionable quality since the system as a whole could be more constrained than the weakest component itself. Additionally, specific user requirements, such as volumetric rendering in our case, were not part of the standard Unity assets and have to be purchased from third parties. The mentioned drawbacks motivated us to build up our own solution and share the experiences and hints with the community. The main achievements presented in this paper could be highlighted as follows: (i) the presentation of main VanityX's architectural layers including the engine, components, libraries, and services, all built on the top of DirectX 12, together with the implementation UML diagrams realized on the Universal Windows Platform (UWP), and (ii) the presentation of application structures including deployment on standard desktop computers and HL2 devices, together with their performance indices, measured on two different GPUs.

Hence, the conceptual model for a 3D rendering and computational platform based on DirectX 12 is developed and demonstrated. At this stage, all the core components are implemented and their functionalities tested. Production level usage is planned through entrepreneurship with the mentioned MetaverseMed project in the field of medicine. The main purpose of the platform is to support the collaboration of medical teams to perform tasks more efficiently. The application demonstrates that VanityX supports both surface-based and direct volume rendering as well as the triangulation techniques that convert

clouds of points into a set of triangles, needed due to the project's functional requirements. As we have already mentioned, such techniques in typical commercial game engines such as Unity are generally left for third-party libraries.

In the sections that follow, the architecture of the platform is minutely described in a top-to-bottom manner, from the higher abstraction level to the low-level details related to implementation. The next section presents VanityX's pipelined software architecture, components, and layers and discusses its purpose with respect to its requirements. It also describes VanityX's main loop and the Vanity object, which acts as a proxy for interfacing with UWP (Universal Windows Platform) constructs and integrating with the rendering process. Section 3 presents VanityX's design and its core C++ and HLSL types, with an accent on wrapping the underlying DirectX 12 COM components. Section 4 deals with the design and implementation of mixed reality on HoloLens 2 directly running on DirectX 12. Section 5 provides insight into the development environment and tools we used, while in Section 6, we turn our attention to testing examples and applications rendered by VanityX on both standard and HoloLens 2 displays.

Finally, we finish off with Section 7, which concludes the paper, discusses our findings and results, and gives suggestions and opportunities for future work.

2. VanityX Platform

As mentioned in the introduction, the platform supports an immersive environment in which relevant information is visualized in real time for the medical team and which would ensure their efficient collaboration. Calculated images, important for exchanging ideas among team members, can be seen via head-mounted displays (HMD) such as Oculus Quest and HoloLens 2 [20].

With reference to Figure 1, such devices allow for mixing real and digital objects on the same scene. In virtual reality (VR), all objects in the scene are digital, and the user cannot see real objects [21]. In augmented reality (AR), we have real and digital objects in the scene with simple interactions via basic human senses, while mixed reality provides more intensive interactions among real and digital objects in the scene, making the user feel more immersed in the scene. The previous definitions are a bit vague, and recently, the term extended reality (XR) has been coined for all three. Mixed reality can be applied in almost any domain such as medicine [21] and, for example, in personal decision support systems [22].

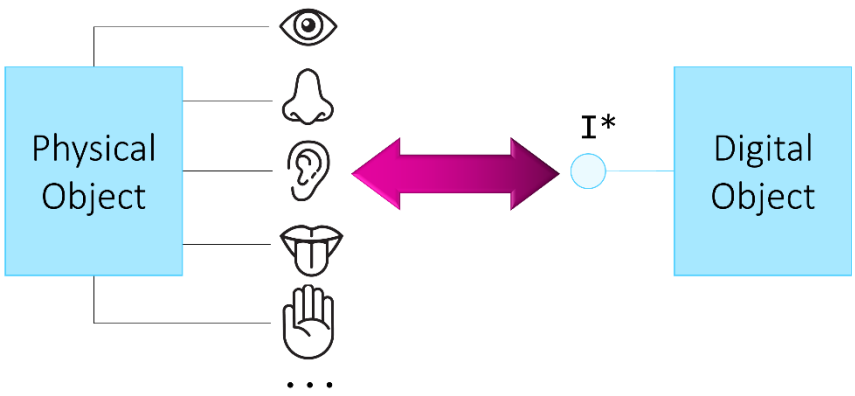

Figure 1. Interactions between physical (e.g., a human) and digital objects. Immersive scenes support a variety of interactions via multiple human senses.

In this way, the functional requirements of the platform are defined and supported by appropriate software solutions that include surface-based and direct volume rendering, the extraction of isosurfaces, dynamic tessellation, and the integration of computation, rendering, and remote services [23,24]. On the other side, software architecture is strongly influenced by quality (nonfunctional) requirements, such as immersive 3D graphics, in

which at least 30 frames per second of high resolution (4K is recommended on flat displays) images are generated. It is also important for the project's goals to support rendering on standard gaming machines (desktops) as well as on mixed reality headsets to ensure better interoperability. Additionally, the platform supports high performance and scalability, assuming fast rendering with low latency for high-poly models and textures, for volumes obtained from high-resolution scanning devices such as MRI and CT and for solutions with multiple interactive users. For programmability and reusability purposes, the system is decomposed into software components (classes) that are incorporated into other components and reused on different hardware architectures and rendering pipelines. The following subsection minutely defines the layers and functionality of the realized software architecture.

2.1. VanityX's Software Architecture

Software architecture is strongly influenced by quality (nonfunctional) requirements. With reference to Figure 2, VanityX provides a 3D engine, computational and rendering services, and an API decomposed into four main layers: a layer that contains core façades and its components such as the device and command objects, a layer that contains the main rendering loop and the corresponding components such as event handlers and timers, a layer that contains the Vanity object and components related to interfacing with the operating system and interacting with the user, and the Vanity 3D library.

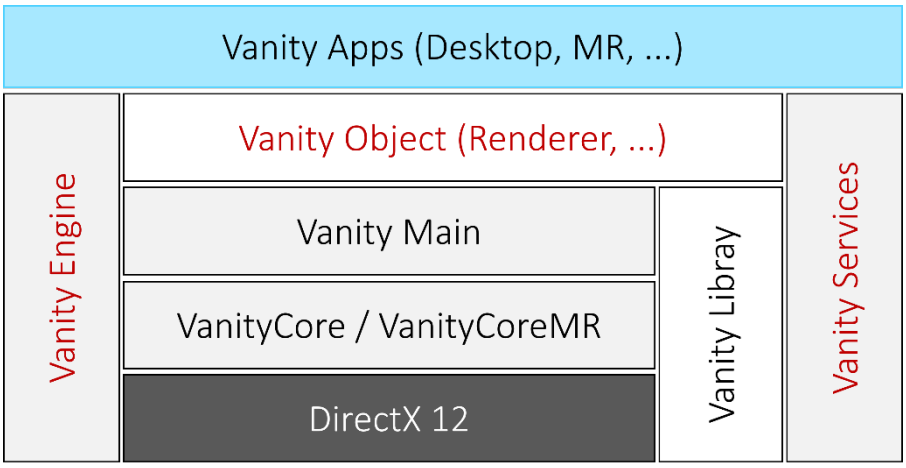

Figure 2. VanityX's main architectural layers, engine, components, libraries, and services, all built on top of DirectX 12. Applications are implemented on the Universal Windows Platform (UWP).

To fulfill low-level quality requirements, the VanityX engine is based on the DirectX 12 library. The DirectX 12 library is based on the Component Object Model (COM)—its components are binary components and therefore are close to metal and very fast. DirectX 12 supports the programming of CPUs in the C++ programming language and the programming of GPUs in High-Level Shader Language (HLSL) [25]. HLSL comes in six DirectX programming stages (shaders): vertex, geometry, compute, tessellation, pixel, and ray tracing (DirectX 12 Ultimate) stages. DirectX 12 provides the DXGI library [26], which manages low-level graphics hardware and features, such as adapters, outputs, and resolutions, that are independent of the DirectX runtime.

Unlike DirectX 11, DirectX 12 allows programmers to explicitly synchronize tasks and assets between the CPUs and the GPUs and therefore bring faster frame rates.

The Vanity Core façade is responsible for managing components that are related to the DirectX 12 runtime and desktop applications, while the Vanity Core MR façade manages DirectX 12 components that work with DirectX 12 runtime and HoloLens 2 applications—the façades are discussed in more detail in Section 3.3. and Section 4.3. The Vanity Main is a layer that implements the components of the main rendering loop—this is explained in

the next section. The Vanity object implements components that provide the entry point into the application and the injection of renderers into the main rendering loop.

The Vanity 3D Library supports common 3D math operations and functions, 3D data structures, and algorithms—on both the CPU and the GPU. Low-level math operations and functions on the CPU are implemented in C++20 and are built on top of the DirectXMath library [27], which supports SIMD operations that are optimized for various versions of Windows. On the GPU, the library is implemented in HLSL. The Vanity 3D library also supports various algorithms such as intersections and collisions, the polygonization of volumetric data, lightening and texturing, standard and proprietary formats, direct volume rendering, and computation on GPUs.

The Vanity services provide computational (e.g., machine learning techniques on multiple GPUs) and rendering solutions to its clients with limited resources and computational power while the Vanity server supports the services via asynchronous message passing [9,28]. Details on VanityX libraries and network services are out of the scope of this paper.

2.2. Vanity 3D Engine

The VanityX engine is a native prototypical low-level real-time 3D engine that provides fast 3D rendering close to hardware-level rendering performance by explicitly managing 3D resources and explicitly synchronizing tasks between the CPU and the GPU in an asymmetric parallel computing pattern. High performance and low latency are its key nonfunctional requirements. We support both surface-based and direct volume rendering of medical images on standard displays as well as on HoloLens 2, Microsoft's MR HMD. VanityX provides two core façades (C++ objects) for accessing underlying core DirectX 12 COM objects for both standard rendering and rendering to HoloLens 2. The Vanity object is a proxy for entering the main rendering loop.

