TU WIEN Informatics

Scripting Automation for Tamashii

Implementierung und Demonstration einer Python-Einbindung in ein modulares Rendering-Framework

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Matthias Preymann
Matrikelnummer 12020638

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: David Hahn, Phd
Dipl.-Ing. Lukas Lipp

Wien, 18. August 2023

Matthias Preymann	Michael Wimmer

TU WIEN Informatics

Scripting Automation for Tamashii

Implementation and demonstration of Python bindings for a modular rendering framework

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Matthias Preymann
Registration Number 12020638

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: David Hahn, Phd
Dipl.-Ing. Lukas Lipp

Vienna, 18th August, 2023

Matthias Preymann	Michael Wimmer

Erklärung zur Verfassung der Arbeit

Matthias Preymann

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 18. August 2023

Matthias Preymann

Danksagung

Zunächst möchte ich mich bei meinen Eltern bedanken, die mir nicht nur das Studium ermöglichen, sondern mich auch sonst in allen Bereichen des Lebens unterstützen und fördern. Ihnen habe ich so viel mehr zu verdanken, als mit dieser Abschlussarbeit in Zusammenhang steht. Im Weiteren möchte ich auch meiner restlichen Familie, meinen Freunden und im Speziellen meinen Mitstudenten meiner Studiengruppe meinen Dank aussprechen, die mir in allen Phasen dieser Arbeit Mut und Motivation zugesprochen haben.

Abschließend möchte ich meinen beiden Betreuern, David Hahn und Lukas Lipp danken, die mir im Verlauf dieses Jahres immer mit Rat und Tat zur Seite standen. Neben ihrer heiteren und herzlichen Art, sowie ihrer Bereitschaft zum freundschaftlichen technischen Diskurs, möchte ich vor allem ihre Hilfsbereitschaft honorieren.

Acknowledgements

First of all, I would like to thank my parents, who not only make it possible for me to study, but also support and encourage me in all other aspects of life. I owe so much more to them than what is merely related to this thesis. Furthermore, I would also like to express my gratitude to the rest of my family, my friends, and especially my fellow students in my study group, who gave me encouragement and motivation throughout all phases of this thesis.

Finally, I would like to thank my two supervisors, David Hahn and Lukas Lipp, who were always there to support and advise me throughout this year. Besides their cheerful and cordial manner, as well as their willingness to engage in friendly technical discourse, I would especially like to honor their continued helpfulness.

Kurzfassung

Wir erweitern das wissenschaftliche Rendering-Framework Tamashii um eine Schnittstelle zur Skriptsprache Python, um den Prozess der Dokumentation und des Vergleichens von Ergebnissen unterschiedlicher Ansätze zu automatisieren, die Entwicklung von experimentellem Code zu vereinfachen und eine nahtlose Integration mit bestehenden Bibliotheken zu ermöglichen. Das Framework ist eine Forschungsplattform, die die Implementierung verschiedener graphics processing unit (GPU)-beschleunigter (differenzierbarer) Rendering-Aufgaben ermöglicht und derzeit vom Institut für Computergrafik der TU Wien entwickelt wird. Tamashii bietet eine große Anzahl an vorgefertigten Funktionen für diese Arbeitsabläufe, die von Forschern genutzt werden können, um ihre eigenen benutzerdefinierten Implementierungen zu erstellen. Der Schwerpunkt dieser Arbeit liegt auf der Integration der Programmiersprache Python in das Framework, so dass alle Projekte davon profitieren, die Tamashii für ihre Forschung nutzen. Mit dieser Erweiterung können die Schritte des Ladens von Szenen, des Konfigurierens des Rendering-Prozesses und des Exports der generierten Daten nun durch Skripting gesteuert werden. Darüber hinaus eröffnet die Möglichkeit, Python direkt einzusetzen, den Zugang zu einem ganzen Ökosystem von Software und Bibliotheken von Drittanbietern, das viele gängige Projekte für Optimierungs- und maschinelle Lernalgorithmen umfasst. In dieser Arbeit identifizieren wir die erforderlichen Merkmale und Eigenschaften einer solchen Schnittstelle, erläutern den Entwurfsprozess, geben technische Details zur Realisierung und evaluieren schließlich das Ergebnis, indem wir seine Verwendung demonstrieren.

