Interactive Visualization Pipeline Construction Environment for Virtual Reality Visualizations

Sandro Massa
sandro.massa@student.uva.nl

July 10, 2022, 65 pages

Academic supervisor: dr Robert G. Belleman, r.g.belleman@uva.nl
Host organisation/Research group: Visualization Lab at Science Park,
Computational Science Lab

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

Virtual Reality (VR) has enabled analysis and communication of data through immersive and interactive visualizations and environments. The development of which is an iterative process that usually requires the user to work on a desktop workstation, move away from it to put on a VR visor to test, then remove it again and go back to their workstation. Such a workflow is inefficient, making focus and flow states for the user hard to achieve. Furthermore, tools reducing such burden on the user either do not exist or are usable for prototypes at best.

The Visualization ToolKit (VTK) is a rich library for the development of scientific visualizations. It is used in VR applications [1, 2] and is greatly supported. Unity is a game engine widely supported and used, with a large development community, which is also used for scientific visualization [2, 3]. Efforts to integrate the two exist and are promising [2, 3], however lack in performance and general purpose usability.

We propose an architecture integrating VTK and Unity, enabling users to fully access the former from within the latter, comprising the UI components for an interactive VR IDE for scientific visualizations. Furthermore, the architecture is agnostic of the versions of either and does not require tweaking between versions. Finally, it supports user extensions that can then be easily called from C# scripts in Unity.

We demonstrate that such architecture reaches the target performance set for HMDs such as the HTC Vive visor, and that it can easily support advanced visualization applications. The final system achieves its objective in making VR development more efficient. It provides support for widely used software in the community, and is capable of supporting a VR workbench as presented in our architecture.

Contents

Chapter 1

Introduction

Virtual Reality (VR) offers great opportunities, especially when it comes to visualizing information and applications, that enable students, practitioners and researchers to better understand and manipulate data. The main advantages over classic screens are its capacity of creating fully interactive and immersive environments, as well as leveraging the human innate spatial comprehension and interaction capabilities [4]. These make VR suited for all applications where interaction and immersion are critical, ranging from educational [5] to medical [6, 7] purposes. Another important aspect of VR that arises directly from its interactive and immersive environments, is that it has good support for activities that require the user to stay active, such as didactic and educational activities. [5].

To aid creating visualizations, both for VR and not, a number of toolkits and frameworks exist. Focusing on scientific visualization, there are several toolkits such as The Visualization ToolKit (VTK) and The Insight ToolKit (ITK). VTK is a powerful and extensive open source toolkit for (3D) visualization, is a widely known and used library which is based on the creation of visualization pipelines [8]. ITK [9] instead, offers libraries in a variety of programming languages focused on creating visualization for different purposes. Examples are volume rendering [2, 10] for medical and engineering purposes and parallel model computation and rendering [11]. As such, these toolkits become also an important resource for VR environments' development [2].

A number of tools already exist for development with these libraries: environments such as ParaView[1], MeVisLab[2] and Delft Visualization and Image processing Development Environment (DeVIDE) [12] integrate either one or the other or both and more of these toolkits. Even though some integration exists for VR development [13, 14], these fall short of making use of the powerful capabilities of VR and are only portings. These still require the user to jump in and out of the environment to make adjustments at the desktop workstation and can be both wasteful in time and break workflow [1, 4, 15, 16]. Coupled with the fact that development of good visualizations is an iterative process, requiring the developer to tweak the pipeline multiple times over, it makes for an inefficient process at best.

This thesis proposes a way of integrating these toolkits, in particular focusing on VTK, with general purpose VR applications development environments, both lowering the technical barrier of using these toolkits and their restrictions when it comes to VR, as well as making use of the capabilities this technology has to offer. To achieve this, we look at previous attempts with this toolkit and its existing environments [1, 15, 16], other environments [2] and similar specific VR development environments [17].

We develop a VR pipeline development environment inspired by ParaView desktop's approach, where the user is able to both edit and render the visualization without exiting the VR environment and using an intuitive block-based editor to create the pipeline. More specifically, we develop an integration between the VTK library and Unity, a general purpose game engine [18], that is used for the generation of the virtual world, as well as handling the UI components that allow editing of the VTK objects.

[1] `https://www.paraview.org/`
[2] `https://www.mevislab.de/`

1.1 Problem statement

Creation of VR visualizations in particular is, to this day, cumbersome, requiring the user to mostly work from a desktop screen, which is not ideal to design a VR visualization, due to its different nature. Another big issue is the fact that to move between testing and usage station (i.e. the VR headset, and the desktop development station) is a non-trivial job that is time-consuming at best, and at worst a factor slowing down and degrading the development process. Creating such an environment represents an important step for the visualization community, removing barriers for VR development, and a challenge from a Software Engineering standpoint. Furthermore, from a research perspective a number of questions open up which are not yet fully explored.

A key characteristic of any good VR environment is performance: in order to have a decent user experience, and limit the influences of motion sickness and other problems that arise from poorly designed VR environments [19], the software needs to reach high performance thresholds. Guidelines for general applications for VR suggest achieving a frame rate of at least 90 FPS [20], but depending on the headset even higher frame rates may be needed, i.e. with newer versions of the Valve Index or the Pimax 5K plus. Considering that a VR Head-Mounted Display (HMD) is made of two displays, one per eye, the target FPS is double the suggested threshold, posing serious design challenges.

1.1.1 Research questions

Designing an environment as the one we describe opens up a number of questions, especially what are the challenges with it its maintainability, as it involves connecting two third-party code bases. We investigate similar works in order to fully develop a stable core of the environment we envision. A very first step has already been accomplished by finding ways to embed visualization libraries in VR development engines that is the base of our environment. Examples of this are ActiViz [3] and Wheeler et al.'s VtkToUnity [2], both interfacing VTK with Unity.

These solutions though aim not at creating development environments, but are oriented at supporting the development of specific applications. For example ActiViz is a commercial tool which has performance limits due to its use of copying operations from CPU to GPU [21], whereas Wheeler et al.'s VtkToUnity, part of the 3DHeart project [2], lacks generality, as VTK is called through specialized code and only the VTK feature needed are actually visible from Unity.

RQ1 In what way can interfacing The Visualization ToolKit with Unity Game Engine be done such that VR usability and performance standards are upheld?

By fully interfacing the toolkit with the game engine we intend that the engine has full access to the capabilities of VTK, while this last one is able to capture events arising within the environment. From previous attempts at achieving this [1, 2, 15, 16], we found that a major issue comes from using different versions of the involved libraries, which were not always traceable back to a simple API change. As such, our intent is to investigate what factors contribute to this instability and how we can limit their impact on the solution.

RQ2 What are the factors contributing to maintainability issues of interfacing The Visualization ToolKit and Unity Game Engine?

RQ2.1 How can the impact of these factors on the maintainability of the system be limited?

Finally, as UI and UX are out of the scope of this thesis, yet important to our successful investigation of the aforementioned research questions, thus they are investigated alongside this thesis project by someone else. Due to the constraints on a work such as a Master's Thesis, we report on the difficulties of such a Software development Process.

1.1.2 Research method

In order to investigate our research questions, we design a software system based on precedent work. The design is guided by three main principles which reflect the constraints we put in our research questions:

1. **Integration**: the two main components of the system we develop already exist, VTK and Unity. A link between the two exists, but is imperfect, and does not allow the capabilities of both to be fully leveraged in the end system. This may be sufficient for some applications, but where a full interfacing of the two is necessary, the existing approaches require a great development overhead. Our objective is to reduce this overhead by maximizing the integration between these two components;
2. **Performance**: a key problem with VR applications is achieving the performances that are sufficient for a decent user experience. A number of guidelines exist on what these performances should be in terms of frames per second (FPS) [20, 22]. As we set out to create a usable IDE that should be usable by users to create their visualizations within a VR environment, we have the objective of achieving an average FPS count of the recommended values for the game engine we select. Another critical aspect of performance is response time: queries and operations on the pipeline have to be executed as fast as possible to avoid rendering issues and long waiting periods;
3. **Maintainability**: as previously expressed, one of our main concerns is maintainability. Due to the nature of the system we develop, our concern is to maximize its maintainability for long term support. We first and foremost investigate which factors contribute to hardly maintainable solutions, and we then discuss how we can limit them.

Given these drivers, and our goal of creating a long-lasting IDE that would enhance the quality of life of the VR visualization development community. Our first concern is to investigate the factors contributing to the maintainability issues of previous solutions proposed by performing a small literature review, collecting a number of previous solutions, and examining them to find out their weak spots. We compile a list of these factors and how much they impact the system, how easily they can be avoided or solved and what components they involve.

Based on the results of this first part, we design an architecture with the objective of minimizing the factors impacting maintainability. At this stage, we also focus on laying the grounds of a performing system, and as such we discuss a number of design choices, potentially based on architectures from different strands of Computer Sciences. We attempt to keep the concerns as separate and encapsulated as possible with the intent of enabling the software to be distributed.

Finally, we develop a proof of concept that we test for performance on a number of visualizations on three main operations: creation of the visualization, rendering of the visualization and a combination of the two. We then compare the results of our benchmarks with the guidelines provided by the developer of Unity and/or manufacturer of the VR headset we choose, as well as with the results of previous research on this topic [1, 2].

Alongside this main research journey, we also collect, in the form of Action-Research, a diary of the difficulties we encountered in researching, designing and implementing the software, as well as managing the two different development branches that are to be highly decoupled. Based on our experience, we discuss what were the weak points of our approach and what are in hindsight potential precautions and solutions we could have taken to enhance productivity and success chances.

1.2 Contributions

Our research makes the following contributions:

1. Analysis of the factors contributing to the unmaintainability of software bridging VTK and Unity,
2. Development of a potentially distributed architecture for performing VR visualization creation and rendering systems,

1.3 Outline

In Chapter 2 we describe the background of this thesis and analyze previous work researching factors contributing to maintainability issues of this kind of software. Chapter 3 describes the design challenges and choices of the system. Chapter 4 describes the development of an architecture able to support the performance required by our software while meeting its requirements. Results of the literature review,

development, testing and benchmarking are shown in Chapter 5 and discussed in Chapter 6. Finally, we present our concluding remarks in Chapter 7 together with future work.

Chapter 2

Background

This chapter presents the necessary background information for this thesis. First, we define some basic terminology that we use throughout this document. Next, we research prior work in the field of VR that interfaces VTK and Unity or develops IDEs for visualizations within a VR environment. We also examine the software and solutions produced by these previous studies and extract which factors contribute to potential maintainability issues and which on the contrary make them more maintainable.

2.1 Terminology

We define a set of terms that we use throughout this thesis, based on the *Glossary of virtual reality terminology* [23] as a main source for our terminology.

Virtual Reality (VR) A computer system used to create an artificial world which is immersive and interactive for the user.

Immersion The feeling of the user of being part of a virtual environment.

Virtual Environment Realistic simulations of interactive scenes.

Interactivity The ability of the user to navigate and manipulate (objects in) the environment.

Visualization Toolkit Library containing functionality aimed at creating and manipulating data visualizations.

Game Engine A software that enables the creation of virtual environments.

2.2 The Visualization Toolkit

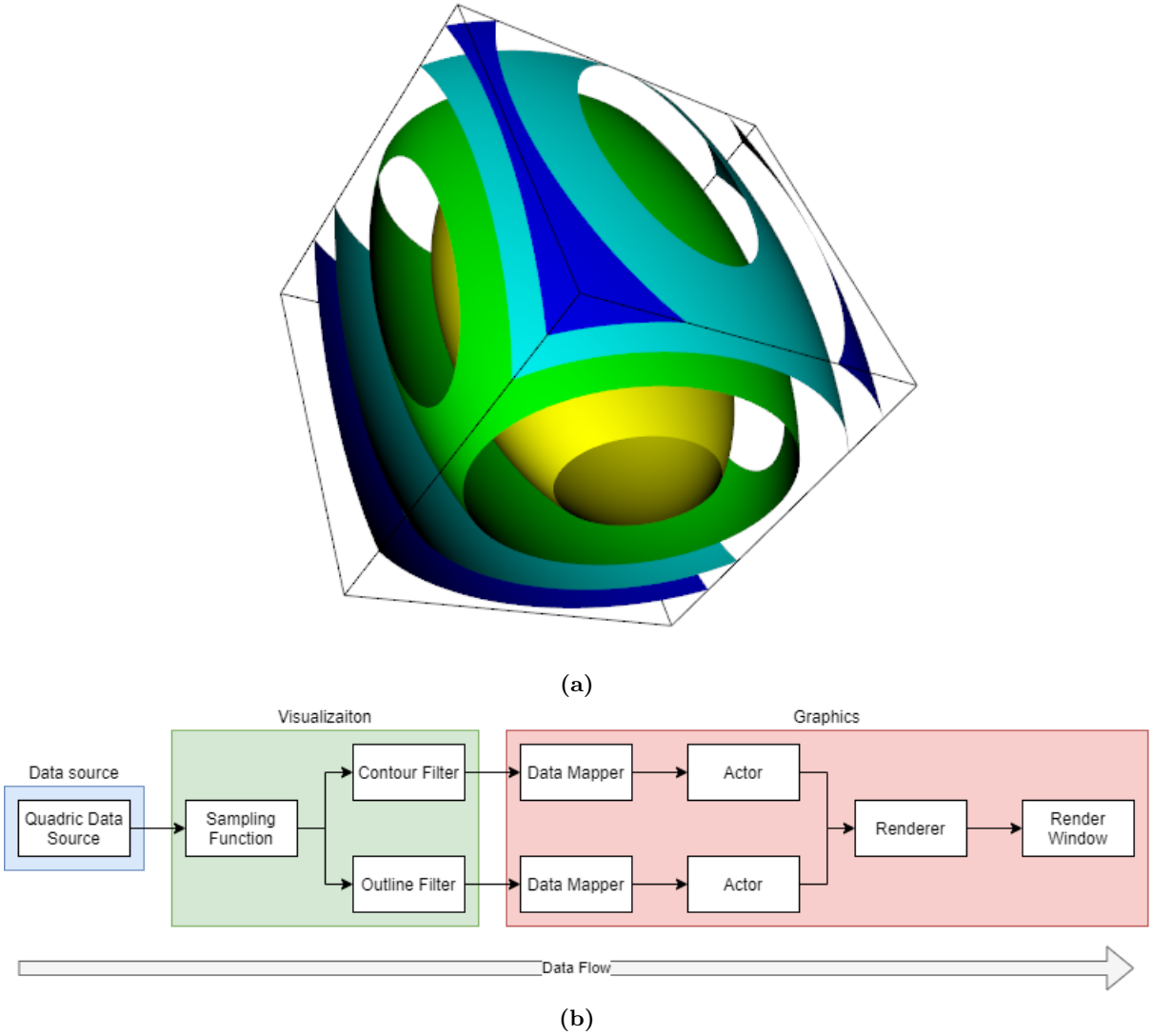
As the necessity to produce ever more complex visualizations grows, both in commercial and scientific settings, a number of visualization toolkits have been developed to support the development of these visualizations. These libraries tend to specialize on some form of visualization; for example, The Visualization ToolKit (VTK) focuses on scientific visualization [15].

These toolkits come in a variety of flavors, languages and/or wrappings, and it is thus out of the scope of this thesis to analyze and develop upon all of them. We discuss research done on all three to create IDEs for or within VR environments, but we ultimately choose to use for a proof of concept.

The Visualization ToolKit (VTK) is a library oriented towards "*manipulating and displaying scientific data*"[1]. It creates visualizations by means of filter pipelines, a technique proposed by Haber and McNabb [24]. These filters manipulate input data through some set of transformations and return the composed data which can then be visualized.

VTK has three different stages of the pipeline which combined return a visualization rendering. The first stage is the generation or the import of a data source and transforming the raw data in some more readable raw data if needed, like structuring it in a model. Subsequently, the raw data is then

[1] `https://vtk.org/`



(a)

(b)

Figure 2.1: VTK visualization (a) and pipeline (b) of a sampled Quadric function.

transformed through filters that compose the data in order for it to be visualized, creating the actual visualization. Finally, the visualization is rendered through the application of mappers that transform the computational representation into a graphical representation. An example of a Quadric 3D function is shown in Figure 2.1, based on an example proposed in the VTK example repository[2], the exact code is presented in Appendix A.

Upon VTK, a number of wrappers have been produced, allowing the library to be used not only in C++, but also in Java, Python, TCL and the .NET Framework[3]. These wrappers allow the usage of the *wrappable* code of the library from one of the aforementioned languages. This process is to be manually done at build time of the library or using one of the wrapping tools accessible from the documentation of VTK.

2.3 Unity Game Engine

Game Engines can be defined as "*software frameworks for game development*" [25]. These frameworks though are not all equal, and as such we focus on those defined as *General Purpose Game Engine (GPGE)s*, which we define as *frameworks for the creation of virtual environments*, which is a fundamental

[2]Available at `https://kitware.github.io/vtk-examples/site/Cxx/VisualizationAlgorithms/ContourQuadric/`.
[3]`https://vtk.org/Wiki/VTK/Wrappers`

Figure 2.2: HTC Vive HMD with cameras and controllers.

component of games. This also means that these software are not restricted to game development purposes, but are used for a variety of projects that entail the simulation of virtual worlds. These frameworks perfectly fit our requirement of creating an immersive and interactive environment for our VR IDE.

A number of GPGEs exist, such as Unity Engine [18] and Unreal Engine [26]. We use Unity as the engine on which we base our solution. This is ideal for our research as previous research already exists discussing the integration of VTK and this engine [2]. Furthermore, it is a very well established, used and supported engine.

We chose to use Unity due to the number of advantages that come with the engine. First and foremost, Unity has already a decent VR integration with the OpenVR[4] and OpenXR[5] plugins. The VR community is already quite established and, in general, the development community is very active and offers valuable support.

The documentation for the engine is comprehensive and freely available with a number of examples that facilitate development, and its API supports the implementation of custom native C++ code through easy drag-n-drop of the shared libraries in the editor and an intuitive interface to import its functionalities in C#. Finally, thanks to OpenGL 2 context sharing, it is possible to render through external code, such as VTK, which suites our need to visualize objects created in native C++ libraries.

2.4 HTC Vive Head-mounted Display

Our objective is to have a generic environment that can be used with any HMD. For our development, however, we focus on the HTC Vive series as these are the HMDs we have access for development and testing. These VR headsets have a 90 Hz refresh rate on the entire line and can be used with controllers, allowing for interactive usage. In particular, we focus on the HTC Vive, that we use for both development and testing. This headset is shown in Figure 2.2.

2.5 Related Work

A number of projects already attempted interfacing visualization libraries with frameworks, be it GPGEs or IDEs, while others attempted the creation of visualization editors, albeit for different purposes and/or constraints. In this section we discuss these previous attempts and how they contribute to laying the ground for our development.

OculusVTK Visual editor was developed in 2016, is a similar attempt to create a visual editor for OculusVTK carried out in Python [1]. The solution allows the user to create simple pipelines and

[4] `https://github.com/ValveSoftware/unity-xr-plugin`
[5] `https://docs.unity3d.com/Packages/com.unity.xr.openxr@0.1/manual/index.html`

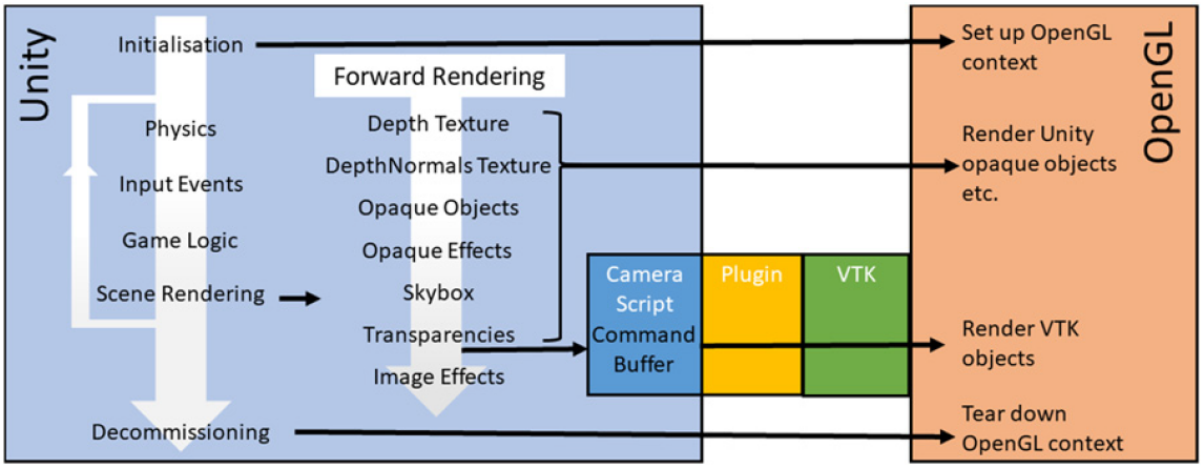


Figure 2.3: Wheeler et al.'s rendering architecture [2].

manipulate them within a visual environment and see the result of the operations. The authors carry out a number of experiments with pipelines, varying from simple arrows to stream tracers and a DICOM image rendering, where results showed a consistent 58-60 FPS were achieved.

While the objectives of the authors are quite similar to ours, they do not focus on the same constraints we set for our project, i.e. performance and the designing of a distributable and parallelizable architecture. Furthermore, their solution focuses on Oculus, whereas our solution aims at being headset-agnostic.

The power of the solution proposed lays in its usage of Python's capability of introspecting on its modules and harvesting data in order to achieve a more general approach to interfacing with VTK. We base our solution partly on this software, as we discuss in Section 4.3.1.

On the other hand, this solution is limited to the usage of the OculusVTK C++ boilerplate for OculusVR. The code is maintained by SURFsara and offers an extension of the OculusVR SDK for Oculus HMDs. Compared to the support of Unity, this limits the generality of the solution, which we aim at extending.

Also, due to the usage of OculusVTK, the solution proposes a complex building system prone to errors and issues related to building platforms, versions and library installations. Our objective is to streamline the building of our solution, if any is needed, and make our solution version-agnostic towards its dependencies.

Furthermore, the Python scripts are limited to only working through introspection on `vtkAlgorithm` subclasses, which limits the user to using such powerful tool on a subset of all available features of VTK. We aim at a general solution, thus either extending the approach up to any `vtkObjectBase` subclass or to allow access to other features through the offered API.

