frontiers in Virtual Reality

ORIGINAL RESEARCH
published: 16 April 2021
doi: 10.3389/frvir.2021.640470

UNREALHAPTICS: Plugins for Advanced VR Interactions in Modern Game Engines

Janis Rosskamp, Hermann Meißenhelter*, Rene Weller*, Marc O. Rüdel, Johannes Ganser and Gabriel Zachmann**

Computer Graphics and Virtual Reality, Faculty 03: Mathematics/Computer Science, University of Bremen, Bremen, Germany

OPEN ACCESS

Edited by:
Mar Gonzalez-Franco,
Microsoft Research, United States

Reviewed by:
Oyewole Oyekoya,
Hunter College (CUNY), United States
Wolfgang Stuerzlinger,
Simon Fraser University, Canada

****Correspondence:***
Janis Rosskamp
j.rosskamp@cs.uni-bremen.de
Hermann Meißenhelter
hmeiss51@cs.uni-bremen.de
Rene Weller
weller@cs.uni-bremen.de
Gabriel Zachmann
zach@cs.uni-bremen.de

Specialty section:
This article was submitted to Technologies for VR, a section of the journal Frontiers in Virtual Reality

Received: *11 December 2020*
Accepted: *16 March 2021*
Published: *16 April 2021*

Citation:
Rosskamp J, Meißenhelter H, Weller R, Rüdel MO, Ganser J and Zachmann G (2021) UNREALHAPTICS: Plugins for Advanced VR Interactions in Modern Game Engines. Front. Virtual Real. 2:640470. doi: 10.3389/frvir.2021.640470

We present UNREALHAPTICS, a plugin-architecture that enables advanced virtual reality (VR) interactions, such as haptics or grasping in modern game engines. The core is a combination of a state-of-the-art collision detection library with support for very fast and stable force and torque computations and a general device plugin for communication with different input/output hardware devices, such as haptic devices or Cybergloves. Our modular and lightweight architecture makes it easy for other researchers to adapt our plugins to their requirements. We prove the versatility of our plugin architecture by providing two use cases implemented in the Unreal Engine 4 (UE4). In the first use case, we have tested our plugin with a haptic device in different test scenes. For the second use case, we show a virtual hand grasping an object with precise collision detection and handling multiple contacts. We have evaluated the performance in our use cases. The results show that our plugin easily meets the requirements of stable force rendering at 1 kHz for haptic rendering even in highly non-convex scenes, and it can handle the complex contact scenarios of virtual grasping.

Keywords: virtual reality, unreal engine, haptic feedback, grasping, plugin architecture, contact point, collision detection

1. INTRODUCTION

With the rise of affordable consumer devices, such as the Oculus Rift or the HTC Vive, there has been a large increase in interest and development in the area of virtual reality (VR). The new display and tracking technologies of these devices enable high fidelity graphics rendering and natural interaction with virtual environments. Modern game engines like Unreal or Unity have simplified the development of VR applications dramatically. They almost hide the technological background from the content creation process so that today, everyone can click their way to their own VR application in a few minutes. However, consumer VR devices primarily focus on outputting information to the two main human senses: seeing and hearing. Also, game engines are mainly limited to visual and audio output. Inputs processed in game engines are typically: key presses, mouse clicks, or mouse movement, controller button presses, and joystick movement. The sense of touch and a variety of untypical input devices are widely neglected.

For instance, the lack of haptic feedback can disturb the immersion in virtual environments significantly. Moreover, the concentration on visual feedback excludes a large number of people from the content created with the game engines: those who cannot see this content, i.e., blind and visually impaired people. Another important interaction technique in VR is to use our most versatile interaction tool directly, the human hand, to perform, e.g., natural grasping interactions. It can help to train (Gomes de Sá and Zachmann, 1999) and inspect (Moehring and Froehlich, 2011) objects in virtual environments more accurately and naturally. Moreover, a common way to train a robot to perform certain tasks is to apply human example grasps inside a virtual environment. Usually, modern game engines lack such fine detailed human interactions, like grasping (Lin et al., 2016).

The main reasons why such advanced input methods are widely neglected in the context of games are that haptic devices and sophisticated input devices for natural interaction methods like Cybergloves are still comparatively bulky, expensive, and do not support plug and play.

Moreover, they differ in several properties from typical input devices for games, e.g., update-rate, latency, accuracy, and resolution. The update-rate is an important property for fine interaction. It describes how many measurements are made per second and is coupled with the application and physics. Although many game engines have a built-in physics engine, they are most usually limited to simple convex shapes. They hardly deliver the complex contact information necessary to handle multiple simultaneous contacts in a stable way that typically appears during grasping. Moreover, the built-in physics engines are usually relatively slow: for the visual rendering loop it is sufficient to provide 60–120 frames per second (FPS) to guarantee smooth visual feedback. Our sense of touch is much more sensitive with respect to the temporal resolution. Here, a frequency of preferably 1,000 Hz is required to provide acceptable force feedback. It is required to decouple the physically-based simulation from the visual rendering path to reach those update rates.

In this paper, we present UNREALHAPTICS a plugin system to enable applications with specialized input devices and the demand for fast and accurate force calculations, e.g., high-fidelity haptic rendering, in a modern game engine. Following the idea of decoupling the simulation part from the core game engine, UNREALHAPTICS consists of three individual plugins:

- A plugin that we call DEVIO: It is used to implement the communication with the VR hardware devices.
- The computational bottleneck during the physically-based simulation is the collision detection. Our plugin called COLLETTE builds a bridge to an external collision detection library that is fast enough for high update rates.
- Finally, FORCECOMP computes the appropriate forces and torques from the collision information.

This modular structure of UNREALHAPTICS allows other researchers to easily replace individual parts, e.g., the force computation or the collision detection, to fit their individual needs. We have integrated UNREALHAPTICS into the Unreal Engine 4 (UE4), but the basic concept is also valid for other game engines. We use a fast, lightweight, and highly maintainable and adjustable event system to handle the communication in UNREALHAPTICS.

In this paper, we will discuss two example applications, which are using UNREALHAPTICS. While the first use case focuses on haptic rendering, our second use case shows how individual components of UNREALHAPTICS can be exchanged for other non-haptic applications like grasping in VR. For the collision detection we use the state-of-the-art collision detection library CollDet (Zachmann, 2001) that supports complexity-independent volumetric collision detection at haptic rates. Our force calculation relies on a penalty-based approach with both 3- and 6-degree-of-freedom (DOF) force and torque computations. Our results show that UNREALHAPTICS is able to compute stable forces and torques for different 3- and 6-DOF devices in Unreal at haptic rates.

2. RELATED WORK

In section 4, we present two applications using our plugins. The first is using haptic feedback, and the second demonstrates usage for grasping. We will discuss haptic and then grasping related work here.

Game engines enable the rapid development with high-end graphics and easy extension to VR to a broad pool of developers. Hence, they are usually the first choice when designing demanding 3D virtual environments. Obviously, this is also true for haptic applications. Consequently, there exist many (research) projects that already integrated haptics into such game engines, e.g., Morris et al. (2004), Andrews et al. (2006), and de Pedro et al. (2016) to name but a few. However, they usually have spent much time developing single-use approaches that are hardly generalizable and thus, not applicable to other programs.

Only a very few approaches provide comfortable interfaces for the integration of haptics into modern game engines. Kollasch (2017) and User ZeonmkII (2016) provide plugins that serve as interfaces to the *3D Systems Touch* (formerly *SensAble PHANToM Omni*)[1] *via* the *OpenHaptics* library (3D Systems, 2018). OpenHaptics is a proprietary library that is specific to 3D Systems' devices, which means that other devices cannot be used with these plugins. Another example is a plugin for the PHANToM device presented in The Glasgow School of Art (2014), also based on the OpenHaptics library. While these plugins provide communication with some haptic devices, they do not provide a framework for a wide variety of input devices. Additionally, they are not integrated into the game engine and are missing vital elements for haptics like fast collision detection. Physics engines are often deeply integrated into game engines, and the rendering frame rate is coupled with the physics calculation. There seems to be no research on replacing the in-build physics engine in game engines like Unreal and Unity. We have only found some experimental showcases but no real project or plugin. In the case of the Unreal Engine, Steve Streeting, an independent game developer, described in his blog[2], how he has integrated bullet physics on top of PhysX. This might use unnecessary performance since PhysX is still active, and unfortunately, bullet physics does not support haptics.

Outside the context of game engines, there are a number of libraries that provide force calculations for haptic devices.

[1]Phantom, O. Sensable Technologies, Inc. Available online at: http://www.sensable.com (accessed November 11, 2020).

[2]Steeve Streeting. Available online at: https://www.stevestreeting.com/2020/07/26/using-bullet-for-physics-in-ue4 (accessed November 11, 2020).