Abstract

We extend the Tamashii scientific rendering framework with an interface to the Python scripting language to automate the process of documenting and comparing results of different approaches, simplifying the development of experimental code, and seamlessly integrating with existing libraries. The framework is a research platform enabling the implementation of various graphics processing unit (GPU) driven (differentiable) rendering tasks and is currently under development by the institute for computer graphics at TU Wien. Tamashii offers a large set of premade functionality common to these workflows that can be leveraged by researchers to create their own custom implementations. The focus of this thesis is the integration of the Python programming language into the framework in a way that benefits all projects utilizing Tamashii in their research. With this addition the steps of loading scenes, configuring the rendering process and exporting the generated data can now be controlled using scripting. Furthermore, with the ability to employ Python directly, access to the whole ecosystem of third party software and libraries opens up, which includes many prevalent projects for optimization and machine learning algorithms. In this thesis we identify the required features and properties of such an interface, explain the design process, give technical details on the realization and finally evaluate the result by demonstrating its use.

Contents

CHAPTER 1

Introduction

The Tamashii scientific rendering framework is a modular research platform that is currently developed by the institute for computer graphics at TU Wien. Its aim is to provide common functionality needed to experiment with novel ideas and algorithms in this field, so that researchers can focus on their innovations instead of recreating common fundamentals each time. For this reason, Tamashii's source code is written in C++, an object oriented systems programming language, that is well suited for high performance applications, which require direct control over the computer's resources and native access to the operating system. Other languages like C and Rust are also commonly used for similar purposes, and although they follow different philosophies in their design, good runtime performance is their common goal. However, despite the many upsides of these kinds of languages, like performance and good support, they are not a golden bullet for any kind of software development problem. One of their main strengths is, introducing only minimal abstractions on top of the computing hardware. But this characteristic often forces developers to write more complex and longer code. Hence, these codebases are more difficult and error-prone to use, harder to maintain and require a lot of knowledge and experience to utilize them effectively. In contrast to that, there are higher-level scripting languages. These languages trade runtime performance and static analysis like type-safety for shorter, more expressive and simpler code. They are therefore easier to use and allow for quicker development, as they intentionally hide many complexities and details, such as the management of dynamic resources. This difference makes them ideal for experimental and highly customized applications that come with frequent code alterations and therefore need quick iteration cycles. It also for example allows researchers to focus on their problem in question like a specific algorithm, instead on implementing details such as resource management or performant filesystem operations.

For this reason, it is common practice for software projects that have to deal with both requirements, to incorporate an interface that facilitates the interoperability between a

high performance language for computationally expensive operations, and a scripting language that serves as an environment for realizing the remaining logic of the application. The proliferation of apps built with the Electron framework [Fc23] is a remarkable example of the utility of combining a performance oriented core system with a modern and convenient scripting language. Similarly, we aim for Tamashii to employ this strategy to further its goal of improving the development experience for researchers in the field of computer graphics. Here, we can split the implementation of graphics tasks into a computational part consisting of the rendering algorithms, and a test driving part containing code supporting parameterization or documentation for example. While the former requires the computational performance of a low-level systems language like C++, the latter can also be accomplished in a language that focuses more on developer comfort. We intend to improve the research and development experience with Tamashii by introducing access to a scripting language, that allows controlling, automating and extending of algorithms written in C++. Thus, we put a high priority on interoperability with custom graphics tasks created as Tamashii implementations.