VtkToUnity was presented 2018 by Wheeler et al.
[2], who developed a Native Unity plugin that allowed the user to exploit VTK's features in the engine's scripts[6]. The plugin's aim is to introduce volume rendering through VTK and uses the OpenGL context sharing technology to enable the visualization library to directly render within the Unity scene. The plugin's approach is to add a rendering callback function after the transparency rendering stage, as optimal for volume rendering. The produced architecture is summarized in Figure 2.3.

The authors carry out experiments to evaluate the performances of the solution. Their objective frame rate is of 90 FPS using a HTC Vive Head-Mounted Display (HMD), which aligns with Unity's own guidelines [20] as well as the 90Hz refresh rate of the HMD [27]. With simpler renderings the solution is of its own able to achieve the desired frame rate, but with more complex scenes it does not. To overcome this, the authors propose the usage of setting the desired frame rate using VTK's own `vtkExternalRenderingOpenGLRenderWindow::SetDesiredUpdateRate` to solve this.

[6] `https://gitlab.com/3dheart_public/vtktounity`

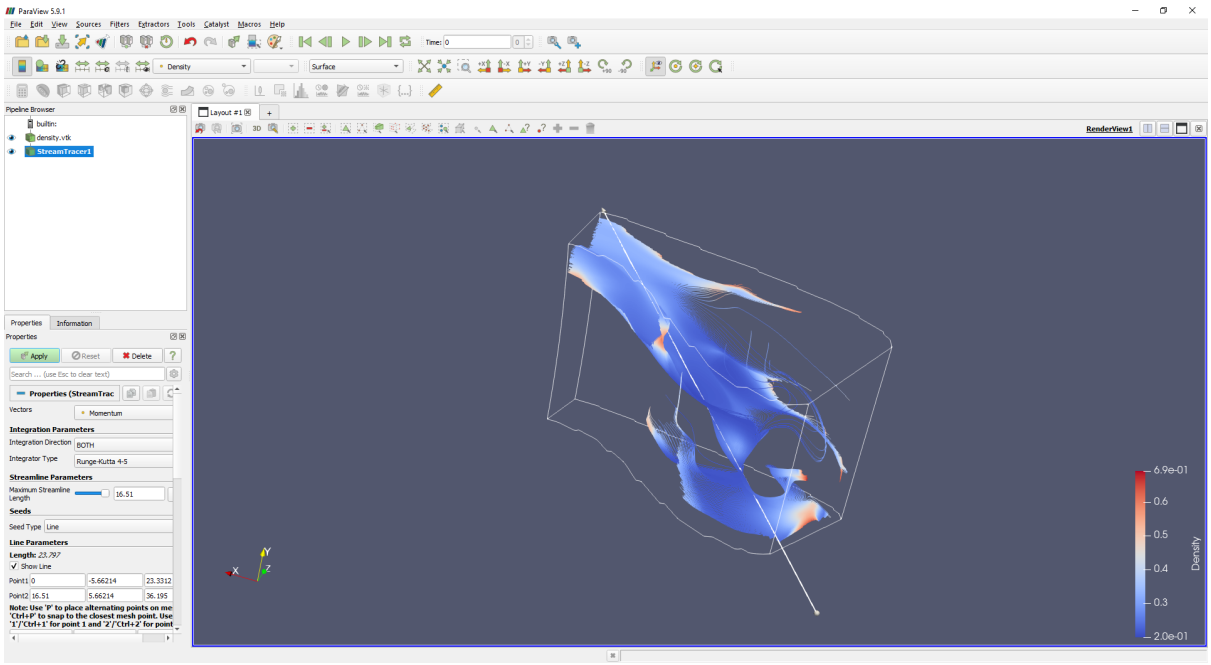


Figure 2.4: ParaView visualization of a StreamTracer using a structured grid point dataset.

This solution presents some limitations, as the C++ native code locks the shared API at every call, and as such limits the ability to leverage parallelization of the software, which could potentially aid the rendering quality while the objective frame rate is set. Furthermore, the plugin focuses on volume rendering in particular, while our solution aims at a more general solution.

An issue with the maintainability of the VtkToUnity plugin is its hard-coding of features and reliance on patches to VTK's code to make it work correctly. For a specific application this is not a problem, but as our objective is to offer a generic solution to access VTK from Unity, this is not viable, as it introduces couplings to versions and particular algorithms and pipelines, and the potential for bugs and clones.

Our solution aims at creating a bridge that would allow the user to create pipelines in Unity without having hard-coded features limiting the user capabilities.

ActiViz is a VTK wrapper for .NET[7], developed by Kitware, the company that is also maintaining VTK. This wrapper is ready to be used with Unity, and exposes the same functionality of the library as other wrappers. This allows already for a good integration of the library and Unity.

The downside of this solution is its performance: as the wrapper requires CPU to GPU copies, with complex visualizations the performances drop significantly. This has already been discussed by Wheeler et al. [2] and is one of the reasons that lead them to develop their own solution. On the other side, this is an official wrapper maintained by Kitware, the same creators of VTK, and it has already an integration for Unity.

ParaView is an IDE for the creation of visualizations mainly based on VTK[8], also developed by Kitware. This uses the same paradigm as VTK for the creation of visualizations, that is by the means of pipelines. An example of a ParaView visualization can be seen in Figure 2.4. The software supports a VR porting for the visualizations which feels clunky, as it is a rendering of 2D UIs in a 3D immersive and interactive environment, which hinders user experience.

Of the previous solutions, only Dreuning's work addresses the issue of allowing the creation of an integrated development environment in VR [1]. This is not an issue in and of itself, but it makes for a cumbersome and time-consuming development environment for VR applications. The solutions all miss

[7] https://www.kitware.eu/activiz/
[8] https://www.paraview.org/

one of the key characteristics we envision for our environment, be it the performances necessary for VR applications, the generality of an IDE or the support for multiple HMDs.

Chapter 3

Requirements and Challenges

After analyzing the different approaches taken by previous work, we define a number of requirements and challenges our solution has to meet and overcome. Most of these generate complexity in the system, and make for potential issues for long term support. For this reason, we first discuss the requirements that drive our choices when facing the challenges this project presents, and finally we validate the decisions we take to overcome these issues.

3.1 Requirements

In order to develop the environment we envision, we first define the requirements our software meets. We present them under four umbrella terms representing how they impact our choices and process. Under these umbrellas, we explain how in particular these impact our solution, and define the necessary terminology we use.

3.1.1 Usability

As first and foremost, Usability is the key requirement of our environment. As one of our objectives is to lower the technical barrier to developing VR applications, the software we present needs to be crafted with particular care to its usability. The issue with this, is the difficulty we have defining what usability is in the first place. A number of definitions have been proposed in the past, though scope of the term is yet to be determined [28]. Thus, in order to avoid ambiguity, we refer to the term as defined in the ISO/IEC 25010:2011:

> **Usability** Degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use [29].

As we develop our environment, we consider two levels of usability of our software. The first level, or high level, refers to the usability as seen from the end-user who has none or low technical knowledge and needs our environment to develop his application, thus referring to the UI of the environment. The second level, or low level, refers to the usability as seen from a skilled user who has technical knowledge and aims at modifying or expanding the environment, or use it with more freedom, and thus develops also with code within our environment.

As such, to cater to the high level usability, the software should be usable out of the box. In particular, the UI should be intuitive and easy to use. For this reason, the use of Heads-up Displays (HUD) within the VR environment should be limited, to avoid cluttering the user's viewport, and the tools should be part of the environment, interactable and movable, in order to make all the environment's objects tools that the user can use in similar way. This also limits the number of gestures and inputs the user has to memorize, as well as having adaptable and customizable configurations of what information the UI should show and where this is available, making the experience tailored to the user.

However, as our main focus is the engine, we also define the low level requirements, as this remains the main access-point to VTK used in the environment. In particular, the API of our software should

be as expressive as possible, meaning the user can write any software the usage of just VTK would allow them. Furthermore, the information inserted in the API should be as explicit as possible, limiting black box operations the user is not informed of, and limiting the need for further documentation. For example, the types in which the parameters are expressed in generic calls should be explicitly specified in the function calls, with constants available to the user that represent these types, without them having to query the documentation in order to know how this representation works.

3.1.2 Generality

To lower the coupling between our software and VTK, we want it to be version-agnostic. This coupling is already weakened by the fact the library is developed to be backwards-compatible and the only major change to the API that broke this, happened between versions 4 and 6 of the library[1]. In order to make the software version-agnostic, the access to the library should be general, i.e. our solution should limit direct calls to VTK's API, preferring adaptable solutions, e.g. introspection, adapter generation, etc.

As usability is also one of the requirements of our software, the benefits of a general access approach are not limited to its maintainability, and generality as a whole also impacts the usability of the environment, as the other way around is true. Referring to the high level usability, we presented an example referring to the UI of the environment. Such UI was also described as general, as it would be composed of environment objects that could be rearranged by the user to customize its own workstation. We mirror this approach of customizability and generality to all the aspects of our solution.

This is all the more critical when looking at the way we interface with VTK. Hard-coding its features would possibly make for the best performances, but would make the system a huge collection of functions, as it would be necessary to wrap the calls for every class and method accessible to the user. Making the access to VTK general would both boost its version-agnosticism and its maintainability.

Expanding on the UI, the system should be able to show a UI for any type of operation the user can execute with VTK. The user should not be responsible for implementing UI components for VTK features that are not foreseen in our code, and as such the way the UI is handled and generated should also be general, it should be able to generate necessary components on-the-fly. This is also part of the full access we just presented in the previous paragraph. In terms of usability, full access makes the environment usable at low level, whereas UI generation at high level.

3.1.3 Maintainability

In order to have a maintainable system, we have to take into consideration a number of metrics and apply conventions that make our solution more robust and enable long-term support. We focus on four characteristics, some of which we want to minimize, some maximize, that we see fit to achieve these two objectives. We aim at minimizing Source Lines of Code (SLOCs), that measure the size of the code base in lines of source code, and coupling with external software, and at maximizing separation of concerns and readability.

The SLOCs have a strong connection to the maintainability of the codebase and as such we aim at keeping its value low [30]. We aim at achieving this by limiting code clones and re-using code wherever possible. Based on the Software Improvement Group's Maintainability Model, we aim at a codebase of less than 322 thousand SLOCs[2].

Another metric we aim at keeping low is coupling. In particular, we aim to keep the coupling level loose by having data coupling with external libraries and reducing the number of libraries necessary [31]. Furthermore, we aim at lowering the impact of these couplings by using the Facade Pattern and keeping the coupling at the edges of the software [32]. The internal coupling of the system is necessarily common-environment coupling as data has to be accessed directly from different modules, i.e. for efficient rendering and updating purposes.

In order to keep the software maintainable, it should be easy for a developer to clearly understand the code. For this reason, we use some coding conventions in order to make the code more comprehensible.

[1] `https://vtk.org/Wiki/VTK/VTK_6_Migration/Overview`.

[2] Based on V13 (May 26, 2021) Evaluation criteria `https://www.softwareimprovementgroup.com/methodologies/iso-iec-25010-2011-standard/`.

First and foremost, each function should come with a comment clearly specifying the responsibility of the function and what it should be used for. Moreover, the size of each function should be as short, and its responsibilities as concise as possible.

Finally, function and methods should cater to as little and concise responsibilities as possible, being focused on very few things. This may clash with our aim at keeping the SLOCs count low, but it balances of the complexity introduced in small yet packed functions that enact a lot of responsibilities. This may also create interface duplication, as multiple layers expose similar functionality but wrapped in further decorating information. For this reason we introduce a convention on how the interfaces should be specified.

3.1.4 Performance

Our final requirement for our solution is that it is performant. This is particularly important as for a decent user experience with VR applications, these should aim at a stable FPS that mirrors the HMD's refresh rate. In general, the target should be 90 FPS as most HMDs have a 90 Hz refresh rate[3] [20]. Another aspect of performance is memory usage. As VTK objects may become memory-heavy, we should limit how much overhead we introduce on the memory. However, this is a soft requirement, as it may not be possible to achieve decent time performances without keeping a lot of information readily available, and time performance is more critical for user experience.

However, hard-coding VTK's features would make for the best performances, but would make the system plugin a huge collection of functions, as it would be necessary to wrap the calls for every class and method accessible to the user, which is hardly what we aim at. As such, we must accept a compromise between the two, however the objective of a stable 90 FPS is critical for this project.

3.2 Challenges

Given the requirements of the system we aim to create, there are a number of challenges we must face before implementing the software. These are mostly related to its design and technology choices, and as such we will discuss them separately from the actual implementation. These challenges mainly relate to the issue of connecting VTK and Unity and to the ability to guarantee full access to VTK from Unity while upholding version-agnosticism.

3.2.1 Unity Integration

The foremost factor of complexity and maintainability issues is the integration of the system with the Unity Engine. Integrating two rendering components is challenging in two main ways. First to tackle is the issue of sharing resources and the rendering context, as taken separately Unity and VTK have both their own objects, memory spaces and rendering contexts.

To simplify the issue, we can make our solution a part of the Unity environment we are developing by creating a Unity native plugin. These are libraries written in C, C++ or Objective-C that are not constrained by the Unity environment like C# Managed plugins [33]. By making our solution a Unity native plugin, we now have to solve the problem of sharing data between the rendering contexts of Unity and VTK.

As we are taking a similar approach to Wheeler et al.'s, we also follow their solution to solve this issue. In VtkToUnity the sharing of resources between the two rendering contexts is achieved using OpenGL Core ES technology, as it allows for sharing some objects between OpenGL Core conforming contexts [2]. As the specification of these contexts is forward compatible, we also do not need to worry about compatibility between versions [34].

We use as base for our software Wheeler's VtkToUnity plugin, as such we use the same system they developed for sharing data between rendering context and for integrating the event loops of Unity and VTK. This second issue was also tackled by Wheeler et al. by nesting the VTK event loop iteration

[3]Some HMDs may come in varying refresh rates, a list of devices with relative specifications is available `https://en.wikipedia.org/wiki/Comparison_of_virtual_reality_headsets`.

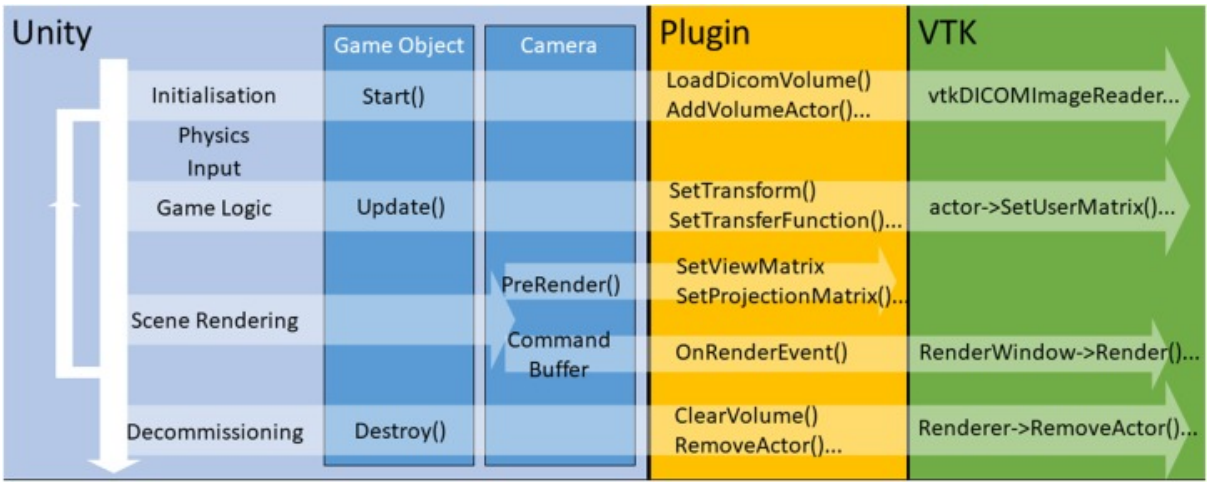


Figure 3.1: Wheeler et al.'s script handling of the VTK event loop [2].

in Unity's using the engine's event queue event handlers and graphics callbacks [2]. The obtained architecture for the plugin is shown in Figure 2.3. Most of the functionality handling these loops is implemented in the C# scripts of Unity, which call the correct native function from the plugin in order to trigger the corresponding event in VTK. A zoom in of the architectural specification can be seen in Figure 3.1 where the particular calls from the scripts is highlighted and what is called in VTK as a consequence for a particular Unity scene developed by Wheeler et al. [2].

3.2.2 VTK Completeness

As already introduced, we aim at fully exposing VTK's functionality to the Unity environment. More formally, we define VTK completeness as the characteristic of an environment that is able to access the full extent of the VTK library, whether it was or not built with its optional modules, and can produce the same visualizations as a native C++ application using VTK would be able to.

In order to achieve such a property, we either generate adapters for all possible software components of VTK that are to be exposed, or we design a system that is able to generically access this information at runtime. As the first makes the system more coupled to the version of VTK with which it is built, and as such adapters are to be generated every time the system is built, it hardly fits our requirements. Thus, we choose to use introspection as our access system to the features of VTK.

As our system uses a C++ plugin to integrate VTK and Unity, and this language does not present an introspective interface, we need to give it access to this feature. The main issue of trying to introduce introspection capabilities to C++ is its memory handling. In particular, non-intrusive solutions that do not require modifications to the compiler generate memory overheads [35] and/or do not present complete introspection systems, as constructs such as `union` cannot be easily handled [36].

Using solutions that would modify the C++ compiler are also not of our interest as they introduce a very tight coupling with the modifications in place, and potential for unexpected breaking points down the development process that would not be easily traceable back to these modifications. For these reasons we see these solutions as non-starters for our project. Thus, we need to introduce introspection through a further module.

Our approach aims to integrate Dreuning's Python introspection scripts [1] within a C++ native plugin, as this would still allow for a performing system written in C++ to access Python's capabilities when it is necessary to expose parts of VTK to the system. The solution presented by Dreuning is able to achieve decent performances in Python, and has the characteristics we need.

3.2.3 Integrating C++ and Python

Introducing these Python scripts within the C++ plugin can be achieved through two different approaches: keeping the Python component separate and make it communicate with the C++ native

plugin or embedding the Python interpreter within the C++ code itself.

The option of synchronizing different modules would be best for maintainability purposes, as it would maximize separation of concerns, as well as isolating modules and reducing coupling, as they would need to synchronize data at worst, whereas the embedding of the Python interpreter would make the coupling tighter. This being said, the redundancy of data created introduces a memory overhead that is not negligible. This overhead is necessary as the VTK objects would need to be accessible from both components, and as separate modules they would need to synchronize them.

With the embedding, the issue would be to share the memory area of the VTK objects of the C++ plugin with the Python interpreter. This is made easy by VTK though, as the wrapped Python objects keep a reference to the C++ objects they wrap. This allows us to embed the interpreter within the program's memory area and avoids us to have to manually manage the sharing of objects between the two parts. As such, we opt to embed the Python interpreter as it makes for the best performances and does not impair maintainability to a significant degree.

3.2.4 Python/C API

As the developers of Python already foresaw a use for Python tightly coupled with other languages, as well as the possibility for users to extend the language, the interpreter can be embedded using natively supported calls that are part of what is called the Python/C API [37]. This API gives access to the ability to instantiate a Python interpreter within a C/C++ software that shares with the interpreter its memory, in order to allow both to access data in a fast and controlled manner.

However, this API is verbose, especially while working with objects, which made us consider the usage of a third party library called Boost::Python, which extends the Python/C API, further simplifying the embedding of the interpreter. The most useful feature which would immensely aid the maintainability and readability of our code is its automatic handling of the Python reference count of objects.

In order for Python's garbage collector to properly dispose of an object, the language decorates each instance with a reference count, which is a counter that keeps track of how many variables are referencing the object. Once this counter reaches zero, the garbage collector disposes of it and frees its memory. The Python/C API leaves the handling of these counters to the user through the usage of `Py_DECREF` and `Py_INCREF` macros.

While this freedom allows for more sophisticated uses of the interface, it also makes the code more complex to read, making the codebase larger in terms of SLOCs and more complex as it requires controlling references and being careful not to create memory leaks. This is made more complicated as some functions of the API increase the reference count while others *steal* its reference from the caller, making the system prone to memory leaks. This is somewhat helped by the introduction of the `Py_XDECREF` macro which decreases if not null or already zero.

On the other hand, while Boost::Python aids in making the code more readable and maintainable, it also introduces a further coupling with a third party library, which comprises a multitude of functionality that is not required for our software, introducing breaking points. Furthermore, the structures wrapped around `PyObject`, the controls and the exception handling system introduced by the library results in overhead on both memory and speed of the execution, which is not ideal in our performance-critical environment.

For even further support, VTK ships with a module to facilitate Python/C++ integrations exposing functions that easily wrap C++ and unwrap Python VTK objects, as each wrapped Python object keeps a reference to the C++ object, and from the pointer to the object the wrapper can be generated.

As a proof of concept, to showcase the capabilities of our system, we do not use Boost::Python, so to limit external coupling and overheads and because VTK already offers an integration of the API that makes the usage of the Python/C API easier. We recognize though that the advantages of the library are not trivial, and it should be explored as a potential option for future updates to the project.

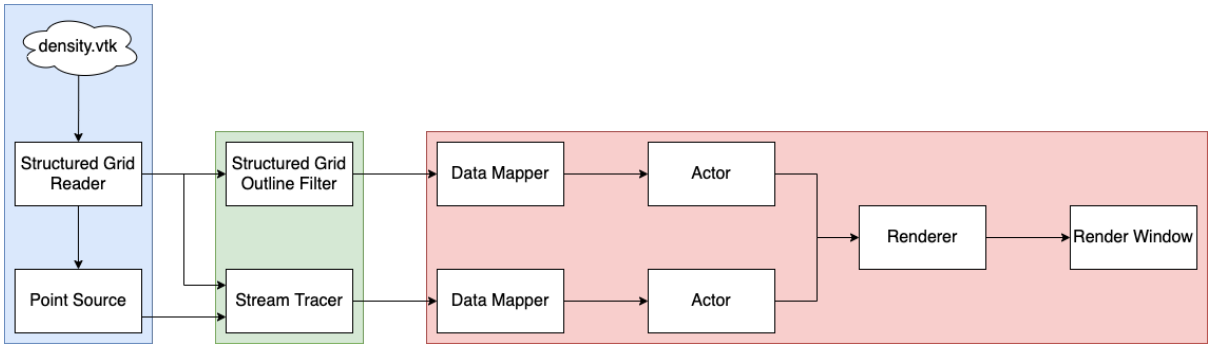


Figure 3.2: VTK pipeline of a Stream Tracer on a density dataset.

