UnrealHaptics: A Plugin-System for High Fidelity Haptic Rendering in the Unreal Engine

Marc O. Rüdel, Johannes Ganser, Rene Weller, and Gabriel Zachmann

University of Bremen, Germany
http://cgvr.informatik.uni-bremen.de/

Abstract. We present UnrealHaptics, a novel set of plugins that enable both 3-DOF and 6-DOF haptic rendering in the Unreal Engine 4. The core is the combination of the integration of a state-of-the-art collision detection library with support for very fast and stable force and torque computations and a general haptics library for the communication with different haptic hardware devices. Our modular and lightweight architecture makes it easy for other researchers to adapt our plugins to their own requirements. As a use case we have tested our plugin in a new asymmetric collaborative multiplayer game for blind and sighted people. The results show that our plugin easily meets the requirements for haptic rendering even in complex scenes.

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 the 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 are primarily focused on outputting information to the two main human senses: seeing and hearing. Also game engines are mainly limited to visual and audio output. The sense of touch is widely neglected. This 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.

The main reasons why the sense of touch is widely neglected in the context of games are that haptic devices are still comparatively bulky and expensive. Moreover, haptic rendering is computationally and algorithmically very challenging. Although many game engines have a built-in physics engine, they are most usually limited to simple convex shapes and they are relatively slow: for the visual rendering loop it is sufficient to provide 60-120 frames per second

(FPS) to guarantee a smooth visual feedback. Our sense of touch is much more sensitive with respect to the temporal resolution. Here, a frequency of preferably 1000 Hz is required to provide an acceptable force feedback. This requirement for haptic rendering requires a decoupling of the physically-based simulation from the visual rendering path.

In this paper, we present UnrealHaptics to enable 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 Haptico: it realizes the communication with the haptic hardware.
- 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 haptic rendering.
- 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). We use a fast, lightweight and highly maintainable and adjustable event system to handle the communication in UnrealHaptics.

As a use case we present a novel asymmetric collaborative multiplayer game for sighted and blind players. In our implementation, Haptico integrates the CHAI3D library that offers support for a wide variety of available haptic devices. For the collision detection we use the state-of-the-art collision detection library CollDet [27] 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

Game engines enable the rapid development with high end graphics and the 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. [2], [15], [13] to name but a few. However, they usually have spent a lot of time in developing single use approaches which are hardly generalizable and thus, not applicable to other programs.

Actually, there exist only a very few approaches that provide comfortable interfaces for the integration of haptics into modern game engines. We only found [11] and [22] that provide plugins for UE4 that serve as interfaces to the *3D*

Systems Touch (formerly *SensAble PHANToM Omni*) [16] via the *OpenHaptics* library [1]. OpenHaptics is a proprietary library that is specific to 3D Systems' devices, which means that other devices cannot be used with these plugins. Furthermore, the plugins are not actively maintained and seem to not be working with the current version of UE4 (version 4.18 at the time of writing). Another example is a plugin for the PHANToM device presented in [20], also based on the OpenHaptics library. Like the other plugins, it is no longer maintained and was even removed from Unity's asset store [21]. During our research, we could not find any actively maintained plugin for a commonly used game engine that supports 3- or 6-DOF force feedback.

Outside the context of game engines, there are a number of libraries that provide force calculations for haptic devices. A general overview is given in [10]. One example is the CHAI3D library [4]. 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 by 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 meshes surface. 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 [3]. Due to this method's simplicity, 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 older library is the H3DAPI library [7]. Same as CHAI3D, it is extensible in both the device and algorithm domain. However by default H3DAPI supports less 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. [18]. It is based on the eXtreme Virtual Reality (XVR) engine, utilising the CHAI3D library, in order to allow rapid application development independent from the specific haptic interface. Unfortunately, the toolkit has not been further developed and there is no documentation to be found, since their homepage went down.