One such graphics task is the optimization of light sources in a predefined scene to match specific lighting objectives including the lighting direction, illumination intensity, and color measured near mesh vertices. Different optimization strategies can be applied to find optimal solutions in this defined problem space with varying effectiveness. In recent years major advances in the research of machine learning (ML) were made like neural radiance fields, which are now being applied in the field of rendering. These ideas could be explored as a computationally more efficient heuristic for these kinds of optimization problems. However, such research could greatly benefit if it had the opportunity to rely on code provided by other projects and hinges on the ability to effectively test each variant. In this case, Python enables access to the most commonly used libraries for ML applications and offers adequate automation capabilities. Since ML research mostly happens in the Python domain, effective interaction with the language opens the door to the wealth of modern algorithms, pre-trained models and additional infrastructure, necessary for data preparation, training, objective formulation and evaluation.

But creating a scripting interface for an existing application of substantial size such as Tamashii is a task with multiple degrees of freedom, that has many potential solutions. There are many things to consider that depend on the envisioned general use-case, the technologies employed in the pre-existing codebase, desirable and already present core paradigms in the design and even specific third-party projects whose compatibility needs to be taken into account. While this list is not exhaustive, it shows how diverse the aspects of the given problem space are. Even the selection of the programming language used for scripting is not an obvious choice, as there are several mature and capable options available. However, given the focus on differentiable programming and rendering, the preferred selection of tooling and languages in academic and open-source projects mostly boils down to a few major ones. Furthermore, while the paradigms guiding the design of the interface are also a matter of taste to a certain degree, they are mostly predetermined by the underlying technologies already in use. One example is the

approach for resource management during runtime, most notably memory management. The designs of programming languages are usually highly opinionated in this regard, hence making it one of their fundamental traits. This circumstance can make interoperability, as it is required for a scripting interface crossing the boundaries of two different language environments, a challenging task that necessitates knowledge of the different approaches in use.

By enabling researchers to express more of their work in a simpler and more descriptive language, we expect to improve their workflow and development experience with the Tamashii framework. The Python interface we propose is intended to facilitate the creation of experimentation code that automates interaction with and testing of Tamashii implementations. Additionally, the interface allows making use of libraries provided through the script language for extending the features of Tamashii and the implementation. As a way to evaluate the results of our efforts, we present three examples that show possible applications and demonstrate the abilities of the interface.

This thesis is structured into three sections, that each focus on separate aspects of the introduction of modular Python bindings to Tamashii. First, we summarize the relevant theoretical background, next we outline the process of conceptualizing and then furthermore realizing the infrastructure for connecting to Python, and finally we demonstrate and evaluate the result. An introduction to related topics and previously published work is given in Chapter 2. We explain the concept behind scripting languages, how they differ from other kinds of programming languages, and how they can enrich the development experience, also giving examples on their domain specific strengths. Further, we give the reader an overview on the basics of differentiable programming, the challenges it tries to solve and highlight implementation approaches. This topic is expanded with differentiable rendering, which relies on the former, to solve problems in the field of computer graphics related to optimization and ML. Lastly, we illustrate the research fields of surrogate modeling and neural rendering on the basis of the surrogate modeling toolbox (SMT) for Python and neural radiance fields (NeRF). Then, in Chapter 3 we give a deeper background on the Tamashii rendering framework, its application, technologies and the challenges currently faced. Moreover, we summarize the principles of the Python programming language and the significant parts of the inner workings of its reference implementation. Based on that, we describe the techniques to extend Python's interpreter with custom code, and what bindings libraries offer to improve its application programming interface (API). The following Chapter 4 lays out the design, which we derive from the principles explained in literature, specific needs imposed by Tamashii and its unique use case, and experience gathered from testing code written before direct access to Python. At the end, we state and explain the decisions that led to the ultimate structure of the interface provided by the bindings. In Chapter 5 we discuss concrete parts of the implementation that are of particular interest as they solve requirements distinct to Tamashii. There, we recount the creation of a new library for the management of global configuration variables and we lay out the specifics of the memory management of objects sharing live state. We create bindings for the main components

of the Tamashii framework, plus custom bindings for one of the research implementations as a proof of concept, as shown in Chapter 6. Here, three examples are made, that use the bindings to achieve testing and automation tasks. In the end, Chapter 7 discusses the work, concludes the findings gathered, lists shortcomings and gives insights into final thoughts. Finally, we give an outlook into potential future work related to this thesis, which suggests several topics that tie in with the limitations of this work.