3.3 Validation

Before discussing design and implementation of our plugin, we validate our choices to tackle the challenges we face in this project. Because we base our plugin on VtkToUnity, the integration with Unity through the usage of a native plugin is already validated by their results, achieving the recommended performance of 90 FPS. On the other hand, the validation of the Python embedding choice is not available, and as such we develop a small test confirming our choice.

3.3.1 Python Embedding tests

On the other hand, the embedding of the Python interpreter is a common choice and is discussed in the official guidelines of Python itself[4] and can achieve decent performances. In order to determine the overhead introduced by the embedding, we compare the execution times of an embedded application with the same application in pure Python and repeat this for direct access to VTK and using the Dreuning's introspection scripts.

The application we run is a stream tracer of a density dataset for VTK applications. The final visualization is composed of the outline and the streamlines. The pipeline can be seen in Figure 3.2. This pipeline has been implemented in Python 3.7 directly using the VTK library and accessing it through a modified and expanded version of Dreuning's scripts, the embedded version of the Python interpreter in C++ executing Python calls directly to VTK, and finally again through C++ by calling the functionality exposed by the modified Dreuning's scripts. The code for each of these tests is presented in Appendix C and the results are shown in Table 3.1.

As shown in by the results, the best solution for overall execution would be a native written Python software. However, stripping the execution time of the one-time loading and finalizing delays, the execution of the native and introspective versions of the algorithm take virtually the same time. What is more surprising, is the 50.68% faster execution of the C++ introspective application using an embedded Python interpreter. The native C++ version using the embedded Python interpreter to call VTK natively in Python shows the best performances as the only real time spent executing code are the Python interpreter's loading and finalizing.

As such, to both accommodate our requirement of generality and performance, we see the embedding of the Python interpreter into the C++ native plugin for Unity and leveraging of the Python introspective capabilities as a good candidate for our implementation. However, taken into account the performances that C++ can achieve, and to further give expert users the possibility to leverage the languages strong suits, we incorporate in our design of the system a component that allows the user to write custom code that can be easily added to the plugin to run code natively.

[4] `https://docs.python.org/3/extending/embedding.html`

Table 3.1: Performance tests' results using Python and C++ with and without introspection, implementing the pipeline from Figure 3.2. Each benchmark has been run 1000 times.
*** Time required to load the necessary structures for embedding Python and using the Introspector.**
**** Cumulative times of the specified operation type.**

(a) Python 3.7.8 implementation accessing directly VTK resources and methods.

	Mean	std	min	50%	max
Loading*	0	0	0	0	0
Instantiation**	0.346051	0.014965	0.316300	0.343050	0.435700
Update	525.841453	7.759079	510.060500	525.217500	616.136600
Setting**	0.032811	0.001585	0.030600	0.032500	0.058100
Getting**	0.214747	0.012234	0.201500	0.209500	0.304900
Connecting**	0.026682	0.002228	0.022300	0.026300	0.044900
Finalizing*	0	0	0	0	0
Total execution time	526.718930	7.764707	510.964400	526.091400	617.007400
VTK execution time	526.718930	7.764707	510.964400	526.091400	617.007400

(b) Python 3.7.8 implementation accessing VTK through introspection.

	Mean	std	min	50%	max
Loading*	1890.429543	19.276412	1847.309400	1887.602000	2059.148200
Instantiation**	0.732206	0.030549	0.684000	0.726600	1.064900
Update	527.823380	9.644450	512.147300	526.707950	652.077100
Setting**	0.068662	0.003141	0.064600	0.068100	0.110400
Getting**	0.220781	0.012619	0.208500	0.215800	0.341100
Connecting**	0.026044	0.003457	0.022300	0.024800	0.053600
Finalizing*	0	0	0	0	0
Total execution time	2421.561489	23.739328	2362.667700	2417.877550	2611.760000
VTK execution time	531.131946	9.659824	515.358300	529.996600	655.820200

(c) C++11 implementation embedding Python accessing directly VTK resources and methods.

	Mean	std	min	50%	max
Loading*	336.828093	8.202800	322.595000	334.397500	399.382000
Instantiation**	0.298408	0.015259	0.264000	0.298000	0.412600
Update	0.003026	0.000261	0.002600	0.003000	0.007400
Setting**	0.033670	0.004690	0.030800	0.033400	0.135800
Getting**	0.003260	0.000307	0.002500	0.003200	0.005100
Connecting**	0.005054	0.000793	0.004200	0.005000	0.026200
Finalizing*	17.801718	0.535997	16.879800	17.667700	21.718900
Total execution time	355.044973	8.151684	341.223000	352.565000	421.322000
VTK execution time	0.415162	0.020545	0.370700	0.415450	0.548600

(d) C++11 implementation embedding Python access to VTK through introspection.

	Mean	std	min	50%	max
Loading*	2247.011260	28.316549	2197.630000	2241.280000	2522.560000
Instantiation**	0.666370	0.033469	0.614600	0.655350	0.966600
Update	267.402500	4.770508	258.869000	266.831000	316.591000
Setting**	0.077507	0.004178	0.070100	0.076400	0.119000
Getting**	0.279808	0.017314	0.255800	0.274200	0.447800
Connecting**	0.047544	0.006922	0.036400	0.046900	0.187700
Finalizing*	23.925579	0.989636	23.330900	23.770100	45.308700
Total execution time	2539.422550	30.844464	2488.160000	2533.210000	2825.620000
VTK execution time	268.485711	4.784796	259.920400	267.901450	317.774000

Chapter 4

Design

This chapter introduces the architecture we developed based on the discussion of the requirements and challenges from Chapter 3. We discuss in detail the implementation of the core components that make the backend of our solution, as the most critical challenge we aim to tackle is to achieve the required performances to make our software usable. We will also discuss the frontend's design, in order to complete our discussion of the architecture, but its implementation and related challenges are out of the scope of this thesis.

4.1 Components

Based on our discussion of the challenges in Chapter 3, we opt for an architecture with components that each have a precise separate concern [38]. This stems from maintainability driven considerations arising from previous work's code, where components were highly coupled with both the visualization library and the engine [1, 15, 39]. On the other hand, VtkToUnity already implements its code separating the Unity interface from the VTK and visualization logic within the plugin, moving most of its complexity in the C# scripts.

The most basic design of our architecture requires two main endpoints, that are VTK and Unity, a native layer of interfacing between the two and a managed Unity plugin to manage the invocation of native code from managed level and to implement the VR app's logic. The bare-bones implementation was developed by Wheeler et al. [2], on which our solution is based. This gives us an infrastructural basement on which our further components are then built.

To achieve our goal of full integration of VTK, we cannot rely on hard-coded functions, which would impair the generality of the system, as well as its maintainability. As such, our solution comprises an introspecting component that enables the gathering of metadata on VTK.

On the other hand, the expression of such metadata also needs to be lowly coupled with the particular implementation of VTK, and as such a component is introduced to enable the generation on-the-fly of UI that fits the I/O operations required to support the development environment. The design of the UI parts themselves is out of the scope of this thesis; we will only discuss its design, which enables the creation of the UIs.

Finally, this generality may come at the price of performance as introspection may be slow and interfacing may not be complete through it, as we discuss in Section 4.3.1, and the UI generation may be slow when working with big and complex pipeline filters. A first experiment in Section 3.3 shows that the performance on the creation of a somewhat complex pipeline using introspection, which yielded promising results. However, these performances may not be enough in all cases to support the use we envision for this software. As such, to enable users to enhance their experiences, both tailored, more user-friendly UIs and focused and efficient adapters that create a direct link between the infrastructure and VTK features are introduced in the architecture.

The final architecture is composed of a total of six components, as shown in Figure 4.1:

- An **Infrastructure Layer** which acts as bridge between VTK and Unity. It implements the Unity

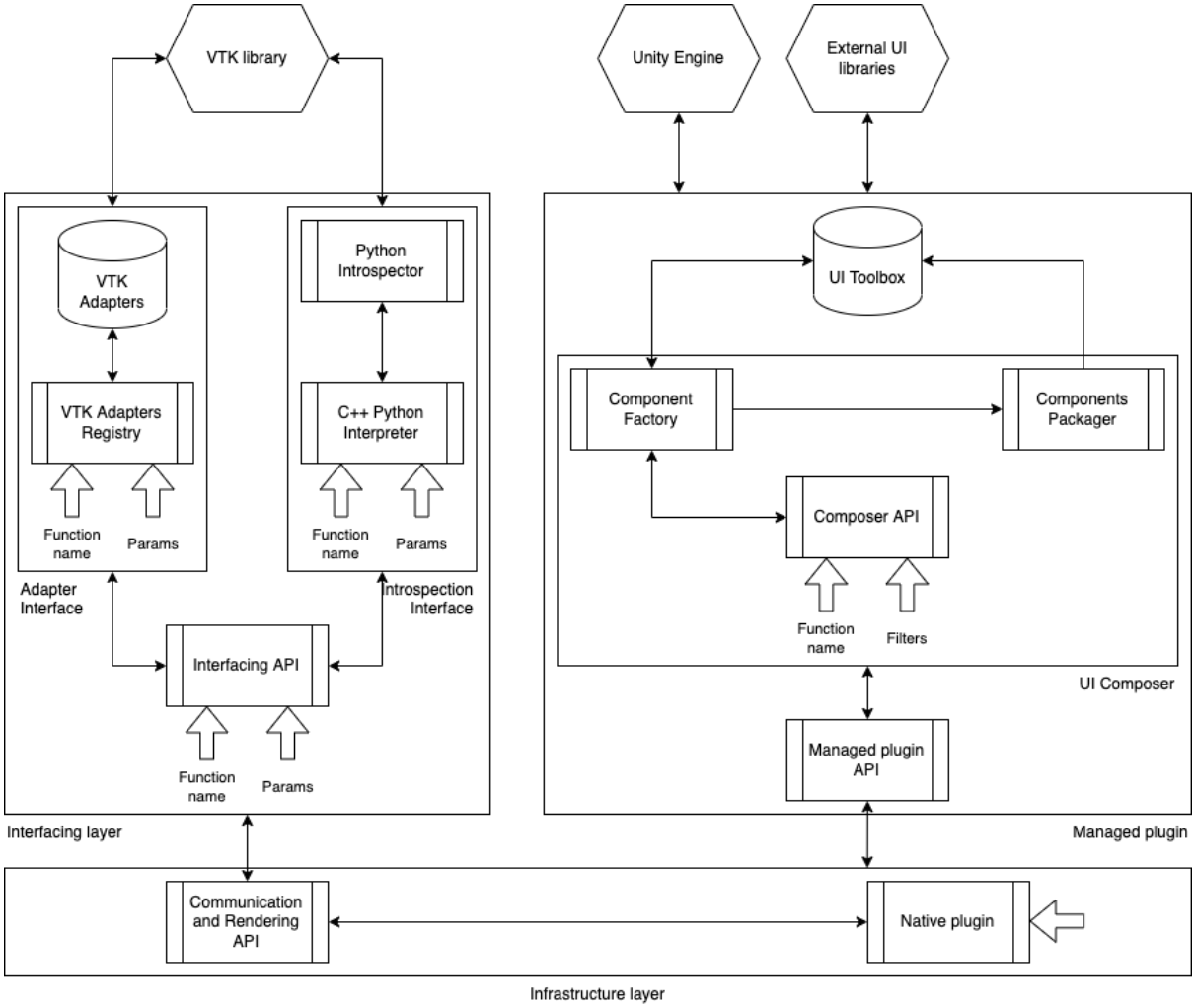


Figure 4.1: The architecture design of the plugin, showing inner working details of the various components. The arrows annotate the information flow between the parts.

native API and the rendering of the VTK objects;

- An **Adapter Interface** which allows users to load custom-written code for VTK and makes it usable from within Unity. It is the Tier 1 **Interfacing Layer**, and will be queried first when calls to the plugin are made;
- An **Introspection Interface** loads the VTK Python plugin at load-time and allows the user to access it whenever Tier 1 fails to find a routine to accomplish requests, acting as Tier 2 of the **Interfacing Layer**;
- A **Managed plugin** which allows Unity to access the functionality exposed by the *Infrastructure Layer*;
- A **UI Toolbox** which maintains a number of UI prefabs used to allow user interaction with VTK objects. This component is part of the **Managed plugin**;
- A **UI Composer** which accesses the *UI Toolbox*'s prefabs to reactively show appropriate UIs upon user interaction with VTK objects. This component is part of the **Managed plugin**.

4.2 Infrastructure

The first component we tackle is the simplest yet the most crucial of this plugin. The infrastructure layer acts as a dispatcher for the requests coming from the managed plugin, routing them either to the rendering API or further down towards the VTK access components. As a reference, the component is

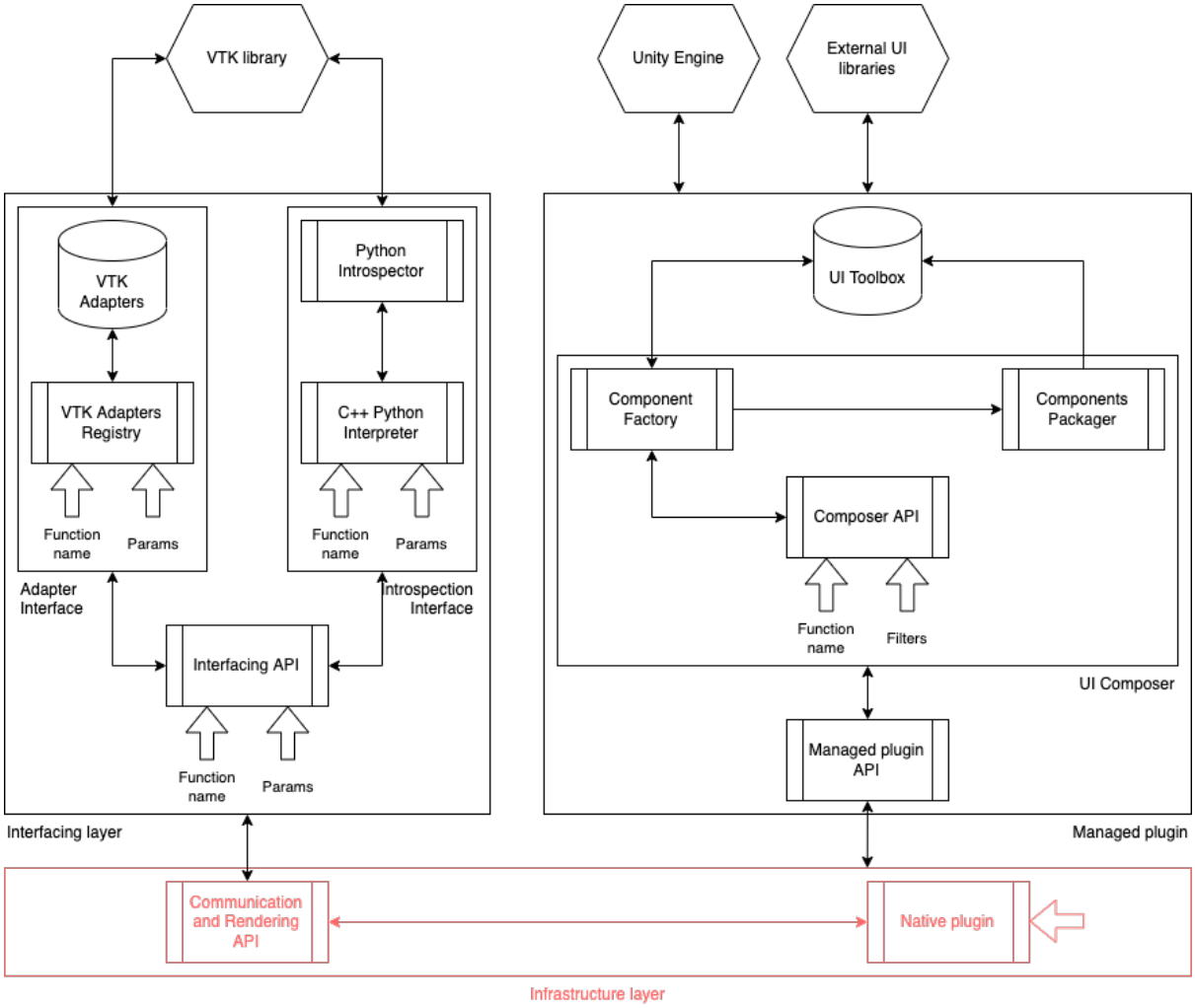


Figure 4.2: Architecture with the infrastructure layer highlighted.

highlighted in the architecture in Figure 4.2.

The infrastructure layer is the component responsible for allowing interaction between all other parts of the system. Its main job is to give a common entry-point from Unity to the VTK interfaces and decide which to call, acting as a dispatcher. As this is the critical point of the system where communication flows, it is also the one that mostly affects parallelization and distribution capabilities. Thus, the calls should be as little blocking as possible, allowing flows to stop only at endpoints.

Wheeler et al.'s VtkToUnity [2] already partly implements this infrastructure layer and already exposes part of the VTK interface as well. Unfortunately, their calls are highly blocking, as once the C++ native code executes, the Plugin calls each `lock()` on the API and thus parallel calls are impossible. On the other hand, the plugin achieves decent performances that fall within the Unity guidelines for VR [20], while allowing for good maintainability and extension possibilities.

As such, we base the infrastructure layer on a refactoring of the VtkToUnity plugin which aims at the following results:

1. Decoupling of the communication and dispatching from the interfacing responsibilities;
2. Refactoring the interfacing code in adapters that constitute the foundation of the adapter-based interfacing;
3. Move as much business logic of the infrastructure in non-blocking calls, preferably with no blocking at the infrastructure layer;
4. Move blocking logic at interface level.

We use the same approach as in the VtkToUnity plugin to create the C++ native plugin and its interfacing with Unity. Our implementation is then reduced to the minimal API necessary to use the VTK functionality expressed by the interface mechanisms described in Section 4.3. Our objective is to make the calls as general as possible, giving the user the most liberty and access as possible. For the naming and implementation, we follow a similar design as the Python/C API [37]. The interface is documented in Appendix D.

In order to keep the readability and intuitiveness of the system decent, we apply this same approach to the API of all components. This is to limit the parsing operations, so that middleware has to apply as little transformation to the parameters as possible, as well as to make the maintainability of the system easier, as understanding data flows and meaning of values becomes easier with consistent naming and patterns.

4.3 Interfacing layer

The core feature of our system is its ability to access VTK and expose it fully to the VR development environment. To achieve this, while keeping the system easily maintainable and performing, we use a two-way access system. The first access route uses a lookup to check whether custom adapters exist that can satisfy the request from the caller, while the second uses introspection capabilities on the library to find the appropriate method, function or class that can satisfy the request.

In order to limit the amount of routes in the infrastructure level and overheads, the interfaces follow a strict prioritization policy. Tier 1 is the Adapter Interface, where a number of custom written adapters can be added and used by the plugin. If the infrastructure finds a suitable adapter, it will use it to answer the request. If no adapter is available, the query will be sent to Tier 2, the Introspection Interface, to be computed. However, if an adapter is found, but it cannot satisfy the request, the system will not query the introspection interface. Instead, it will return an error, and it will be the user's responsibility to either query the introspection interface directly or to expand the adapter. Other than the performance rationale behind this choice, the other main reason is that complexity in our system is kept at the peripheral components, with the clear intent of the dispatcher to be as simple and straight-forward as possible, limiting complexity in that central middleware.

4.3.1 Introspection Interface

The introspection interface is the low priority of the two VTK access tiers. It acts as the main route to connect to VTK and access its features and the most used component of the native part of our plugin for the average use-case of this system. It comprises two main parts, the Python `Introspector` script, which uses the Dreuning implementation to access VTK introspectively, and the C++ wrapper that handles the memory used by the VTK objects and the Python Interpreter embedded in the system. As a reference, the component is highlighted in red in the architecture in Figure 4.3.

Design

As we discussed in Section 3.2, solutions already exist that implement introspective systems to access VTK in Python. In particular, we modify and extend the Python scripts from Dreuning's solution in order to make use of its `ClassTree`. The issue we find with this implementation is that it is limited to `vtkAlgorithm` classes, and as such leaves out a chunk of the features of the VTK. As extending these scripts is out of the scope of this thesis, we instead introduce a feature to overcome this limitation that would nevertheless be implemented, i.e. the ability to pipe the result of a VTK call to other calls. This does not allow the user to instantiate objects of classes that are not `vtkAlgorithm`, however it allows already for a big part of the left out features to be accessed.