A general overview is given in Kadleček and Kmoch (2011). One example is the CHAI3D library (CHAI3D, 2016b). It is an open-source library written in C++ that supports a variety of devices by different vendors. It offers a common interface for all devices that can be extended to implement custom device support. For its haptic rendering, CHAI3D accelerates the collision detection with mesh objects using an axis-aligned bounding box (AABB) hierarchy. The force rendering is based on a finger-proxy algorithm. The device position is proxied by a second virtual position that tries to track the device position. When the device position enters a mesh, the proxy will stay on the surface of the mesh. The proxy tries to minimize the distance to the device position locally by sliding along the surface. Finally, the forces are computed by exerting a spring force between the two points (CHAI3D, 2016a). Due to the simplicity of the method it only returns 3-DOF force feedback, even though the library generally allows for also passing torques and grip forces to devices. Nevertheless, we are using CHAI3D in our use case, but only for the communication with haptic devices. A comparable, slightly newer library is the H3DAPI library (H3DAPI, 2019). Same as CHAI3D, it is extensible in both the device and algorithm domain. However, by default, H3DAPI supports fewer devices and likewise does not provide 6-DOF force feedback. A general haptic toolkit with a focus on web development was presented by Ruffaldi et al. (2006). It is based on the eXtreme Virtual Reality (XVR) engine, utilizing the CHAI3D library to allow rapid application development independent from the specific haptic interface.

All approaches mentioned above are limited to 3-DOF haptic rendering and do not support 6-DOF rendering. Sagardia et al. (2014) present an extension to the *Bullet* physics engine for faster collision detection and force computation. Their algorithm is based on the Voxmap-Pointshell algorithm (McNeely et al., 1999). Objects are encoded both in a voxmap that stores distances to the closest points of the object as well as point-shells on the object surface that are clustered to generate optimally wrapped sphere trees. The penetration depth from the voxmap is then used to calculate the forces and torques. In contrast to Bullet's build-in algorithms, this approach offers full 6-DOF haptic rendering for complex scenes. However, the Voxmap-Pointshell algorithm is very memory-intensive and susceptible to noise (Weller et al., 2010).

Our second application is grasping. There are two approaches for grasping, one is based on gesture and the other is physics-based. Gesture grasps can achieve real-time computations but lack natural interaction because they are limited to two states: the grasp and release of the object. The fingers might also penetrate the object, or there is even no contact with the object while grasping it. The physics-based approaches have a trade-off between accuracy and interactivity since they are very computationally intensive. Such a physics-based approach was presented in Verschoor et al. (2018), with a library called CLAP, that was integrated into the Unreal Engine. The hand is modeled as a soft body, whereas the object is rigid. The overall simulation reached real-time, but for natural haptic feedback, this is not sufficient. A hybrid approach was presented in Liu et al. (2019), where authors introduced a caging-based system to find a better balance between realism and performance. The hand was modeled with cylinders, and a grasp is triggered when the center of the collision contact points lies within the grasped object. Moreover, they showed a glove that has a Vive Tracker, a network of IMUs, and some vibration motors. They have integrated this system in UE4 and achieved stable grasps in real-time. The performance was evaluated only qualitatively by grasping different kinds of objects.

In the following, we will give a short overview of the structure of this paper. We will first show the UNREALHAPTICS plugin architecture by discussing the three main plugins and their communication interface in section 3. After that, we show an UE4 specific implementation. We will then present two example applications where the plugin is used. The first application (section 4.1) is haptic rendering, demonstrating a performance with haptic rates. The second use case (section 4.2) shows a grasping application with detailed collisions.

3. UNREALHAPTICS

Modern game engines support the most common devices, such as joysticks or head-mounted displays (HMDs), making them easy to set up. More specialized input, like haptic devices or elaborate hand tracking, has to be set up manually. Additionally, these applications are often in need of custom physics and high-performance collision detection. To fill that gap and provide an easy-to-use, adjustable, and generalizable framework, we have developed UNREALHAPTICS. Our software can be used in games, research, or business-related contexts, either whole or in parts. We developed our system in the Unreal Engine because of the following reasons:

- It is one of the most popular game engines with a large community, regular updates, and good documentation.
- It is free to use in most cases, especially in a research context where it is already heavily used (Mól et al., 2008; Reinschluessel et al., 2017).
- It is fully open-source, thus can be examined and adapted.
- It offers programmers access on the source code level while game designers can use a comfortable visual editor in combination with a visual scripting system called *Blueprints*. Thus, it combines the advantages of open class libraries and extensible IDEs.
- It is extendable *via* plugins.
- It is built on C++, which makes it easy to integrate external C++-libraries. This is convenient because C++ is still the first choice for high-performance libraries, e.g., haptic rendering.

Figure 1 presents the previous state before our plugins: on the one side, there are different haptic devices available with their libraries. On the other side, there is the game engine in which we want to integrate the devices. To interact with the virtual environment using input, such as haptic devices, we need (i) communication with the device, (ii) fast collision detection, and (iii) stable force computation. We solve these three major challenges by using a modular design to fulfill our goal of a flexible and adjustable system. Each module handles one of these

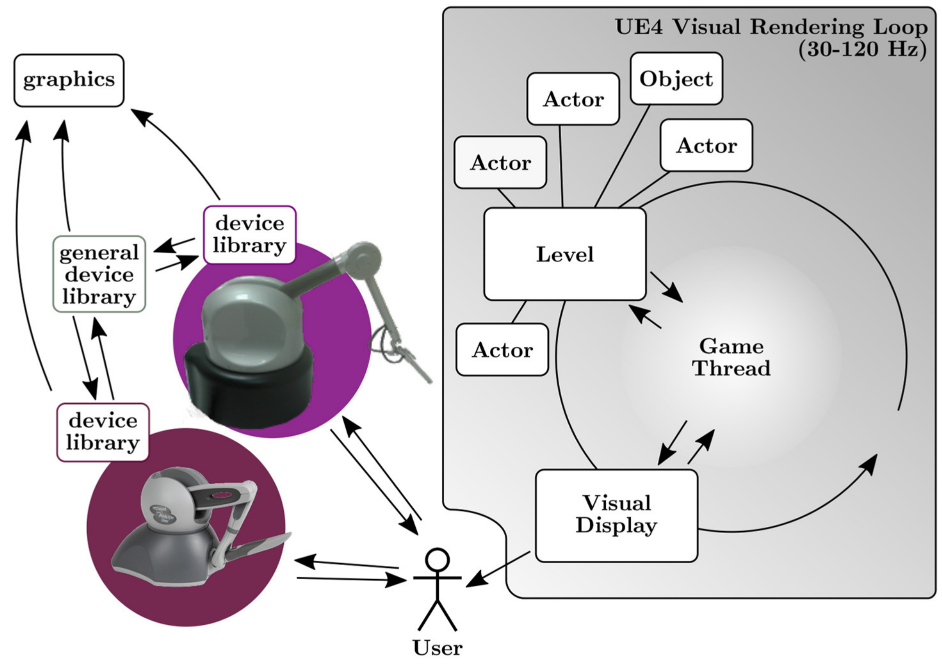


FIGURE 1 | A typical haptic integration without UNREALHAPTICS. **(Left)** Different haptic devices available with their libraries. **(Right)** Scheme of UE4, which we want to integrate the devices with.

tasks. Of course, this structure is not only valid for haptics but interaction with other input as well. Our plugins are responsible for the following tasks:

- A plugin called DEVIO that handles the communication with the input device by adding a general layer to initialize devices and to receive and send data to the device during runtime.
- A plugin called COLLETTE that communicates with an (external) collision detection library. Initially, it passes geometric objects from Unreal to the collision library (to enable it to potentially compute acceleration data structures etc.). During runtime, it updates the transformation matrices of the objects and collects collision information.
- FORCECOMP, a force rendering plugin that receives collision information and computes forces and torques. The force calculation is closely related to the collision detection method because it depends on the provided collision information. However, we decided to separate the force and torque computation from the actual collision detection into separate plugins because this allows an easy replacement, e.g., if the simulation is switched from penalty-based to impulse-based. While its main task is to compute forces, it can also be used to process collision data in general, which will be shown in section 4.2.3.

The list of plugins already suggest that communication plays an important role in the design of our plugin system. Hence, we will start with a short description on this topic before we detail the implementations of the individual plugins.

3.1. Unreal Engine Recap

Unreal Engine 4 is a game engine that comprises the engine itself as well as a 3D editor to create applications using the engine. We will start with a short recap of UE4's basic concepts.

Unreal Engine 4 follows the component-based entity system design. Every object in the scene (3D objects, lights, cameras, etc.) is at its core a data-, logic-less entity (in the case of UE4 called *actors*). The different behavior between the objects stems from *components* that can be attached to these actors. For example, a `StaticMeshActor` (which represents a 3D object) has a mesh component attached, while a light source will have different components attached. These components contain the data used by UE4's internal systems to implement the behavior of the composed objects (e.g., the rendering system will use the mesh components, the physics system will use the physics components etc.).

Unreal Engine 4 allows its users to attach new components to actors in the scene graph which allows extending objects with new behavior. Furthermore, if a new class is created using UE4's C++-dialect, variables of that class can be exposed to the editor. By doing so, users have the ability to easily change values of an instance of the class from within the editor itself, which minimizes programming effort.

Unreal Engine 4 not only provides a C++ interface but also a visual programming language called *Blueprints*. *Blueprints* abstract functions and classes from the C++ interface and present them as "building blocks" that can be connected by execution lines. It serves as straightforward way to minimize programming effort and even allows people without programming experience to create game logic for their project.