The following is typical client code for (1) instantiating the Vanity object, (2) passing a renderer to it in a dependency injection manner, and (3) starting the rendering loop calling method Run():

```
auto renderer = std::make_unique<Renderer>( . . . );
Vanity vanity{ std::move(renderer) };
vanity.Run();
```

To the best of our knowledge, the VanityX engine is currently the only 3D engine supporting mixed reality (HoloLens 2) functionality on top of the DirectX 12 library.

3. Vanity Core Components

To support programmability, reusability, and unit and integration tests, core VanityX classes are implemented as object-oriented wrappers around the underlying DirectX 12 COM objects, and because of this, the users of VanityX are not supposed to explicitly deal with the underlying COM objects. For example, a COM object that implements interface ID3D12Device2 is encapsulated in VanityX's Device class via the winrt::com_ptr template—a reference-counted COM smart pointer. Likewise, a COM object that implements interface IDXGISwapChain4 is encapsulated in the SwapChain class. DirectX 12 creational free functions that instantiate COM objects, such as D3D12CreateDevice(), are encapsulated in VanityX's methods.

Resources are buffers and textures. VanityX implements constant buffers, vertex buffers, index buffers, front and back buffers, and depth stencil buffers. Specific descriptor types such as descriptors for render targets and depth stencil buffers are cohesively refactored into individual classes that inherit from class DescriptorHeapBase. VanityX supports multidimensional textures in various texture formats as well as samplers that can be configured for various address modes and filters.

3.1. Vanity Object

The Vanity object is the entry point into Vanity's 3D applications. It is primarily responsible for hiding the complexity of managing interactions with UWP [29] and passing the scene renderer into the main rendering loop.

With reference to Figure 3, the Vanity object does not directly instantiate an AppView object—this is the job of the ViewFactory object. Its class implements the IFrameWorkViewSource interface whose method CreateView() will instantiate an AppView object. The Vanity object instantiates a ViewFactory object and passes it to a CoreApplication object calling method Run(). The CoreApplication object represents an application by UWP. It creates the AppView object by calling method CreateView(). An IRenderer object represents the actual scene renderer—it is passed from the Vanity object to the AppView object.

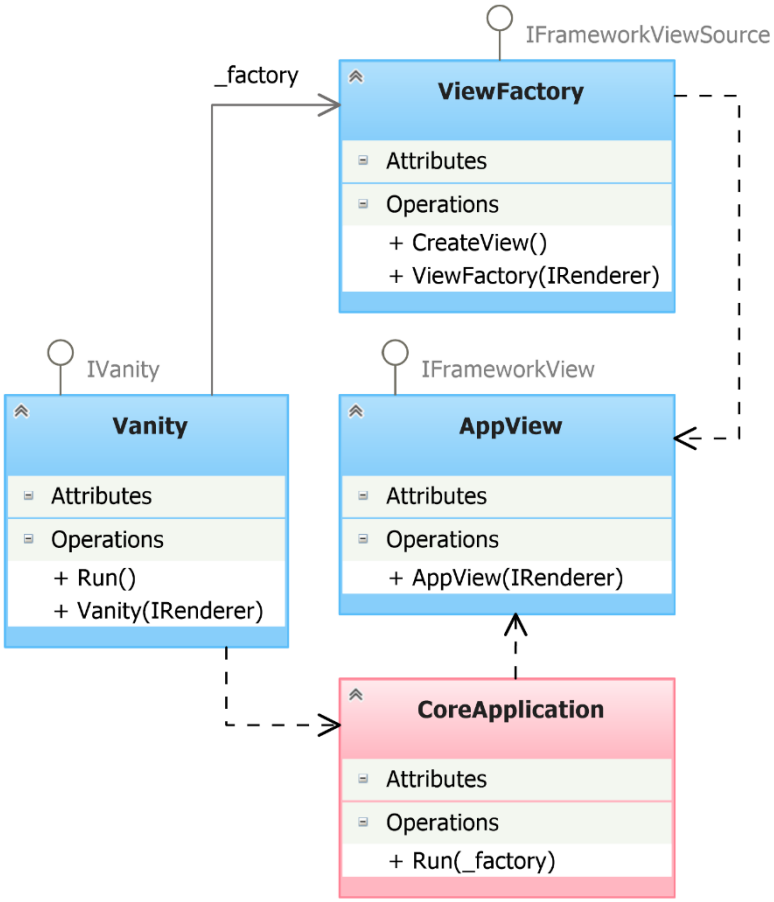

Figure 3. The VanityX object depicted via Unified Modeling Language (UML). It manages objects for interfacing with UWP and provides the connections to the underlying rendering pipeline.

The AppView manages the states of a UWP application, which can be in Not Running, Running, and Suspended states. During activation, the application loads data asynchronously and enters the Running state. On suspending, the application stores data and enters the Suspended state. On resuming, the application restores data and enters the Running state again. From the Suspended state, the application can be automatically terminated, e.g., when the system is low on memory. VanityX components are designed to be extended by inheriting their classes.

3.2. Main Rendering Loop

The AppView object allows for creating the main application window, handling various input events, and implementing the lifecycle phases of the application such as activation, suspending, resuming, and termination. During the activation of an AppView

object, a sequence of methods is automatically invoked by the UWP runtime. For example, method Initialize() instantiates the main loop object and injects the renderer into the main rendering loop, while method Run() enters the main rendering loop.

The primary responsibility of the main rendering loop is the execution of the graphics pipeline. It implements a standard execution pattern in which the pipeline must be initialized, periodically executed, and shut down. With reference to Figure 4, the AppView object instantiates the main loop object (class MainLoop) and passes (via a shared pointer) a scene renderer (an instance of class Renderer) object to it. During the initialization process, the main loop object instantiates a VanityCore object, or a VanityCoreMR object for HoloLens 2 applications, and a Timer object, which makes use of the hardware counters for the calculations of various timings and statistics.

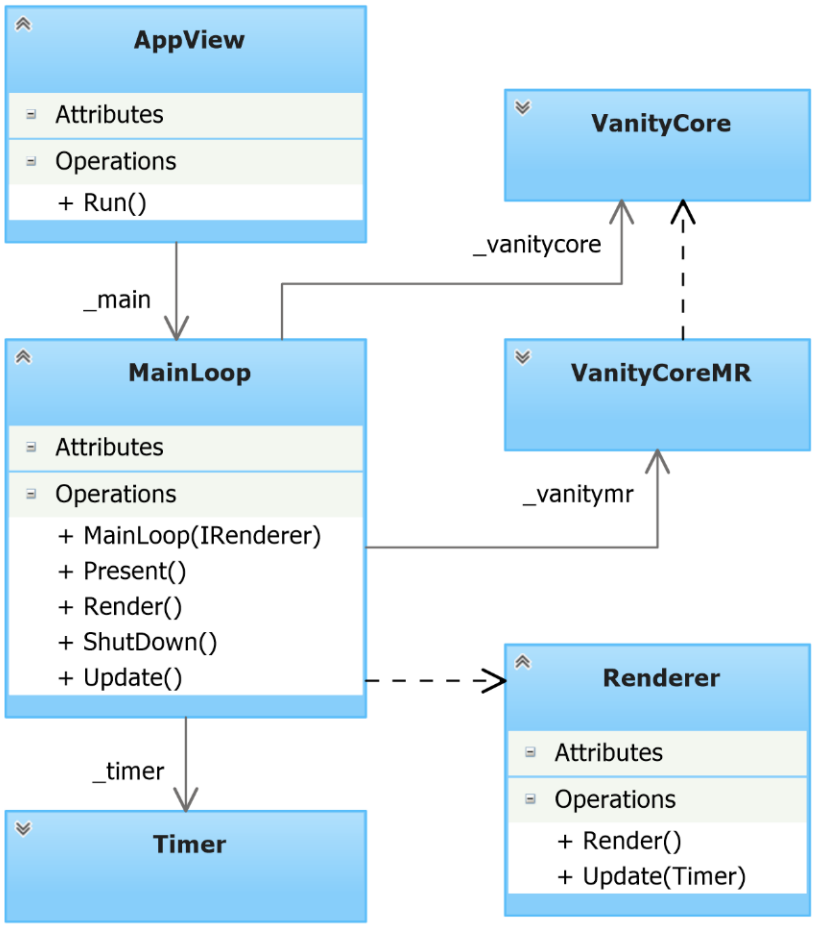

Figure 4. VanityX's main rendering loop integrates the rendering process and event management from a variety of input sources.

In each rendering iteration, (1) events from dispatcher CoreWindow are processed, (2) network messages from the server are processed, (3) the scene is updated for the current frame, (4) the scene is rendered into the back buffer, and finally, (5) the scene is presented onto the screen, as shown in the code snippet below:

```
dispatcher.ProcessEvents(...);
client.ProcessMessages(...);
auto currentframe = _main->Update(previousframe);
auto ok = _main->Render(currentframe);
if(ok) _main->Present(currentframe);
```

3.3. Vanity Façade

The Vanity façade is responsible for creating core Vanity objects related to the core types of the underlying DirectX 12 graphics pipeline and for providing easy access to them. It is implemented via the VanityCore class.

With reference to Figure 5, to support a higher frame rate, the Vanity façade implements triple buffering. Its method Present() presents the content of the front buffer (the swap chain) on the screen. A DescriptorHeapBase object implements a heap where resource descriptors (views) are stored. Descriptors can be of various types such as constant buffer view (CBV) and shader resource view (SRV). Heap types (default, upload, readback, and custom) determine how resources can be accessed by the CPU and the GPU.