From Dreuning's solution we adapt the `ClassTree` implementation in order to strip it of its UI related code, as well as the `PipelineObject`, which we use as the introspective wrapper for the VTK objects we instantiate to hold the getter and setter calls. A further script has been produced to facilitate the calls from C++, exposing the required features for: (a) creating a (wrapped) VTK object and return the C++ object's address; (b) getting the description of a VTK object, i.e. its attributes and their types,

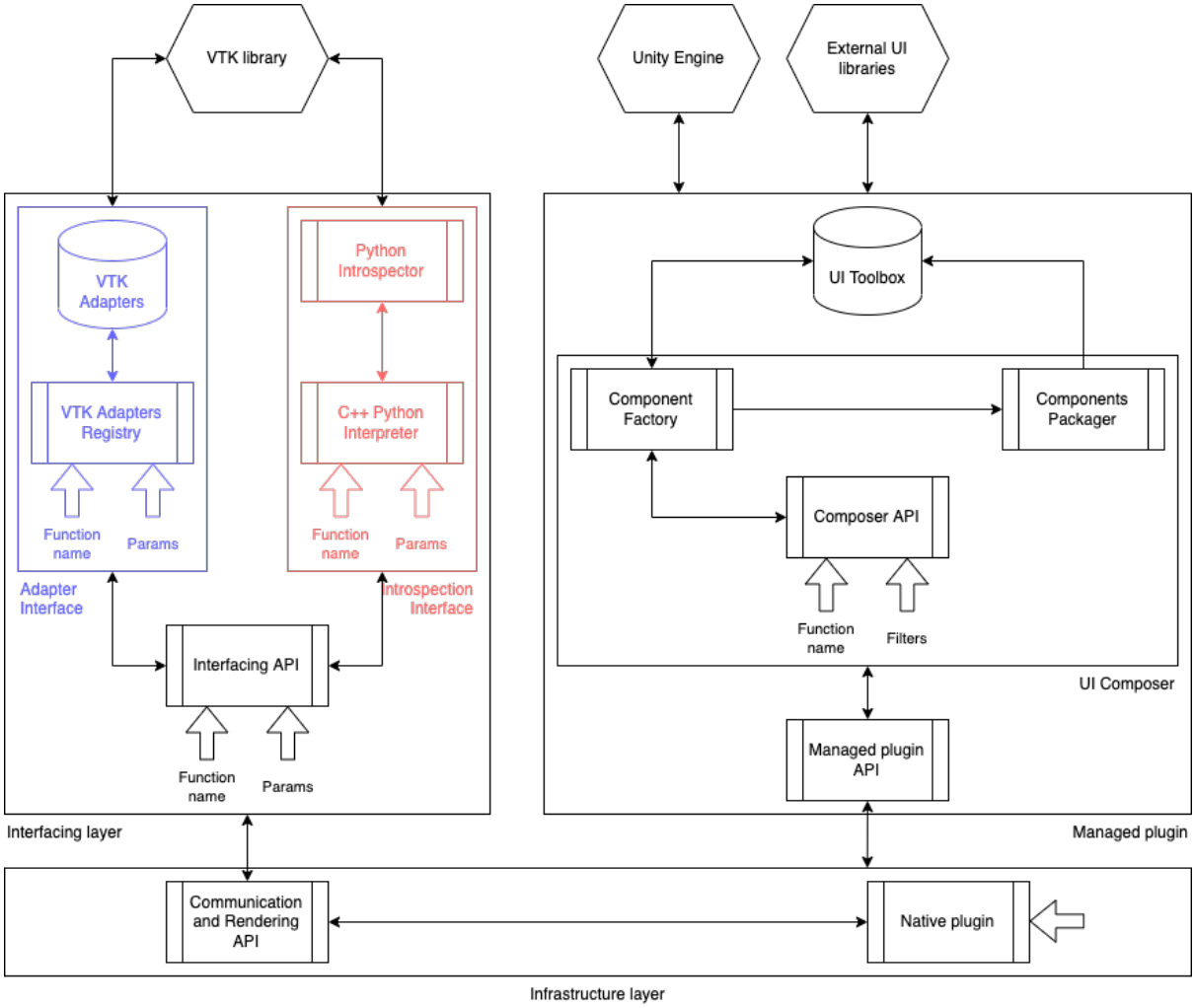


Figure 4.3: Architecture with the introspection interface highlighted in red and the adapters interface in blue.

used by the UI Composer to generate UIs on-the-fly; (c) get the value of a VTK object's attribute; (d) set the value of a VTK object's attribute; and (e) delete the wrapper's information once the VTK object gets deleted in the environment.

In order to access the Python scripts from the C++ native plugin, we embed the Python 3.7 interpreter in the C++ implementation. This is achieved by initializing the interpreter as part of the process, using a subset of the software's memory to run Python. This, combined with the ability to wrap the C++ VTK objects using the `vtkPythonUtils` class into Python and then access them back again, allows us not to worry about synchronizing memory and processes, boosting both memory and time performances.

The implementation instantiates a `Introspector` class from Python, using it as access-point for the introspective methods. Using the `PyObject_CallMethod` we are able to use the methods of the `Introspector`, starting with `createVtkObject` which instantiates a wrapped VTK object, using the class name, that carries the information of the getter and setter methods of the `vtkAlgorithm` class. These wrapped objects, that we will now call *nodes*, are registered in a structure that maps the pointer of the C++ object to its wrapped Python counterpart, for fast lookup access [40]. The C++ interface exposes calls that allow, having the node available, to call getters, setters, to execute the VTK update of the object, retrieve the output port, delete the object, execute a generic or piped call. This last one means that you can execute, for example, `object.GetOutput().GetCenter()` with single access to the `Introspector` and use non-introspectable objects as intermediate values, expanding the possible calls that the plugin can execute.

To avoid ambiguity when calling these methods, and to avoid passing excessive data through the

Symbol	Meaning
O	A VTK object. The value from the Managed plugin is the ID of the object in the infrastructure layer's registry. The introspection interface receives the pointer to the VTK object.
d	A C integer or long value.
f	A C float or double value.
s	A C char or LPCSTR value.
b	A C++ bool value.
%#	is any of the previous symbols, # is the number of values. It represents a tuple of values, e.g.: f3 corresponds to (fff) in the Python/C API.

Table 4.1: Format symbols used in the calls to the plugin.

Unity native interface, the methods that retrieve values from the `Introspector` use a postfix notation to determine whether the call is to return a native value that can easily be parsed to and from a string representation (`_AsString`), an ID of a VTK object (`_AsVtkObject`), or either if the result is to be discarded or the method being called has `void` return type (`_AsVoid`).

When calling any method that uses parameters of generic types, we use a similar convention as the Python/C API so to make the maintainability of the code easier, as these are strictly intertwined and usually used alongside. As such, calls are structured as follows: the first parameter is the pointer to the object on which the method is to be called, the second parameter is the name of the method to be called, or a specifier that acts as a dispatching value (i.e. dispatching either to `SetInputConnection` or `SetSourceConnection`), third is the string format of the following parameters (e.g. a method that has three doubles as parameters would have as format `"fff"`), and finally a vector containing the parameters. Special treatment is reserved for the `"O"` object parameters that are first passed as string representation of integer values, then parsed to integers by the infrastructure layer, which represents the ID of the VTK object in the registry, and finally two vectors are sent to the introspection interface, one with the `vtkObjectBase *` objects and the other with the rest of the parameters.

For a complete description of the formatting values, see Table 4.1, while the interface of the introspection interface is documented in Appendix D.

Benefits and Limitations

The obvious benefit of this implementation is that it allows us to broadly achieve VTK-completeness in our system. The Python component gives the plugin access to the `vtkAlgorithm` classes, and through piped calls to VTK it is possible to implement virtually any visualization a native application could make.

A more subtle benefit of the usage of Python is that it is an interpreted language, and the interpreter being embedded in the C++ code makes it easier to maintain and expand the `Introspector` capabilities. Changes to such parts of the introspection interface would not require the plugin to be re-built every time, whereas any change to the C++ code requires the plugin to be built and deployed in the Unity environment again. Furthermore, Python being a high level language, the actual implementation becomes more readable and understandable, making the component easier to work on.

As already introduced in Section 4.2 on the infrastructure layer, we use a common convention on functions' and parameters' names and meaning, very similar to the Python/C API. Keeping this standard is not only beneficial to the maintainability of the system, as the code is easier to read, but also to its performances as the passage from our C++ to the Python/C API calls and vice versa is straight-forward, requiring limited parsing.

The API proposed does not limit the user's ability to access VTK. Indeed, the calls that are exposed make no assumption on what the objects are used for and how, and they create as little barriers as possible. The main issue is that the calls support only a limited amount of types as parameters, compared to the Python/C API. On the other hand, these are representative of most of the types used within VTK classes and as such we believe this to be a minor limitation and that it does not impair our solution.

Lastly, thanks to the fact that most of the implementation of the C++ introspection interface uses little classes from VTK and mostly classes that tend to be not modified a lot by the VTK developers, like the `vtkObjectBase` class, which is the root of all classes in the library. As such, updating the component to newer versions of VTK requires only the recompilation of the plugin and having the correct version of the Python wrapper installed on the machine.

The main limitation of this implementation is that it is not strictly VTK complete. Any class that is not inheriting from `vtkAlgorithm` is not introspected on. These classes could be made available through generic calls through Python, but this would potentially introduce overheads at runtime that are not acceptable in such a performance critical system. As such, we accept this limitation of the system as it makes for a good compromise between our requirements of VTK completeness and performance. And thanks to the fact this is all achieved through Python, the system could potentially be made strictly VTK complete without touching the C++ code and as such left for future work.

4.3.2 Adapter Interface

Next to the introspection interface, the adapter interface comprises a collection of adapters added by the user. These are a way of allowing users to expand the plugin, taking advantage of the pure C++ performance edge over the C++ and Python introspection interface. Furthermore, it allows the users to introduce "templates" in the plugin, i.e. creating adapters that reduce repetitive and verbose calls to a single call to the C++ plugin. As a reference, the component is highlighted in blue in the architecture in Figure 4.3.

Design

The adapter interface is one of the two main ways in which expert users can tailor their experience with the system. These are pieces of C++ code that can be introduced in the native plugin's adapters folder and be called from the Unity environment. Particular use cases for adapters are:

1. The creation of custom loggers for diagnostics and flow analysis;
2. Implementation of *templates* at native level that enable faster instantiation and define their own parameters and routines;
3. Implementation of custom algorithms that compound VTK features.

The adapters are Singleton instances that define first and foremost a unique string identifying which component they are adapting. Such ID represents the name of the object the code adapts. For instance, an adapter for the source class `vtkConeSource` should be identified by the class' name. Furthermore, each adapter has to implement access methods for getting descriptions of what attributes the class has, getting the values of each of those attributes and setting such values as well.

Optionally, the adapter can implement rendering information in case the adapter is not for a specific VTK class but for a template or custom algorithm, in which case information is needed on how to render and update the objects generated by the adapter.

In order to implement the defined adapters, we first define a basic adapter class which is used for only use case number 3, i.e. the adapting of VTK defined objects. This class does not define rendering and update information, as these are already defined by VTK. We implement `vtkAdapter` as shown in listing 4.1.

The class has a unique identifier in the form of a `LPCSTR` (Long Pointer to Const STRing) which is to be set when instantiating the adapter. The class also carries with it a descriptor of the VTK class that specifies what attributes the class has and what is their type. This descriptor is used by the Unity Managed plugin as we discuss in Section 4.4.2.

Alongside these functions, there are two templates that define the types of `getter` and `setter` methods used by adapters. These functions take as input a pointer to the actor that renders the VTK object and the first returns a string representation of the value of the attribute while the second takes a further argument that is the new value to which attribute is to be set.

The generic calls for getting and setting attributes are respectively `GetAttribute` and `SetAttribute`.

```
class vtkAdapter
{
public:
    template <typename T> using getter = std::stringstream(T::*)(vtkObjectBase *);
    template <typename T> using setter = void (T::*)(vtkObjectBase *, LPCSTR);

    virtual ~vtkAdapter() { }

    inline LPCSTR GetAdaptingObject()
    {
        return m_vtkObjectName;
    }

    virtual void SetAttribute(
        vtkObjectBase *object,
        LPCSTR propertyName,
        LPCSTR newValue) = 0;

    virtual LPCSTR GetAttribute(
        vtkObjectBase *object,
        LPCSTR propertyName) = 0;

    virtual LPCSTR GetDescriptor() const = 0;

    virtual vtkObjectBase *NewInstance() = 0;

    virtual LPCSTR CallMethod_AsString(
        vtkObjectBase *object,
        LPCSTR method,
        LPCSTR format,
        const char *const *argv) const = 0;

    virtual vtkObjectBase *CallMethod_AsVtkObject(
        vtkObjectBase *object,
        LPCSTR method,
        LPCSTR format,
        const char *const *argv) const = 0;

    virtual void CallMethod_AsVoid(
        vtkObjectBase *object,
        LPCSTR method,
        LPCSTR format,
        const char *const *argv) = 0;

protected:
    // The name of the VTK object (as written in the wiki) for which
    // the class acts as an adapter
    LPCSTR m_vtkObjectName;

    vtkAdapter(
        LPCSTR vtkObjectName)
    {
        m_vtkObjectName = vtkObjectName;
    };
};
```

Listing 4.1: C++ implementation of the vtkAdapter class.

These functions act as dispatchers for the particular operations, that can either be directly implemented into the function but, as good practice, we later show an example of how we recommend these adapters should be implemented.

The actual attribute to change is encoded in a `LPCSTR` that contains the name of the attribute exactly as written in the C++ code. A difference from the `GetAttribute` generic call and the particular `getter` template is that the return value is not returned but a specific parameter acts as return buffer, as this value is not to be used inside the C++ native code but to be sent through to the C# interface in Unity.

Adapters also implement a `CreateObject` method allowing the instantiation of VTK objects through Adapters. The interface is the same as the Introspection Interface's `CreateObject` plus an array of

arguments that can be passed and defined in each adapter.

Finally, the `CallMethod` family of methods allows the user to implement generic methods that can be called from the Unity environment. In particular, these methods are meant to be used to accommodate the needs of use cases number 2 and 3. These use a similar signature to the `VtkResource_CallMethod` methods from the plugin's API, so that the parameters can be directly passed down without the need of unnecessary parsing at the infrastructure level. The responsibility for correct parsing is left to the user to implement, so that they can also have a higher degree of freedom of what they can actually code.

Alongside the adapters, the `vtkAdapterUtility` provides the access point to the register of the adapters, which is the collection of the adapters wrapped in a Singleton pattern available to the system. This allows to couple adapters to a single point of entry and masks the adapters to the infrastructure layer, making it easy to execute on a separate service. The implementation of this class is generated by a Python script at build time, available in Appendix B. The interface of the utility is shown in Listing 4.2.

```
1 class vtkAdapterUtility
2 {
3 public:
4     static vtkAdapter *GetAdapter(LPCSTR vtkAdaptedObject);
5
6 private:
7     static const std::unordered_map<LPCSTR, vtkAdapter *> s_adapters;
8 };
```

Listing 4.2: C++ interface of the vtkAdapterUtility class.

As visible, the adapter only exposes the function `GetAdapter` that returns the implementation of the adapter for the unique ID requested if such object exists, `NULL` otherwise. The adapters are stored in an unordered map, as it has faster access times than the ordered counterparts on the average case [40, 41]. This depends on the hashing function, but as we use default types for keys, and we do not replicate them, the average case is almost certainly guaranteed. The map is populated on instantiation and the entries are generated as Singletons, as shown in Listing 4.3.

```
1 const std::unordered_map<LPCSTR, vtkAdapter*> vtkAdapterUtility::s_adapters =
2 {
3     { Singleton<vtkConeSourceAdapter>::Instance()->GetAdaptingObject(), Singleton<
          vtkConeSourceAdapter>::Instance() },
4     // More adapters...
5 };
```

Listing 4.3: Example of adapters' register instantiation.

Benefits and Limitations

These adapters have the objective of allowing the user to customize their system and tailor it to their needs. The user can define different ways in which the native plugin interacts with VTK. The ability to customize the software to these extents enables users to create code that can be re-used and distributed. Within a community of users, this allows them to exchange knowledge and tools that could not be achieved without this feature.

The main driver behind adapters is to make the solution maintainable as well as generic. We do not foresee all potential uses of the adapters, as it is not our objective. Their implementation is kept to the bare minimum and some use cases are presented which are, to us, the most obvious. We set out to not limit the potential for these components, and require the implementation of what is needed to make it work. This caters to our requirement of a usable system, as it makes it easier to tailor to specific needs of users.

The level of generality of this component however is also one of its main limitations. As we cannot foresee all the possible use-cases, and to avoid limiting users' options, we do not enforce many restrictions on what adapters can and cannot do. Furthermore, we can do little to enforce standards and recommended ways of using adapters, for the same reasons as before. As such, where users have great potential for new features, that also have little support from the system to maintain and debug them.

Another limitation, at least compared to the introspection interface, is that this tier has no guarantee to compile with different versions of VTK, as the features used are not guaranteed to be present or be the same across versions. Although VTK developers have as one of their requirements the new versions of the library to be backwards compatible as already discussed in Chapter 3, this does not mean that they are forwards compatible, and adapters developed with newer versions of VTK are not guaranteed to work with older ones, and we do not set a way to solve this issue. This is because adapters are tools for the users, and they should be responsible for paying attention to this kind of details.

The other great advantage of adapters over introspection is that it can make use of the performances of pure C++ code. As we discussed in Chapter 3, pure C++ code running VTK algorithms achieves far greater performances on the execution of pipelines than our approach using Python. As such, adapters enable the users to take advantage of such performances, removing the barrier of also having to code their Unity plugin along.

4.4 Managed Plugin

The integration of VTK with Unity makes use of both a native plugin, which we already discussed, and a managed plugin, which is the focus of this section. In particular, this part of the system is responsible for making the native plugin API more user-friendly, and connecting the rendering cycles of the two rendering contexts. The setup is almost identical to the VtkToUnity plugin, with a different API for VTK.

The basic API of the managed plugin is identical to the native plugin API, except the usage of the `Marshal`, a .NET library used to work with unmanaged memory areas. In Unity, this library is used in particular to handle passing parameters back and forth between the managed and native parts of the environment, like defining how strings have to be handled passing down to C++. In our implementation we use this exclusively to specify strings to be handled as long pointer strings, as we represent them as `LPCSTR` (Long-Pointer Constant STRing) in our C++ code.

As a further design discussion, we present now the UI Toolbox and UI Composer components that are part of the managed plugin. As the UI is out of the scope of this thesis, we will not discuss particular implementations of these components, as they will be presented in a separate thesis. Designing these components, we mirror our approach to accessing VTK, as our drivers remain the same. In particular, the UI Toolbox is the component that mirrors the Adapters registry, while the UI Composer mirrors the Introspection Interface.

For the API of the managed plugin, see Appendix D. The minimal API is identical to the native plugin API, with the types being those of C#, while the support and quality of life methods introduced in C# to avoid verbose code is separately documented.

4.4.1 UI Toolbox

The UI Toolbox is the main access to UI components of the managed plugin. It acts as a database of factories that generate specific UIs when called upon. As a reference, the component is highlighted in blue in the architecture in Figure 4.4.

As UIs are in general more complex than accessing introspectively a library, as they are made of more components and constrained by interaction and usage limits that depend on the device being used, the environment, the input devices and so on, the ability to introduce custom-tailored UI components is necessary. Thus, we propose the usage of a registry of UI prefabs developed in Unity as first-tier access to the UI of the environment.

A registry mapping from the classname to the prefab factory `IComponentFactory` acts as the access-point for the UI instantiation. When a part of the pipeline is created, the UI UI Toolbox first checks the registry to determine whether a factory for the entire UI exists. If this exists, the factory is called and a `Create` function generates the UI based on the prefab. The UI is then linked to the particular part of the pipeline by the manager in order to keep track of the changes and to avoid the changes being either lost or set to the wrong VTK object. This is also to have the UI ready but not shown, and only made visible through `Show` when necessary and hidden through `Hide` when changing focus.

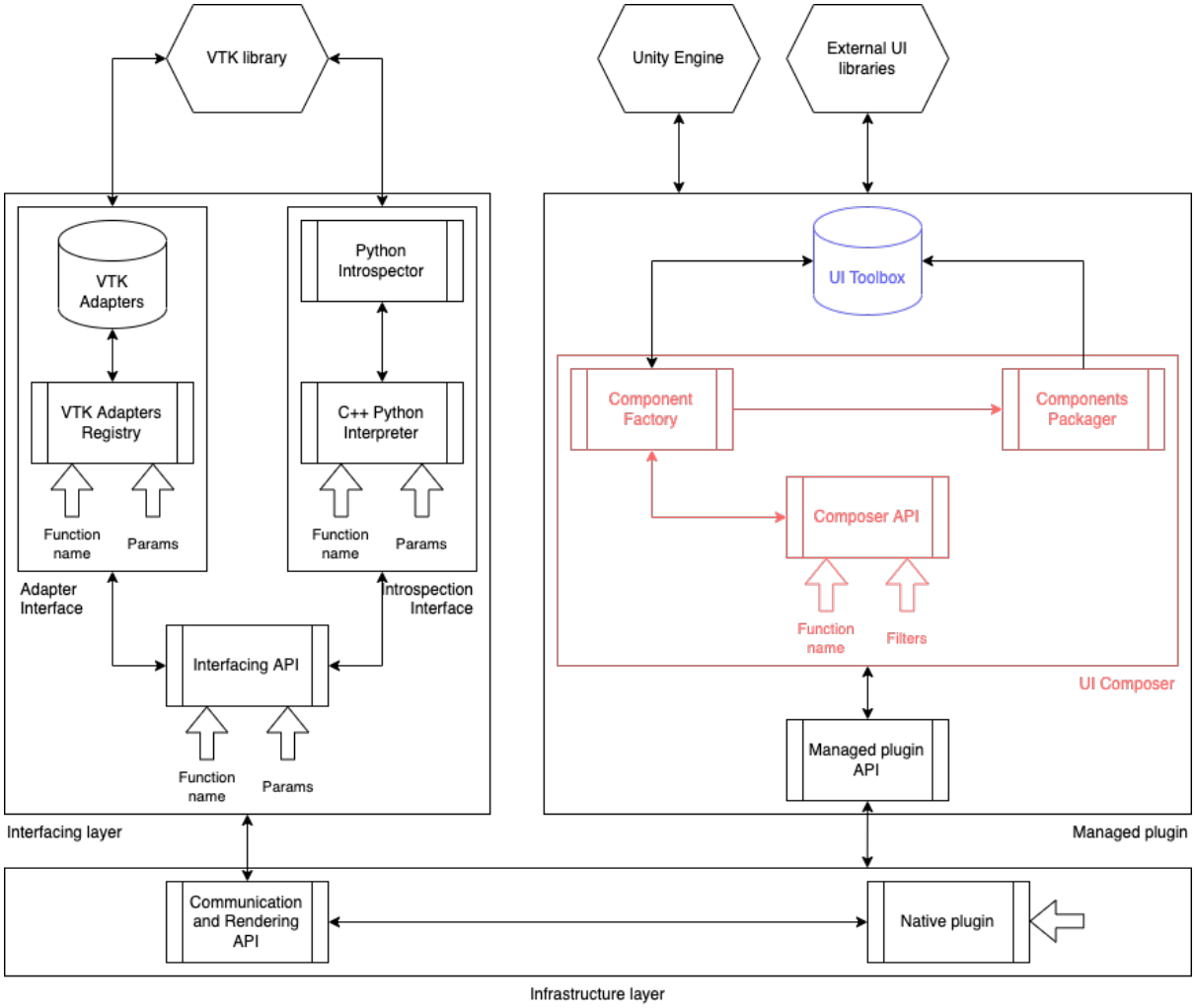


Figure 4.4: Architecture with the UI Toolbox highlighted in blue and the UI Composer in red.

4.4.2 UI Composer

Finally, the UI Composer is the last part of the plugin we will discuss. This is a factory of factories, generating UI components that are a collection of minimal input UIs that are the base of the UI Toolbox. As a reference, the component is highlighted in red in the architecture in Figure 4.4.