When extending UE4 with custom classes, the general idea is noted in Epic Games (2020a): programmers extend the existing systems by exposing the changes *via* blueprints. These can be used by other users to create game behavior. Our plugin system follows this ideas.

Furthermore, UE4 allows developers to bundle their code as plugins in order to make the code more reusable and easier to distribute (Epic Games, 2020b). Plugins can be managed easily within the editor. All classes and blueprints are directly accessible for usage in the editor. We implemented our system as a set of three plugins to make the distribution effortless and allow the users to choose which features they need for their projects.

Finally, UE4 programs can be linked against external libraries at compile time, or dynamically loaded at runtime, similar to regular C++ applications. We are using this technique to base our plugins on already existing libraries. This ensures a time-tested and actively maintained base for our plugins.

3.2. Design of the Plugin Communication

As described above, our system consists of three individual plugins that exchange data. Hence, communication between the plugins plays an important role. Following our goal of flexibility, this communication has to meet two major requirements.

- The plugins need to communicate with each other without knowledge about the others' implementation because users of our plugins should be able to use them individually or combined. They could even be replaced by the users' own implementations. Thus, the communication has to run on an independent layer.
- Users of the plugins should be able to access the data produced by the plugins for their individual needs. This means that it must be possible to pass data outside of the plugins.

To fulfill both these requirements, we implemented a messaging approach based on *delegates*. A *delegator* is an object that represents an event in the system. The delegator can define a certain function signature by specifying parameter types. *Delegates* are functions of said signature that are bound to the delegator. The delegator can issue a broadcast which will call all bound delegates. Effectively, the delegates are functions reacting to the event represented by the delegator. A delegator can pass data to its delegates when broadcasting, completing the messaging system.

The setup of the delegates between the plugins can be handled, for example, in a custom controller class within the users' projects. We describe the implementation details for such a controller in section 3.6.

3.2.1. Our Light Delegate System

Unreal Engine 4 provides the possibility to declare different kinds of delegates out of the box. However, these delegates have a few drawbacks. Only Unreal Objects (declared with the `UOBJECT` macro etc.) can be passed with such delegates, limiting their use for more general C++ applications. They also introduce several layers of calls in the call stack since they are implemented around the reflection system of the UE4. This may influence performance when many delegates are used.

To overcome these problems we implemented our own lightweight `Delegator` class. It is a pure C++ class that can take a variable number of template arguments which represent the parameter types of its delegates. A so called *callable* can be bound with the `addDelegate(...)` function. Our solution supports all common C++-callables (free functions, member functions, lambdas, etc.). The delegates can be executed with the `broadcast()` function which will execute delegates one after another with just a single additional step in the call stack. The data are always passed around as references internally, preventing any additional copies.

3.3. Devio Plugin—Device Interface

DEVIO provides a common abstraction layer for input devices. To use a device in UNREALHAPTICS, one has to include the correct communication library and implement the abstract functions. With these functions, we gain full control of our device with either Blueprints or C++ Code. Additionally, the plugin provides bidirectional data transfer, i.e., data can be received and sent to the device.

DEVIO mainly consists of three parts: The device manager, the device thread, and the device interface. The device manager provides the user interface and is the only part used by the developer. It is represented as a UE4 actor in the scene and is used to send and receive data, e.g., positions or forces. If necessary, the execution loop of the plugin can be separated from the game thread of the UE4. This device thread allows higher framerates compared to the game thread. While this is not required for most devices, it is crucial for haptics, which can be seen in **Figure 2**. When new device data are available, a delegator-event `OnTransform` is broadcasted, which passes the data to the device manager in every tick. Users of the plugin can hook their own functions to this event, allowing them to react to the new device data. A second delegator-event `ForceOnHapticTick` is broadcasted, which allows users to hook functions like force computation into the device thread. Our own FORCECOMP plugin uses this mechanism, which is further described in section 3.6.

3.4. Collette—Collision Detection Plugin

The physics module included in UE4 has two drawbacks that makes it unsuitable for some devices:

1. It runs on the main game thread, which means it is capped at 120 FPS.
2. Objects are approximated by simple bounding volumes, which is very efficient for game scenarios but too imprecise to compute the collision data needed for precise physics computations.

In order to circumvent these problems, we bypass the physics module of the UE4.

Our COLLETTE plugin does exactly that. We do not implement a collision detection in this plugin, but provide a flexible wrapper to bind external libraries. In our use case, we show an example how to integrate the CollDet library (see

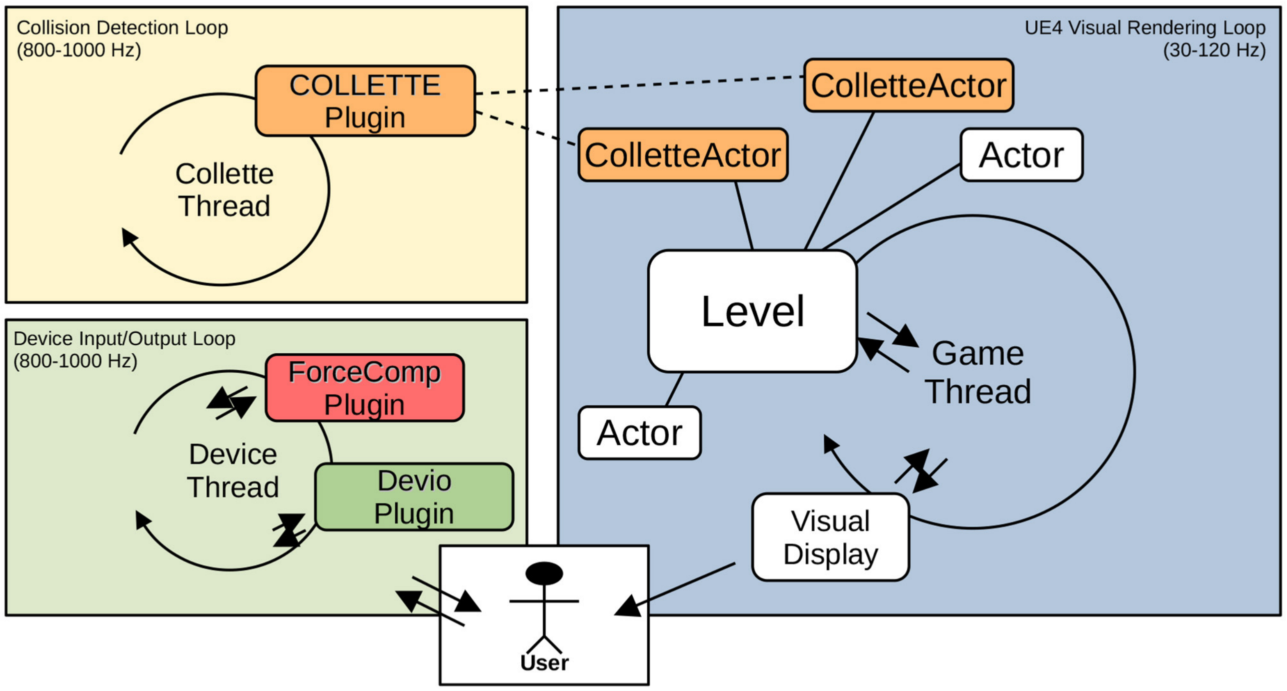
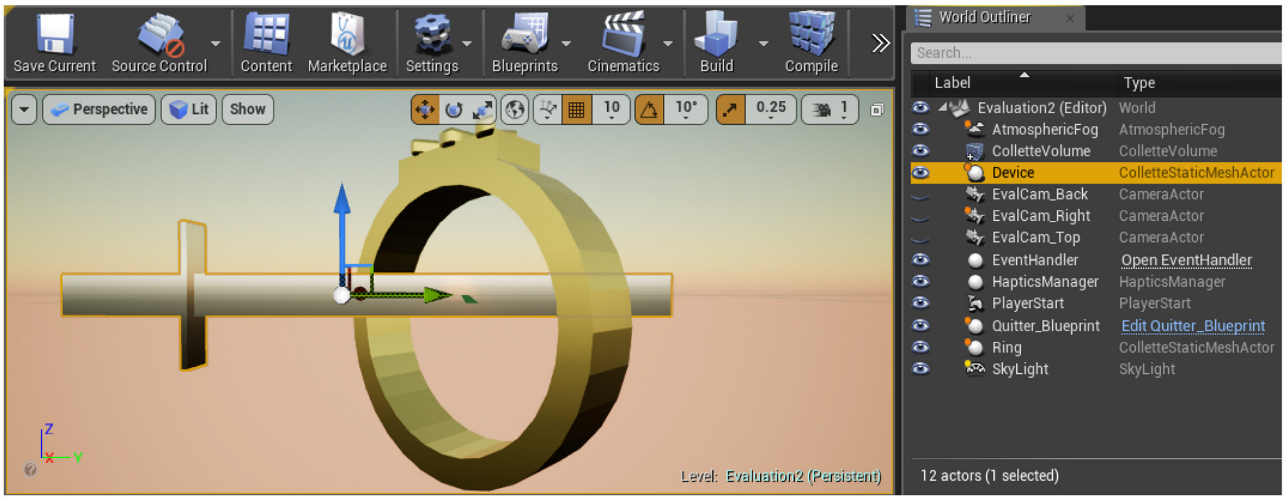


FIGURE 2 | The basic structure of our plugin system with three threads. (Right) The UE4 game thread that is responsible for the visual feedback and runs with up to 120 Hz. (Left) The haptic rendering thread and the collision detection thread. The device communication included in DEVIO and the FORCECOMP plugin run at 1,000 Hz for a stable update rate. We decided to put the collision detection in its own thread in order to not disturb the device communication, e.g., in case of deep collisions that require more computation time than 1 ms. The collidable objects in the Unreal scenegraph are represented as `ColletteStaticMeshActors` that are derived from Unreal's built in `StaticMeshActors`.