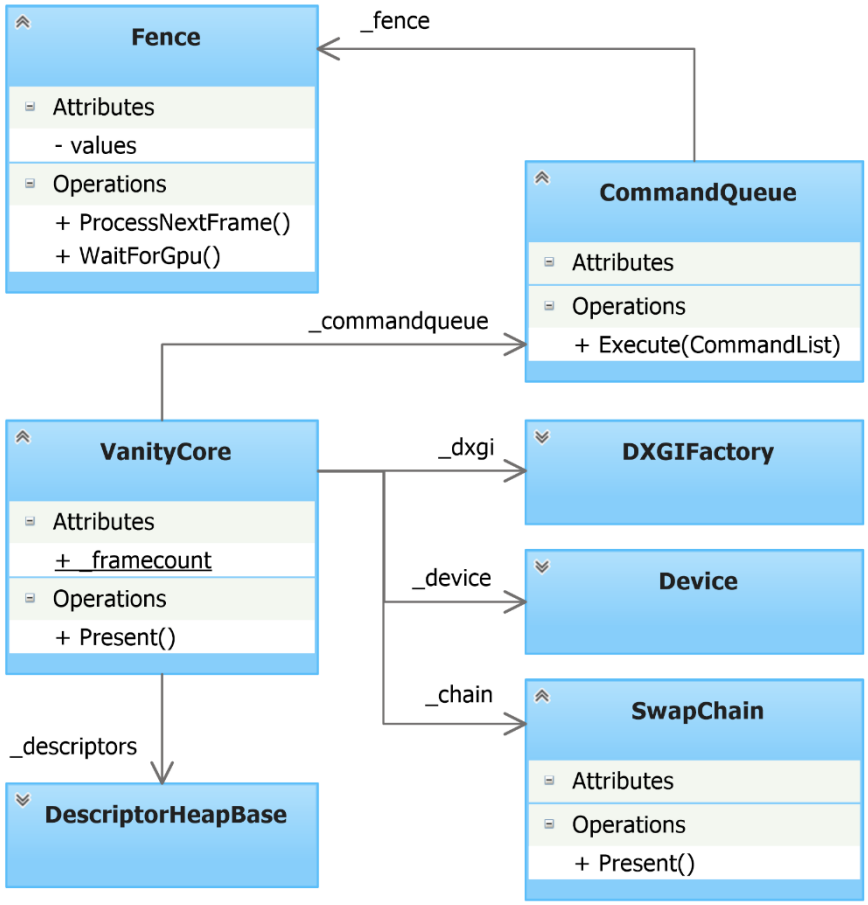

Figure 5. The Vanity façade implements the Façade design pattern, providing a simpler interface to the underlying 3D objects.

The Device class encapsulates the functionality of DirectX 12 objects, which are used for things such as creating resources, managing feature levels, and multisampling. The SwapChain class implements triple buffering and handles things such as presenting front buffers onto the display and resizing displays. The DXGIFactory class manages low-level hardware features such as video adapters, outputs, and resolutions—it is, for example, used for creating a SwapChain object.

The commands are submitted to a CommandQueue object. A CommandAllocators object manages the allocations and deallocations of commands that are not immediately executed by the GPU since it must synchronize with the CPU (they may run in parallel). The synchronization is implemented via a Fence object that encapsulates a long integer value and methods for the actual cooperation and synchronization between the processors—the interaction is based on the traditional producer–consumer coordination pattern.

3.4. Scene Object

With reference to Figure 6, the SceneObject class represents an object (model) that can be rendered. It contains a Mesh object that represents the model's geometry, multiple Texture objects that represent the textures to be mapped onto the object, multiple Material objects that describe its interactions with the light, and a World object that defines its location in the scene. A Mesh object defines the topology of rendering primitives, a method for asynchronous loading models and assets, a method for uploading the geometry to the GPU, a method for binding the geometry to the pipeline, and a method for drawing the geometry.

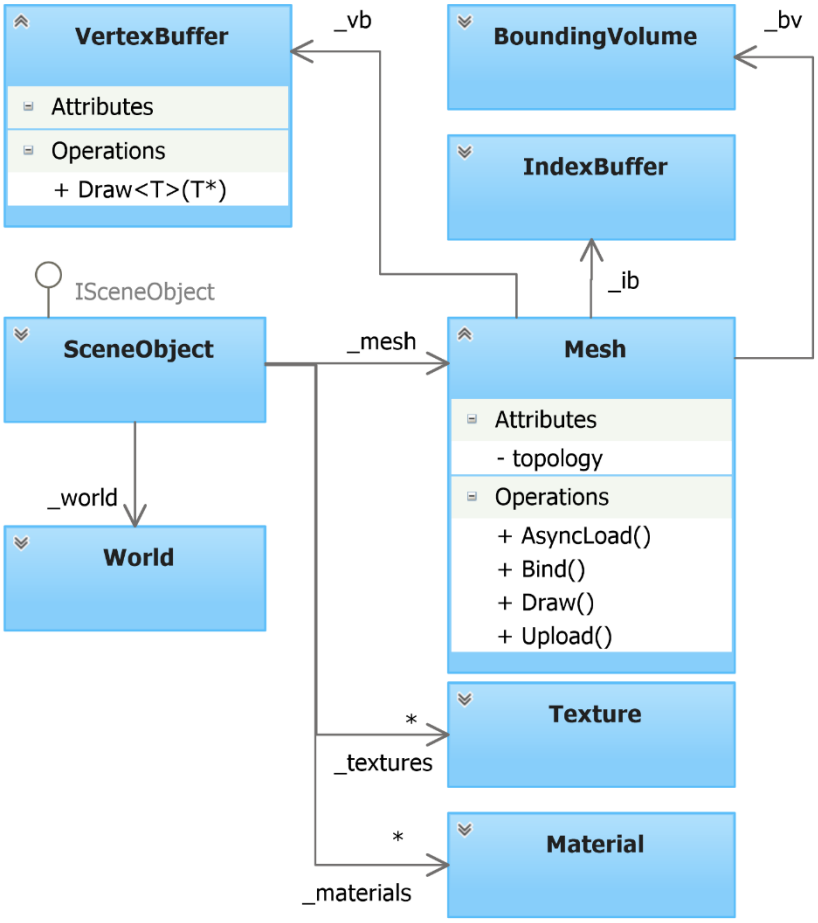

Figure 6. The scene object manages meshes, their locations on the scene, and corresponding assets such as textures and materials.

The VertexBuffer and IndexBuffer objects manage buffers for vertices and indices, respectively, while a BoundingVolume object is used for calculating intersections (collision) with other objects on the scene. A VertexBuffer object contains a Draw<>() method, which draws the scene object. It is parameterized by the type of the vertex (T), and a vertex can have multiple attributes such as position, normal, texture coordinates, tangents, etc.

3.5. Pipeline Management

Resources are not directly bound to shader registers.

A root signature is an array of parameters that point to resource descriptors. It describes the logical layout of pipeline resources, specifying which resources, such as buffers, textures, and samplers, can be bound to the pipeline, as well as the GPU registers the resources will be mapped to. For example, a world matrix for each frame is transferred as a resource to the GPU; for triple buffering, we have three world matrices on the resource

heap, and each matrix has a descriptor on the descriptor heap. A root signature contains a parameter, which, depending on the frame, points to the corresponding resource descriptor.

Class RootSignatureBase implements the core behavior of the root signature, while class DescriptorHeapBase implements the core behavior of the descriptors heap such as binding descriptors and parameters to commands. Client apps inherit those classes and add/override application-specific features. The pipeline accepts vertices and produces pixels (fragments). Vertices and indices are encapsulated in classes VertexBuffer and IndexBuffer, respectively. A pipeline state is used to initialize configurable stages and to set programmable shaders—all DirectX 12 shaders are supported, including the compute shader. Class PipelineState implements the functionality for managing shaders and configurable stages such as the rasterizer, depth, and blend states.

3.6. Managing Commands

Commands execute the rendering pipeline and start parallel tasks. VanityX provides four classes for managing commands: CommandList, CommandQueue, CommandAllocators, and Fence.

A CommandList object is used to add commands that can be directly executed on the GPU, as well as command bundles that are preprocessed and optimized by the GPU. The CPU and the GPU can run in parallel. The synchronization object, via events and a set of values, implements a producer–consumer coordination pattern in which the GPU is idle when the queue is empty while the CPU is idle when the queue is full. The GPU and the CPU make use of waiting and signaling primitives to synchronize their work.

With reference to Figure 7, a CommandList object can be in four states. Calling method Create() on it, it enters the Empty state. Adding (recording) a command, the command object transits to the Ready state and stays in that state as long as new commands are added to it. Before being executed, the command object must be closed by calling method Close() on it. By calling the Execute() method on the command object, it transits to the Submitted state. A command object can be reused by calling method Reset(), in which case it enters the Empty state again and can be used for recording new commands.