All approaches mentioned above are limited to 3-DOF haptic rendering. Sagardia et al. [19] present an extension to the *Bullet* physics engine for faster collision detection and force computation. Their algorithm is based on the Voxmap-Pointshell algorithm [12]. 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 clustured 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 hap-

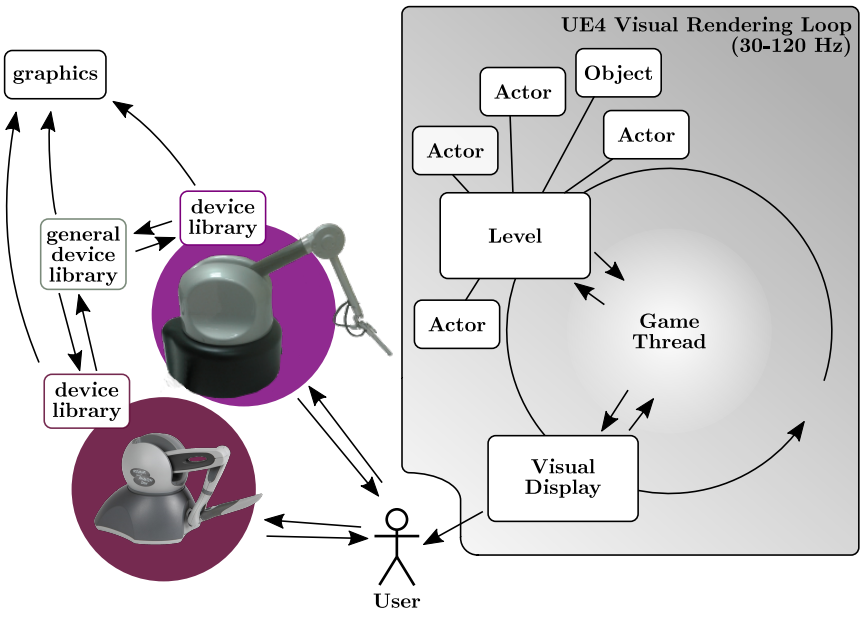


Fig. 1. A typical haptic integration without UnrealHaptics. Left: different haptic devices available with their libraries. Right: Scheme of UE4, which we want to integrate the devices with.

tic rendering for complex scenes. However, the Voxmap-Pointshell algorithm is known to be very memory intensive and susceptible to noise [23].

3 UnrealHaptics

The goal of our work was to develop an easy-to-use and simultaneously adjustable and generalizable system for haptic rendering in modern game engines. This can be used in games, research or business related contexts, either as whole or in parts. We decided to use the Unreal Engine for development because of several reasons:

- it is one of the most popular game engines with a large community, regular updates and a good documentation,
- it is free to use in most cases, especially in a research context where it is already heavily used [17], [14],
- 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 graphical editor in combination with a graphical scripting system called *Blueprints*. Thus, it combines the advantages of open class libraries and extensible IDEs
- it is extendable via plugins,

– and finally, it is build on C++, which makes it easy to integrate external C++-libraries. This is convenient because C++ is still the first choice for high-performance haptic rendering libraries.

Our goals directly imply a modular design for our system. The main challenges when including haptics into programs are fast collision detection, stable force computation and communication with hardware devices. 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 UE4 in which we want to integrate the devices. Consequently, our system consists of three individual plugins that realizes one of these tasks. In detail these are:

– A plugin called Haptico, which realizes the communication with haptic hardware, i.e. it initializes haptic devices and during runtime receives positions and orientations and sends forces and torques back to the hardware.
– 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 which receives collision information and computes forces and torques that are finally send to Haptico. 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.

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

UE4 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.

UE4 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.).

UE4 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.

UE4 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 the UE4 with custom classes, the general idea is noted in [6]: 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 [5]. 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