Label	Type
Evaluation2 (Editor)	World
AtmosphericFog	AtmosphericFog
ColletteVolume	ColletteVolume
Device	ColletteStaticMeshActor
EvalCam_Back	CameraActor
EvalCam_Right	CameraActor
EvalCam_Top	CameraActor
EventHandler	Open EventHandler
HapticsManager	HapticsManager
PlayerStart	PlayerStart
Quitter_Blueprint	Edit Quitter_Blueprint
Ring	ColletteStaticMeshActor
SkyLight	SkyLight

12 actors (1 selected)

FIGURE 3 | Unreal's editor view of one of our scenes. The sword is controlled by the phantom device, whereas the ring is static. Both are `ColletteStaticMeshActor`, which is visible on the right side.

section 4.1.2). Like DEVIO, COLLETTE can run in its own thread. Thus, high framerates even for complicated scenes can be achieved. The plugin uses a `ColletteStaticMeshActor` to represent collidable objectsas shown in **Figure 3**. This is an extension to UE4's `StaticMeshActor`. It supports loading additional pre-computed acceleration data structures to the actor's mesh component when the 3D asset is loaded. For instance, in our use case, we load a pre-generated sphere tree asset from the hard drive which is used for internal representation of the underlying algorithm.

The collision pipeline is represented by a `ColletteVolume`, which extends the UE4 `VolumeActor`. We decided to use a volume actor because it allows to restrict collision detection checks to defined areas in the level. To register collidable objects with the pipeline, they can be registered with an `AddCollisionWatcher(...)` blueprint function to the collision detection pipeline. The function takes references to the `ColletteVolume` as well as two `ColletteStaticMeshActor`s.

During runtime, the collision thread checks registered pairs with their current positions and orientations. If a collision is determined, the class `ColletteCallback` broadcasts an `OnCollision` delegator-event. Users of the plugin can easily hook their own functions to this event, allowing reactions to the collision. Blueprint events cannot be used here as they are also executed on the game thread and thus run at a low frequency. The event also transmits references to the pair of `ColletteStaticMeshActor`s involved in the collision, as well as the collision data generated by the underlying algorithm. This data can then be used, for example, to compute collision response forces.

3.5. ForceComp Plugin

The force calculation is implemented as a free standing function which accepts the data from two `ForceComponent`s that can be attached especially to `ColletteStaticMeshActor`s and depends on the current transform of the `ColletteStaticMeshActor`. The `ForceComponent` provides UE4 editor properties needed for the physical simulation of the forces: For instance, the mass of the objects, a scaling factor, or a damper (see section 4.1.3). We have separated the force data from the collision detection. This allows users to use the COLLETTE plugin without the force computation.

3.6. Controlling Data Flow *via* Events

We already mentioned that we use a delegate-based event system to organize the data flow between the three plugins. In order to manage the events, we use an `EventHandler` actor. This guarantees a maximum of flexibility and avoids that the plugins depend on the specific implementation. Basically, the `EventHandler` has references to all involved components and game objects like actors and events. Our `EventHandler` supports drag-and-drop in the Unreal editor window, and hence, there is no coding required to establish these references. For instance, if we want to add a collidable object in the scene, we simply have to drag a `ColletteStaticMeshActor` instance on the `EventHandler` instance in the editor window.

In addition, the `EventHandler` implements various functions that it binds to the events of the plugins during initialization. For example, it provides functions for the two most important events: the `OnTransform` event sent by the device thread and for the `OnCollision` event of the `ColletteVolume` actor. The `OnTransform` event broadcasts the position and orientation data automatically to the virtual representation, e.g., a hand. This has the same effect as if the representation would be updated directly in the device thread. Moreover, the `OnTransform` event also evokes a second delegate function from FORCECOMP that computes the collision forces based on this data. When finished, it may pass the forces back to the `DeviceManager`, which applies them to the associated device (see **Figure 4** for a simplified example).

The `OnCollision` delegator event of the `ColletteVolume` actor sends the collision data to the attached function of the `EventHandler` and finally stores it in shared variables. By doing this, the device thread will execute the delegate after it has updated the virtual tool's transform. The delegate itself reads the data from the shared variables and checks if a collision occurred. If so, the collision information is used to compute forces. The `Collette` thread is synchronized with the force computation. After a force is calculated the `Collette` thread stops waiting, reads the device data, and checks for collisions. This way the frequency of the collision thread is bounded to the device thread frequency (1,000 Hz).

With this solution, however, we keep the concrete implementations of the plugins separate from each other. **Figure 5** shows and example for the event handling between FORCECOMP and DEVIO. This modular and customizable approach guarantees a very flexible data flow between the different plugins that can be easily defined by the user within the editor. In the following sections, we want to explore UNREALHAPTICS for two concrete applications, the first being haptic rendering with a haptic device and the second grasping with a hand-tracking device.

4. APPLICATIONS

In this section, we demonstrate how UNREALHAPTICS can be used in applications. The first example focuses on haptic rendering, while the second example shows a grasping application.

4.1. Haptic Rendering

We applied UNREALHAPTICS to an application with support for haptic rendering. For haptic rendering, we use our plugin in a setup with three threads: one for the main game loop, including the visual rendering in Unreal, one for the haptic rendering that covers DEVIO and FORCECOMP and one for the collision detection. We decided to run the collision detection independently in its own thread to guarantee stable haptic rendering rates even in the case of deep interpenetrations where the collision detection could exceed the 1ms time frame. **Figure 3** shows this three-thread scenario. However, it is easy to use COLLETTE also in the haptic rendering thread—or to even use a fourth thread for FORCECOMP—by simply adjusting the configuration in the `EventHandler`. This example shows how the actual collision detection libraries, force rendering, and communication libraries can be integrated into our framework.

4.1.1. Device Communication *via* CHAI3D

We use the CHAI3D library to connect to haptic devices. As already mentioned in section 2, this library supports a wide variety of haptic devices, including the PHANToM and the *Haption Virtuose* (Haption, 2020) which we used for testing.

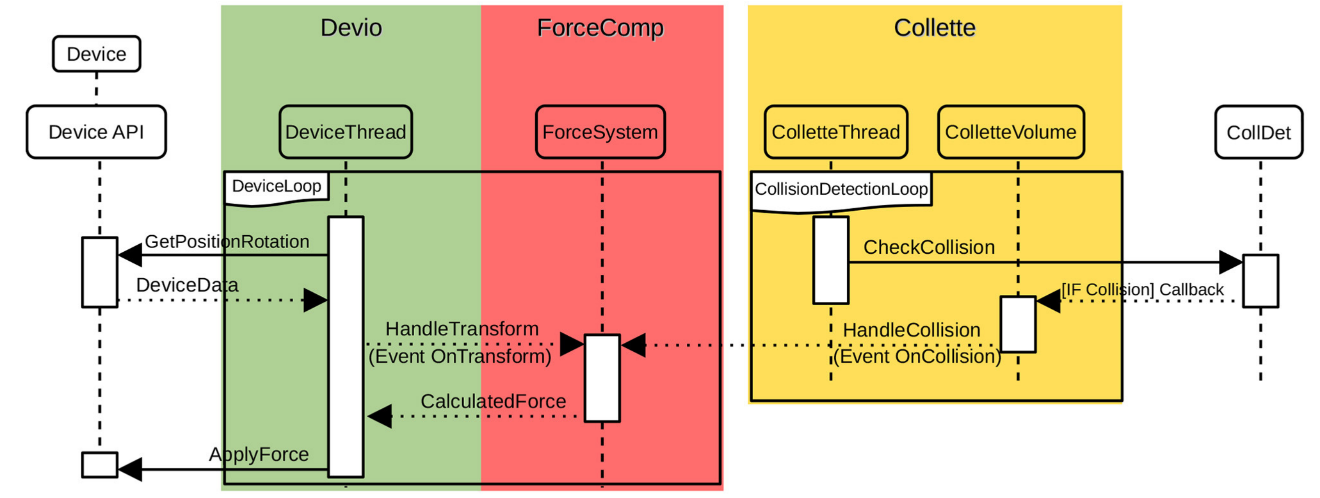


FIGURE 4 | A simplified sequence diagram of the communication of FORCECOMP, COLLETTE, and DEVIO in case of a collision: DEVIO receives the current position and orientation from the device and informs FORCECOMP *via* a `OnTransform` event. `ColletteVolume` in COLLETTE evokes an `OnCollision` event and passes the collision data to FORCECOMP. FORCECOMP computes appropriate forces and torques and passes them back to DEVIO that finally, applies them to the device. Please note, due to space constrains, we did not include transformations that are send from DEVIO to the respective `ColletteStaticMeshActors`. Moreover, we omitted the `EventHandler` in this example.