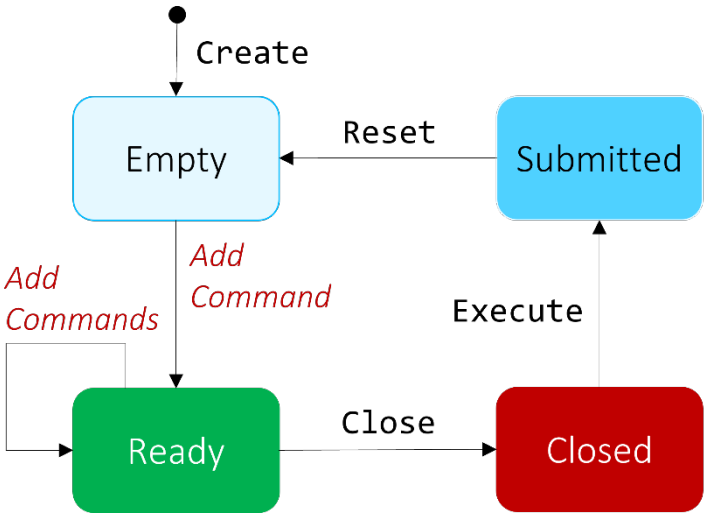

Figure 7. The states of a CommandList object represented via a final state machine.

4. Mixed Reality with VanityX

The mixed reality part of VanityX and the corresponding applications are designed, implemented, and tested to run on the Microsoft HoloLens 2 immersive headset.

With reference to Figure 8, the Vanity Server provides various computational and rendering services to the applications (engines) running on HoloLens 2 devices. The server also interacts with a variety of medical devices, such as ROBOT, to retrieve medical data,

images, and videos in the DICOM format [30] and transfer them to its MR applications. To allow medical staff participating in medical procedures without wearing HoloLens 2 devices to see the images being rendered, the apps can mirror their screens onto external ones.

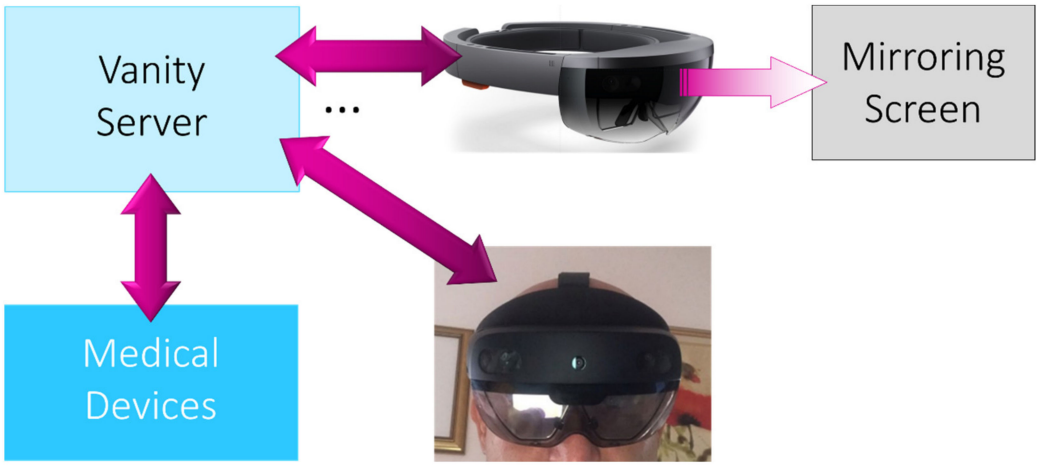

Figure 8. VanityX platform supporting multiple HoloLens2 headsets.

4.1. HoloLens 2

HoloLens 2 is an amazing HMD device—it is an untethered holographic computer.

The heart of HoloLens 2 is the holographic processing unit (HPU) 2.0 [31,32] that sits in the front enclosure. It receives data from various sensors, e.g., from the depth, head-tracking, and eye-tracking cameras, processes them, and sends the processed data, via the PCIe, to the SOC application processor (the Qualcomm Snapdragon 850 CPU with the Adreno 630 GPU) located in the back enclosure. The application processor runs the app, calculates (renders) the image, and sends the rendered image to the holographic processor, which performs the late-stage reproduction (LSR) algorithm, correcting the position of the hologram and thereby improving the hologram stability, and then sends the image to the display module.

The HPU 2.0 performs various tasks related to scene management and control, such as head and eye tracking; hand tracking and various gesture readings; spatial audio, which makes the sound appear exactly from the position where the holograms are located; spatial anchors [33]; and scene understanding, where it can recognize things such as walls and floors. HoloLens 2 provides a MEMS (micro-electromechanical systems) laser display with 47 pixels for the degree of sight—its vertical field of view beta is 29°, its horizontal field of view alpha is 43°, and its resolution for each eye is 2K (1363 × 2021).

4.2. MetaverseMed

Our current research and development activities in mixed reality are a part of the MetaverseMed project [19]. We are focused on implementations of medical IT solutions for HoloLens 2 on the top of the VanityX platform and their integration with medical procedures. Our native engine is currently built on top of the HolographicSpace API [34], but we also, for the sake of comparisons, implemented some features using the OpenXR API [35] and the Unity engine.

For our prototype, we choose the native HolographicSpace API over the native OpenXR API and commercial 3D engines. The Open API is the standard, and standards are very useful when developing software for various devices—our XR platform is HoloLens 2. Unity is currently based on DirectX 11 and therefore does not support explicit parallelism (parallel rendering tasks) and a predictable transfer of pipeline resources between the CPU and the GPUs.

Figure 9 shows our testing of medical solutions based on the VanityX platform in a real medical environment—interactions are implemented using the MRTK library [36] with the Unity engine. By overlapping medical scans, in the form of 3D holograms, on the patient's body parts (organs), MetaverseMed will primarily empower medical staff, accelerating diagnosis and improving the accuracy of surgical operations. Holograms blend digital objects, such as medical images and volumes, with physical patients, making them look transparent and therefore opening up opportunities for new medical scenarios that were not possible before. Medical staff wearing HoloLens 2 headsets can always see their surroundings, walk around, and interact with people, physical objects, and holograms. MR training [37] will enable innovative ways of learning and sharing knowledge among medical staff and students.

Figure 9. Testing project MetaverseMed in a real medical environment. In the Cyber room, the surgeons and radiologists practice and test eye tracking, head movements, voice commands, and hand commands. Courtesy of prof. dr. Dragan Schwarz from private hospital Radiochirurgia, Zagreb, Croatia.

Integrating VanityX holograms with the CyberKnife 7 radiation delivery system [38] is our most challenging research—we will enable the surgeons and radiologists to observe and manipulate medical holograms inside the patients as if they were transparent, helping them improve the precision of diagnoses and surgical operations.

4.3. Vanity MR Façade

The Vanity MR façade represents core MR features, and it is implemented via the VanityCoreMR class. It is responsible for creating core VanityX objects related to things such as the holographic camera, spatial coordinates, pipeline resources, and descriptor heaps.

Figure 10 shows some basic components of the Vanity MR Façade. Via the VanityCore object, the VanityCoreMR accesses VanityX objects related to core types of the underlying DirectX 12 graphics pipeline. The Fence class, in conjunction with command-based classes, provides synchronization mechanisms (producer–consumer primitives) between the CPU

and the GPU. The ViewProjection class represents both the view and projection matrices for the left and right eyes. Class Buffer represents constant buffers for transferring camera matrices for triple buffering, while class DescriptorHeapBase manages views for the camera's buffers.

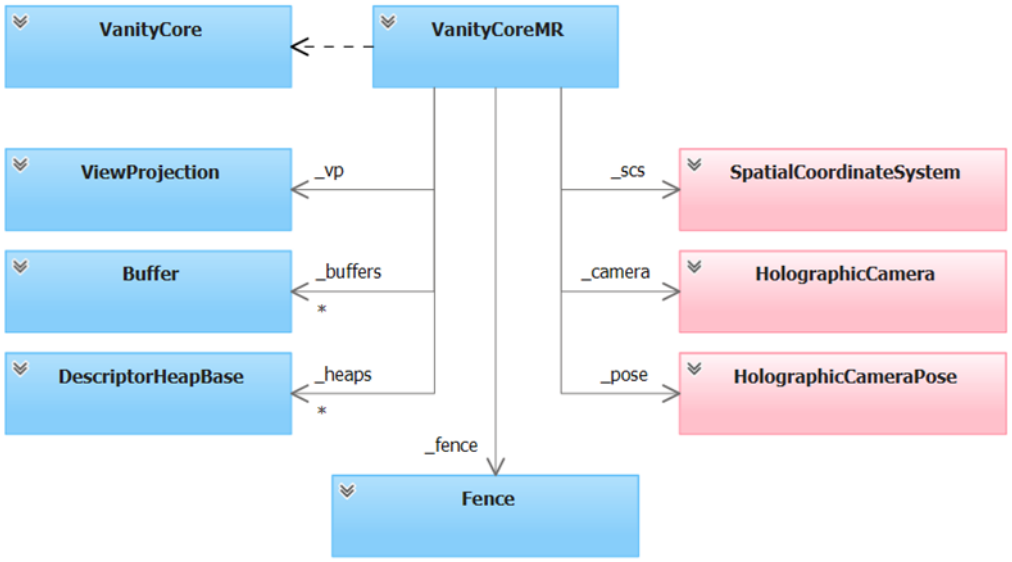

Figure 10. VanityX's MR Core Façade implements the Façade design pattern, providing a simpler interface to MR-based components (*—multiplicity).

The HolographicCamera and the HolographicCameraPose classes are a part of the UWP Holographic API [39]. The former provides rendering functionality for each frame on the holographic display, which is represented via class HolographicDisplay, while the latter represents the predicted position of the holographic camera for a particular frame. The SpatialCordinateSystem class represents the user surroundings and positions (anchors) where coordinate systems can be anchored and shared across devices—the class is a part of the UWP Perception API [40].

4.4. VanityX vs. Unity

In addition to our approach, mixed reality applications for HoloLens 2 can be developed natively using the OpenXR API and using commercial engines such as Unity and Unreal Engine.