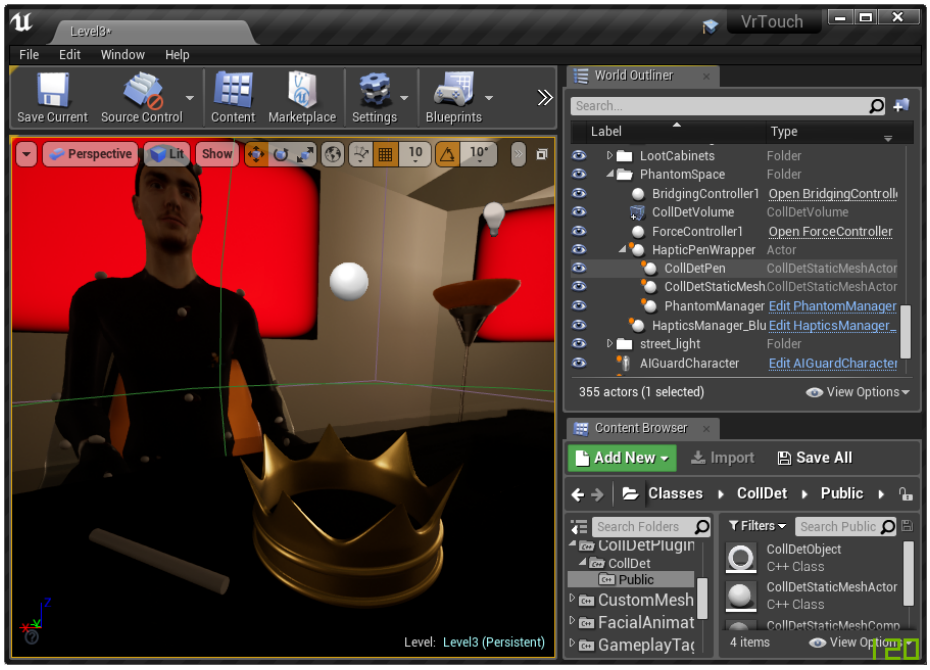


Fig. 2. Unreal's editor view of the game. On the left side, you see the Phantom player in the virtual environment. In front of him are the virtual tool (pen) and a `ColletteStaticMeshActor` to be recognized (crown). On the right, the scene graph is displayed with our custom classes.

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.

Our Light Delegate System UE4 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 UE4's reflection system. This may influence performance when many delegates are used. Finally, we experienced problems at runtime: UE4-delegators temporarily forgot their bound functions which led to crashes when trying to access the addresses of these functions.

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 is always passed around as references internally, preventing any additional copies.

3.3 Haptico Plugin — Haptic Device Interface

HAPTICO enables game developers to use haptic devices directly from UE4 without implementing a connection to the device manually. It automatically detects a connected haptic device and allows full control via either Blueprints or C++ Code. This includes the retrieval of positions and orientations from the device and the sending of forces and torques to the device, thanks to the underlying CHAI3D library.

HAPTICO consists of mainly three parts: The haptic manager, the haptic thread and the haptic device interface. The haptic manager is the only user interface and represented as an UE4 actor in the scene. It provides functions to apply forces and torques to the device and to get informations such as position and rotation of the end effector. To be used for haptic rendering the execution loop of the plugin must be separated from UE4's game thread which runs at a low frequency. The plugin uses its own haptic thread internally. The haptic thread reads the positional and rotational data from the device, provides it for the haptic manager and applies the new force and torques retrieved from the haptic manager to the device in every tick. When new haptic data is available a delegator-event `OnTransform` is broadcasted, which passes the device data to the haptic manager in every tick. Users of the plugin can easily hook their own functions to this event, allowing to react to the moved device. A second delegator-event `ForceOnHapticTick` is broadcasted, which allows users to hook force calculation functions into the haptic 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 haptic rendering:

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 haptic rendering.

This leads to the realization that for haptic rendering, UE4's physics module has to be bypassed.