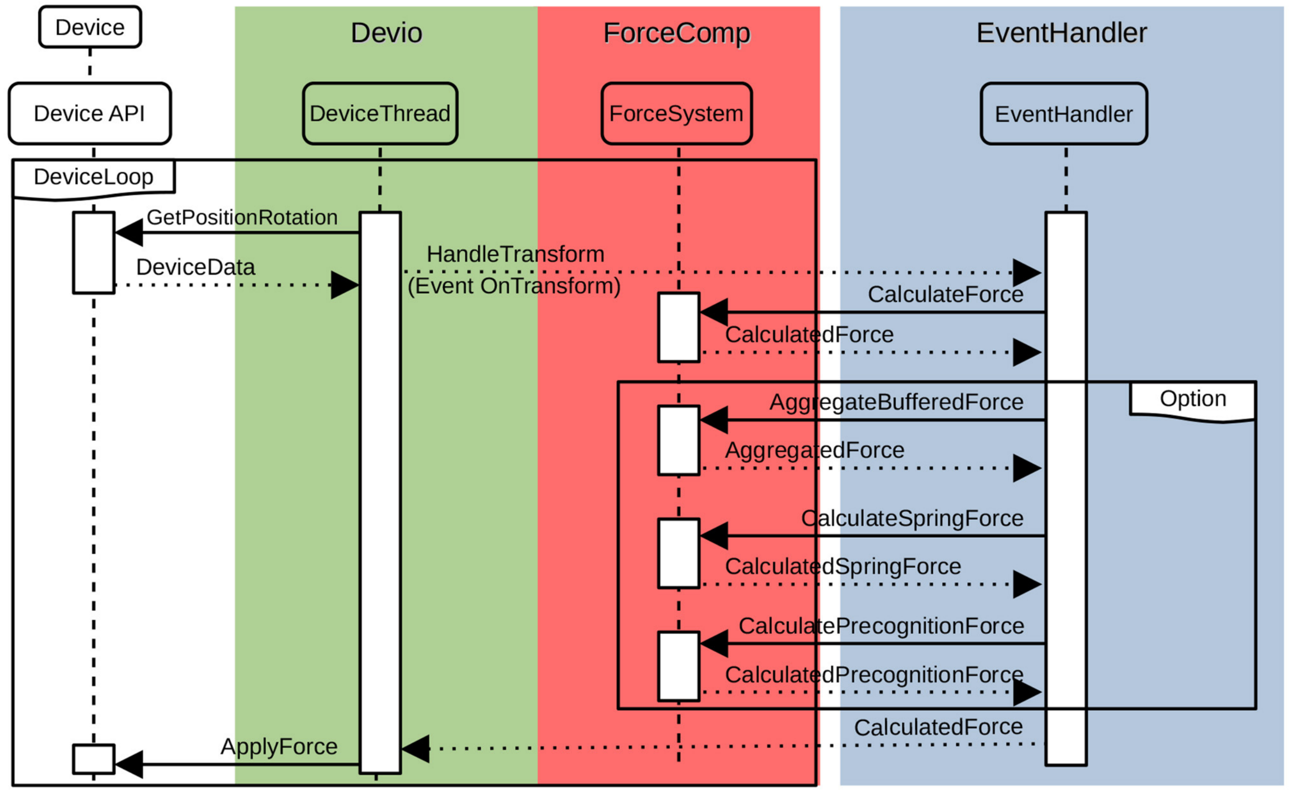


FIGURE 5 | A simplified sequence diagram of the communication of FORCECOMP, DEVIO, and our `EventHandler` that also shows the flexibility of our system. Initially, DEVIO reads the configuration from the haptic library and evokes an `OnTransform` event. This is passed to the `EventHandler` that calls the callable `HandleTransform` function that has initially registered for this event. It is easy to register more than one functions for the same event, e.g., to toggle friction or virtual coupling. The results are finally transferred back to DEVIO *via* the `EventHandler`.

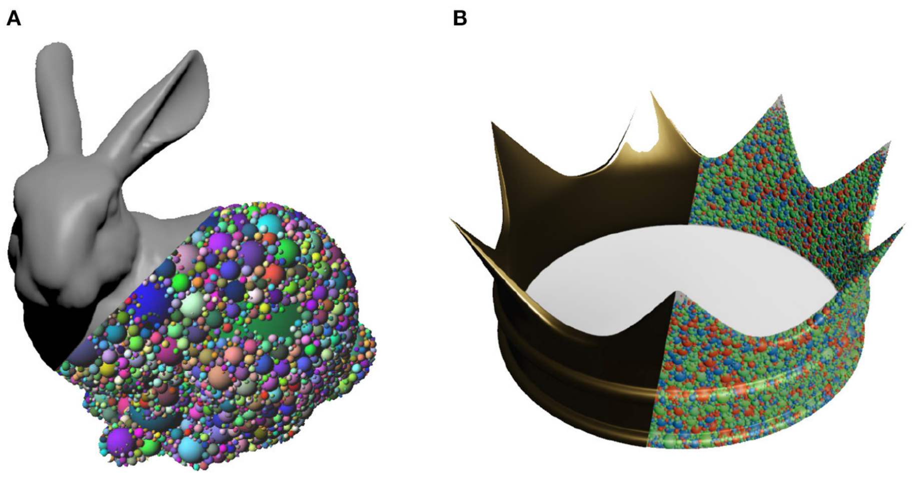


FIGURE 6 | Objects from our test scenes and their inner sphere representations: **(A)** Stanford bunny and **(B)** Crown.

DEVIO links CHAI3D as a third-party library at compile time. We primarily use CHAI3D's *Devices* module as an interface to the hardware devices, especially to set and retrieve positions and rotations. We did not use CHAI3D's force rendering algorithms as they do not support 6-DOF force calculation.

4.1.2. Collision Detection With CollDet

CollDet is a collision detection library written in C++ that implements a complete collision detection pipeline with several layers of filtering (Zachmann, 2001). This includes broad-phase collision detection algorithms like a uniform grid or convex hull pre-filtering and several narrow phase algorithms like a memory-optimized version of an AABB-tree, called Boxtree (Zachmann, 2002), and DOP-trees (Zachmann, 1998). For haptic rendering, the *Inner Sphere Trees* (ISTs) data structure fits best. Unlike other methods, ISTs define hierarchical bounding volumes of spheres *inside* the object based on a polydisperse sphere packing (see **Figure 6**). This approach is independent of the object's triangle count, and it has shown to be applicable to haptic rendering. Beyond the performance, the main advantage is the collision information provided by the ISTs: they do not simply deliver a list of overlapping triangles but give an approximation of the objects' overlap volume. This guarantees stable and continuous forces and torques (Weller et al., 2010). The source code is available under an academic-free license.

COLLETTE's `ColletteVolume` is, at its core, a wrapper around CollDet's pipeline class. Instead of adding CollDet objects to the pipeline, the plugin abstracts this process by registering the `ColletteStaticMeshActor`s with the volume. Internally, a `ColletteStaticMeshActor` is assigned a `ColID` from the CollDet pipeline through its `ColletteStaticMeshComponent`, so that each actor represents a unique object in the pipeline. When the volume moves the objects and checks for collisions in the pipeline, it passes the IDs of the respective actors to the CollDet functions that implement the collision checking. Analog to CHAI3D, COLLETTE links to the CollDet library at compile time.

4.1.3. Force Calculation

Force and torque computations for haptics usually rely on penalty-based approaches because of their performance. The actual force computation method is closely related to the collision information that is delivered from COLLETTE. In the case of the ISTs, this is a list of overlapping inner spheres for a pair of objects. In our implementation, we apply a slightly modified volumetric collision response scheme as reported by Weller and Zachmann (2009):

For an object A colliding with an object B we compute the restitution force $\vec{F}_A$ by

$$\begin{aligned}\vec{F}_A &= \sum_{j\cap i\neq\emptyset} \vec{F}_{A_i} \\ &= \sum_{j\cap i\neq\emptyset} \vec{n}_{i,j}\cdot \max\left(\mathrm{vol}_{i,j}\cdot\left(\varepsilon_c - \frac{\mathrm{vel}_{i,j}\cdot\varepsilon_d}{\mathrm{Vol}_{total}}\right),0\right)\end{aligned} \tag{1}$$

where (i,j) is a pair of colliding spheres, $\vec{n}_{i,j}$ is the collision normal, $\mathrm{vol}_{\mathrm{i,j}}$ is the overlap volume of the sphere pair, $\mathrm{Vol}_{\mathrm{total}}$ is the total overlap volume of all colliding spheres, $\mathrm{vel}_{\mathrm{i,j}}$ is the magnitude of the relative velocity at the collision center in direction of $\vec{n}_{i,j}$. Additionally, we added an empirically determined scaling factor ε_c for the forces and applied some damping with ε_d to prevent unwanted increases of forces in the system.