For our research, we analyzed the Unity development process for MR applications and compared that process with our approach. In Unity, MR applications can be implemented in C Sharp or in JavaScript—we strongly prefer the first approach. Unity provides an XR custom rendering pipeline and the Mixed Reality Toolkit (MRTK) [36], which simplifies things related to the holographic space and 3D interactions in the scene. Unity performs two compilations. The first compilation is from the Mono development environment to a UWP application using the IL2CPP compiler, while the second compilation is from the UWP application to an ARM64 application, which is then, using Visual Studio 2022, deployed to the HoloLens 2 headset.

Compared to Unity, VanityX's MR approach has the following advantages:

- VanityX is a computation and rendering platform with a server that provides computational and rendering services and an engine that can be configured to run on desktop and HoloLens 2 devices. Unity is a 3D engine.
- Unity MR applications are based on DirectX 11 (support for DirectX 12 in the current version (2021.2.14f1) is still experimental), and therefore, they utilize neither explicit parallelism for transferring graphic resources between the CPU and the GPU nor the explicit synchronization of tasks between the CPU and the GPUs, as VanityX does.

- VanityX supports both surface-based and direct volume rendering as well as triangulation techniques that convert clouds of points into a set of triangles. Such techniques in Unity are generally left for third-party libraries.

5. Development Environments

The development environment for VanityX is based on Visual Studio 2022 and its various tools, such as debuggers, performance analyzers, loggers, and unit testing tools. For the sake of performance and explicit asymmetric parallelism, VanityX is based on DirectX 12. Programming language C++20 is used to implement code on the CPU while HLSL 6.* is used for implementing programmable stages (shaders) of the underlying rendering pipeline. Desktop applications are tested on the NVIDIA Quadro RTX 4000 GPU and NVIDIA RTX 1080Ti, while mixed reality applications are tested on the HoloLens 2 simulator [41], the emulator version 21H1 [42], and the development edition's HoloLens 2 immersive headset.

5.1. Application Structure

A VanityX application instantiates a Vanity object passing a scene renderer to it, which renders the scene, via the pipeline, onto the screen.

With reference to Figure 11, the renderer object inherits from the RendererBase class, which implements interface IRenderer and provides basic rendering functionality such as creating, initializing, and destroying window-based resources. The renderer object, via its base class, inherits access to the VanityCore object and via that object has access to essential engine objects such as DXGI, Device, and Command.

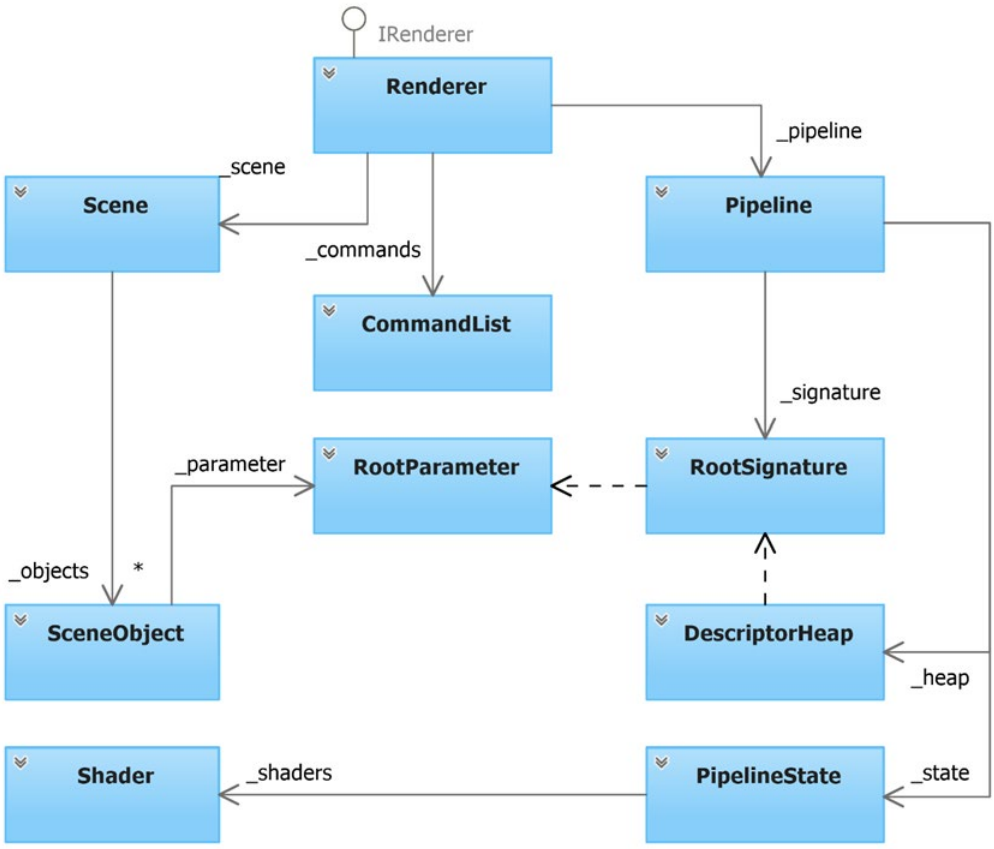

Figure 11. The structure of an application that is rendered on both desktops and HoloLens 2 headsets. A scene is a collection of multiple scene objects (e.g., holograms).

Scene and pipeline objects are instantiated by the renderer object. The scene object, via the co_wait operator and the std::future<> template, asynchronously loads and maintains scene assets such as models, textures, and volumes while the pipeline object asynchronously loads assets such as vertex and pixel shaders and binds them to the rendering pipeline. To update resources on the GPU and draw the scene onto the back buffer, the main rendering loop periodically calls the Update() and Render() methods from the renderer object.

The rendering object implements the observer pattern and, inside the main loop, listens to various events it is registered for. For example, a SourcePressed event handler manages gestures, such as a tap gesture. Input events can also be checked asynchronously.

5.2. Optimization

The VanityX engine and the applications based on it are optimized at three levels to make them run more efficiently and to use fewer resources.

At the system level, we improve the utilization of the CPU and the GPU by implementing explicit multitasking, explicit manipulation (placement) of graphic assets such as meshes and textures, and triple buffering.

At the algorithmic level, we implement efficient 3D algorithms and corresponding data structures. The former encompasses various algorithms such as intersections and collision detection, while the latter includes data structures such as k-d trees as well as spatial data structures that efficiently, in a hierarchical manner, manage the scene and the scene objects. Both surface-based and volumetric models (cloud of points) are converted into our proprietary binary formats (.vxm and .vxc) for faster reading, writing, and transferring. To minimize the allocations from the free store, we prefer utilizing member objects to implement one-to-one associations and custom allocators to implement associations with multiple objects of the same class.

At the micro level, we utilize various techniques such as compile-time computation via immediate functions, unfolding of recursive function calls, SSE (streaming SIMD extensions) instructions, flattening branches, loop unrolling, and online compilations (via the fxc compiler) to generate the best instruction set of the underlying GPU.

5.3. Debugging and Performance

To debug C++ code on the CPU, we use custom logging by tracing and recording VanityX calls with multiple channels and verbosity and Visual Studio 2022′s debugger.

PIX for Windows is used for capturing and analyzing the Direct3D 12 calls that it makes during a given frame. PIX implements a kind of offline (post-mortem) monitoring, where the analysis is performed after the execution—it is contrasted to online monitoring [28], where tools can gather information from the system being monitored and can manipulate its dynamic behavior. We use PIX events (regions of time) and markers from the WinPixEventRuntime API to demarcate regions being monitored. Function PIXBeginEvent() labels the start of a user-defined region while function PIXEndEvent() labels the end of that region. During the launching, PIX instruments the application being observed by injecting the capture layer into it. Using PIX, we record monitoring data in the form of GPU captures and use those traces for various debugging and analysis purposes, such as visualizing bottlenecks and pipeline stalls.

With reference to Figure 12, we captured a GPU frame related to the graphics queue and show the API calls that implement explicit synchronization between the CPU and the GPU. The Wait() call is issued on the ID3DCommandQueue object. It makes the command queue object wait until the specified fence reaches or exceeds the requested value. After the scene has been drawn, the Signal() call is issued on the command queue object to set the fence to the specified value. The Signal() call is issued again after the scene is presented to the screen. The synchronization on the CPU side is performed via VanityX's fence object, as described in Section 3.6.

```
Signal(obj#10,406)  {this->ID3D12CommandQueue obj#2,return->S_OK}
Wait(obj#11,402)  {this->ID3D12CommandQueue obj#2,return->S_OK}
ResourceBarrier(1,...)  {this->ID3D12GraphicsCommandList obj#8}
ClearRenderTargetView(res#1,...,0,nullptr)  {this->ID3D12GraphicsCommandList obj#8}
ClearDepthStencilView(res#2,D3D12_CLEAR_FLAG_DEPTH,1,0,0,nullptr)  {this->ID3D12GraphicsCommandList obj#8}
DrawIndexedInstanced(1344,1,0,0,0)  {this->ID3D12GraphicsCommandList obj#8}
DrawIndexedInstanced(171498,1,0,0,0)  {this->ID3D12GraphicsCommandList obj#8}
DrawIndexedInstanced(8448,1,0,0,0)  {this->ID3D12GraphicsCommandList obj#8}
DrawIndexedInstanced(9792,1,0,0,0)  {this->ID3D12GraphicsCommandList obj#8}
DrawIndexedInstanced(11040,1,0,0,0)  {this->ID3D12GraphicsCommandList obj#8}
DrawIndexedInstanced(76620,1,0,0,0)  {this->ID3D12GraphicsCommandList obj#8}
DrawIndexedInstanced(18672,1,0,0,0)  {this->ID3D12GraphicsCommandList obj#8}
DrawIndexedInstanced(27744,1,0,0,0)  {this->ID3D12GraphicsCommandList obj#8}
ResourceBarrier(1,...)  {this->ID3D12GraphicsCommandList obj#8}
Signal(obj#12,405)  {this->ID3D12CommandQueue obj#2,return->S_OK}
Present(obj#16,0,<unknown>)  {this->ID3D12SharingContract obj#2}
```

Figure 12. Recording a drawing of a human abdomen consisting of eight organs. Each organ is drawn using Direct3D 12′s method DrawIndexedInstanced()—the first parameter is the number of indices that defines the mesh geometry (triangles).