If the UI Toolbox fails, a generic system to generate UIs gets called. This requires more resources and time to execute, as it takes the descriptor of the VTK object, which maps the names of the attributes to their types, and then creates a minimal UI with this information. The UI Composer uses a registry of minimal components that are part of the environment that implement interfaces to edit native simple values, e.g. integers, decimals, booleans, strings and so on. These components are also registered with the UI Toolbox, for faster access.

The components generated this way are then used to instantiate a `ComposedUI` object, that contains the information of the different parts that make the UI and then add this to the UI Composer's registry, acting as a `IComponentFactory` for later calls to the UI Composer on the same classname to avoid generating the same factory twice.

In order to accelerate and expand the UI Composer's ability, the mappings are not hard-coded; the UI Composer still uses the UI Toolbox as a lookup for the basic parts used to generate the UI, and thus custom-tailored basic UIs can be added to the UI Toolbox to override the natively implemented ones. In this way, the component for a given string in the UI Toolbox's registry is always the last one registered, making the native ones the lowest priority ones.

The UI Toolbox is designed in order to accommodate our maintainability requirement, and as such this modular registry comes in handy to set some conventions in the development of UIs using this system that make them re-usable. In particular, UI components should be registered either if they allow the manipulation and/or visualization of a single parameter or when they combine other UI components. The first ones constitute the minimal set of the UI components, which respond to a particular input/output, like `float`, `int`, or `string`, while the others respond to complex interactions like showing the UI for a `vtkConeSource` object, or visualizing a `stringList`. For the naming of the components we encourage camel case with the first letter lowercase as by VTK's convention.

These conventions also make for the best operability of the UI Composer that will be able to generate more complex UIs, using composed UIs instead of simple input methods. To allow for further customization, we propose that the names can be postponed with filters that specify how the input is inserted, i.e. a float can be inputted through `slider` or `textField` for example. However, this is to make the modularity more useful to the user, rather than to the UI Composer, as we do not design how these filters could be used by it.

Chapter 5

Results

In the previous chapters we presented an architecture that satisfies the requirements of a VR visualization development workstation. We also presented an implementation of the engine of such an environment that satisfies in particular its performance and generality requirements. To validate our work, we run similar tests to those presented by Dreuning for their OculusVTK environment [1]. We compare the results of these tests with those from Dreuning and the performances of VtkToUnity [2]. Finally, we compare those results with the findings from Chapter 3 on the performances of VTK in C++ and Python.

5.1 Methodology

We have developed two Unity scenes using our plugin to test two visualizations using VTK. In Section 5.2 we introduce these scenes. The implementation follows a similar structure to the examples provided by VtkToUnity, but offer less interaction. To each of the VTK objects in the scene, a rotation is applied in order to evaluate the behavior.

As we are using an HTC Vive headset, we can use the SteamVR provided layer to collect the performance data. Unfortunately, this utility requires us to take these values manually or to record through other software the readings, as there is no way to dump the amount of values we need from this utility. For this reason, we developed a small script in C# for sampling the FPS values and store them in a buffer that is dumped when the execution of the scene is terminated. We dump at the end of the execution to avoid the IO operations to influence the FPS values. Another script is written to dump profiling values instead. It follows the same structure as the FPS reader, but instead times the execution of the calls to our implementation.

The scenes are executed manually and navigated through-out the test to evaluate performance over a realistic usage of the environment. The visualization produced are rendered both from afar and nearby, through the motion of the user. The visualizations are also scaled using a Unity proxy object that contains the VTK objects, as well as applying a constant spin to the VTK object from within Unity, to account for Unity operations influencing VTK objects. In one scene we also apply repeatedly transformations to the VTK objects using our plugin, in order to take into account real-time changes to the objects.

There should be no graphical artifacts when VTK objects change during runtime and the rendering, scaling and transformation of the objects should not impair usage of the environment. We will finally compare our experience of the environment with the data collected to validate our engine.

The results from these tests should validate in part our performance requirement. Considering we are only instantiating and updating one object at the time, we have to check that the time it takes to execute such operations would not account for major disruption in usage when multiple objects are on-screen. We think it to be more interesting to view the profiling data from the scenes over a full test with multiple objects, as we can have a more clear picture of what the limits of the system are.

Nonetheless, we will perform a stress test with an increasing number of objects to complete the discussion. This test will not be run in a VR environment, considering that, if significant differences are found between the two, we can already account for that comparing these results with the previous two.

We expect our tests to show that the scenes using the Introspection Interface are indeed slower, use more memory or both compared to previous solutions. However, we also expect it to yield a usable environment and as such achieve our objective of developing a powerful enough engine to support general purpose VR visualization development environments. We also expect there to be a significant difference in the execution times compared to the native runs from Chapter 3, as we introduce delay both in accessing VTK and with Unity's rendering cycle. However, seeing as the execution times from the native tests were promising, we expect this not to be an issue.

We also expect our system to be able to support a somewhat large number of simple VTK objects, as the profiling conducted on the C++ with Python introspection shows the operations take times that are insignificant compared to the 11.1ms limit we have to reach for the 90 FPS target. If we consider more complex objects such as stream tracers, further optimizations will likely be required for a completely fluid experience with multiple of such objects and live editing, which will be discussed in Chapter 7.

5.2 Materials

To test our design, we implemented the engine of the environment, i.e. the Infrastructure layer and its integration with Unity as described in Chapter 4, and developed two test scenes in Unity using the plugin. In order to test them, we used Python 3.7 to run the scripts and embed the interpreter in a C++11 native plugin. We built the system using CMake 3.14 and MS Visual Studio Community 2015. We tested the implementation with VTK 8.1.2 and 9.0.2 to test its version-agnosticism, and Unity 2019.3.5f1, under Windows 10.

The environment runs on a workstation built with an Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz, NVIDIA GeForce GTX 1080 Ti and 32.0 GB of RAM. We used a first generation HTC Vive HMD to test the scenes, which has a 90 Hz refresh rate with a combined 2160x1200 pixels. For the tests we used a density dataset file provided by Boston University Tech in a 2008 workshop on Paraview[1] which we edited in order to have the points centered around (0, 0, 0).

We track performances using a C# script in the Unity editor called through a static object loader that collects the FPS count every second and dumps them into a log file. We then process the file in order to produce the following information from the data: the FPS distribution, maximum, minimum, mean and median values. Furthermore, we compare the results with the recommended values from Unity and HTC. During the tests, we move the camera at different distances from the user and executing delayed updates of the objects in the scene to see how these updates impact the user experience.

The experiments are composed of one scene where a cone is rendered through a `vtkConeSource` and its parameters adjusted so to have a higher resolution. Every two seconds, it's height, radius and resolution are changed. A second scene is composed of a `vtkStreamTracer` generated from the density dataset. We first test the scene using a low number of points for the stream tracer, and then again with a much larger number.

5.3 Tests

We discuss the results of each single test we run. In particular, we focus on analyzing the performances and profiling of each of the scenes. For each test we show the relevant pieces of C# code creating and manipulating the objects. The FPS counter we use for collection is a common piece of code in the Unity community, and we will not examine it. The profiling instead is carried out in two different ways. The first, consistent with the tests in Chapter 3, computes the time using a timing function on the plugin calls and dumps those to a spreadsheet. The second uses Unity's integrated profiler to generate relevant metrics on the scene's runtime. All the listings shown are stripped of timing and profiling code for readability purposes.

[1] Available online at `https://www.bu.edu/tech/support/research/training-consulting/presentations/visualizationworkshop08/`.

FPS metrics	
Count	7405
Max	99
Min	2
Mean	87.7627
Median	89.0
Standard deviation	3.1823
5th percentile	88.0
25th percentile	89.0
50th percentile	89.0
75th percentile	89.0

(a)

FPS metrics processed	
Count	7172
Max	92
Min	86
Mean	88.9769
Median	89.0
Standard deviation	0.5110
5th percentile	88.0
25th percentile	89.0
50th percentile	89.0
75th percentile	89.0

(b)

Table 5.1: Description of the data distribution of FPS in the cone source scene. (a) does not account for outliers, while (b) does.

5.3.1 Cone Source

With the Cone source scene, we aim at confirming our environment works well at rendering and manipulating simple objects, while achieving the target performances. The relevant implementation is shown in Appendix E. This particular scene is also updated every two seconds, changing the cone's height, radius and resolution every two seconds. This is done through a coroutine for which the code is shown in Appendix E. The objective is to evaluate the performance of the plugin under real-time editing. Finally, the scene is run both with and without HMD to evaluate how much this influences the plugin's performances.

From our manual tests, the scene appeared well rendered, the updates to the VTK object did not cause lag to the rotation or the rendering of the object, no glitches or artifacts were generated that were not expected, and the experience within the environment felt smooth. The data collected by the scripts further confirms the first-hand experience, as is visible in Figure 5.1.

In order to understand the graphs, below are the relevant information from the dataset. In particular, the execution of the test lasted for circa 83 seconds. Table 5.1a relays the rest of the data. As visible, the values validate our design. The system reaches on average and most of the time the objective FPS values, in particular staying stable around 89.0 FPS.

Although this value is not exactly 90 FPS, as would be most desirable, considering the possible rounding error and that these values get dumped as integer values, it is likely that those values approached closely 90 FPS, and thus we believe that error accounts for this difference.

What is surprising to see, is the minimum of 2 FPS and quite a large standard deviation. However, we account for such variation considering two main factors. First, we know that our plugin requires a certain loading time to be fully operational, and this loading time is blocking to our scene, as the VTK objects cannot be instantiated yet at this point. Second, we have not cleaned the dataset of potential outliers, and thus we may be looking at sharp FPS falls that are not perceivable while using the environment, as proven by the fact that our experience was not disrupted by those drops.

Furthermore, we are collecting metrics and running quite some debugging scripts on top of our plugin, and as such these values could also be explained through all the boilerplate code running. We thus attempt to clean up the dataset in order to remove these factors, i.e. use the profiling data to remove the loading period from the dataset, and we use Z-Scores of threefold the standard deviation to remove outliers from the dataset.

With these considerations, we analyze again the dataset and produce new results, shown in Table 5.1b, that indeed confirm our suspicions and, in fact, validate our design and implementation. We still do not account for the metrics and debugging code, but considering the results we believe those to be trivial and not influencing our results to an interesting degree.

The amount of time *shaved off* the beginning of the dataset represents the loading time of the software

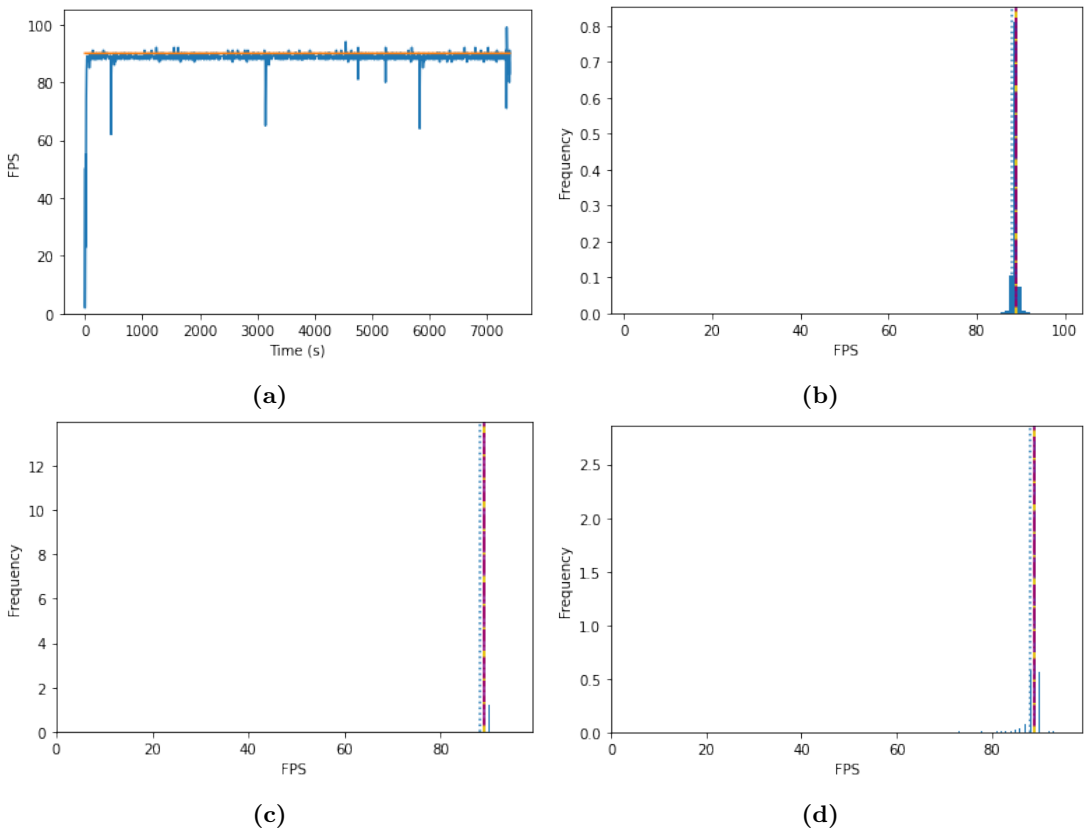


Figure 5.1: Time series (a), (c) and histogram distributions (b), (d) of the FPS from the cone source scene with and without outliers respectively. The orange line in (a) and (c) represents the 90 FPS target. The dotted lines in (b) and (d) represent: 5th percentile in blue, 25th in red, 50th in purple and 75th in yellow.

that we do consider for runtime analysis. Furthermore, the missing FPS drops are the outliers in the collected metrics, which amount to 57 frames in total not considered over the 3498 of the entire test (1.63%). The standard deviation and the mean values also show the app executes as expected under normal load.

We can see both from analyzing the profiling data and the FPS that there are some anomalies causing FPS drops and slower operations than expected. The overlapping graphs can be seen in Figure 5.2, where the execution frame of the given value has been computed taking into account each update executes 2 seconds after the previous has terminated (estimated at 180 frames as the error is neglectable). We can see that there are some similarities in the moments when these anomalies where detected, however not enough to hint to an issue with our system.

The remaining values from the profiling are slightly higher than those obtained in Chapter 3 for the C++ with Python introspection, but this is as expected and as such validating our results. In particular, it appears it would be acceptable to update a VTK object around 5 to 10 times per frame, which would in total mean between 450 and 900 operations per second. We believe these values could be improved on, but are already an acceptable result.

5.3.2 Stream tracer

The Stream tracer scene has the objective to validate the plugin with more complex sources. The scene has been run both with 100 and 1000 points, and the FPS results analyzed as for the cone source scene. The profiling data mainly focuses on update times of the VTK object and their analysis will be discussed later. The relevant implementation is shown in Appendix E.

The points represent the sources from which the Stream tracer's visualization is computed. Considering the algorithm is computationally demanding, adding source points increases the stress under which

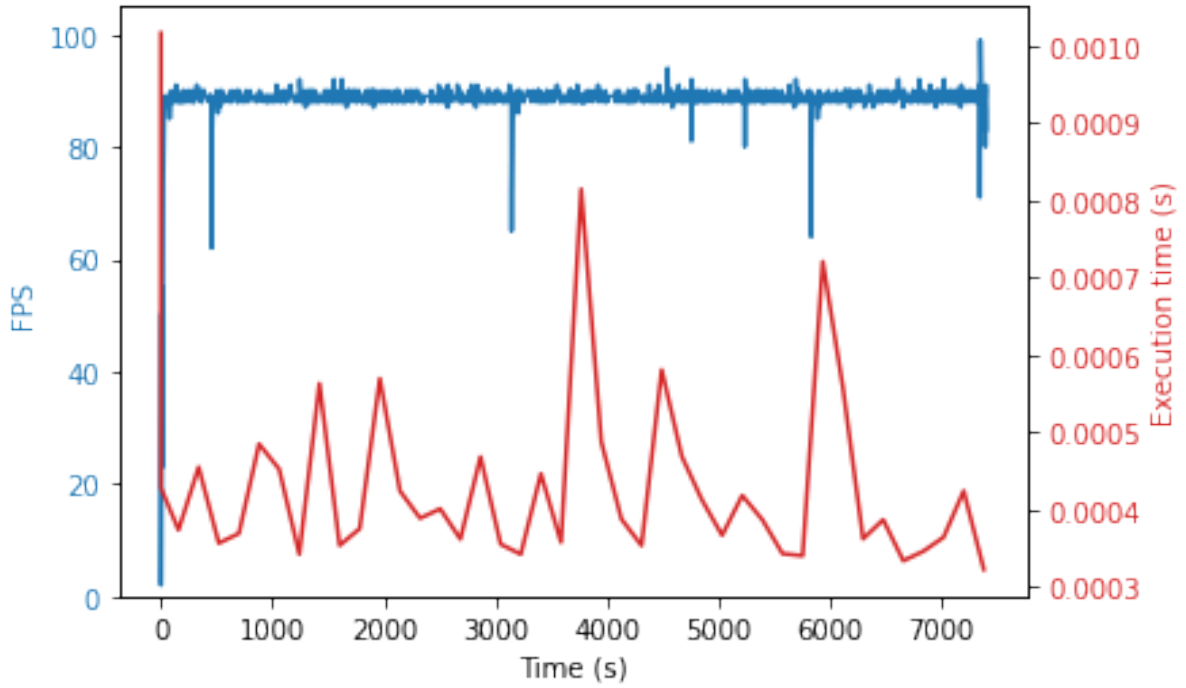


Figure 5.2: Overlapping graphs showing the drops in FPS coincide with spikes in computation time of VTK operations.

FPS metrics	
Count	3946
Max	106
Min	2
Mean	88.6607
Median	89.0
Standard deviation	4.2957
5th percentile	88.0
25th percentile	89.0
50th percentile	89.0
75th percentile	89.0

(a)

FPS metrics processed	
Count	3716
Max	92
Min	86
Mean	88.9997
Median	89.0
Standard deviation	0.5038
5th percentile	88.0
25th percentile	89.0
50th percentile	89.0
75th percentile	89.0

(b)

Table 5.2: Description of the data distribution of FPS in the 100 points stream tracer scene. (a) does not account for outliers, while (b) does.

the system is put. Increasing the number of points allows us to test how well the system holds with more complex, crowded and computationally slow visualizations, and thus under a real-world application load.

Starting from the 100 points scene, we can clearly see that the system is more than capable to sustain the rendering. The results are fully in line with our expectations and there is not any interesting result to highlight. The description of the measurements are listed in Table 5.2, while the time series and distribution can be seen in Figure 5.3. Finally, Figure 5.4 shows the rendering inside the VR environment.

Regarding the profiling, the setup operations creating the stream tracer object take around the time we would expect, as they are almost identical to those obtained from the C++ with Python embedding and using introspection test from Chapter 3. The update operations at each frame were almost neglectable, with a maximum time of 0.15 ms.

More interesting is the 1000 points rendering. The distribution of the FPS is more spread out and the standard deviation even without outliers is still significant. Although it reaches high mean and median

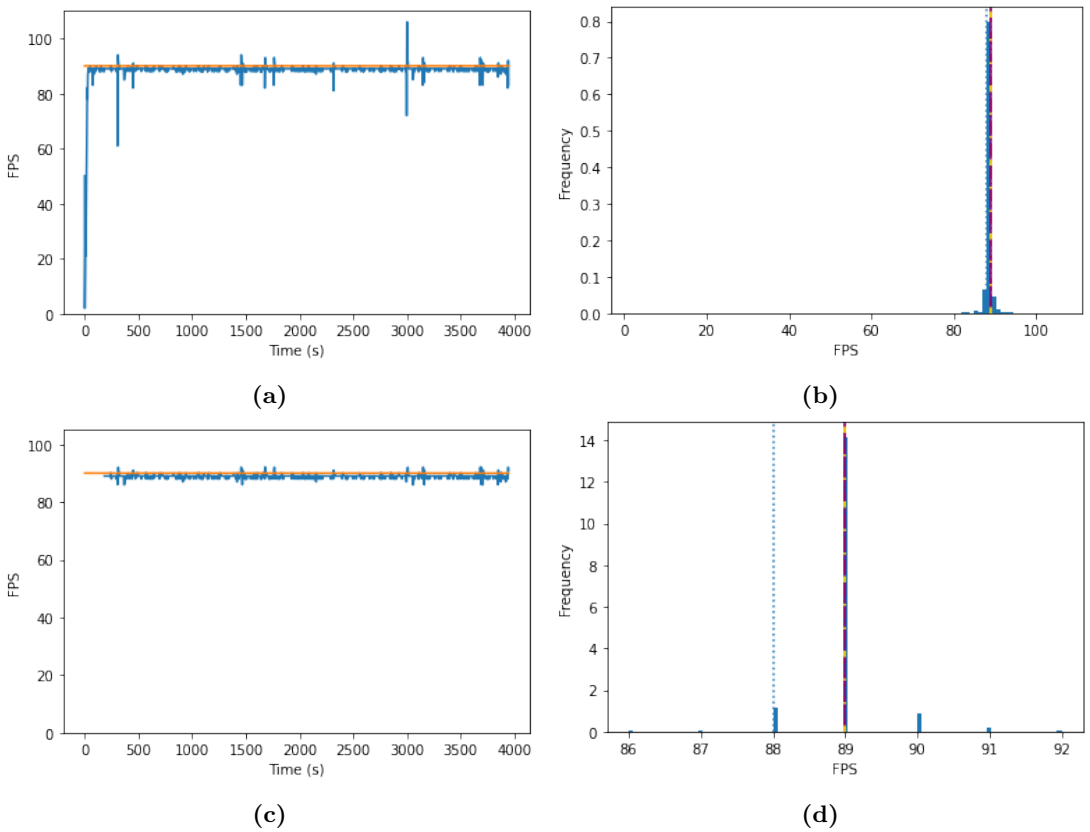


Figure 5.3: Time series (a), (c) and histogram distributions (b), (d) of the FPS from the 100 points stream tracer scene with and without outliers respectively. The orange line in (a) and (c) represents the 90 FPS target. The dotted lines in (b) and (d) represent: 5th percentile in blue, 25th in red, 50th in purple and 75th in yellow.

values, with solid 90 FPS above the 25th-percentile, the lower end is also more pronounced, with a drop to a minimum of 65 FPS in the processed dataset.

Taking the profiling into account, we can see that the updating operations called on the objects do not take significant amounts of time, with a maximum of just below 0.4 ms. Considering these results, and the weight of the rendering itself, we believe these to be promising results, however they show further work in optimization is required in order to produce a well-rounded system.