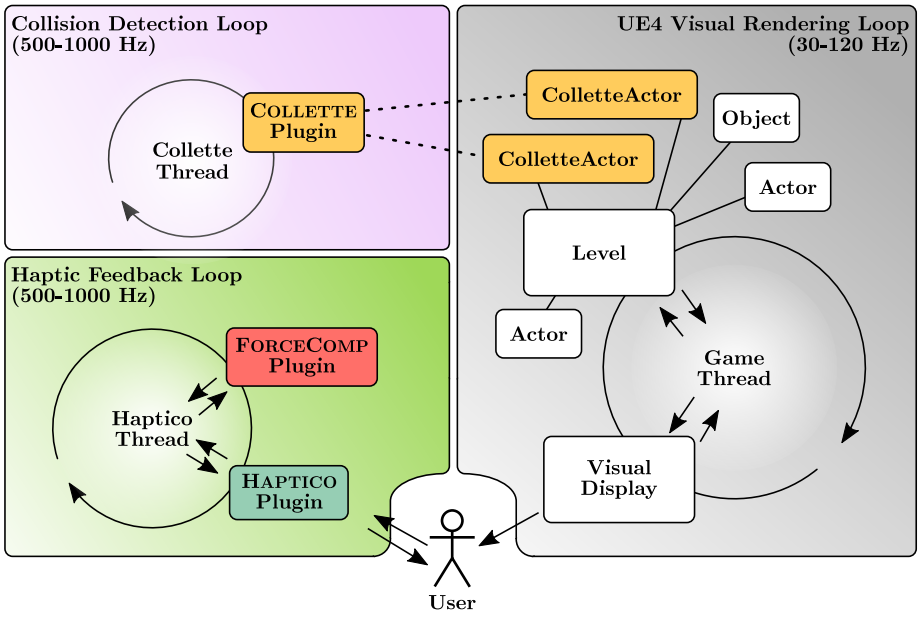


Fig. 3. The basic structure of our plugin system with three threads. Right: The UE4 game thread that is responsible for the visual feedback and runs with up to 120Hz. Left: The haptic rendering thread and the collision detection thread. The haptic rendering that included the HAPTICO and the FORCECOMP plugin runs at 1000Hz for a stable haptic feedback. We decided to put the collision detection in its own thread in order to not disturb the haptic rendering e.g. in case of deep collisions that require more computation time than 1 msec. The collidable objects in the Unreal scenegraph are represented as `ColletteStaticMeshActor`s that are derived from Unreal's built in `StaticMeshActor`s.

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 Section 4.2). Like HAPTICO, COLLETTE can run in its own thread. Thus, the frequency needed for haptic rendering can be achieved.

The plugin uses a `ColletteStaticMeshActor` to represent collidable objects. 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. This is especially useful for asymmetric multiplayer scenarios as described in Section 4.

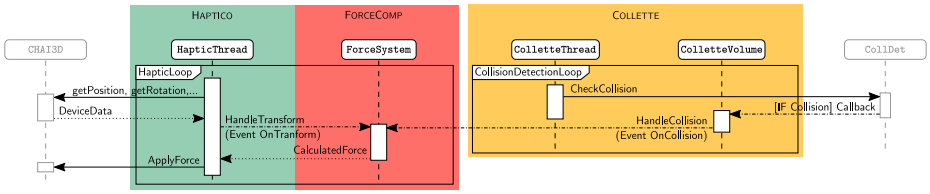


Fig. 4. A simplified sequence diagram of the communication of ForceComp, Collette and Haptico in case of a collision: Haptico receives the current position and orientation from the device and informs ForceComp via a `OnTransform` event. `ColletteVolume` in Collette evokes an `OnCollision` event and passes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

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 4.2). 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 mange 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, hence, there is no coding required to establish these references. For instance, if we want to attach an mesh to the haptic device to use it as a virtual tool. In this case, 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 haptic thread and for the `OnCollision` event of the `ColletteVolume` actor. The `OnTransform` event broadcasts the position and orientation data to the virtual tool automatically. This has the same effect as if the virtual tool would be updated directly in the haptic 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 passes the forces back to the `HapticManager`, which applies them to the associated haptic device (see Fig. 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 haptic 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

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 Haptico.

Overall, a typical setup with our plugin system consists of three threads: one for the main game loop including the visual rendering in Unreal, one for the haptic rendering, that covers Haptico and ForceComp and one for the collision detection. We decided to run collision detection independently in it's own thread in order 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 spend a fourth thread for ForceComp) by simply adjusting the configuration in the `EventHandler`.

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.

4 Use Case

We applied the UnrealHaptics to a real-world application with support for haptic rendering. This example shows how actual collision detection libraries, force rendering and communication libraries can be integrated into our plu-