5.4. Deployment to HoloLens 2

The target architecture for VanityX desktop applications is x64. HoloLens 2 mixed reality applications are based on the ARM64 architecture, built on a desktop computer, and deployed to and started on the HoloLens 2 headset. Using Visual Studio 2022, VanityX's mixed reality applications can be deployed via Wi-Fi or via USB.

Applications running on HoloLens 2 in the debugging mode can be remotely controlled from the developing computer via standard debugging techniques such as stepping through, watching variables, and setting breakpoints.

Once deployed, mixed reality applications can be started from HoloLens 2′s Start Menu using taps, air taps, and voice commands.

6. Test Cases

Three types of testing were used: unit testing, integration testing, and end-to-end testing. To test our components (units) individually and independently, we use the Google Test framework integrated with Visual Studio 2022, while our integration and end-to-end testing are performed via 3D test applications created for particular use case scenarios.

6.1. Surface-Based Rendering

In this kind of rendering, we are viewing only the surface (an approximation) of the models being rendered and their interaction with light. A vertex is a basic element making up a 3D object. It defines a point in space using x, z, and y coordinates as well as its other attributes such as color, normal, texture, etc. Using vertices, programmers can create basic rendering primitives such as points, lines, and triangles, and using the graphics primitives, programmers can compose more complex 3D objects.

In VanityX, a mesh is a collection of triangles. In addition to meshes, which define the geometry of a model, models can also contain other features that define their location and appearance on the scene, such as materials, textures, and how they are affected by light. VanityX supports surface-based rendering in the solid and wire-frame modes by directly configuring the DirectX 12 rasterizer state in real time.

With reference to Figure 13, we show a human skull of a catholic saint that is rendered in the solid mode and a human abdomen. The skull model represents a skull (cranium) of a catholic relict—it is created by a professional 3D scanner and is a part of the Interactive Virtual Shrine (IVS) project [43], which provides anthropological and religious aspects of catholic relicts. Its vertex attributes are position, normal, and texture coordinates, and it is rendered in the solid mode. The abdomen model is created by a DCC tool; its vertex attributes are position, normal, and color; and it is rendered in the wire-frame mode without culling back triangles.

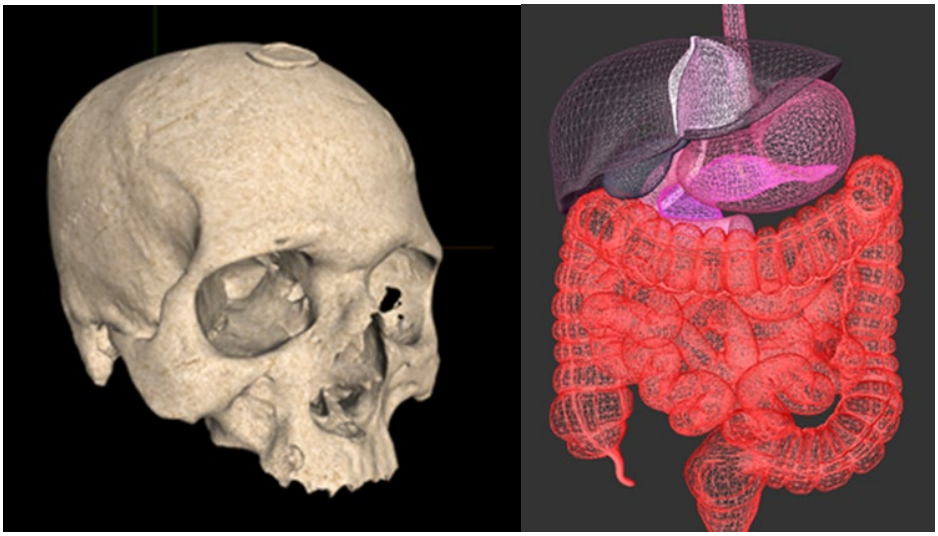

Figure 13. Rendering triangle meshes with VanityX. The relict on the left side is stamped with the papal seal. It is provided by courtesy of Reverend Marijan Jelenić, the catholic church in Vodnjan, Croatia. On the right, we render a human abdomen.

Table 1 and Figure 14 show the rendering times for each individual organ of a human abdomen on different GPUs.

Table 1. Durations of rendering human organs on the NVIDIA RTX Quadro 4000 GPU.

Organ	Number of Vertices	Number of Indices	Duration (ns)
Bile duct	896	1344	256
Falciform ligament	5632	8448	1312
Gall bladder	6528	9792	1184
Pancreas	7360	11,040	1120
Liver	12,448	18,672	2944
Stomach	18,496	27,744	3808
Small intestine	51,080	76,620	11,328
Colon	114,332	171,498	25,952

Figure 14. Drawing times. The horizontal axis represents the number of indices for each individual organ; curve Series1 represents measurements made on Quadro, while curve Series2 represents measurements performed on GTX 1080Ti.

6.2. Rendering Medical Imaging

Direct volume rendering (DVR) is best suited for medical and other scanned images. A medical image represents a thin slice of a human body taken from a scanning device such as computed tomography (CT) or magnetic resonance imaging (MRI). Each slice is composed of individual pixels arranged into a rectangular grid—combining individual 2D images into a 3D grid, we obtain clouds of points. An element in the 3D grid is called a voxel (volume element).

Many techniques including deep and machine learning [44] have been developed for the segmentation [45], reconstruction [46], and registration [47] of images, including medical ones. VanityX supports various volume rendering techniques such as texture-based rendering, ray-cast rendering, and rendering using transfer functions. With reference to Figure 15, we render a slice of a human abdomen (liver) and a human skull. The slice of a human abdomen is anonymized and rendered using texture-based rendering, where each axial slice is represented as a texture mapped (sampled) onto a quad. The human skull is rendered using transfer functions that map scalar data onto color and opacity to depict relevant features of the model being rendered.

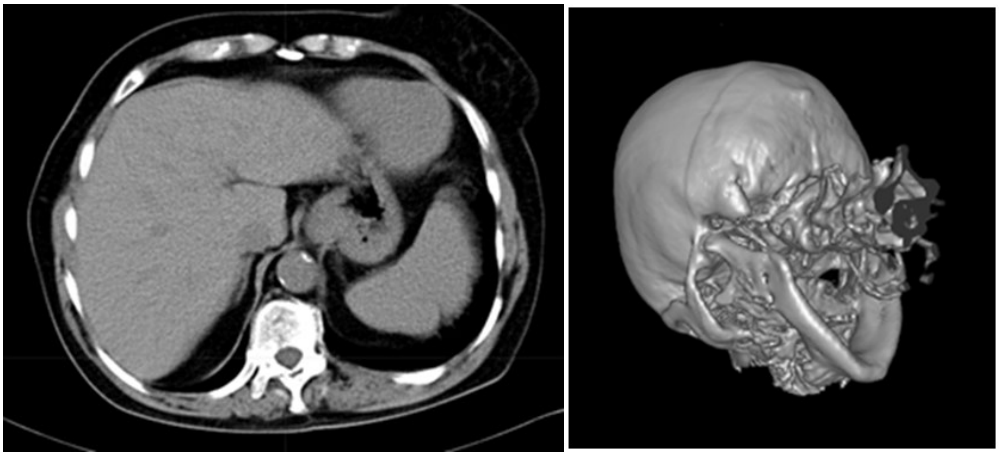

Figure 15. Volume rendering via textures and transfer functions. On the left is a model provided courtesy of Dragan Schwarz from hospital Radiochirurgia, Croatia, while the second model is a free one.

6.3. Isosurfaces

A scalar field is a function that assigns numbers such as densities and attenuation to points in R^3. For volume visualization, computer graphics make use of implicit surfaces (isosurfaces) that are explicitly defined via field functions. There are many techniques for constructing isosurfaces from scalar fields, such as Delaunay triangulations [48] and marching cubes [49]. VanityX makes use of the marching cubes algorithm to construct isosurfaces via triangle meshes from volumetric data acquired from CT and MRI scanning devices.

With reference to Figure 16, we show a 3D surface-based model that is triangulated using the marching cube algorithm. The model represents a triangle mesh of a human brain triangulated from a free volumetric model. Triangle meshes in VanityX are implemented via DirectX triangle lists. A triangle mesh must be two-manifold, where each edge is shared by two faces—only boundary edges may belong to one face. Traversal to neighboring faces is allowed only through edges, and passing through vertices is not allowed.

In some cases, triangle meshes generated by the marching code algorithm are not two-manifold and therefore might be rendered with holes, so we developed a simple 3D tool to visualize marching cubes cases and spot potential errors. The right picture in Figure 16 shows a case where five voxels have bigger values than the threshold, and we use three triangles for creating a part of the isosurface.