The datasets' descriptions are visible in Table 5.3, while Figure 5.5 shows the graphs of said datasets. Furthermore, Figure 5.6 shows the profiling data from this scene. It is important to note two things: first, the first graph does not contain both the update operations called every frame on the stream tracer, neither the first update on the reader object, as its value would make all other values unreadable; second, all the values show that the operations ran by our plugin take considerably little time, and as such we believe most of the lag is caused by optimizations needed at the Unity managed level. The removed values are 0.2771s and 0.2768s for the 100 and 1000 points tests respectively.

As visible from the comparison of profiling data, even though the second scene has a tenfold increase in the amount of points used for generating the stream tracer, times of generation and update are almost identical, evidence of the fact that the system's bottleneck is the Unity managed level and not our engine. We will discuss further root cause investigation in Chapter 7.

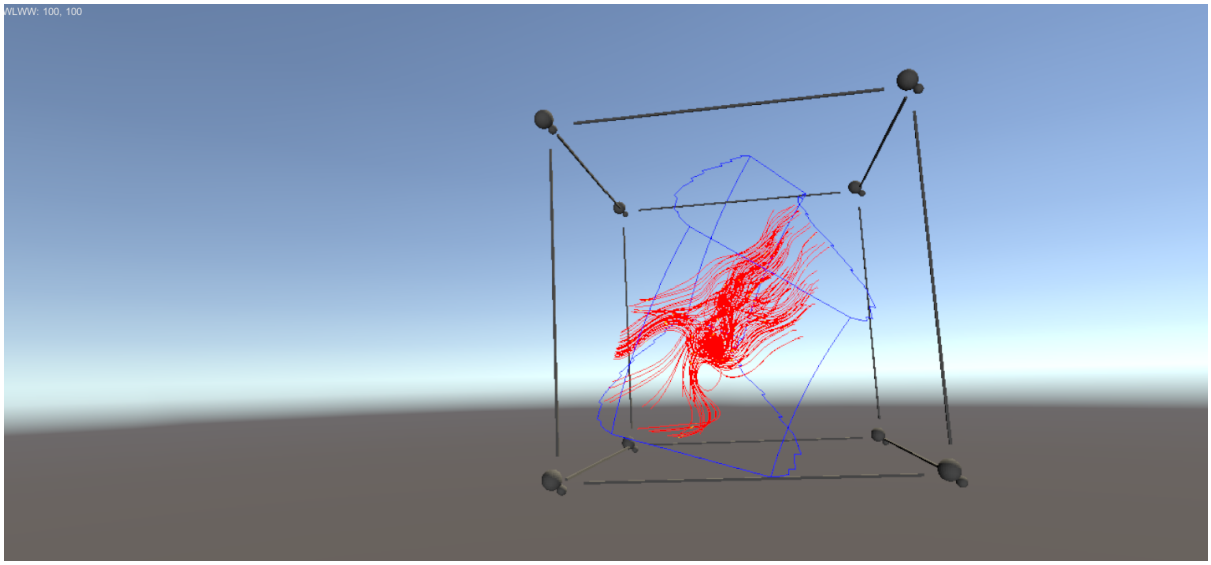


Figure 5.4: Image of the stream tracer within the VR environment.

FPS metrics	
Count	8983
Max	101
Min	2
Mean	88.9054
Median	90.0
Standard deviation	6.5988
5th percentile	75.0
25th percentile	89.0
50th percentile	90.0
75th percentile	92.0

(a)

FPS metrics processed	
Count	8683
Max	101
Min	72
Mean	89.5098
Median	90.0
Standard deviation	4.8100
5th percentile	84.0
25th percentile	89.0
50th percentile	90.0
75th percentile	92.0

(b)

Table 5.3: Description of the data distribution of FPS in the 1000 points stream tracer scene. (a) does not account for outliers, while (b) does.

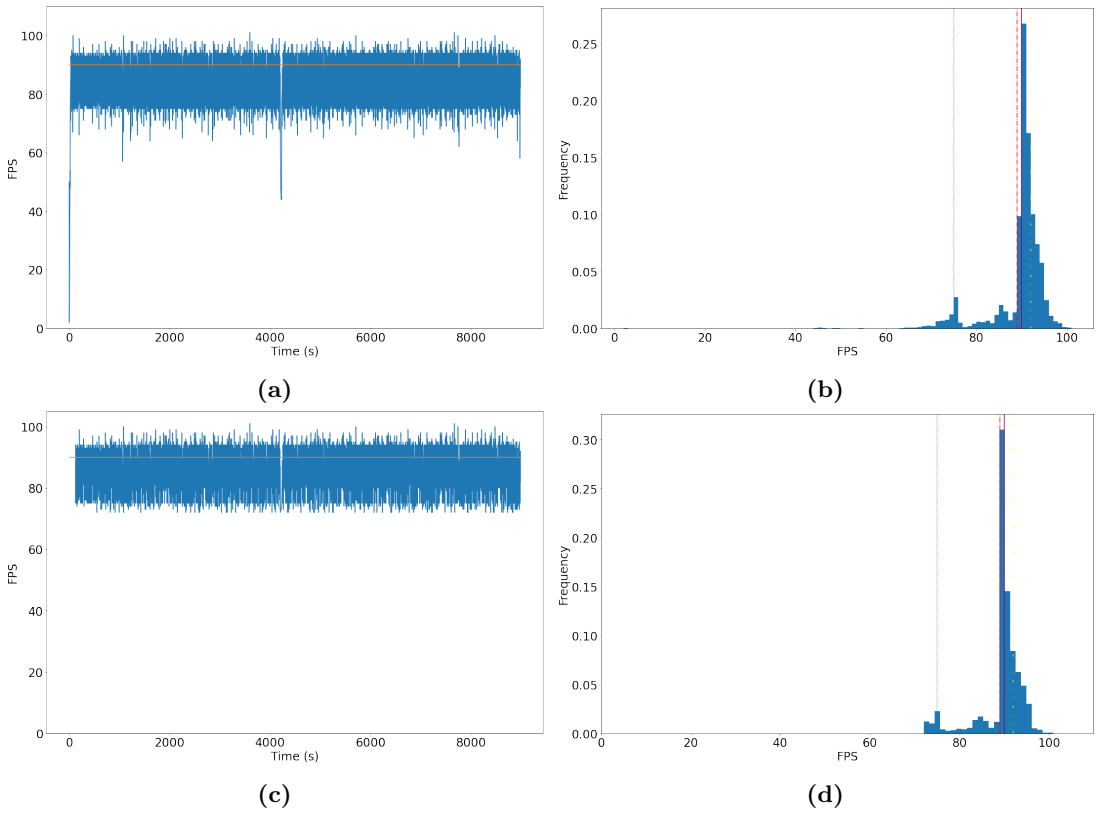


Figure 5.5: Time series (a), (c) and histogram distributions (b), (d) of the FPS from the 1000 points stream tracer scene with and without outliers respectively. The orange line in (a) and (c) represents the 90 FPS target. The dotted lines in (b) and (d) represent: 5th percentile in blue, 25th in red, 50th in purple and 75th in yellow.

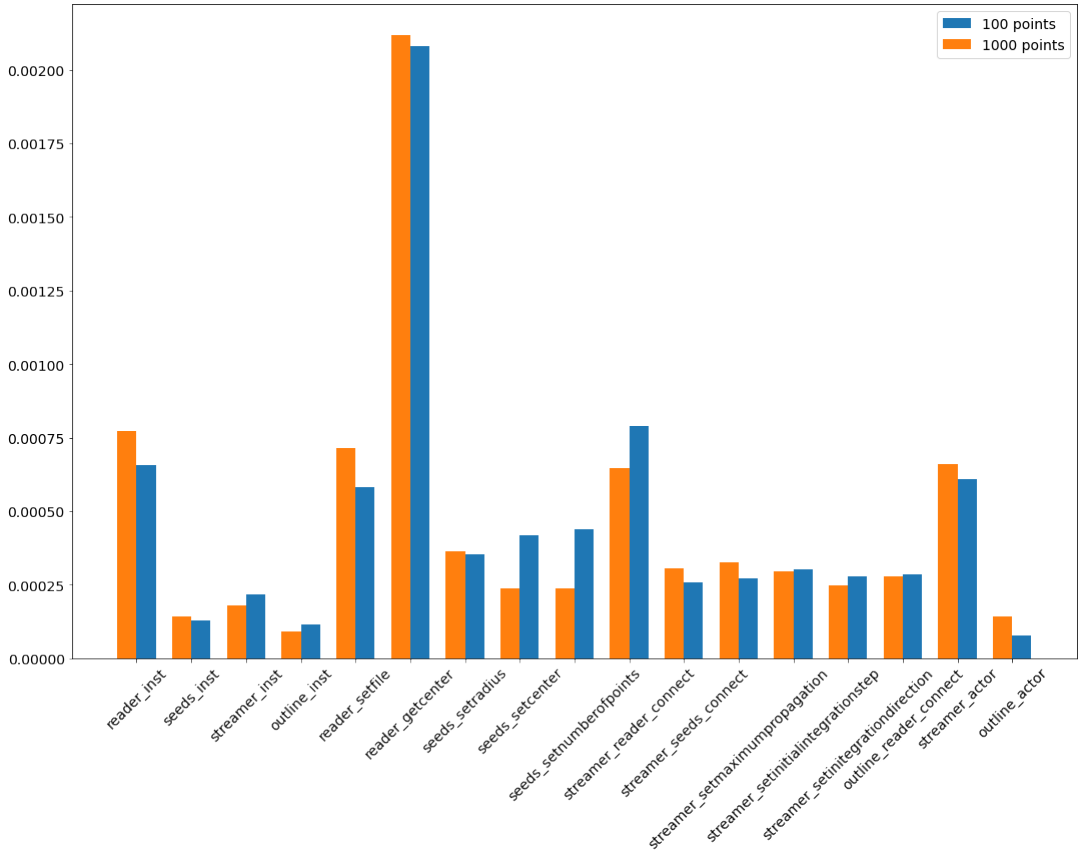


Figure 5.6: The comparison of time taken by the 100 and 1000 points stream tracers to load.

Chapter 6

Discussion

In Chapter 5, we introduced our validation of our design and our expectations. In this Chapter we discuss whether our solution meets our expectations, what unforeseen results we obtained and what are the limitations of and threats to our design and implementation. We will mostly focus on performances and generality of the engine. Usability is discussed in Chapter 7 as it heavily relates to the future development of the UI environment we described in Chapter 4.

6.1 Expectations met

Our software is capable of visualizing and editing in real time VTK objects without significant losses in performance. The overall design works reliably in instantiating, accessing and updating VTK objects and can be easily used alongside Unity.

We built and ran the code in several VTK versions (8.1.0, 8.1.2, 8.2.0, 9.1.0) without hiccups, and we were also able to run the plugin with two different releases of Unity. The ability to access any VTK object works seamlessly.

Alongside its versatility and version agnosticism, the system achieves the desired performances. During the manual tests, we did not experience lag or motion sickness and the environment rendering appeared smooth, without artifacts and anomalies. The little interactivity introduced through Unity managed level allowed us to explore different distances and scales of rendering the VTK objects, that did not impair the usage of the system.

Overall, the state of the plugin shows that the creation of the system we envision is possible and that the possible bottlenecks we foresaw in the beginning of the project were either not an issue or could be overcome.

Finally, a final shortcoming of our software lays in its maintainability. Although the different modules composing our architecture can be considered as separate, most of them still depend on either Unity or VTK. While for the first one this is pretty much scoped down already in the API definition and routing, this cannot be said for the second as infrastructure layer, Introspection Interface and adapters all depend on VTK. Ideally, the infrastructure layer should be decoupled from the library.

However, we are not certain this is desirable, as doing so could lead to further overheads and complexity in the system. Considering that the use-cases for our plugin entail the need for high performances and lightweight code, we believe that this is not an urgent matter and that this issue is mitigated by the fact that these are not tight couplings and only depend on which library Introspection and Adapter Interface are working with.

On the maintainability of the codebase, we ran CLOC [42], an opensource utility program to count lines of code. The entire codebase is less than 6000 SLOCs, with 2795 of which are mainly boilerplate code necessary for implementing a Unity C++ plugin. The Introspection Interface, comprising both the C++ and Python code, is made of 1517 SLOCs, while the adapter code is bundled with the main plugin and has a further 162 SLOCs of header files. The C# helper functions to access the plugin amount at a total of 871 SLOCs.

If we consider the libraries we need to run the code, i.e. the Unity headers, the renderer library (GLEW) and our full C++/C# plugin, the codebase reaches short of 40 SLOCs. Our code additions are fully documented in the header files, with further documentation inline where algorithms may appear difficult.

6.2 Shortcomings

The main shortcomings that our software presents compared to our objectives lay in the limitations of the Introspection Interface module and in the technical barrier for the usage of the software.

The Introspection Interface module is not able to fully expose a quick access system to VTK, as only the VTK objects are loaded at instantiation, while a lot of the utilities are not. However, this is a temporary limitation due to the necessity to expand on the Introspection Interface module. This is out of the scope of this thesis, as such work relies on known technologies and approaches, but requires further analysis of the VTK package and is material for a different thesis.

This being said, it still limits the capabilities of the software, which for now are expanded through a workaround, allowing the plugin to directly access the introspection feature of Python. Which is not ideal, as it introduces a non-trivial overhead in the loading of the symbols in runtime, and should thus be avoided as much as possible.

The second main issue relates to the technical barrier. Although we have reduced the technical skills required to approach the development of VTK visualizations in Unity for VR, this was not to the extent of our expectations. The managed level interface still requires the user to carefully specify the types of the parameters passed and does not do much in the sense of error handling.

6.3 Threats to validity

The main threats to the validity of this research are the limitations of the introspection layer, already introduced in Chapter 4. In particular, of itself the introspection layer does not strictly meet our definition of VTK-completeness. As one of our objectives was to have a general system that could access the full array of tools provided by VTK, this threatens the results we achieved.

However, if we consider the entire system instead of focusing only on the single components, we argue the system provides all the necessary tools to make the system strictly VTK-complete, and that the introspection layer provides an out-of-the-box solution that caters to most needs of visualization developers. More particular needs may require the usage of adapters, but as those are small code snippets, they can be easily shared and re-used, and thus a repository of adapters can be created.

Furthermore, the Python Introspection Interface loads most information and instantiation, thus giving a fast and general access to most resources, whereas a more general solution may require longer times to actually generate this information, and may not be as performant as a smaller, narrower solution. We expand on the possible paths forward on the Introspection Interface in Chapter 7.

When it comes to our validation, our experiments do not cover all the array of uses of our software. Furthermore, we tested performances without factoring in the interactiveness of the environment we envision. However, we presented an engine which is able to reach the desired performances and enables further research into the creation of a VR visualization development environment. If performances should not reach the necessary thresholds for bigger projects, there is ample room for improvement.

Also, our solution can leverage fully the modules of VTK, allowing usage of parallel and distributed rendering of VTK objects. We believe that, as a preliminary research in the creation of such a complex environment, we already demonstrated that visualization and VR technologies are mature enough to start the creation of this environment, and with our implementation we filled an important gap. On top of this, our solution is not exclusively aimed to such an environment as the one we envision, and can be used for multiple development projects, and we believe it will help standardize the VR visualization development landscape by catering to multiple needs of the community.

Another threat to the validity of our project are the results of the tests. Although in general they show our solution to reach its goals, some anomalies exist that we can only partly explain. These anomalies

seem rare but could influence the experience of the user. We will discuss future steps to investigate and solve these issues in Chapter 7, however we acknowledge that in the current state the software could present unpredictable and disruptive behavior.

Finally, a known bug exists in the code due to which an OpenGL error occurs: `OPENGL INVALID_STATE_ERROR`. This does not influence the usage of the system, as it was possible to use it. However, this may be a hint towards explaining some of the issues that arose during the tests and must be anyway investigated in order to achieve a reliable and usable software.

Chapter 7

Conclusion

We have developed a fully functional coding environment that allows users to easily integrate VTK and Unity for visualization development. Our system is already able to uphold VR performance and usability standards. As such, it is capable of supporting VR environments and thus ready as an engine for an immersive VR visualization development environment.

Our design enabled the code to be maintainable and modular, making it easier to scope changes to the system and investigate bugs and issues. The ability to switch between native C++ code and the Python interpreter allows full leverage of VTK's features without losing the ability of reaching high performances.

Analyzing previous implementations we were able to determine threats to the maintainability and usability of our software long-term, mainly relate to the usage of deprecated libraries, development of monolithic software and heavy reliance on particular versions of the libraries or frameworks used. Compared to these, our system is capable of reaching the same performances while fully exposing the API of VTK to the user in a well-supported, maintained and known environment such as Unity.

The new approach proposed to create such integration is not dependent on a particular library or engine used. As such, we argue our architecture is agnostic of which are used, and can be easily used with different ones, still retaining the advantages presented in Chapter 4.

Considering the state of the art today, we have presented a first iteration of an architecture for visualization libraries and game engines that can be easily reproduced and used for VR environments. Furthermore, we have presented an implementation of such an architecture that achieves the requirements for an immersive VR visualization development environment, which can already be used by the VR community for the development of integrated applications.

7.1 Future work

The development of our environment is still in progress. The system presents some obvious shortcomings that need to be addressed in followup research. Starting with the limitations of the Introspection Interface.

The python module lacks the ability of fully mapping and loading the VTK library. A preliminary attempt to simply reproduce the approach [1] already used failed due to cyclic dependencies inside the class tree. As such, a more sophisticated approach should be developed.

Alongside this, the access to the Introspection Interface should be more carefully controlled in the C++ native code, limiting how the C# managed plugin can access it. Considering the overhead that symbol loading presents, allowing users to generically access this feature is not ideal, as non-expert users could abuse such feature, resulting in performance issues. Also, such feature should not be necessary nor desirable, as the system should expose the full API without the need of shortcuts.

Furthermore, a number of anomalies have been detected during tests, in particular FPS and performance drops that were barely perceivable inside the environment, that could though hinder the user experience; and the presence of a known bug that results in OpenGL errors at each frame. These could

potentially be linked and as such further investigation should be carried out.

Unfortunately, our preliminary attempts to scoping down which component caused our system to throw the error failed, however we believe it lays in the actual rendering and the updating of the scene objects, as we amply tested the second without any success in finding the bug.

The performance issues may be caused by external factors, and we believe further testing should be carried out, especially to ensure the system is usable by less technically trained users, representing the more wide VR visualization development community, as well as to explore the capabilities of the software with more complex and interesting visualizations.

The system achieved the results as illustrated without making use of parallelization or distribution technologies. While the code of the native plugin was thought with those in mind, the plugin is not yet in a state where those technologies can be used. The Infrastructure Layer as presented in Chapter 4 still follows VtkToUnity's blocking interface [2], and thus further development is still necessary.

Finally, the development of the UI components presented in Chapter 4 is still to be tackled. We proposed a way of implementing such components that would mirror the agnosticism, versatility and usability of the integration. We believe that creating such a modular system would allow users to more precisely tailor the experience to their needs and as such we believe that such a research route should be explored.

Acknowledgements

I am deeply grateful to Mr Fosco Cancelliere for his support while writing this thesis, who acted as a proofreader and guide in this journey. I am greatly thankful to Ms Francesca Galvagno as well, as she stood alongside me and helped me through my most trying times while working on this project and being my support during this year, due to both Covid and non-Covid related hardships.

Special thanks go to my supervisor, Dr Robert G. Belleman, as he allowed me to explore this interesting field with freedom and responsibility, while being of help and support. Alongside him, I want to thank Dr Ana Oprescu for being a guiding and cheerful light during this trying year of University, and although facing an unconventional education, she kept motivating and rooting for us to succeed, boosting the morale of the cohort and people around her in general.

Finally, I want to thank Mr Youri Reijne and Mr Simon Baars for being such good and supporting friends, and such helpful presences throughout the academic year.

Bibliography

[1] H. Dreuning, "A visual programming environment for the Visualization Toolkit in Virtual Reality," English, Publication Title: Universiteitsbibliotheek van Amsterdam Type: University archive, bscthesis, University of Amsterdam, 2016. [Online]. Available: https://scripties.uba.uva.nl/ (visited on 01/16/2021).

[2] G. Wheeler *et al.*, "Virtual interaction and visualisation of 3D medical imaging data with VTK and Unity," en, *Healthcare Technology Letters*, vol. 5, no. 5, pp. 148–153, 2018, ISSN: 2053-3713, 2053-3713. DOI: 10.1049/htl.2018.5064. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1049/htl.2018.5064 (visited on 01/06/2021).

[3] Kitware, *ActiViz - 3D Visualization Library for .Net C# and Unity \textbar Kitware*, en-US, Publication Title: Kitware Europe, 2008. [Online]. Available: https://www.kitware.eu/activiz/ (visited on 01/06/2021).

[4] R. G. Belleman, "Interactive Exploration in Virtual Environments," en, Ph.D. dissertation, University of Amsterdam, 2003. [Online]. Available: https://dare.uva.nl/search?identifier=a3d8718e-ab66-41e0-9d41-68803ab4cc8c (visited on 01/17/2021).

[5] M. Hussein and C. Natterdal, "The Benefits of Virtual Reality in Education- A comparision Study," Enlgish, bscthesis, University of Gothenburg, Jun. 2015. [Online]. Available: https://gupea.ub.gu.se/bitstream/2077/39977/1/gupea_2077_39977_1.pdf (visited on 01/15/2020).

[6] A. L. Faria, A. Andrade, L. Soares, and S. B. i Badia, "Benefits of virtual reality based cognitive rehabilitation through simulated activities of daily living: A randomized controlled trial with stroke patients," en, *Journal of NeuroEngineering and Rehabilitation*, vol. 13, no. 1, p. 96, Nov. 2016, ISSN: 1743-0003. DOI: 10.1186/s12984-016-0204-z. [Online]. Available: https://doi.org/10.1186/s12984-016-0204-z (visited on 01/16/2021).

[7] C.-H. Chen, M.-C. Jeng, C.-P. Fung, J.-L. Doong, and T.-Y. Chuang, "Psychological Benefits of Virtual Reality for Patients in Rehabilitation Therapy," *Journal of Sport Rehabilitation*, vol. 18, no. 2, pp. 258–268, May 2009, ISSN: 1543-3072, 1056-6716. DOI: 10.1123/jsr.18.2.258. [Online]. Available: https://journals.humankinetics.com/view/journals/jsr/18/2/article-p258.xml (visited on 01/15/2021).