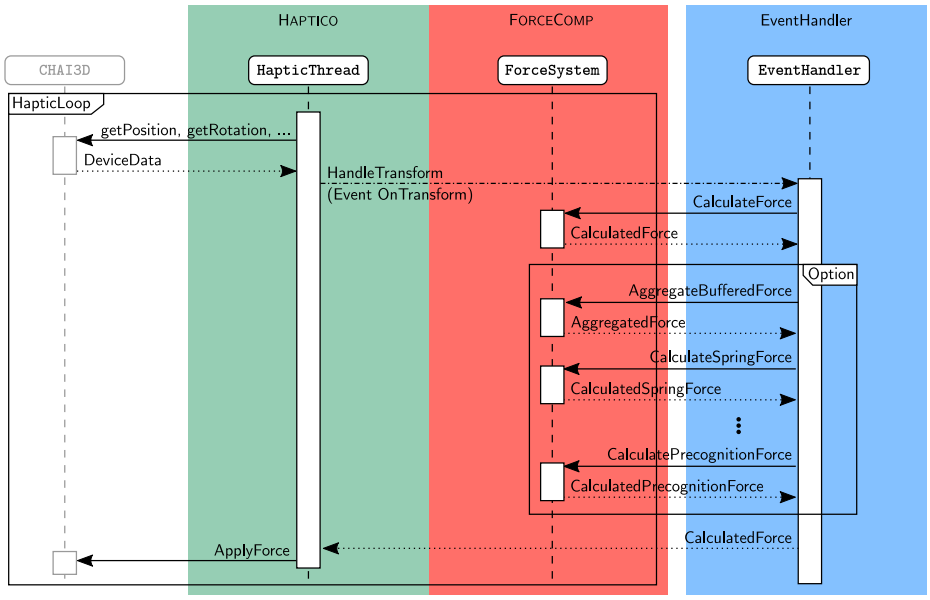


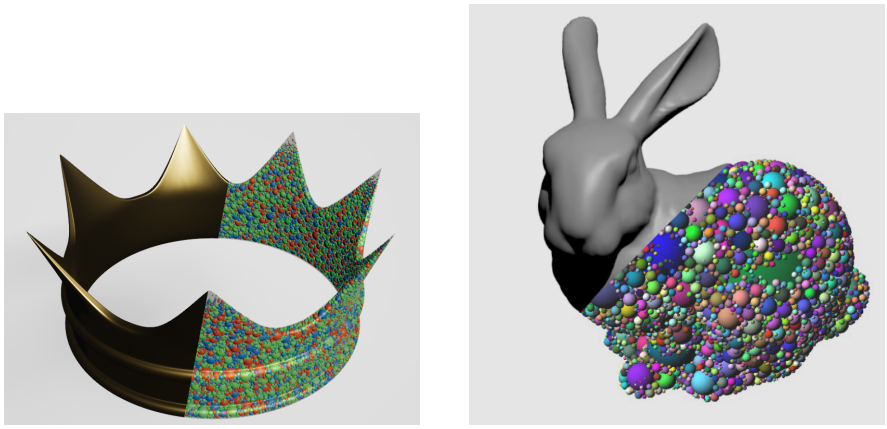
Fig. 5. A simplified sequence diagram of the communication of ForceComp, Haptico and our texttttEventHandler that also shows the flexibility of our system. Initially, Haptico reads the configuration from the haptic library and evokes an `OnTransform` event. This is passed to the `EventHandler` that calls 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 Haptico via the `EventHandler`.

gin system. Our use case is an asymmetric virtual reality multiplayer game [9] where a visually impaired and a seeing player can interact collaboratively in the same virtual environment. While the seeing person uses a head mounted display (HMD) and tracked controllers like the HTC Vive hand controllers, the blind person operates a haptic force feedback device, like the PHANToM Omni.

4.1 Game Idea

An extensive research involving interviews with visually impaired people was done to understand their perspective for a good game before going into development phase. It turned out that most people we interviewed attach great importance to a captivating storyline and ambiance. Therefore we included believable recordings and realistic sound effects to achieve an exciting experience.

The game takes place in a museum owned by a dubious relics collector. A team of two professional thieves, Phantom and Vive, attempt to break into the museum in order to steal various valuable artifacts. The blind player takes control over Phantom, a technician, particularly skilled in compromising security



(a) Half-section of the crown (b) Half-section of the bunny

Fig. 6. Objects from our game application and their inner sphere representations: A crown and a model of the Stanford bunny that has to be detected by the Phantom player.

systems and an expert for forgeries. Vive is played by the sighted player using an HMD. He is a professional pickpocket and a master of deceiving people.

For every exhibit in the museum, there are several fake artifacts that look exactly the same as the real ones. Since Vive is incapable of differentiating between real and fake artifacts, it is the job of Phantom to apply his skills here. Also, several guards patrol in the premises for possible intruders (see Figure 7). Vive has to be careful not to get spotted or make too much noise as these guards are highly sensitive to sounds. Vive's job is to break the displays, collect the artifacts while distracting the guards and bring them to Phantom. Phantom's job on the other hand is to recognize the right artifact using his shape recognition expertise. The goal of the game is to steal and identify all the specified artifacts before the time runs out.