CHAPTER 2

Related Work

Thanks to a healthy and steadily growing ecosystem of scripting languages, there is a wide selection of languages to choose from when integrating one of them into a project. This chapter discusses the principles behind scripting languages in general and highlights Lua and Python as prevalent examples in use today. Moreover, we mention projects utilizing them based on their domain specific needs, and lastly demonstrate the family of vectorized scripting languages as representatives of languages focused on a narrower, more specialized problem space. We discuss the ways we can realize dynamic resource management and contrast the characteristics of automatic versus manual memory management. In addition, we show state of the art implementations of garbage collection (GC) systems of well-known runtime environments and illustrate their working principles illustrated. Furthermore, we introduce the field of differentiable programming and give an overview of its underlying theory. Several projects implementing automatic differentiation are discussed, and the relationship between gradient computation and numeric optimization, and likewise machine learning, is described. The topic is then expanded on with differentiable rendering, naming projects that approach the problem in different ways. Then, we display neural rendering as an emerging field of research with a focus on neural radiance fields and how others have improved on the original publication. Finally, we summarize the basics of surrogate modeling, the way it can help to speed up optimization and simulation tasks in engineering workflows. There, we show the SMT Python module as an illustrative library that implements modern modeling algorithms.

2.1 Scripting Languages

There is a myriad of different programming languages following many diverse ideas trying to satisfy varying requirements. These conditions include safety, execution speed, ease-of-use and learnability, development speed, expressiveness, compatibility, verifiability, extensibility and many others. As most of these aspects oppose each other in one way or

another, focusing on certain traits impacts the others, forcing trade-offs and giving each language unique advantages and disadvantages. These properties also allows them to be assorted into families [Sco09]. For example, a language designer putting high importance on speed during runtime, needs to consider the computing hardware underlying the execution of the program. Either this hypothetical language has to forgo high level constructs and abstractions, or a compilation step of some sort is necessary. While there is again more than one option like full ahead of time (AOT) compilation or employing a just in time (JIT) compiler, time will be spent compiling ultimately degrading the development or user experience during startup, interoperability gets more complicated and questions about security are raised when executing unknown code outside an interpreter's confines known as a sandbox.

Scripting languages are a family of programming languages that are designed for scripting and automation tasks. Their main use-case is sequencing commands for other components, serving as a coordinator across large codebases that offer complex functionality. However, they are fully capable languages allowing the creation of complex and feature-rich libraries and applications on their own. Therefore, their design makes decisions towards fast, accessible, and scalable development. The goal is to hide technical details by offering high level interfaces, allow for rapid development by using an interpreter eliminating a compilation step, utilizing expressive syntax that incorporates high level data structures, and offer domain specific features to easily convey problems in their field of use. Further, they try to reduce mental load on the developer by automating resource management through systems like GC and avoiding undefined behavior by doing run time checks and clearly communicating errors. This assistance however comes with drawbacks, primarily less runtime performance. Due to that, scripting environments often rely on tightly integrated libraries written in better performing languages that provide implementations for computationally expensive algorithms.

A well-established option in the scripting language family is Lua ("moon" in Portuguese) [IdFC23], developed by the Department of Computer Science at PUC-Rio[1] to replace two other domain specific languages with a single general purpose one. It has seen wide adoption in many areas of software development including academic projects like LuaTorch, which is the precursor to the popular PyTorch library [PGM+19a]. Lua has a lightweight (<300kiB when compiled for 64bit), embeddable and performant interpreter with a JIT compiler as an option. Thanks to its small size and novel register-based interpreter which allows for more efficient code execution [IdFC07], it is very popular with game engine developers who embed the complete runtime environment into their software. However, there are also many other products that rely on Lua like Adobe's Photoshop Lightroom. Another project with historical significance is Torch-Lua [CKF11] with its last version Torch7 which provides a Lua environment for the development of numerical computer programs on top of a C++ core library. While Torch7 is not actively developed anymore it became the predecessor of the now widely used PyTorch [Chi21] library.