Only positive forces are considered preventing an increase in the overlapping volume of the objects. The total restitution force is then computed simply by summing up the restitution forces of all colliding sphere pairs.

Torques for full 6-DOF force feedback can be computed by

$$\vec{\tau}_A = \sum_{j\cap i\neq\emptyset} \left(C_{i,j} - A_m\right)\times \vec{F}_{A_i} \tag{2}$$

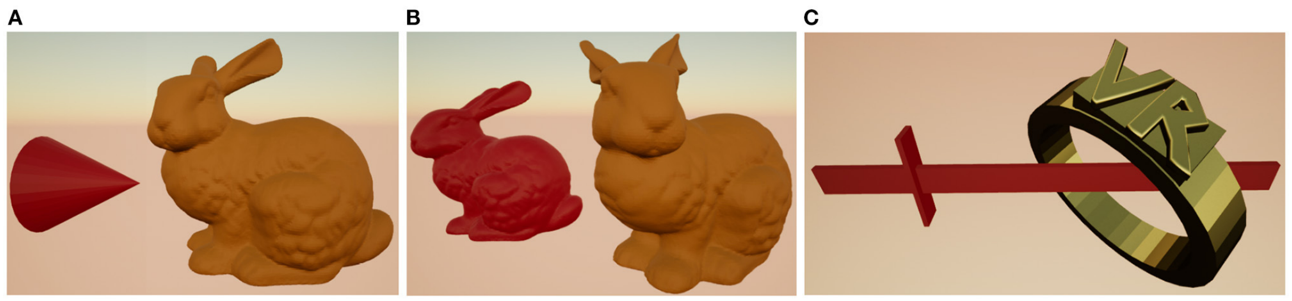


FIGURE 7 | In order to evaluate the performance of our plugins, we have used multiple complex test scenes: **(A)** Cone and bunny, **(B)** Two bunnies, and **(C)** Sword and ring. The user controls in each scene a reddish object with the phantom device to touch the other object.

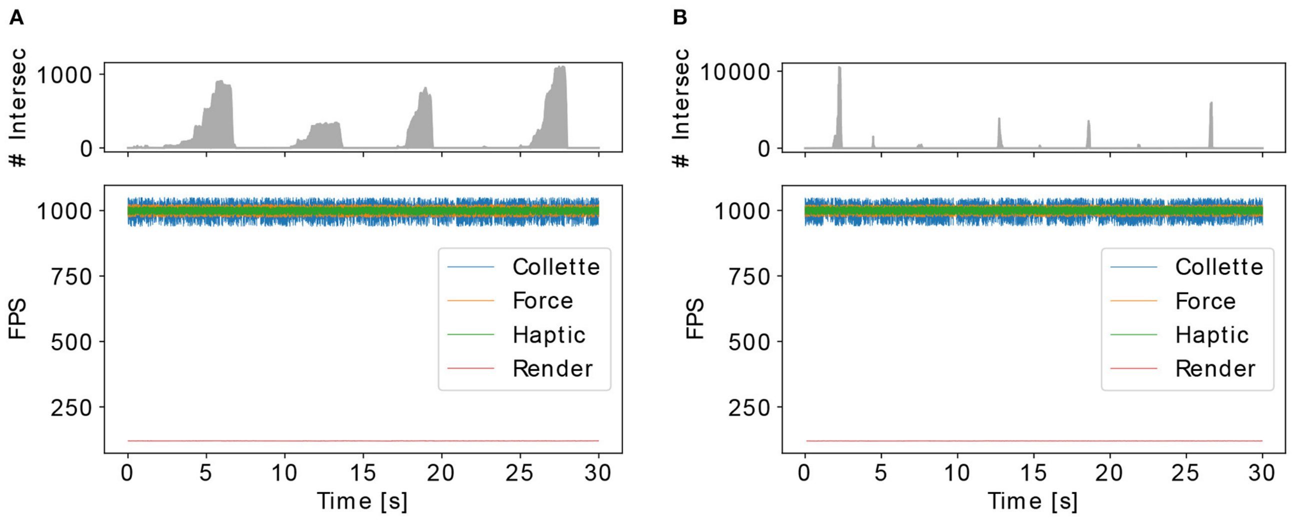


FIGURE 8 | Performance of our plugins in the cone and bunny scene. Here, we have used different sphere packing resolutions (approximation of volume) per object. Top graph shows the number of intersecting spheres. In **(A)**, we have used 10k spheres and in **(B)** we have used 100k spheres. We have achieved haptic frame rates even with large number of spheres, and also we have had a constant rendering frame rate (120 FPS).

where $C_{i,j}$ is the center of collision for sphere pair (i,j) and A_m is the center of mass of the object A. Again, the total torques of one object are computed by summing the torques of all colliding sphere pairs (Weller and Zachmann, 2009).

4.1.4. Performance

We have evaluated the performance of our implementation in UE4 on an AMD Ryzen 7 2700X (8 Cores) with 32 GB of main memory and a NVIDIA GeForce RTX 2070 running Microsoft Windows 10 Professional.

We used three different test scenes: the user explores the surface of an object (in our example, the Stanford bunny) with a Phantom device. In our test scenes, we represented the end effector with a red color (see **Figure 7**).

We achieved almost always a frequency of 850–1K Hz for the force rendering and haptic communication thread. It only dropped slightly in case of situations with a lot of intersecting pairs of spheres. The same appears for the collision detection that slightly dropped to 850 Hz in situations of heavy interpenetrations. It is similar to the results reported in Weller et al. (2010), where a simple OpenGL test scene was used. This shows that our architecture does not add significant processing overhead (see **Figures 8**, **9**).

4.2. Grasping

Due to the modular structure of UNREALHAPTICS, we can easily adapt parts of the plugin to account for different use cases. In the following, we will show how UNREALHAPTICS can be applied to a natural grasping application in VR. In this case, we change our input device to a hand tracking device, i.e., a Cyberglove, which enables natural grasping of objects in VR (**Figure 10**).

More precisely, we aim at investigating the human grasping of different object to transfer it to robots performing everyday activities. To do that, in this first step, we record sophisticated human grasping data in VR. This is done by generating heat maps during grasp experiments of the virtual objects (Tenorth et al., 2015). To record a heat map, we determine the contact points of the individual parts of the hand on object. Heat maps

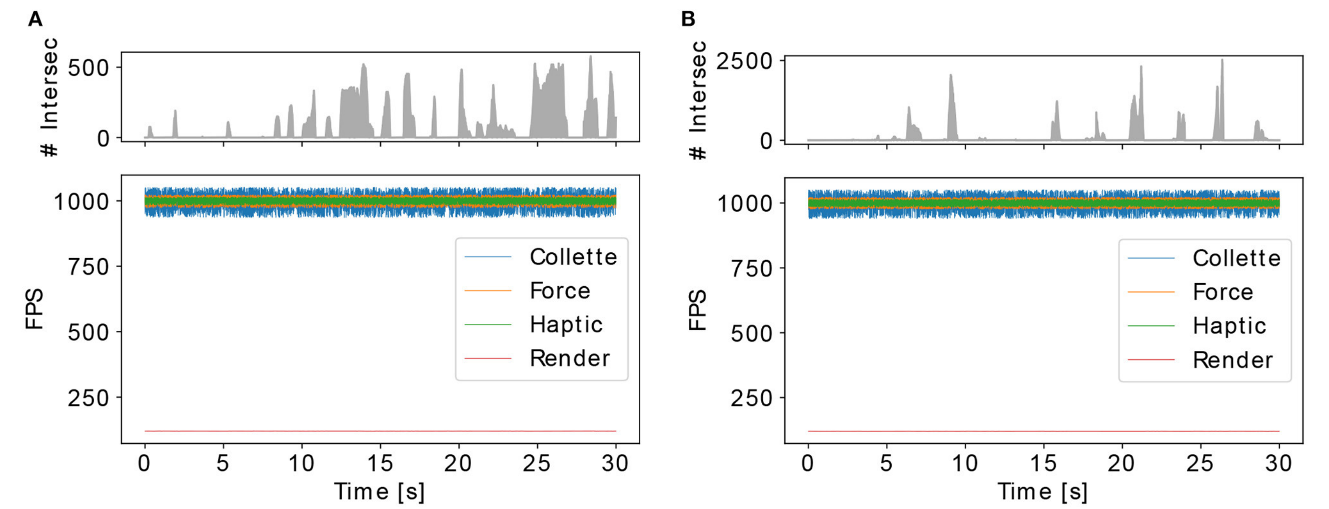


FIGURE 9 | Performance of our plugins in different scenes with 100k spheres per object. Top graph shows the number of intersecting spheres. In **(A)**, we have used two bunnies and in **(B)** a sword and ring. For these different objects in the scene, we have achieved haptic frame rates and constant 120 FPS for rendering.