[8] W. Schroeder, K. Martin, and B. Lorensen, *The visualization toolkit: an object-oriented approach to 3D graphics ; [visualize data in 3D - medical, engineering or scientific ; build your own applications with C++, Tcl, Java or Python ; includes source code for VTK (supports Unix, Windows and Mac)*, eng, 4. ed. Clifton Park, NY: Kitware, Inc, 2006, OCLC: 255911428, ISBN: 978-1-930934-19-1.

[9] M. McCormick, X. Liu, J. Jomier, C. Marion, and L. Ibanez, "ITK: Enabling reproducible research and open science," *Frontiers in Neuroinformatics*, vol. 8, 2014, ISSN: 1662-5196. DOI: 10.3389/fninf.2014.00013. [Online]. Available: http://journal.frontiersin.org/article/10.3389/fninf.2014.00013/abstract (visited on 01/21/2021).

[10] R. A. Drebin, L. Carpenter, and P. Hanrahan, "Volume Rendering," *SIGGRAPH Computer Graphics*, vol. 22, no. 4, pp. 65–74, Jun. 1988, ISSN: 0097-8930. DOI: 10.1145/378456.378484. [Online]. Available: https://doi.org/10.1145/378456.378484.

[11] M. Dutra, P. S. Rodrigues, G. A. Giraldi, and B. Schulze, "Distributed Visualization Using VTK in Grid Environments," in *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, 2007, pp. 381–388. DOI: 10.1109/CCGRID.2007.43.

[12] C. P. Botha, "DeVIDE: The Delft Visualisation and Image processing Development Environment," Delft Technical University, Tech. Rep., 2004. [Online]. Available: http://graphics.tudelft.nl/Publications-new/2004/B004a.

[13] S. Sua *et al.*, "Virtual reality enabled scientific visualization workflow," in *2015 IEEE 1st Workshop on Everyday Virtual Reality (WEVR)*, 2015, pp. 29–32. DOI: 10.1109/WEVR.2015.7151692.

[14] N. Shetty *et al.*, "Immersive ParaView: A community-based, immersive, universal scientific visualization application," in *2011 IEEE Virtual Reality Conference*, 2011, pp. 239–240. DOI: 10.1109/VR.2011.5759487.

[15] D. Kruis, "Creating interactive visualization pipelines in Virtual Reality," English, Publication Title: Universiteitsbibliotheek van Amsterdam Type: University archive, bscthesis, University of Amsterdam, 2017. [Online]. Available: https://scripties.uba.uva.nl/search?id=631450 (visited on 01/16/2020).

[16] J. Schutte, "Virtual Reality Interactions for Scientific Data Visualizations," bscthesis, University of Amsterdam, Jun. 2018. [Online]. Available: http://visualisationlab.science.uva.nl/wp-content/uploads/2021/09/Jan-Schutte-Virtual-Reality-Interactions-for-Scientific-Data-Visualizations_633527576.pdf.

[17] K. C. VanHorn, M. Zinn, and M. C. Cobanoglu, *Deep Learning Development Environment in Virtual Reality*, 2019. arXiv: 1906.05925 [cs.LG].

[18] J. K. Haas, "A History of the Unity Game Engine," Worcester Polytechnic Institute, 100 Institute Road, Worcester MA 01609-2280 USA, Tech. Rep., Mar. 2014.

[19] E. Viire, "Health and Safety Issues for VR," *Communications of the ACM*, vol. 40, no. 8, pp. 40–41, Aug. 1997, ISSN: 0001-0782. DOI: 10.1145/257874.257882. [Online]. Available: https://doi.org/10.1145/257874.257882.

[20] U. Technologies, *VR Best Practice*, 2020. [Online]. Available: https://learn.unity.com/tutorial/vr-best-practice (visited on 01/17/2021).

[21] C. Gregg and K. Hazelwood, "Where is the data? Why you cannot debate CPU vs. GPU performance without the answer," in *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, IEEE, 2011, pp. 134–144.

[22] E. Games, *Virtual Reality Best Practices*, en-US, 2021. [Online]. Available: https://docs.unrealengine.com/en-US/SharingAndReleasing/XRDevelopment/VR/DevelopVR/ContentSetup/index.html#vrandsimulationsickness (visited on 01/17/2021).

[23] C. Manetta and R. A. Blade, "Glossary of virtual reality terminology," *International Journal of Virtual Reality*, vol. 1, no. 2, pp. 35–39, 1995.

[24] R. B. Haber and D. A. McNabb, "Visualization idioms: A conceptual model for scientific visualization systems," *Visualization in scientific computing*, vol. 74, p. 93, 1990.

[25] C. Politowski, F. Petrillo, J. E. Montandon, M. T. Valente, and Y.-G. Guéhéneuc, "Are game engines software frameworks? A three-perspective study," *Journal of Systems and Software*, vol. 171, p. 110 846, 2021.

[26] Epic Games, *Unreal engine*, version 4.22.1, Apr. 25, 2019. [Online]. Available: https://www.unrealengine.com.

[27] H. Corp, *Buy VIVE Hardware — VIVE European Union*, https://www.vive.com/eu/product/vive/#js-spec-details, (Accessed on 06/13/2021).

[28] H. M. Hassan and G. H. Galal-Edeen, "From usability to user experience," in *2017 International Conference on Intelligent Informatics and Biomedical Sciences (ICIIBMS)*, 2017, pp. 216–222. DOI: 10.1109/ICIIBMS.2017.8279761.

[29] ISO/IEC 25010, *ISO/IEC 25010:2011, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*, 2011.

[30] I. Heitlager, T. Kuipers, and J. Visser, "A Practical Model for Measuring Maintainability," in *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, 2007, pp. 30–39. DOI: 10.1109/QUATIC.2007.8.

[31] "ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary," *ISO/IEC/IEEE 24765:2017(E)*, pp. 1–541, 2017. DOI: 10.1109/IEEESTD.2017.8016712.

[32] *Design patterns : elements of reusable object-oriented software* (Addison-Wesley professional computing series), eng. Reading, MA: Addison-Wesley, 1995, ISBN: 0201633612.

[33] U. Technologies, *Native plug-ins*, Aug. 2021. [Online]. Available: `https://docs.unity3d.com/Manual/NativePlugins.html`.

[34] K. Group, *OpenGL Context*, Jul. 2021. [Online]. Available: `https://www.khronos.org/opengl/wiki/OpenGL_Context`.

[35] M. de Bayser and R. Cerqueira, "A System for Runtime Type Introspection in C++," in *Programming Languages*, F. H. de Carvalho Junior and L. S. Barbosa, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 102–116, ISBN: 978-3-642-33182-4.

[36] T. R. Chuang, Y. Kuo, and C.-M. Wang, "Non-intrusive object introspection in C++: Architecture and application," in *Proceedings of the 20th International Conference on Software Engineering*, 1998, pp. 312–321. DOI: `10.1109/ICSE.1998.671360`.

[37] P. Software Foundation, *Python/C API Reference Manual*. [Online]. Available: `https://docs.python.org/3/c-api/index.html`.

[38] W. L. Hursch and C. V. Lopes, "Separation of Concerns," Tech. Rep., 1995.

[39] P. Düking, H.-C. Holmberg, and B. Sperlich, "The Potential Usefulness of Virtual Reality Systems for Athletes: A Short SWOT Analysis," *Frontiers in Physiology*, vol. 9, p. 128, 2018, ISSN: 1664-042X. DOI: `10.3389/fphys.2018.00128`. [Online]. Available: `https://www.frontiersin.org/article/10.3389/fphys.2018.00128`.

[40] cppreference.com, *Std::unordered_map*, `https://en.cppreference.com/w/cpp/container/unordered_map`, (Accessed on 06/18/2021).

[41] cppreference.com, *Std::map*, `https://en.cppreference.com/w/cpp/container/map`, (Accessed on 06/18/2021).

[42] A. Danial, *Cloc*, version 1.92, 2021. [Online]. Available: `https://github.com/AlDanial/cloc`.

Acronyms

FPS frames per second. 5
GPGE General Purpose Game Engine. 8, 9
HMD Head-Mounted Display. 1, 4, 10, 12
IDE Integrated Development Environment. 1, 5, 7, 11, 12
ITK The Insight ToolKit. 3
SLOCs Source Lines of Code. 14, 41, 42
UI User Interface. 1, 4
UX User Experience. 4
VR Virtual Reality. 1, 3–5, 7, 9, 11, 12, 22, 23, 42
VTK The Visualization ToolKit. 1, 3–5, 7–11, 14–18, 20–24, 26, 28, 33, 42, 52

Appendix A

Example VTK pipeline

```
#include <vtkQuadric.h>
#include <vtkSampleFunction.h>
#include <vtkContourFilter.h>
#include <vtkOutlineFilter.h>
#include <vtkPolyDataMapper.h>
#include <vtkActor.h>
#include <vtkProperty.h>
#include <vtkRenderWindow.h>
#include <vtkRenderer.h>
#include <vtkRenderWindowInteractor.h>
#include <vtkImageData.h>
#include <vtkNew.h>
#include <vtkPolyData.h>

int main(int arc, char* argv)
{
   vtkNew<vtkQuadric> quadric;
   quadric->SetCoefficients(.5,1,.2,0,.1,0,0,.2,0,0);

   vtkNew<vtkSampleFunction> sample;
   sample->SetSampleDimensions(200,200,200);
   sample->SetImplicitFunction(quadric);

   vtkNew<vtkContourFilter> contours;
   contours->SetInputConnection(sample->GetOutputPort());
   contours->GenerateValues(5,0.0,1.2);

   vtkNew<vtkPolyDataMapper> contMapper;
   contMapper->SetInputConnection(contours->GetOutputPort());
   contMapper->SetScalarRange(0.0,1.2);

   vtkNew<vtkActor> contActor;
   contActor->SetMapper(contMapper);

   vtkNew<vtkOutlineFilter> outline;
   outline->SetInputConnection(sample->GetOutputPort());

   vtkNew<vtkPolyDataMapper> outlineMapper;
   outlineMapper->SetInputConnection(outline->GetOutputPort());

   vtkNew<vtkActor> outlineActor;
   outlineActor->SetMapper(outlineMapper);
   outlineActor->GetProperty()->SetColor(0,0,0);

   vtkNew<vtkRenderer> ren;
   vtkNew<vtkRenderWindow> renWin;
   renWin->AddRenderer(ren);

   vtkNew<vtkRenderWindowInteractor> iren;
   iren->SetRenderWindow(renWin);

```

```
52     ren->AddActor(contActor);
53     ren->AddActor(outlineActor);
54     ren->SetBackground(1,1,1);
55
56     renWin->Render();
57     iren->Start();
58
59     return EXIT_SUCCESS;
60 }
```

Listing A.1: Example of Quadric Contour in VTK used to produce Figure 2.1.

Appendix B

vtkAdapterUtility code generator

In order to generate at before build a register of all the adapters added to the plugin, a Python script has been created to automate this process. This will generate the necessary support class used within the plugin, i.e. `vtkAdapterUtility`. This class comprises a fixed header file and a source file that changes depending on the installed adapters. The Listing B.1 shows the code of such script. An example of generated source file is shown in Listing B.2

```
import os

adapter_utility_h_name = "vtkAdapterUtility.h"
adapter_utility_cpp_name = "vtkAdapterUtility.cpp"
adapter_utility_class_name = "vtkAdapterUtility"
adapter_utility_getter_name = "GetAdapter"
adapter_utility_register_name = "s_adapters"
adapter_utility_register_type = "const std::unordered_map<LPCSTR, vtkAdapter*>"

adapter_utility_h = open(adapter_utility_h_name, "w")
adapter_utility_cpp = open(adapter_utility_cpp_name, "w")



### GENERATING HEADER FILE

adapter_utility_h.write("#pragma once\n")
adapter_utility_h.write("\n")
adapter_utility_h.write("/* This file has been automatically generated through the Python header generation utility\n")
adapter_utility_h.write(" * \n")
adapter_utility_h.write(" * This file contains the necessary information to allow the VtkToUnity plugin to know\n")
adapter_utility_h.write(" * that the adapters exist and it can call them. As such, this should be generated every\n")
adapter_utility_h.write(" * time the plugin is built to avoid losing any adapters in the compilation.\n")
adapter_utility_h.write(" */\n")
adapter_utility_h.write("\n")
adapter_utility_h.write("\n")

adapter_utility_h.write("// Utility includes\n")
adapter_utility_h.write("#include <unordered_map>\n")
adapter_utility_h.write("\n")
adapter_utility_h.write("#define NOMINMAX\n")
adapter_utility_h.write("#include <windows.h>\n")
adapter_utility_h.write("\n")
adapter_utility_h.write("#include \"../Singleton.h\"\n")
adapter_utility_h.write("#include \"../vtkAdapter.h\"\n")
adapter_utility_h.write("\n")


adapter_utility_h.write("\n")
adapter_utility_h.write("\n")
adapter_utility_h.write("// This class is used to register the adapters\n")
```

```
42 adapter_utility_h.write(f"class {adapter_utility_class_name}\n")
43 adapter_utility_h.write("{\n")
44
45 # Generating the class for the utility operations
46
47 adapter_utility_h.write("public:\n")
48
49 # begin public area
50
51 adapter_utility_h.write(f"\tstatic vtkAdapter* {adapter_utility_getter_name}(\n")
52 adapter_utility_h.write("\t\tLPCSTR vtkAdaptedObject);\n")
53 adapter_utility_h.write("\n")
54
55 # end public area
56
57 adapter_utility_h.write("private:\n")
58
59 # begin private area
60
61 adapter_utility_h.write("\t// Map with all the adapters registered in this folder\n")
62 adapter_utility_h.write(f"\tstatic {adapter_utility_register_type} {
       adapter_utility_register_name};\n")
63
64 # end private area
65
66 adapter_utility_h.write("};\n")
67
68
69
70 ### GENERATING SOURCE FILE
71
72 adapter_utility_cpp.write("/* This file has been automatically generated through the
       Python header generation utility\n")
73 adapter_utility_cpp.write(" * \n")
74 adapter_utility_cpp.write(" * This file contains the necessary information to allow the
       VtkToUnity plugin to know\n")
75 adapter_utility_cpp.write(" * that the adapters exist and it can call them. As such,
       this should be generated every\n")
76 adapter_utility_cpp.write(" * time the plugin is built to avoid losing any adapters in
       the compilation.\n")
77 adapter_utility_cpp.write(" */\n")
78 adapter_utility_cpp.write("\n")
79 adapter_utility_cpp.write("\n")
80 adapter_utility_cpp.write(f"#include \"{adapter_utility_h_name}\"\n")
81 adapter_utility_cpp.write("\n")
82 adapter_utility_cpp.write("// Adapters' header files found in the folder (.h and .hpp)\n
       ")
83
84 # Generating the includes of the adapters' header files
85
86 classes = []
87
88 for f in os.listdir("."):
89     if f != adapter_utility_h_name and ( f.endswith(".h") or f.endswith(".hpp") ):
90         adapter_utility_cpp.write(f"#include \"{f}\"\n")
91         class_name = os.path.splitext(f)[0]
92         classes.append(class_name[:1].upper() + class_name[1:])
93
94 adapter_utility_cpp.write("\n")
95 adapter_utility_cpp.write("\n")
96
97 # Creating the adapters' mapping
98
99 adapter_utility_cpp.write(f"{adapter_utility_register_type} {adapter_utility_class_name
       }::{adapter_utility_register_name} =");
100 adapter_utility_cpp.write("\n{\n")
101
102 # begin s_adapters init
103
104 for clss in classes:
105     adapter_utility_cpp.write(f"\t{{ Singleton<{clss}>::Instance()->GetAdaptingObject(),
           Singleton<{clss}>::Instance() }},\n")
106
```

```
107 # end s_adapters init
108
109 adapter_utility_cpp.write("};\n")
110
111 adapter_utility_cpp.write("\n")
112 adapter_utility_cpp.write("\n")
113
114 adapter_utility_cpp.write(f"vtkAdapter* {adapter_utility_class_name}::{
        adapter_utility_getter_name}(\n")
115 adapter_utility_cpp.write("\tLPCSTR vtkAdaptedObject)\n")
116 adapter_utility_cpp.write("{\n")
117
118 # begin GetAdapter
119
120 adapter_utility_cpp.write("\tauto itAdapter = s_adapters.find(vtkAdaptedObject);\n")
121 adapter_utility_cpp.write("\tif (itAdapter != s_adapters.end())\n")
122 adapter_utility_cpp.write("\t{\n")
123 adapter_utility_cpp.write("\t\treturn itAdapter->second;\n")
124 adapter_utility_cpp.write("\t}\n")
125 adapter_utility_cpp.write("\telse\n")
126 adapter_utility_cpp.write("\t{\n")
127 adapter_utility_cpp.write("\t\treturn NULL;\n")
128 adapter_utility_cpp.write("\t}\n")
129
130 # end GetAdapter
131
132 adapter_utility_cpp.write("}\n")
```

Listing B.1: generate-header.py script

```
1  /* This file has been automatically generated through the Python header generation
       utility
2   *
3   * This file contains the necessary information to allow the VtkToUnity plugin to know
4   * that the adapters exist, and it can call them. As such, this should be generated
       every
5   * time the plugin is built to avoid losing any adapters in the compilation.
6   */
7
8
9  #include "vtkAdapterUtility.h"
10
11 // Adapters' header files found in the folder (.h and .hpp)
12 #include "vtkConeSourceAdapter.h"
13
14
15 const std::unordered_map<LPCSTR, vtkAdapter*> vtkAdapterUtility::s_adapters ={
16     { Singleton<vtkConeSourceAdapter>::Instance()->GetAdaptingObject(), Singleton<
          vtkConeSourceAdapter>::Instance() },
17 };
18
19
20 vtkAdapter* vtkAdapterUtility::GetAdapter(
21     LPCSTR vtkAdaptedObject)
22 {
23     auto itAdapter = s_adapters.find(vtkAdaptedObject);
24     if (itAdapter != s_adapters.end())
25     {
26         return itAdapter->second;
27     }
28     else
29     {
30         return NULL;
31     }
32 }
```

Listing B.2: Example vtkAdapterUtility.cpp

Appendix C

Python/C++ Performance tests

To evaluate the best approach towards introducing introspection into our project, we developed four tests that determine the performances: Python directly accessing VTK (Listing), Python using introspection to access VTK (Listing), C++ using the embedded Python interpreter to directly access VTK (Listing) and C++ using the embedded Python interpreter using introspection to access VTK (Listing). In both Python and C++ we use helper functions that time the execution of the functions we call. Their implementations are respectively shown in Listings C.5 and Listings C.6. Finally, the introspective C++ code uses functions prefixed with `PyVtk_` which are the same functions presented in our plugin's implementation, adapted not to be run inside a Unity plugin[1].

```
reader = timed_execution("reader_inst", vtkStructuredGridReader)
timed_execution("reader_setfile", reader.SetFileName, ("density.vtk",))
timed_execution("reader_update", reader.Update)

seeds = timed_execution("seeds_inst", vtkPointSource)
timed_execution("seeds_setradius", seeds.SetRadius, (3.0,))
timed_execution("seeds_setcenter_reader_getoutput_getcenter", seeds.SetCenter, (reader.
    GetOutput().GetCenter(),))
timed_execution("seeds_setnumberofpoints", seeds.SetNumberOfPoints, (100,))

streamer = timed_execution("streamer_inst", vtkStreamTracer)
timed_execution("streamer_setinputconn_reader_getoutputport", streamer.
    SetInputConnection, (reader.GetOutputPort(0),))
timed_execution("streamer_setsourceconn_seeds_getoutputport", streamer.
    SetSourceConnection, (seeds.GetOutputPort(0),))
timed_execution("streamer_setmaxpropagation", streamer.SetMaximumPropagation, (1000,))
timed_execution("streamer_setinitialintegstep", streamer.SetInitialIntegrationStep,
    (.1,))
timed_execution("streamer_setintegdirboth", streamer.SetIntegrationDirectionToBoth)

outline = timed_execution("outline_inst", vtkStructuredGridOutlineFilter)
timed_execution("outline_setinputconn_reader_getoutputport", outline.SetInputConnection,
     (reader.GetOutputPort(0),))
```

Listing C.1: Native Python VTK benchmark script

```
introspector = timed_execution("introspector_inst", Introspector)

reader = timed_execution("reader_inst", introspector.createVtkObject, ("
    vtkStructuredGridReader",))
timed_execution("reader_setfile", introspector.setVtkObjectAttribute, (reader, "FileName
    ", "s", "density.vtk"))
timed_execution("reader_update", introspector.updateVtkObject, (reader,))

seeds = timed_execution("seeds_inst", introspector.createVtkObject, ("vtkPointSource",))
timed_execution("seeds_setradius", introspector.setVtkObjectAttribute, (seeds, "Radius",
     "f", 3.0))
```

[1]The full code is available on GitHub at `https://github.com/EnlitHamster/Python-Embedding`.

```
center = timed_execution("reader_getoutputgetcenter", introspector.genericCall, (
    introspector.vtkInstanceCall(reader, "GetOutput", ()), "GetCenter", ()))
timed_execution("seeds_setcenter_reader_getoutput_getcenter", introspector.
    setVtkObjectAttribute, (seeds, "Center", "f3", introspector.outputFormat(center)))
timed_execution("seeds_setnumberofpoints", introspector.setVtkObjectAttribute, (seeds, "
    NumberOfPoints", "d", 100))

streamer = timed_execution("streamer_inst", introspector.createVtkObject, ("
    vtkStreamTracer",))
timed_execution("streamer_setinputconn_reader_getoutputport", introspector.
    vtkInstanceCall, (streamer, "SetInputConnection", (introspector.
    getVtkObjectOutputPort(reader),)))
timed_execution("streamer_setsourceconn_seeds_getoutputport", introspector.
    vtkInstanceCall, (streamer, "SetSourceConnection", (introspector.
    getVtkObjectOutputPort(seeds),)))
timed_execution("streamer_setmaxpropagation", introspector.setVtkObjectAttribute, (
    streamer, "MaximumPropagation", "d", 1000))
timed_execution("streamer_setinitialintegstep", introspector.setVtkObjectAttribute, (
    streamer, "InitialIntegrationStep", "f", .1))
timed_execution("streamer_setintegdirboth", introspector.vtkInstanceCall, (streamer, "
    SetIntegrationDirectionToBoth", ()))

outline = timed_execution("outline_inst", introspector.createVtkObject, ("
    vtkStructuredGridOutlineFilter",))
timed_execution("outline_setinputconn_reader_getoutputport", introspector.
    vtkInstanceCall, (outline, "SetInputConnection", (introspector.
    getVtkObjectOutputPort(reader),)))
```