[1] Pontifćia Universidade Católica do Rio de Janeiro/Pontifical Catholic University of Rio de Janeiro

PyTorch uses Python as its scripting language, which is another general purpose interpreted programming language. It was first developed by Guido van Rossum and focuses on being easy to learn by beginners but enjoys great adoption in nearly all fields of software development today. In the field of machine learning Python is especially prevalent [Sca23] [RPN20].

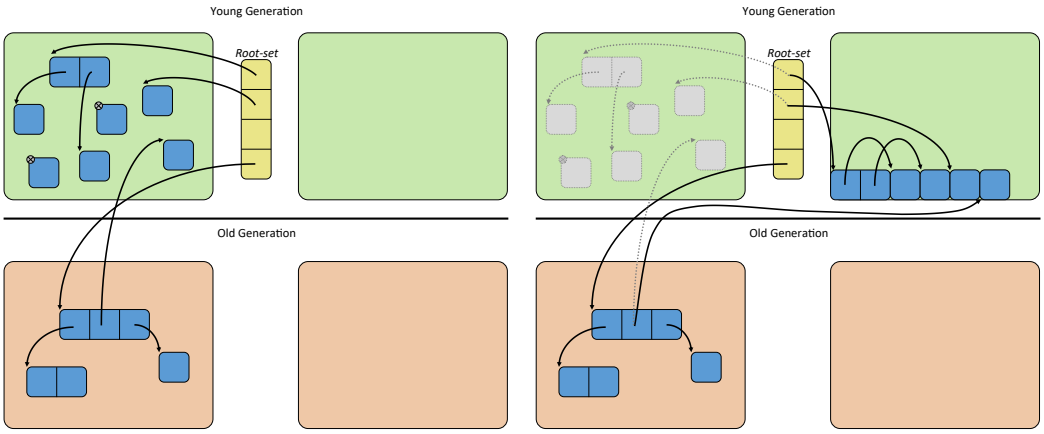
While Lua and Python follow a style oriented towards general purpose problems, there are also more domain specific languages which target a narrower problem space. The languages R and MATLAB are examples from this category. Both are vectorized languages that natively interact with multi-dimensional data objects like vectors and matrices. R's strengths lie in the field of statistical computing and data visualization, where it is the de facto standard for academic work. MATLAB (short for matrix laboratory) finds many applications in engineering tasks like technical simulations and signal processing. Developed by Cleve Moler, it has been a commercialized product offered by MathWorks since 1984. GNU Octave tries to be a compatible, free and open alternative to MATLAB developed by the community and licensed under the GNU General Public License (GPL) [Eat88].

2.2 Memory Management

As previously mentioned, scripting languages opt for automatic resource management in most cases to free the developer from the mental load of often error-prone manual management. Programs written in this way do not explicitly declare when values are dropped or become unused. Instead, the program flow just stops referencing the associated memory sections. Normally this would mean, that fresh previously unused memory is needed whenever new values have to be stored, which would then increase the program's memory consumption as it executes, eventually exhausting the computer's hardware resources, a so-called "memory leak". For this reason, a garbage collector runs in the background invisible to the program that detects memory sections no longer in use and makes them available again for fresh allocations.

There are multiple approaches to realize such automatic memory reclamation with varying levels of complexity and advantages depending on the specific use-case as described by Wilson [Wil06]. The common Mark-and-Sweep algorithm acts in two phases: First all values in the dynamic heap memory that qualify for collection are found and marked. Subsequently, in the second step said values are freed, leaving empty spaces behind that can be filled with new data. To perform the first half, the garbage collecting algorithm has to find a root set of references that point into the heap. For example, local variables on the stack, global and static variables can be candidates to check. Next all references part of any objects found need to be followed as well. Cyclic graphs of references can be handled by enforcing that every object is only marked once. One downside of this method is the high amount of fragmentation left behind, which leads to ineffective memory utilization and possibly unideal locality properties.

Compacting garbage collectors try to prevent this, by moving surviving objects to a



(a) There are two unreferenced objects in the young generation before collection starts.