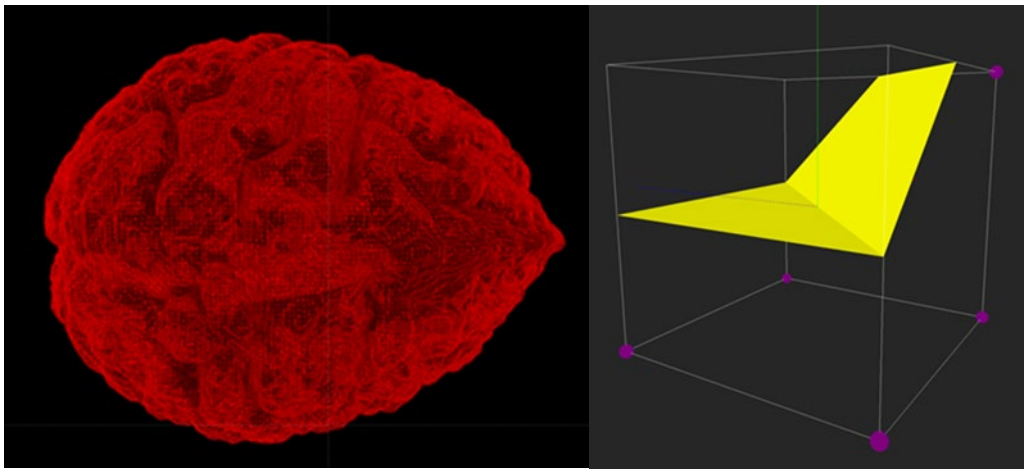

Figure 16. On the left, we show a triangulated human brain. The model is rendered in the wire-frame mode without back culling. On the right, we inspect a marching cubes case. Two polygons (three triangles) are used to cover five voxels.

6.4. Rendering on HoloLens 2

All previous examples run on HoloLens 2. In this example, we demonstrate how a mesh of a human abdomen is rendered on the HoloLens 2 immersive headset.

Figure 17 shows how a model of a human abdomen and its organs (holograms) are rendered in the solid mode on the HoloLens 2 physical device. The same rendering image is achieved using the HolographicSpace and OpenXR APIs. Figure 13 shows how the same model of the human abdomen is rendered on a standard flat display in the wire-frame mode. Note that the colors of organs are not real.

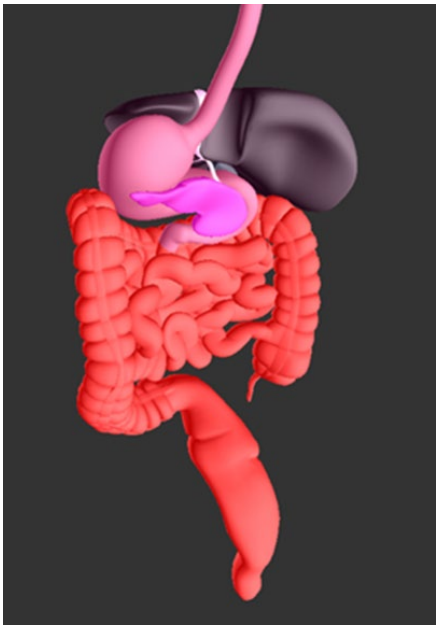

Figure 17. Rendering a model of human abdomen on HoloLens 2 in solid mode with culling back vertices. The textures are not real—they are chosen to highlight organs such as the pancreas and liver.

With reference to Figure 11, the Renderer class represents the rendering functionality for each frame on the holographic display. A Scene class manages eight scene objects that, via triangle meshes, represent organs of a human abdomen: *liver*, *stomach*, *small intestine*, *colon*, *bile duct*, *gall bladder*, *pancreas*, and *falciform ligament*. The scene object asynchronously

loads models, shaders, and pipeline resources. The Pipeline class manages the underlying graphics pipeline. It makes use of a PipelineState object to set the programmable pipeline stages (*shaders*). The CommandList class executes the pipeline, providing explicit asymmetric multiprocessing between the Qualcomm Snapdragon 850 CPU and the Adreno 630 GPU in exchanging resources and executing tasks in parallel.

The RootSignature class, via the RootParameter objects, represents the resources that can be bound to the rendering pipeline as well as the resources that can be bound to a particular programmable stage. A DescriptorHeap object holds the positions of individual organs (*holograms*) in the world space relative to the holographic camera as well as the view and projection matrices for each eye—all pipeline resources are allocated to support triple buffering.

With reference to Figure 9, physicians, radiologists, and surgeons wearing a HoloLens 2 headset can interact with the holograms by using gestures including taps and voice commands such as *Select*, which will bring the hologram of interest to the position of the gaze, which can be obtained via eye-tracking mechanisms.

7. Discussions and Conclusions

Our ideas and concepts for creating an agile real-time 3D rendering and computing platform that runs close to metal and supports surface-based rendering, direct volume rendering, and mixed (*extended*) reality on HoloLens 2 have proved to be very successful and enlightening. We described the design, architecture, and implementation choices of our platform as well as of the 3D engine in detail and compared our decisions and choices with other approaches. It is worth mentioning that the platform is currently a prototype in which only core functionalities are implemented, and the production level usage is planned through the entrepreneur company alongside the MetaverseMed project [21] in the field of medicine.

VanityX is based on network services and explicit asymmetric multiprocessing between the CPU and the GPU, explicit control of how pipeline resources are distributed and exchanged, and asynchronous loading of 3D assets. It supports two rendering paradigms: surface-based rendering, where models are constructed via triangle meshes with variable vertex types, and direct volume rendering, where models are represented via a set of scalars that can be rendered using texture-based rendering, ray-casting techniques, and transfer functions. Using isosurface generation techniques, we generate triangle meshes from volumes of scalars. It is worth mentioning that the implementation of these possibilities is not trivial and is the subject of very intensive research activity [6].

VanityX provides an object-oriented programming model that hides the underlying DirectX 12 COM objects and makes use of two programming languages that are close to hardware: C++ and HLSL. We support two platforms: desktops with flat screens with high resolutions and mixed reality using the HoloLens2 immersive headset. We deploy directly to the HoloLens 2 and therefore allow for standard software practices such as debugging and profiling the source code. Since our findings and results have opened new questions and identified new research directions, they strongly motivate further investigations.

Future research will be in two directions: (1) general 3D research and (2) investigations of using mixed reality in medicine. We will further investigate techniques and algorithms to support 3D rendering and the use of parallel computing techniques on both CPUs and GPUs, such as implementing rendering on multiple GPUs and implementing various polygonization techniques on GPUs using the tessellation stage. Our future research in analyzing medical imaging will use machine learning algorithms, especially those that can recognize malignancies. We also aim to use ray-casting techniques from DirectX 12 for our volume rendering techniques.

The primary business domain for which we will continue our research in mixed reality is medicine, via project MetaverseMed. It will primarily empower medical radiologists, clinicians, and surgeons by accelerating diagnosis and improving the precision of surgical operations by projecting medical images in the form of 3D holograms on the patient's body

parts, resulting in faster and better care and outcomes. Holograms (*blend*) digital objects, such as medical images and volumes, with physical patients to look like they are a part of the patients. MetaverseMed will also, via simulated training, enable continuous learning and the sharing of experience and knowledge among healthcare workers and students.

Our most challenging investigation is the integration of VanityX holograms with the CyberKnife 7 radiation delivery system to enable medical experts, especially radiologists and surgeons, to see and manipulate medical holograms inside the patients as if they were transparent, thereby helping them improve the accuracy of surgical operations. A big research area will be telehealth to empower multidisciplinary medical teams to collaborate remotely, perform virtual cooperation and consultations, and make (*more accurate*) diagnoses and plans. However, although the system is raised and its functionality established, there are some issues that have to be appropriately evaluated and tested in real settings. One typical example is the evaluation of real-time remote 3D rendering of medical images using GPUs, well founded in [8]. Having in mind that one of the system's key advances is related to team collaboration, determining the maximum load of a specific system based on the quality of service (QoS) is scheduled as a future research task.