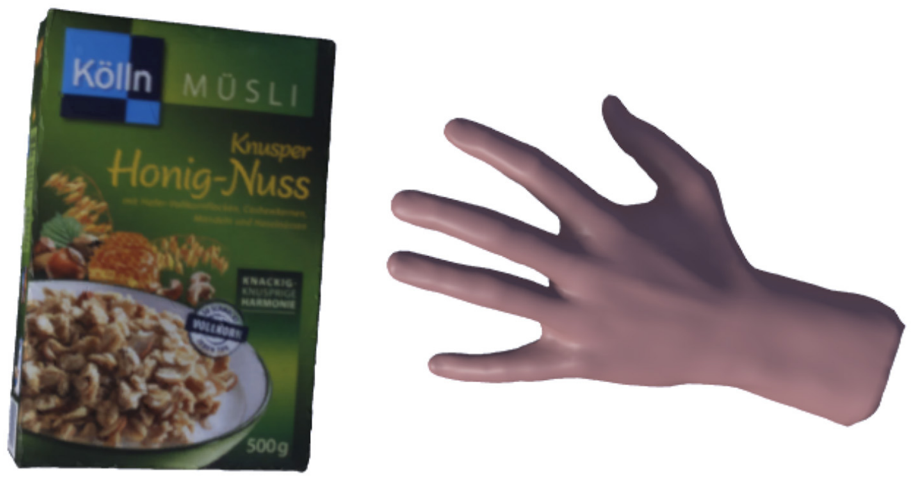


FIGURE 10 | A virtual hand is grasping an object. The hand is controlled by the user's real hand using a Cyberglove.

for a specific object show the combination of contact points from multiple grasping experiments. **Figure 11** show examples of such maps for different objects. The aggregated contact points are drawn on the texture of the object. We represent each finger with a different color, so it is easy to distinguish the grasping positions. Our generated heat maps show possible candidates for grasp points of an object. This data can be used, i.e., by robots to learn stable grasping configurations for a wide variety of objects. We will discuss the plugins in more detail in the following section.

4.2.1. Device Communication

While the CHAI3D library can be used for many different haptic devices, no such library exists for hand tracking with the many fundamentally different tracking methods, such as gloves, optical-based, or marker-based (references). For this application, we decided to use a Cyberglove, which uses bending sensors attached to a glove to track finger motion. We have written a library that connects and retrieves signals from the glove. These signals are then transformed into joint angles by a calibration step. DEVIO links this library, and by implementing the abstract functions, we can use UNREALHAPTICS with a Cyberglove as an input device.

4.2.2. Collision Detection With CollDet

Similar to our haptic application, we use the COLLETE plugin with the CollDet library to precisely detect collisions between the hand and objects. CollDet can easily handle multiple contacts from different fingers and the palm while delivering sophisticated collision information. Each graspable object is represented by a `ColleteStaticMeshActor`. For the virtual hand, 16 `ColleteStaticMeshActors` are registered in the `ColletteVolume`, three actors for each finger and one for

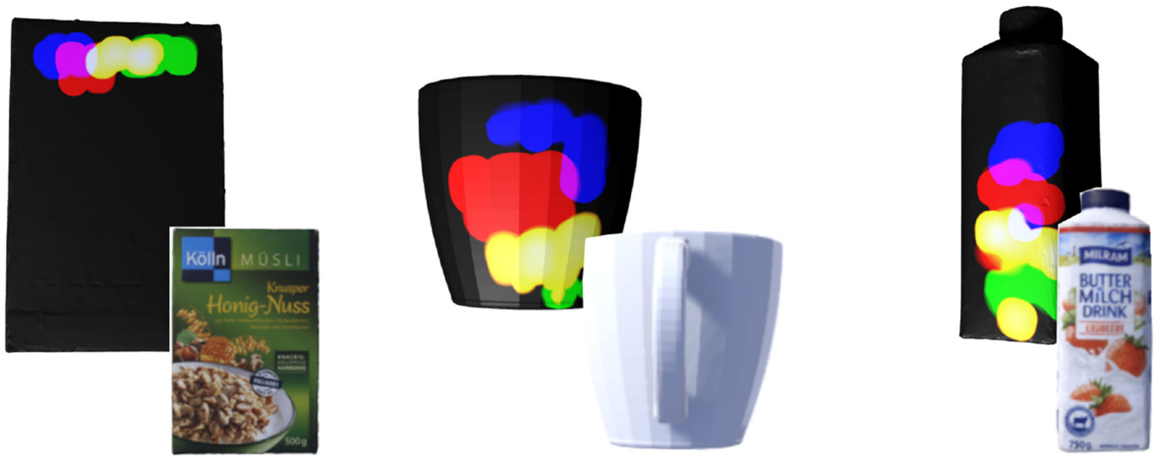


FIGURE 11 | Heatmaps for a cerial box, a cup, and a milk box. The colors denote the finger used for grasping the object: ● index finger, ● middle finger, ● ring finger, ● pinky finger, ● thumb.

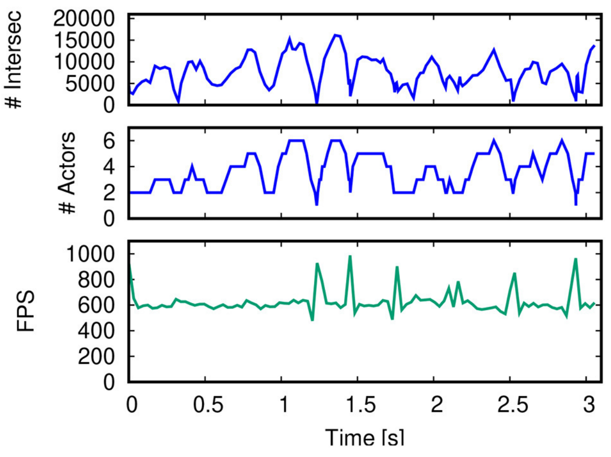


FIGURE 12 | Performance of our plugin for the grasping application with 17 interacting *ColleteStaticMeshActors*. We used multiple grasping operations and only used frames with collisions for the analysis. The top graph shows the number of intersecting spheres. The middle graph shows the number of colliding *ColleteStaticMeshActors*. In the bottom graph, the framerate is shown. The framerate never drops below 550 Hz even for complicated collisions.

the palm. This separation of actors is necessary because fingers are movable, and `ColleteStaticMeshActors` are rigid objects. At first glance, this may look complex, but in the case of collisions, we automatically know which part of the hand is in contact with the object. Moreover, we can also detect self-collisions between fingers easily. Information about the positions of the collision on the object allows both grasping and creating heat maps.

4.2.3. Force Calculation and Heat Maps

To manipulate objects in VR, knowledge of the force acting on objects from the virtual hand is essential. We can only estimate the force in our setup because our hand tracking device has no haptic feedback. Nonetheless, the force calculation introduced in section 4.1.3 using the overlapped volume can be used to approximate the force acting on objects. In this application, the main goal is the generation of heat maps. Hence, during a grasping operation, the contact points of the hand with the object are computed. We do this in the following way: If the hand is in contact with an object, the collision detection returns a list of overlapping spheres. To generate heat maps from pairs of overlapping spheres, we need to compute the corresponding points on the surface of the object. We can use Unreal's linetrace function to determine the nearest point on the surface with a raycast. Using Unreal's complex collision detection, we get a good approximation of the collisions on the surface. For the linetrace, we define as a starting point, the center of the hand's sphere and as an endpoint the center of the object's sphere.

4.2.4. Performance

We, again, tested the performance of our collision detection loop. In comparison with section 4.1.4, the number of ColldetActors increased from 2 to 17, which in principle could all collide simultaneously and increase the complexity significantly. We evaluated the performance on a machine running Windows 10 with an AMD Ryzen 9 3900X (12 cores), 16 GB of main memory, and an NVIDIA GeForce RTX 2080 SUPER. The results are shown in **Figure 12**. To focus only on the interesting case where the hand is grasping an object, we only considered frames with collisions and limited the maximal framerate to 1,000 Hz. Each of the 16 finger parts is packed with 5,000 spheres and the graspable object with 10,000 spheres. The maximum number of intersecting spheres is around 30,000, and the maximum number of colliding actors is 11, as can be seen in the upper graph. The frame rate never drops below 150 Hz, and our application is easily running in real-time.

5. CONCLUSIONS AND FUTURE WORK

We have presented a new plugin system for integrating more specialized VR input devices into modern plugin-orientated game engines. Our system consists of three individual plugins that cover the complete requirements for most devices: communication with different hardware devices, collision detection, and force rendering. Intentionally, we used an abstract design of our plugins. This abstract and modular setup makes it easy for other developers to exchange parts of our system to adjust it to their individual needs. Our results show that our plugin system is stable, and the performance is well-suited for our applications of haptic rendering and virtual grasping, even for complex non-convex objects.

With our plugin system, future projects have an easy way to include special devices in games, research, and business related applications. Even though other developers may decide to use different libraries for their work, we are confident that our experiences reported here in combination with our high-level UE4 plugin system will simplify their integration effort enormously.

However, our system and the current CollDet-based implementation also have some limitations that we want to solve in future developments. Currently, our system is restricted to rigid body interaction. Further work may entail the inclusion of deformable objects. In this case, a rework of the interfaces is necessary because the amount of data to be exchanged between the plugins will increase significantly; instead of transferring simple matrices that represent the translation and orientation of an object, we have to augment complete meshes. Direct access to UE4s mesh memory could be helpful to solve this challenge.