(b) After collection and compaction the two objects are eliminated. All references into the now empty left side of the young generation have to be patched.

Figure 2.1: Operations of a teared compacting GC.

new section of memory where they are tightly packed, as illustrated by Figure 2.1. This action, however, requires patching all memory addresses of references that pointed to these objects, and in addition possibly a lot of time is spent copying. On the other hand, by moving the objects into a compacted area a large completely empty area is left behind shown in Figure 2.1b, that enables very quick allocation similar to a stack, known as Arena allocation. Since the compacted area must be analyzed again eventually too, another optimization can be employed that tries to reduce the time spent marking objects. Based on the assumption that there are distinct lifetime groups of objects multiple tiered compaction areas can be created that are marked with different frequencies. Values that are only allocated to be used once never reach the first compaction section and belong to the youngest generation which is cleaned up immediately. Objects of the older generation stored in the compacted area are marked less frequently and due to occurring in longer sections of the program's flow. These leftovers can be moved to another area where they are checked even less often if they reach a certain age. The objects from this generation are usually values in a program that never change like global constants.

The work of these processes scales with the amount of heap memory in use, which can lead to unexpected behavior of the program execution, if it is halted for the whole duration of marking, collecting and moving. Especially for real time applications such long pauses of so-called stop-the-world garbage collectors are unacceptable. Therefore, even more complicated setups exist that split the work over multiple iterations achieving collection incrementally, or completely in parallel to the program execution.

Well known examples of languages relying on garbage collection are Java, JavaScript and Python. Java‘s runtime environment, the Hotspot Java virtual machine (JVM),

comes with multiple GC implementations, that can be selected on startup. The older Serial, Parallel and CMS collectors have been introduced and deprecated in past Java versions. The fully parallel G1 collector is currently the default, and performs very well compared to its predecessors [PGM19b]. It manages multiple blocks of virtual memory areas, regularly determines the one containing the most garbage, and does the collection and compaction in parallel to keep pause times low [Cor23].

JavaScript is implemented by all modern web browsers as custom JavaScript engines that parse, compile and execute the code. Apple's WebKit features a non-compacting, but generational GC, that performs the marking phase in parallel to the execution of the program. It uses intricate lock-free algorithms and memory layouts to allow for seamless operations of the single mutator and multiple marking threads [Xu22]. The developers of Chromium, which is at the heart of many web browsers such as Chrome, Opera and Microsoft Edge, created the Oilpan GC that is able to do cross-component collection. This strategy tackles the problems of the heterogeneous environment that results from the Blink rendering engine written in C++ and the memory managed by the V8 JavaScript engine. Issues like use-after free and especially leaked memory caused by reference cycles were a common occurrence [DEH+18]. Thus, by allowing Oilpan to also collect objects from the C++ heap, reference cycles between the tightly coupled components become trivial to handle.

The Python programming language also requires a collection scheme to reclaim memory during runtime. The CPython interpreter however does not entirely rely on a mark-and-sweep GC to identify unused objects. Instead, reference counting is the primary approach employed [Sal23]. As Python treats any value as an object including numbers such as integers, there is always a large number of heap allocations to manage, of which many do not reference any other objects. Reference counting works in real-time without pausing and does not require involved sub-systems in the interpreter. However, reference cycles are inevitable when working with complex data structures, which lead to memory leaks if no additional measures are taken. Therefore, a complementing generational GC exists that finds unreachable reference cycles to deallocate. Objects that can be proven to never form cycles, like numbers, are except from the GC's traversal.

Even efficient GC implementations involve significant work during runtime, which is why many languages forgo this concept and require the programmer to manually manage dynamically allocated memory by explicitly freeing it. The C++ programming language works on this principle with the language keywords `new` and `delete`, which constitute the language's fundamental dynamic memory management. For each memory allocation initiated by a `new` expression, a corresponding `delete` one must exist, else memory is leaked during runtime when the reference to the memory is lost. This system is complemented by the Resource acquisition is initialization (RAII) paradigm which uses the statically determined life-times of variables and values to call constructor and destructor methods of objects [SSvcm23b]. This way, more complex automated memory management strategies can be implemented on top of the simple new and delete calls.