Author Contributions: Conceptualization, I.Z.; methodology, I.Z., M.B., V.P. and V.S.; software, I.Z.; validation, I.Z., M.B. and V.P.; formal analysis, I.Z. and V.S.; investigation, I.Z.; writing—original draft preparation, I.Z.; writing—review and editing, M.B. and V.P.; visualization, I.Z. and M.B.; supervision, V.S.; project administration, V.P.; funding acquisition, V.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Akenine-Möller, T.; Haines, E.; Hoffman, N.; Pesce, A.; Iwanicki, M.; Hillaire, S. *Real-Time Rendering*, 4th ed.; A K Peters/CRC Press: New York, NY, USA, 2018; p. 1198.
2. Preim, B.; Botha, C. *Visual Computing for Medicine*; Morgan Kaufmann: Burlington, MA, USA, 2014.
3. Engel, K.; Hadwiger, M.; Kniss, J.M.; Rezk-Salama, C. *Real-Time Volume Graphics*; Bühler, N.M.A.K., Ed.; The Eurographics Association: Goslar, Germany, 2006.
4. Wenger, R. *Isosurfaces: Geometry, Topology, and Algorithms*; CRC Press: New York, NY, USA, 2013; p. 488.
5. Rezende, L.S.O.; Sa, P.H.M.; Macedo, M.C.F.; Apolinario, A.L.; Winkler, I.; Moret S.G., M.A. Volume Rendering: An Analysis based on the HoloLens Augmented Reality Device. In Proceedings of the 2020 22nd Symposium on Virtual and Augmented Reality (SVR), Porto de Galinhas, Brazil, 7–10 November 2020; pp. 35–38. [CrossRef]
6. Weiss, S.; Chu, M.; Thuerey, N.; Westermann, R. Volumetric Isosurface Rendering with Deep Learning-Based Super-Resolution. *arXiv* **2019**, arXiv:1906.06520. [CrossRef] [PubMed]
7. Jung, H.; Jung, Y.; Kim, J. Understanding the Capabilities of the HoloLens 1 and 2 in a Mixed Reality Environment for Direct Volume Rendering with a Ray-casting Algorithm. In Proceedings of the 2022 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW), Christchurch, New Zealand, 12–16 March 2022; pp. 698–699. [CrossRef]
8. Coutinho, E.A.G.; Carvalho, B.M. Evaluation of Real-Time Remote 3D Rendering of Medical Images using GPUs. In Proceedings of the 2020 IEEE 33rd International Symposium on Computer-Based Medical Systems (CBMS), Rochester, MN, USA, 28–30 July 2020; pp. 19–24. [CrossRef]
9. Sunderam, V.; Geist, A.; Dongarra, J.; Mancheck, R. The PVM concurrent computing system: Evolution, experiences, and trends. *Parellel Comput.* **1994**, *20*, 531–545. [CrossRef]
10. Zoraja, I.; Hermann, H.; Sunderam, V. SCIPVM: Parallel distributed computing on SCI workstation clusters. *Concurr. Pract. Exp.* **1999**, *11*, 121–138. [CrossRef]
11. Microsoft. Important Changes from Direct3D 11 to Direct3D 12. 2021. Available online: https://docs.microsoft.com/en-us/windows/win32/direct3d12/important-changes-from-directx-11-to-directx-12 (accessed on 16 March 2023).
12. NVIDIA. *Multi-Adapter Support in DirectX* 12*; NVIDIA Corporation: Santa Clara, CA, USA, 2017.
13. Microsoft. Direct3D 12 Graphics. 2021. Available online: https://docs.microsoft.com/en-us/windows/win32/direct3d12/direct3d-12-graphics (accessed on 16 March 2023).

14. Kronos Group. OpenGL: The Industry's Foundation for High Performance Graphics. 2022. Available online: https://www.khronos.org/opengl/ (accessed on 16 March 2023).
15. NVIDIA. GEFORCE RTX 30 SERIES. Available online: https://www.nvidia.com/en-eu/geforce/graphics-cards/30-series/ (accessed on 16 March 2023).
16. AMD. AMD Radeon™ Graphics. 28 August 2022. Available online: https://www.amd.com/en/graphics/radeon-rx-graphics (accessed on 16 March 2023).
17. Unity. Unity for all. Unity Coorporation, 28 August 2022. Available online: https://unity.com/ (accessed on 16 March 2023).
18. Epic Games. Unreal Engine. Available online: https://www.unrealengine.com/en-US/ (accessed on 16 March 2023).
19. Zoraja, I.; Schwarz, D. MetaverseMed: A Platform for Developing Medical XR Solutions. ZORAJA Consulting, August 2022. Available online: https://www.linkedin.com/posts/activity-6931608664935931904-FZNW?utm_source=share&utm_medium=member_desktop (accessed on 16 March 2023).
20. Microsoft Coorporation. Microsoft Hololens 2. Available online: https://www.microsoft.com/en-us/hololens (accessed on 16 March 2023).
21. Liberatore, M.J.; Wagner, W.P. Virtual, mixed, and augmented reality: A systematic review for immersive systems research. *Virtual Real.* **2021**, *25*, 773–799. [CrossRef]
22. Fuchs, K.; Haldimann, M.; Grundmann, T.; Fleisch, E. Supporting food choices in the Internet of People: Automatic detection of diet-related activities and display of real-time interventions via mixed reality headsets. *Future Gener. Comput. Syst.* **2020**, *113*, 343–362. [CrossRef]
23. Gregory, J. *Game Engine Architecture*, 3rd ed.; A K Peters/CRC Press: New York, NY, USA, 2018.
24. Microsoft Coorporation. Pipelines and Shaders with Direct3D 12. 2022. Available online: https://docs.microsoft.com/en-us/windows/win32/direct3d12/pipelines-and-shaders-with-directx-12 (accessed on 22 June 2022).
25. HigMicrosoft Coorporation. High-Level Shader Language (HLSL). Available online: https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl (accessed on 16 March 2023).
26. Microsoft Coorporation. DXGI. 25 May 2021. Available online: https://docs.microsoft.com/en-us/windows/win32/direct3ddxgi/dx-graphics-dxgi (accessed on 16 March 2023).
27. Microsoft Coorporation. DirectXMath. 30 December 2021. Available online: https://docs.microsoft.com/en-us/windows/win32/dxmath/directxmath-portal (accessed on 16 March 2023).
28. Zoraja, I. *Online Monitoring in Software DSM Systems*; Shaker Verlag: Aeachen, Germany, 2000.
29. Microsoft Coorporation. Universal Windows Platform Documentation. Available online: https://docs.microsoft.com/en-us/windows/uwp/ (accessed on 16 March 2023).
30. DICOM Standard. 9 June 2022. Available online: https://www.dicomstandard.org/current (accessed on 16 March 2023).
31. Terry, E. Silicon at the Heart of HoloLens 2. In Proceedings of the IEEE Hot Chips Symposium (HCS), Cupertino, CA, USA, 18–20 August 2019.
32. Kipman, A.; Pollefeys, M. HoloLens 2: Unpacked. ETH Zürich, 3 October 2019. Available online: https://video.ethz.ch/speakers/global-lecture/2019/61d6198e-bf0d-4970-962a-a56e70482fce.html (accessed on 16 March 2023).
33. Pollefeys, M. Holograms, Spatial Anchors and the Future of Computer Vision with Dr. Marc Pollefeys. MIcrosoft, 10 April 2019. Available online: https://www.microsoft.com/en-us/research/podcast/holograms-spatial-anchors-and-the-future-of-computer-vision-with-dr-marc-pollefeys/ (accessed on 16 March 2023).
34. Microsoft Corporation. Getting a HolographicSpace. 20 January 2021. Available online: https://docs.microsoft.com/en-us/windows/mixed-reality/develop/native/getting-a-holographicspace (accessed on 16 March 2023).
35. KHRONOS Group. OpenXR. Available online: https://www.khronos.org/openxr (accessed on 16 March 2023).
36. Microsoft Corporation. What is Mixed Reality Toolkit 2? 1 July 2022. Available online: https://docs.microsoft.com/en-us/windows/mixed-reality/mrtk-unity/mrtk2/?view=mrtkunity-2022-05 (accessed on 12 July 2022).
37. Schild, J.; Misztal, S.; Roth, B.; Flock, L.; Luiz, T.; Lerner, D.; Herkersdorf, M.; Weaner, K.; Neuberaer, M.; Franke, A.; et al. Applying Multi-User Virtual Reality to Collaborative Medical Training. In Proceedings of the 2018 IEEE Conference on Virtual Reality and 3D User Interfaces (VR), Tuebingen/Reutlingen, Germany, 18–22 March 2018.
38. CyberKnife. What is the CyberKnife®System? CyberKnife. Available online: https://cyberknife.com/cyberknife-how-it-works/ (accessed on 10 July 2022).
39. Microsoft Corporation. Windows.Graphics.Holographic Namespace. Available online: https://docs.microsoft.com/en-us/uwp/api/windows.graphics.holographic?view=winrt-22000 (accessed on 10 July 2022).
40. Microsoft Corporation. Windows. Perception Namespace. Available online: https://docs.microsoft.com/en-us/uwp/api/windows.perception?view=winrt-22621 (accessed on 10 July 2022).
41. Microsoft Corporation. Using the Windows Mixed Reality simulator. 19 October 2021. Available online: https://docs.microsoft.com/en-us/windows/mixed-reality/develop/advanced-concepts/using-the-windows-mixed-reality-simulator (accessed on 11 July 2022).
42. Microsoft Corporation. Using the HoloLens Emulator. 25 May 2022. Available online: https://docs.microsoft.com/en-us/windows/mixed-reality/develop/advanced-concepts/using-the-hololens-emulator (accessed on 8 July 2022).
43. Anđelinović, Š.; Zoraja, I.; Tadić, I. Interaktivno Virtualno Svetište. In Proceedings of the Međunarodni Simpozij o Vodnjanskim Relikvijama, Vodnjan, Croatia, 25 June 2018.

44. Wang, G.; Ye, Y.; Müller, K.; Fessler, J.A. Image Reconstruction is a New Frontier of Machine Learning. *IEEE Trans. Med. Imag.* **2018**, *37*, 1289–1296. [CrossRef] [PubMed]
45. Sirotković, J.; Dujmić, H.; Papić, V. Image segmentation based on complexity mining and mean-shift algorithm. In Proceedings of the 9th IEEE Symposium on Computers and Communications, Funchal, Portugal, 23–26 June 2014.
46. Todorić, I.; Kuzmanić, A.; Bonković, M. Image-based 3D Reconstruction: Exploring Workflow with Open-source Software. In Proceedings of the SoftCOM 2018, Split, Croatia, 13–15 September 2018.
47. Häne, C.; Zach, C.; Cohen, A.; Pollefeys, M. Dense Semantic 3D Reconstruction. *IEEE Trans. Pattern Anal. Mach. Intell.* **2017**, *39*, 1730–1743. [CrossRef] [PubMed]
48. Alexa, M. Conforming Weighted Delaunay Triangulations. *ACM Trans. Graph.* **2020**, *39*, 1–16. [CrossRef]
49. Lorensen, W.E.; Cline, H.E. Marching cubes: A high resolution 3D surface construction algorithm. *ACM SIGGRAPH Comput. Graph.* **1987**, *21*, 163–169. [CrossRef]