In order to identify objects and the differences between fake and real objects in the game, the Phantom player uses a haptic force feedback device to sweep over the virtual collected objects. As soon as the virtual representation of the haptic device collides with an object, UnrealHaptics detects these collisions and renders the resulting forces back to the haptic device. It is therefore possible for visually impaired people to perceive the object similarly to how they would in real life. Adding realistic sounds to this sampling could further improve this experience.

Even if the gameplay is in the foreground in our current use case, it is obvious that almost the same setup can be easily extended to perform complex object recognition tasks or to combine HMD and haptic interaction for the sighted player.

Fig. 7. In-game screenshot of our implemented game. The Phantom player sits at the table recognizing objects. A guard (right) is patrolling the room.

4.2 Implementation Details

The concept behind UnrealHaptics is explained in Section 3. The following sections will give an insight into our concrete implementations for the individual plugins.

Device Communication via CHAI3D The basis for Haptico is the CHAI3D library. As already mentioned in Section 2, this library supports a wide variety of haptic devices, including the PHANToM and the *Haption Virtuose* [8] which we used for testing. CHAI3D is linked by Haptico 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.

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 [27]. This includes broad-phase collision detection algorithms like a uniform grid or convex hull pre-filtering as well as several narrow phase algorithms like a memory optimized version of an AABB-tree, called Box-tree [25], and DOP-trees [26]. For haptic rendering, the *Inner Sphere Trees* 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. The main advantage, beyond the performance, 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 [23]. 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 abstract 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 which implement the collision checking. Like with CHAI3D, COLLETTE links to the CollDet library at compile time.

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 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 [24]:

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

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

where (i, j) is a pair of colliding spheres, $\boldsymbol{n}_{i,j}$ is the collision normal, $vol_{i,j}$ is the overlap volume of the sphere pair, Vol_{total} is the total overlap volume of all colliding spheres, $vel_{i,j}$ is the magnitude of the relative velocity at the collision center in direction of $\boldsymbol{n}_{i,j}$. 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 to prevent 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

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

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 [24].

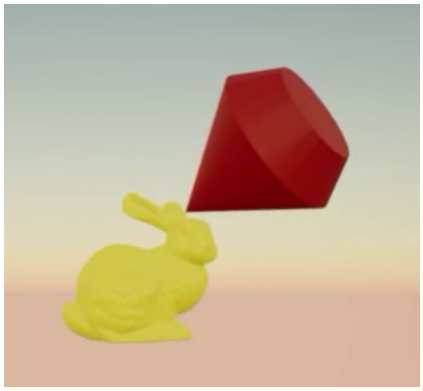


Fig. 8. In order to evaluate the performance of our plugins, we used a complex test scene where the user controls a gemstone with the Phantom device to touch the 3D Stanford bunny.

4.3 Performance

We have evaluated the performance of our implementation in the game on an Intel Core i7-6700K (4 Cores) with 64 GB of main memory and and a NVIDIA GeForce GTX 1080 Ti running Microsoft Windows 10 Enterprise.

We used a typical test scene from our game: the user explores the surface of an object (in our example, the Stanford bunny) with a Phantom device. In our example, we represented the end effector by a gemstone (see Figure 8).

We achieved almost always a frequency of 500-1K 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 500 Hz in situations of heavy interpenetrations. This is similar to the results reported in [23] where a simple OpenGL test scene was used and it shows that our architecture does not add significant processing overhead (see Figure 9).

5 Conclusions and Future Work

We have presented a new plugin system for integrating haptics into modern plugin-orientated game engines. Our system consists of three individual plugins that cover the complete requirements for haptic rendering: 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. In our use case, a collaborative multiplayer VR game for blind and sighted people, we have demonstrated the simplicity of integrating external C++-libraries with our plugins, namely CHAI3D for the communication with the hardware and the collision detection library CollDet. Our results show