Listing C.2: Introspective Python VTK benchmark script

```
PyObject *init_python()
{
   /* Activating virtual environment */
   const wchar_t *sPyHome = L"venv";
   Py_SetPythonHome(sPyHome);

   /* Initializing Python environment and setting PYTHONPATH. */
   Py_Initialize();

   /* Both the "." and cwd notations are left in for security, as after being built in
      a DLL they may change. */
   PyRun_SimpleString("import sys\nimport os");
   PyRun_SimpleString("sys.path.append( os.path.dirname(os.getcwd()) )");
   PyRun_SimpleString("sys.path.append(\".\")");

   /* Decode module from its name. Returns error if the name is not decodable. */
   PyObject *pVtkModuleName = PyUnicode_DecodeFSDefault("vtk");
   if (pVtkModuleName == NULL)
   {
      fprintf(stderr, "Fatal error: cannot decode module name\n");
      return NULL;
   }

   /* Imports the module previously decoded. Returns error if the module is not found.
      */
   PyObject *pVtkModule = PyImport_Import(pVtkModuleName);
   Py_DECREF(pVtkModuleName);
   if (pVtkModule == NULL)
   {
      if (PyErr_Occurred())
      {
         PyErr_Print();
      }
      fprintf(stderr, "Failed to load \"Introspector\"\n");
      return NULL;
   }

   return pVtkModule;
}


PyObject *inst_vtkobj(
```

```
42      PyObject *pVtkModule,
43      LPCSTR classname)
44  {
45      /* Looks for the Introspector class in the module. If it does not find it, returns
            and error. */
46      PyObject* pVtkClass = PyObject_GetAttrString(pVtkModule, classname);
47      if (pVtkClass == NULL || !PyCallable_Check(pVtkClass))
48      {
49          if (PyErr_Occurred())
50          {
51              PyErr_Print();
52          }
53          fprintf(stderr, "Cannot find class \"Introspector\"\n");
54          if (pVtkClass != NULL)
55          {
56              Py_DECREF(pVtkClass);
57          }
58          return NULL;
59      }
60
61      /* Instantiates an Introspector object. If the call returns NULL there was an error
62          creating the object, and thus it returns error. */
63      PyObject *pInstance = PyObject_CallObject(pVtkClass, NULL);
64      Py_DECREF(pVtkClass);
65      if (pInstance == NULL)
66      {
67          if (PyErr_Occurred())
68          {
69              PyErr_Print();
70          }
71          fprintf(stderr, "Introspector instantiation failed\n");
72          return NULL;
73      }
74
75      return pInstance;
76  }
77
78
79  void test_native()
80  {
81      PyObject *pVtkModule = timed_execution<PyObject *>("python_inst", init_python);
82
83      PyObject *pReader = timed_execution<PyObject *>("reader_inst", inst_vtkobj,
            pVtkModule, "vtkStructuredGrid");
84      timed_execution_v("reader_setfile", PyObject_CallMethod, pReader, "SetFileName", "s",
             "density.vtk");
85      timed_execution_v("reader_update", PyObject_CallMethod, pReader, "Update", "");
86
87      PyObject *pSeeds = timed_execution<PyObject *>("seeds_inst", inst_vtkobj, pVtkModule,
             "vtkPointSource");
88      timed_execution_v("seeds_setradius", PyObject_CallMethod, pSeeds, "SetRadius", "d",
            3.0);
89      PyObject *pOutput = timed_execution<PyObject *>("reader_getoutput",
            PyObject_CallMethod, pReader, "GetOutput", "");
90      PyObject *pCenter = timed_execution<PyObject *>("output_getcenter",
            PyObject_CallMethod, pOutput, "GetCenter", "");
91      timed_execution_v("seeds_setcenter", PyObject_CallMethod, pSeeds, "SetCenter", "(d,d,
            d)", pCenter);
92      timed_execution_v("seeds_setnumberofpoints", PyObject_CallMethod, pSeeds, "
            SetNumberOfPoints", "i", 100);
93
94      PyObject *pStreamer = timed_execution<PyObject *>("streamer_inst", inst_vtkobj,
            pVtkModule, "vtkStreamTracer");
95      PyObject *pReaderPort = timed_execution<PyObject *>("reader_getoutputport",
            PyObject_CallMethod, pReader, "GetOutputPort", "i", 0);
96      PyObject *pSeedsPort = timed_execution<PyObject *>("seeds_getoutputport",
            PyObject_CallMethod, pSeeds, "GetOutputPort", "i", 0);
97      timed_execution_v("streamer_setinputconn", PyObject_CallMethod, pStreamer, "
            SetInputConnection", "O", pReaderPort);
98      timed_execution_v("streamer_setsourceconn", PyObject_CallMethod, pStreamer, "
            SetSourceConnection", "O", pSeedsPort);
99      timed_execution_v("streamer_setmaxpropagation", PyObject_CallMethod, pStreamer, "
            SetMaximumPropagation", "i", 1000);
```

```
100     timed_execution_v("streamer_setinitialintegstep", PyObject_CallMethod, pStreamer, "
            SetInitialIntegrationStep", "d", 0.1);
101     timed_execution_v("streamer_setintegdirboth", PyObject_CallMethod, pStreamer, "
            SetIntegrationDirectionToBoth", "");
102
103     PyObject *pOutline = timed_execution<PyObject *>("outline_inst", inst_vtkobj,
            pVtkModule, "vtkStructuredGridOutlineFilter");
104     timed_execution_v("outline_setinputconn", PyObject_CallMethod, pOutline, "
            SetInputConnection", "O", pReaderPort);
105
106     Py_XDECREF(pVtkModule);
107     Py_XDECREF(pReader);
108     Py_XDECREF(pSeeds);
109     Py_XDECREF(pStreamer);
110     Py_XDECREF(pOutline);
111
112     timed_execution_v("python_fin", Py_Finalize);
113 }
```

Listing C.3: Native C++ VTK benchmark source code.

```
1  void test_introspection()
2  {
3
4      PyObject *pIntrospector = timed_execution<PyObject *>("interpreter_init",
           PyVtk_InitIntrospector);
5
6      vtkObjectBase
7          *pReader = timed_execution<vtkObjectBase *>("reader_inst", PyVtk_CreateVtkObject,
               pIntrospector, "vtkStructuredGridReader"),
8          *pSeeds = timed_execution<vtkObjectBase *>("seeds_inst", PyVtk_CreateVtkObject,
               pIntrospector, "vtkPointSource"),
9          *pStreamer = timed_execution<vtkObjectBase *>("streamer_inst",
               PyVtk_CreateVtkObject, pIntrospector, "vtkStreamTracer"),
10         *pOutline = timed_execution<vtkObjectBase *>("outline_inst", PyVtk_CreateVtkObject
               , pIntrospector, "vtkStructuredGridOutlineFilter");
11
12     timed_execution_v("reader_setfile", PyVtk_SetVtkObjectProperty, pIntrospector,
           pReader, "FileName", "s", "density.vtk");
13
14     timed_execution_v("reader_update", PyVtk_ObjectMethod, // pReader->Update()
15         pIntrospector,
16         pReader,
17         "Update",
18         "",
19         std::vector<vtkObjectBase *>(),
20         std::vector<LPCSTR>());
21
22
23     PyObject *pOutput = timed_execution<PyObject *>("reader_getoutput",
           PyVtk_ObjectMethod, // pReader->GetOutput()
24         pIntrospector,
25         pReader,
26         "GetOutput",
27         "",
28         std::vector<vtkObjectBase *>(),
29         std::vector<LPCSTR>());
30
31     LPCSTR center = timed_execution<LPCSTR>("reader_getoutput_getcenter",
           PyVtk_PipedObjectMethodAsString, // pReader->GetOutput()->GetCenter()
32         pIntrospector,
33         pReader,
34         std::vector<LPCSTR>({ "GetOutput", "GetCenter" }),
35         std::vector<LPCSTR>({ "", "" }),
36         std::vector<vtkObjectBase *>(),
37         std::vector<LPCSTR>());
38
39     timed_execution_v("seeds_setradius", PyVtk_SetVtkObjectProperty, pIntrospector,
           pSeeds, "Radius", "f", "3.0");
40     timed_execution_v("seeds_setcenter", PyVtk_SetVtkObjectProperty, pIntrospector,
           pSeeds, "Center", "f3", center);
```

```
41    timed_execution_v("seeds_setnumberofpoints", PyVtk_SetVtkObjectProperty,
          pIntrospector, pSeeds, "NumberOfPoints", "d", "100");
42
43    vtkAlgorithmOutput *pSeedsPort = timed_execution<vtkAlgorithmOutput *>("
          seeds_getoutputport", PyVtk_GetOutputPort, pIntrospector, pSeeds);
44
45    timed_execution_v("streamer_setinputconn_reader", PyVtk_ConnectVtkObject,
          pIntrospector, pReader, (vtkAlgorithm *)pStreamer);
46    timed_execution_v("streamer_setsourceconn_seeds", PyVtk_ObjectMethod, // pStreamer->
          SetSourceConnection(pSeeds->GetOutputPort(0))
47        pIntrospector,
48        pStreamer,
49        "SetSourceConnection",
50        "o",
51        std::vector<vtkObjectBase *>({ pSeedsPort }),
52        std::vector<LPCSTR>());
53
54    timed_execution_v("streamer_setmaxpropagation", PyVtk_SetVtkObjectProperty,
          pIntrospector, pStreamer, "MaximumPropagation", "d", "100");
55    timed_execution_v("streamer_setinitialintegstep", PyVtk_SetVtkObjectProperty,
          pIntrospector, pStreamer, "InitialIntegrationStep", "f", "0.1");
56    timed_execution_v("streamer_setintegdirboth", PyVtk_ObjectMethod, // pStreamer->
          SetIntegrationDirectionToBoth()
57        pIntrospector,
58        pStreamer,
59        "SetIntegrationDirectionToBoth",
60        "",
61        std::vector<vtkObjectBase *>(),
62        std::vector<LPCSTR>());
63
64    timed_execution_v("outline_setinputconn", PyVtk_ConnectVtkObject, pIntrospector,
          pReader, (vtkAlgorithm *)pOutline);
65
66    timed_execution_v("interpreter_fin", PyVtk_FinalizeIntrospector, pIntrospector);
67 }
```

Listing C.4: Introspective C++ VTK benchmark source code.

```
1  time_execution_data = {}
2  def timed_execution(name, func, args = ()):
3      t_start = perf_counter()
4      r = func(*args)
5      time_execution_data[name] = perf_counter() - t_start
6      return r
```

Listing C.5: timed_execution Python function.

```
1  typedef std::chrono::high_resolution_clock::time_point time_var;
2
3  #define DURATION(a) std::chrono::duration_cast<std::chrono::nanoseconds>(a).count()
4  #define TIME_NOW() std::chrono::high_resolution_clock::now()
5
6  std::unordered_map<LPCSTR, double> time_execution_data;
7
8  template<typename R, typename F, typename... A>
9  R timed_execution(LPCSTR name, F function, A&& ...argv)
10 {
11     time_var start = TIME_NOW();
12     R r = function(std::forward<A>(argv)...);
13     time_execution_data.insert(std::make_pair(name, DURATION(TIME_NOW() - start) /
           1000000000.0f));
14     return r;
15 }
16
17 template<typename F, typename... A>
18 void timed_execution_v(LPCSTR name, F function, A&& ...argv)
19 {
20     time_var start = TIME_NOW();
21     function(std::forward<A>(argv)...);
```

```
22     time_execution_data.insert(std::make_pair(name, DURATION(TIME_NOW() - start) /
           1000000000.0f));
23 }
```

Listing C.6: timed_execution Cpp functions.

Appendix D

Interfaces' documentation

In the following pages, we present the API of different parts of the library, in particular Table D.1 describes the API the plugin exposes to Unity, Table D.2 the API the introspector exposes to the Infrastructure layer.

Function signature	Explenation
`int VtkResource_CallObject(` `LPCSTR classname)`	Instantiates an object of type classname, adds it to the object registry at both Introspection and Plugin level and returns the id in the plugin registry.

`int VtkResource_CallObjectAndShow(`
`LPCSTR classname,`
`Float4 color,`
`bool wireframe)`
	Same as above, but also wraps it in an actor with specified color and whether to show the wireframe.

`LPCSTR VtkResource_GetAttrAsString(`
`const int rid,`
`LPCSTR propertyName)`
	Retrieves the value of the specificed attribute and returns it as a string.

`void VtkResource_SetAttrFromString(`
`const int rid,`
`LPCSTR propertyName,`
`LPCSTR format,`
`LPCSTR newValue)`
	Sets a value from a string. The format identifies how the value has to be parsed.

`LPCSTR VtkResource_GetDescriptor(` `const int rid)`	Returns a string description of the object, in particular its attributes' name and type, comma separated.

`LPCSTR VtkResource_CallMethodAsString(`
`const int rid,`
`LPCSTR method,`
`LPCSTR format,`
`const char *const *argv)`
	Calls a method on a given object and returns its return value as a string.

`int VtkResource_CallMethodAsVtkObject(`
`const int rid,`
`LPCSTR method,`
`LPCSTR format,`
`LPCSTR classname,`
`const char *const *argv)`
	Calls a method on a given object and returns its return value as a VTK object pointer.

`void VtkResource_CallMethodAsVoid(`
`const int rid,`
`LPCSTR method,`
`LPCSTR format,`
`const char *const *argv)`
	Calls a method on a given object and ignores the return value.

Function signature	Explenation

`LPCSTR VtkResource_CallMethodPipedAsString(`
`const int rid,`
`const int methodc,`
`const int formatc,`
`const char *const *methodv,`
`const char *const *formatv,`
`const char *const *argv)`
	Calls a methods pipeline starting from a given object and returns its final return value as a string.

`int VtkResource_CallMethodPipedAsVtkobject(`
`const int rid,`
`const int methodc,`
`const int formatc,`
`LPCSTR classname,`
`const char *const *methodv,`
`const char *const *formatv,`
`const char *const *argv)`
	Calls a methods pipeline starting from a given object and returns its final return value as a VTK object pointer.

`void VtkResource_CallMethodPipedAsVoid(`
`const int rid,`
`const int methodc,`
`const int formatc,`
`const char *const *methodv,`
`const char *const *formatv,`
`const char *const *argv)`
	Calls a methods pipeline starting from a given object and ignores its final return value.

`void VtkReousrce_DeleteObject(` `const int rid)`	Deletes an object and its wrappers, freeing its memory space.

`void VtkResource_Connect(`
`LPCSTR connectionType,`
`const int sourceRid,`
`const int targetRid)`
	Connects the sources output to the targets input connection. The connection type identifies the method to be called.

`void VtkResource_AddActor(`
`const int rid,`
`Float4 color,`
`bool wireframe)`
	Adds mapper and actor to the source and adds it to the renderer.

`bool VtkError_Occurred()`	Checks whether an error occurred on the last call of VtkResource.

Function signature	Explenation
`LPCSTR VtkError_Get()`	Returns the last error occurred on VtkResource. It should always be called, in conjunction with `VtkError_Occurred()`, after any VtkResource call.

Table D.1: Plugin API exposed to Unity.

Function signature	Explenation
`vtkObjectBase *CreateObject(` `LPCSTR classname)`	Instantiates an object of class classname and adds it to the mapping from VTK object to Python object.

`LPCSTR GetProperty(`
`vtkObjectBase *pVtkObject,`
`LPCSTR propertyName)`
	Retrieves the value of the specificed attribute and returns it as a string.

`void SetProperty(`
`vtkObjectBase *pVtkObject,`
`LPCSTR propertyName,`
`LPCSTR format,`
`LPCSTR newValue)`
	Sets a value from a string. The format identifies how the value has to be parsed.

`LPCSTR GetDescriptor(` `vtkObjectBase *pVtkObject)`	Returns a string description of the object, in particular its attributes' name and type, comma separated.

`LPCSTR CallMethod_AsString(`
`vtkObjectBase *pVtkObject,`
`LPCSTR method,`
`LPCSTR format,`
`vector<vtkObjectBase *> refv,`
`vector<LPCSTR> argv)`
	Calls a method on a given object and returns its return value as a string.

`vtkObjectBase *CallMethod_AsVtkObject(`
`vtkObjectBase *pVtkObject,`
`LPCSTR method,`
`LPCSTR format,`
`LPCSTR classname,`
`vector<vtkObjectBase *> refv,`
`vector<LPCSTR> argv)`
	Calls a method on a given object and returns its return value as a VTK object pointer.

`void CallMethod_AsVoid(`
`vtkObjectBase *pVtkObject,`
`LPCSTR method,`
`LPCSTR format,`
`vector<vtkObjectBase *> refv,`
`vector<LPCSTR> argv)`
	Calls a method on a given object and ignores the return value.

Function signature	Explenation

`LPCSTR CallMethodPiped_AsString(`
`vtkObjectBase *pVtkObject,`
`vector<LPCSTR> methods,`
`vector<LPCSTR> formats,`
`vector<vtkObjectBase *> refv`
`vector<LPCSTR> argv)`
	Calls a methods pipeline starting from a given object and returns its final return value as a string.

`vtkObjectBase *CallMethodPiped_AsVtkObject(`
`vtkObjectBase *pVtkObject,`
`vector<LPCSTR> methods,`
`vector<LPCSTR> formats,`
`LPCSTR classname,`
`vector<vtkObjectBase *> refv,`
`vector<LPCSTR> argv)`
	Calls a methods pipeline starting from a given object and returns its final return value as a VTK object pointer.

`void CallMethodPiped_AsVoid(`
`vtkObjectBase *pVtkObject,`
`vector<LPCSTR> methods,`
`vector<LPCSTR> formats,`
`LPCSTR classname,`
`vector<vtkObjectBase *> refv,`
`vector<LPCSTR> argv)`
	Calls a methods pipeline starting from a given object and ignores its final return value.

`void DeleteObject(` `vtkObjectBase *pVtkObject)`	Deletes an object and its wrappers, freeing its memory space.
`bool ErrorOccurred()`	Checks whether an error was registered in the error buffer.
`LPCSTR ErrorGet()`	Returns the last error occurred on an introspection call.

Table D.2: Introspective component's API.

Appendix E

Unity test scenes

This chapter presents the relevant code used to create the scenes used to validate the presented plugin.

```
/* SNIPPET A */
// Create a backend VTK object through the Introspector.
int id = VtkToUnityPlugin.VtkResource_CallObject("vtkConeSource");

// Set its attributes.
VtkUnityWorkbenchPlugin.SetProperty(id, "Height", "f", 0.1f.ToString());
VtkUnityWorkbenchPlugin.SetProperty(id, "Radius", "f", 0.1f.ToString());
VtkUnityWorkbenchPlugin.SetProperty(id, "Resolution", "d", 200.ToString());

// Directly connect the source to an actor in the backend.
VtkToUnityPlugin.VtkResource_AddActor(id, colour, false);

/* SNIPPET B */
// Defines the parameters of the Cone source for updating
struct Preset
{
    public float Height;
    public float Radius;
    public int Resolution;
}

// Determines which preset to use
private bool useDefault = true;
private Preset defaultPreset = new Preset();
private Preset transformedPreset = new Preset();

// Co-routine to be run alongside the updates to simulate real-time editing
IEnumerator TransformCone()
{
    for(;;)
    {
        // Changes all cones on the scene and yield again in 2 seconds
        foreach (var idPosition in _shapeIdPositions)
        {
            Preset p = useDefault ? defaultPreset : transformedPreset;
            int id = idPosition.Id;
            VtkUnityWorkbenchPlugin.SetProperty(id, "Height", "f", p.Height.ToString());
            VtkUnityWorkbenchPlugin.SetProperty(id, "Radius", "f", p.Radius.ToString());
            VtkUnityWorkbenchPlugin.SetProperty(id, "Resolution", "d", p.Resolution.
                ToString());

        }
        useDefault = !useDefault;
        yield return new WaitForSeconds(2.0f);
    }
}
```

Listing E.1: C# calls to create a VTK cone source in Unity (Snippet a) and to update it through a co-routine (Snippet b)

```
// Creating the VTK resources
int reader = VtkToUnityPlugin.VtkResource_CallObject("vtkStructuredGridReader");
int seeds = VtkToUnityPlugin.VtkResource_CallObject("vtkPointSource");
int streamer = VtkToUnityPlugin.VtkResource_CallObject("vtkStreamTracer");
int outline = VtkToUnityPlugin.VtkResource_CallObject("vtkStructuredGridOutlineFilter");

VtkToUnityPlugin.VtkResource_SetAttrFromString(reader, "FileName", "s", "Data/density.
    vtk");
VtkToUnityPlugin.VtkResource_CallMethodAsVoid(reader, "Update", "", IntPtr.Zero);

string center = VtkToUnityPlugin.VtkResource_CallMethodPipedAsString(
    reader,
    2,
    2,
    VtkUnityWorkbenchHelpers.MarshalStringArray(new string[] { "GetOutput", "GetCenter"
        }),
    VtkUnityWorkbenchHelpers.MarshalStringArray(new string[] { "", "" }),
    IntPtr.Zero);

VtkToUnityPlugin.VtkResource_SetAttrFromString(seeds, "Radius", "f", 3.0f.ToString());
VtkToUnityPlugin.VtkResource_SetAttrFromString(seeds, "Center", "f3", center);
VtkToUnityPlugin.VtkResource_SetAttrFromString(seeds, "NumberOfPoints", "d", nPoints.
    ToString());

VtkToUnityPlugin.VtkResource_Connect("Input", reader, streamer);
VtkToUnityPlugin.VtkResource_Connect("Source", seeds, streamer);
VtkToUnityPlugin.VtkResource_SetAttrFromString(streamer, "MaximumPropagation", "d", 100.
    ToString());
VtkToUnityPlugin.VtkResource_SetAttrFromString(streamer, "InitialIntegrationStep", "f",
    0.1f.ToString());
VtkToUnityPlugin.VtkResource_CallMethodAsVoid(streamer, "SetIntegrationDirectionToBoth",
     "", IntPtr.Zero);

VtkToUnityPlugin.VtkResource_Connect("Input", reader, outline);

VtkToUnityPlugin.VtkResource_AddActor(streamer, colour, false);
VtkToUnityPlugin.VtkResource_AddActor(outline, colour, false);
```

Listing E.2: C# calls to create a VTK stream tracer in Unity.