Also, our use cases offers interesting avenues for future works. Currently, we plan a user study with blind video game players to test their acceptance of haptic devices in 3D multiplayer environments. Moreover, we want to investigate different haptic object recognition tasks, for instance, with respect to the influence of the degrees of freedom of the haptic device or with bi-manual vs. single-handed interaction.

DATA AVAILABILITY STATEMENT

The raw data supporting the conclusions of this article will be made available by the authors, without undue reservation.

AUTHOR CONTRIBUTIONS

JR: evaluation of grasping and research. HM: evaluation of haptic feedback and research. RW: review and general ideas. MR: some part of implementation and technical help. JG: some part of implementation. GZ: review, general help and supervision. All authors contributed to the article and approved the submitted version.

FUNDING

The research reported in this paper has been (partially) supported by the German Research Foundation DFG, as part of Collaborative Research Center (Sonderforschungsbereich) 1320 EASE—Everyday Activity Science and Engineering, University of Bremen (http://www.ease-crc.org/).

ACKNOWLEDGMENTS

Some part of this work has previously been published at the EuroVR 2018 conference in London, United Kingdom, October, 22–23, 2018 (Rüdel et al., 2018). We extended the original plugin for haptic-based devices to other specialized input devices, e.g., Cybergloves, in this follow-up work.

REFERENCES

3D Systems (2018). *Geomagic OpenHaptics Toolkit*. Available online at: https://www.3dsystems.com/haptics-devices/openhaptics (accessed September 3, 2020).

Andrews, S., Mora, J., Lang, J., and Lee, W. S. (2006). "Hapticast: A physically-based 3D game with haptic feedback," in *Proceedings of FuturePlay 2006* (Ontario, CA).

CHAI3D (2016a). *CHAI3D Documentation–Haptic Rendering*. Available online at: http://www.chai3d.org/download/doc/html/chapter17-haptics.html (accessed March 9, 2020).

CHAI3D (2016b). Available online at: http://www.chai3d.org/ (accessed March 9, 2020).

de Pedro, J., Esteban, G., Conde, M. A., and Fernández, C. (2016). "Hcore: a game engine independent OO architecture for fast development of haptic simulators for teaching/learning," in *Proceedings of the Fourth International Conference on Technological Ecosystems for Enhancing Multiculturality* (New York, NY: ACM), 1011–1018. doi: 10.1145/3012430.3012640

Epic Games (2020a). *Introduction to C++ Programming in UE4*. Available online at: https://docs.unrealengine.com/en-US/Programming/Introduction

Epic Games (2020b). *Plugins*. Available online at: https://docs.unrealengine.com/latest/INT/Programming/Plugins/index.html (accessed March 9, 2020).

Gomes de Sá, A., and Zachmann, G. (1999). Virtual reality as a tool for verification of assembly and maintenance processes. *Comput. Graph.* 23, 389–403. doi: 10.1016/S0097-8493(99)00047-3

H3DAPI (2019). Available online at: http://h3dapi.org/ (accessed March 9, 2020).

Haption, S. A. (2020). *Virtuose 6D Desktop*. Available online at: https://www.haption.com/pdf/Datasheet_Virtuose_6DDesktop.pdf (accessed March 9, 2020).

Kadleček, P., and Kmoch, S. P. (2011). "Overview of current developments in haptic APIs," in *Proceedings of CESCG* (Vienna: Vienna University of Technology).

Kollasch, F. (2017). *Sirraherydya/Phantom-Omni-Plugin*. Available online at: https://github.com/SirrahErydya/Phantom-Omni-Plugin (accessed Septemper 3, 2020).

Lin, J., Guo, X., Shao, J., Jiang, C., Zhu, Y., and Zhu, S. C. (2016). "A virtual reality platform for dynamic human-scene interaction," in *SIGGRAPH ASIA 2016 Virtual Reality Meets Physical Reality: Modelling and Simulating Virtual Humans and Environments* (New York, NY: ACM), 1–4. doi: 10.1145/2992138.2992144

Liu, H., Zhang, Z., Xie, X., Zhu, Y., Liu, Y., Wang, Y., et al. (2019). "High-fidelity grasping in virtual reality using a glove-based system," in *2019 International Conference on Robotics and Automation (ICRA)* (Piscataway, NJ: IEEE), 5180–5186. doi: 10.1109/ICRA.2019.8794230

McNeely, W. A., Puterbaugh, K. D., and Troy, J. J. (1999). "Six degree-of-freedom haptic rendering using voxel sampling," in *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '99* (New York, NY: ACM Press/Addison-Wesley Publishing Co.), 401–408. doi: 10.1145/311535.311600

Moehring, M., and Froehlich, B. (2011). "Effective manipulation of virtual objects within arm's reach," in *2011 IEEE Virtual Reality Conference* (Piscataway, NJ: IEEE), 131–138. doi: 10.1109/VR.2011.5759451

Mól, A. C. A., Jorge, C. A. F., and Couto, P. M. (2008). Using a game engine for VR simulations in evacuation planning. *IEEE Comput. Graph. Appl.* 28, 6–12. doi: 10.1109/MCG.2008.61

Morris, D., Joshi, N., and Salisbury, K. (2004). "Haptic battle pong: High-degree-of-freedom haptics in a multiplayer gaming environment," in *Proceedings of Experimental Gameplay Workshop* (San Jose: Game Developers Conference). Available online at: https://www.microsoft.com/en-us/research/publication/haptic-battle-pong-high-degree-freedom-haptics-multiplayer-gaming-environment-2/

Reinschluessel, A. V., Teuber, J., Herrlich, M., Bissel, J., van Eikeren, M., Ganser, J., et al. (2017). "Virtual reality for user-centered design and evaluation of touch-free interaction techniques for navigating medical images in the operating room," in *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems, CHI EA '17* (New York, NY: ACM), 2001–2009. doi: 10.1145/3027063.3053173

Rüdel, M. O., Ganser, J., Weller, R., and Zachmann, G. (2018). "Unrealhaptics: a plugin-system for high fidelity haptic rendering in the unreal engine," in *International Conference on Virtual Reality and Augmented Reality* (Chem: Springer International Publishing), 128–147. doi: 10.1007/978-3-030-01790-3_8

Ruffaldi, E., Frisoli, A., Bergamasco, M., Gottlieb, C., and Tecchia, F. (2006). "A haptic toolkit for the development of immersive and web-enabled games," in *Proceedings of the ACM Symposium on Virtual Reality Software and Technology* (New York, NY: ACM), 320–323. doi: 10.1145/1180495.1180559

Sagardia, M., Stouraitis, T., and Silva, J. L. E. (2014). "A new fast and robust collision detection and force computation algorithm applied to the physics engine bullet: method, integration, and evaluation," in *EuroVR 2014–Conference and Exhibition of the European Association of Virtual and Augmented Reality*, eds J. Perret, V. Basso, F. Ferrise, K. Helin, V. Lepetit, J. Ritchie, C. Runde, et al. (The Eurographics Association).

Tenorth, M., Winkler, J., Beßler, D., and Beetz, M. (2015). Open-EASE: a cloud-based knowledge service for autonomous learning. *KÜnstl. Intell.* 29, 407–411. doi: 10.1007/s13218-015-0364-1

The Glasgow School of Art (2014). *Haptic Demo in Unity Using OpenHaptics With Phantom Omni*. Online Video. Available online at: https://www.youtube.com/watch?v=nmrviXro65g (accessed September 3, 2020).

User ZeonmkII (2016). *Zeonmkii/Omniplugin*. Available online at: https://github.com/ZeonmkII/OmniPlugin (accessed September 3, 2020).

Verschoor, M., Lobo, D., and Otaduy, M. A. (2018). "Soft hand simulation for smooth and robust natural interaction," in *2018 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)* (Piscataway, NJ: IEEE), 183–190. doi: 10.1109/VR.2018.8447555

Weller, R., Sagardia, M., Mainzer, D., Hulin, T., Zachmann, G., and Preusche, C. (2010). "A benchmarking suite for 6-DOF real time collision response algorithms," in *Proceedings of the 17th ACM Symposium on Virtual Reality Software and Technology* (New York, NY: ACM), 63–70. doi: 10.1145/1889863.1889874

Weller, R., and Zachmann, G. (2009). "A unified approach for physically-based simulations and haptic rendering," in *Sandbox 2009*, ed D. Davidson (New York, NY: ACM), 151. doi: 10.1145/1581073.1581097

Zachmann, G. (1998). "Rapid collision detection by dynamically aligned DOP-trees," in *Proceedings of IEEE Virtual Reality Annual International Symposium; VRAIS '98* (Atlanta, GA), 90–97. doi: 10.1109/VRAIS.1998.658428

Zachmann, G. (2001). "Optimizing the collision detection pipeline," in *Procedings of the First International Game Technology Conference (GTEC)* (Hong Kong).

Zachmann, G. (2002). "Minimal hierarchical collision detection," in *Proceedings of ACM Symposium on Virtual Reality Software and Technology (VRST)* (Hong Kong), 121–128. doi: 10.1145/585740.585761

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.