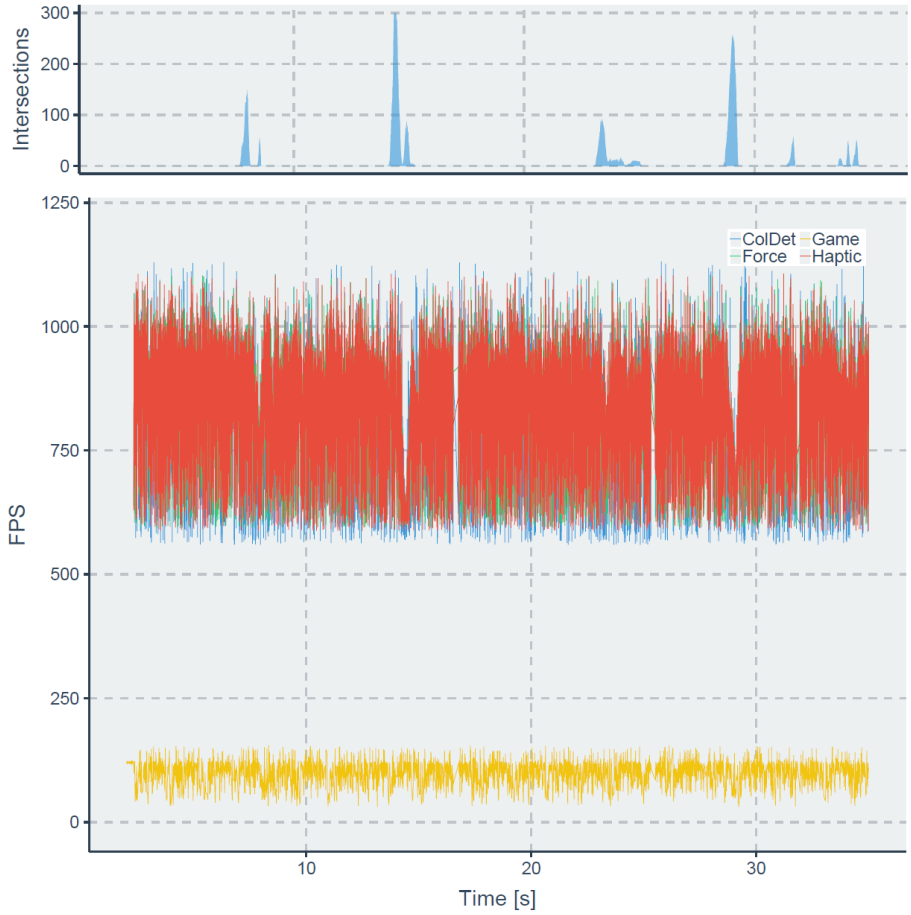


Fig. 9. Performance of our plugins in a typical exploration scene of about 35 seconds total duration in our game. We achieved haptic frame rates even in situations with large penetrations.

that our plugin system works stably and the performance is well suited for haptic rendering even for complex non-convex objects.

With our plugin system, future projects have an easy way to provide haptic force feedback in haptic enabled games, serious games, 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. Moreover, our system is not limited to haptic rendering but it can be also used to integrate general physically-based simulations.

However, our system, and the current CHAI3D and 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 case 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. Finally, other haptic interaction metaphors could also be interesting, e.g. the use of the haptic devices as a virtual cane to enable orientation in 3D environments for blind people.

References

1. 3D Systems: Geomagic OpenHaptics Toolkit (2018), https://www.3dsystems.com/haptics-devices/openhaptics, website
2. Andrews, S., Mora, J., Lang, J., Lee, W.S.: Hapticast: A physically-based 3d game with haptic feedback (2006)
3. CHAI3D: CHAI3D Documentation — Haptic Rendering (2018), http://www.chai3d.org/download/doc/html/chapter17-haptics.html
4. CHAI3D: Website (2018), http://www.chai3d.org/
5. Epic Games: Plugins (17112017), https://docs.unrealengine.com/latest/INT/Programming/Plugins/index.html
6. Epic Games: Introduction to C++ Programming in UE4 (2018), https://docs.unrealengine.com/en-US/Programming/Introduction, website
7. H3DAPI: Website, http://h3dapi.org/
8. Haption SA: Virtuose 6d desktop, https://www.haption.com/pdf/Datasheet_Virtuose_6DDesktop.pdf
9. Juul, J.: The game, the player, the world: Looking for a heart of gameness. PLURAIS-Revista Multidisciplinar **1**(2) (2010)
10. Kadleček, P., Kmoch, S.P.: Overview of current developments in haptic APIs. In: Proceedings of CESCG (2011)
11. Kollasch, F.: Sirraherydya/phantom-omni-plugin (11122017), https://github.com/SirrahErydya/Phantom-Omni-Plugin
12. McNeely, W.A., Puterbaugh, K.D., Troy, J.J.: Six degree-of-freedom haptic rendering using voxel sampling. In: Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques. pp. 401–408. SIGGRAPH '99, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1999). https://doi.org/10.1145/311535.311600, http://dx.doi.org/10.1145/311535.311600
13. Morris, D., Joshi, N., Salisbury, K.: Haptic Battle Pong: High-Degree-of-Freedom Haptics in a Multiplayer Gaming Environment (2004), https://www.microsoft.com/en-us/research/publication/haptic-battle-pong-high-degree-freedom-haptics-multiplayer-gaming-environment-2/

14. Ml, A.C.A., Jorge, C.A.F., Couto, P.M.: Using a game engine for vr simulations in evacuation planning. IEEE Computer Graphics and Applications **28**(3), 6–12 (May 2008). https://doi.org/10.1109/MCG.2008.61
15. de Pedro, J., Esteban, G., Conde, M.A., Fernández, C.: 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. pp. 1011–1018. ACM (2016)
16. PHANTOM, O.: Sensable technologies. Inc., http://www. sensable. com
17. Reinschluessel, A.V., Teuber, J., Herrlich, M., Bissel, J., van Eikeren, M., Ganser, J., Koeller, F., Kollasch, F., Mildner, T., Raimondo, L., Reisig, L., Ruedel, M., Thieme, D., Vahl, T., Zachmann, G., Malaka, R.: 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. pp. 2001–2009. CHI EA '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3027063.3053173, http://doi.acm.org/10.1145/3027063.3053173
18. Ruffaldi, E., Frisoli, A., Bergamasco, M., Gottlieb, C., Tecchia, F.: A haptic toolkit for the development of immersive and web-enabled games. In: Proceedings of the ACM symposium on Virtual reality software and technology. pp. 320–323. ACM (2006)
19. Sagardia, M., Stouraitis, T., Silva, J.L.e.: A New Fast and Robust Collision Detection and Force Computation Algorithm Applied to the Physics Engine Bullet: Method, Integration, and Evaluation. In: Perret, J., Basso, V., Ferrise, F., Helin, K., Lepetit, V., Ritchie, J., Runde, C., van der Voort, M., Zachmann, G. (eds.) EuroVR 2014 - Conference and Exhibition of the European Association of Virtual and Augmented Reality. The Eurographics Association (2014). https://doi.org/10.2312/eurovr.20141341
20. The Glasgow School of Art: Haptic demo in Unity using OpenHaptics with Phantom Omni (2014), https://www.youtube.com/watch?v=nmrviXro65g, online Video
21. The Glasgow School of Art: Unity Haptic Plugin for Geomagic OpenHaptics (HLAPI/HDAPI) (2018), https://assetstore.unity.com/packages/templates/unity-haptic-plugin-for-geomagic-openhaptics-hlapi-hdapi-19580, website
22. User ZeonmkII: Zeonmkii/omniplugin (1732016), https://github.com/ZeonmkII/OmniPlugin
23. Weller, R., Sagardia, M., Mainzer, D., Hulin, T., Zachmann, G., Preusche, C.: A benchmarking suite for 6-dof real time collision response algorithms (2010). https://doi.org/10.1145/1889863.1889874, http://dl.acm.org/ft_gateway.cfm?id=1889874&type=pdf
24. Weller, R., Zachmann, G.: A unified approach for physically-based simulations and haptic rendering. In: Davidson, D. (ed.) Sandbox 2009. p. 151. ACM, New York, NY (2009). https://doi.org/10.1145/1581073.1581097
25. Zachmann, G.: The boxtree: Exact and fast collision detection of arbitrary polyhedra. In: SIVE Workshop. pp. 104–112 (July 1995)
26. Zachmann, G.: Rapid collision detection by dynamically aligned DOP-trees. In: Proc. of IEEE Virtual Reality Annual International Symposium; VRAIS '98. pp. 90–97. Atlanta, Georgia (Mar 1998)
27. Zachmann, G.: Optimizing the collision detection pipeline. In: Procedings of the First International Game Technology Conference (GTEC) (2001)