These strategies are implemented in the standard template library (STL) using templated,

class-based reference types called smart pointers, which hold an internal pointer to a memory allocation and provide an interface for moving and sharing ownership of a memory block. If it can be guaranteed that no owners are left, the memory can be freed and returned to the system [SSvcm23e]. They realize this behavior with the help of reference counting, where each holder increments a counter dedicated to the allocation when it takes ownership. When ownership is relinquished, the same counter is decremented until it reaches zero at some point marking the object safe to be deallocated. The STL offers several pointer types that all center around this general concept, but allow for more fine-grained control and optimization, which are described in the later Chapter 3.5.

While the most common Python implementation named CPython, also bases its memory management of data handled within Python scripts on a reference counting scheme, as we describe later in Chapter 2.2, this does not result in inherent compatibility with C++ code. Due to C++'s flexibility in memory management approaches, the guarantees CPython makes to Python developers, and the fact that the API connecting them utilizes the C programming language, resource management across this boundary is a non-trivial problem to solve. Smart pointers do help in this regard, as they make reasoning about and movement of ownership more clear and easier to enforce, but substantial planning and care has to be put in to end up with a reliable, safe and maintainable interface. C++ programs and libraries such as Tamashii require therefore substantial re-engineering in many cases to allow their runtime data to be exposed and exchanged across the language boundary in a sound and convenient manner.

2.3 Differentiable Programming

With the foundational overview on the motivation of scripting languages and the most important technical aspects of different resource management approaches out of the way, the mathematical and algorithmic theory that underlies many of Tamashii's use-cases can be highlighted, called differentiable programming. A mathematical function maps its input parameters to a resulting value, which can be plotted as a shape like a curve or surface when working with one or two dimensional values respectively. The function's slope at every point can also be described by a function called the derivative. The derivative relates changes in the function's inputs to the change of its outputs. In case of functions that take a vector of multiple inputs and yield a single scalar value, this is also called a gradient. This property can provide valuable information about a function's characteristics, as its gradient describes crucial qualities. For example, when a function's value reaches an extreme point like a minimum or maximum its slope value will be zero before it changes its direction away from the extreme point. With this elementary knowledge a strategy can be developed that finds zeros in the derivative to locate extreme points of the function. This operation is called continuous optimization which is useful when a maximization or minimization problem can be formulated as a function. While there are many different algorithms that are able to aid in this process based on genetic or heuristic approaches, gradient based optimization can outperform the others for smooth continuous functions.

As numeric optimization is an important discipline in computer science, getting an additional insight on a given problem space by using derivatives can be very helpful. If the gradients of a function to optimize are known or can be computed, multiple algorithms exist that can take advantage of this information to find extreme points. A simple approach is gradient descent which works on the intuitive idea of following the downwards slope until a minimum is reached [Rud17]. However, finding suitable starting points to prevent only stopping in local minima and selecting a step size to efficiently sample the function and its derivative are non-trivial problems.

The Adam algorithm is a prevalent optimizer based on function gradients. It is able to adapt its step size by using averaged internal moments and can handle large numbers of parameters, while being computationally and memory efficient [KB14]. These properties make it popular in the field of machine learning with neural network (NN), where researchers employ it during the training phase. In the so called back-propagation step, iterative optimization of the neural weights happens in respect to a loss function that describes the NN's current performance.

Therefore, the ability to optimize a function this way hinges on the ability to find its derivative function by a process called differentiation. However, in the field of computer science, most functions representing mappings of inputs to outputs are not denoted as algebraic (closed-form) expressions, which prevents traditional means of differentiation. Instead, algorithms are formulated as sequential imperative program statements that manipulate memory in most cases. Neural network topologies are also often described this way by expressing the flow of data through each layer and activation function as a vectorized function call.

There is more than one way to differentiate a sequential program. As traditional symbolic differentiation requires the function to be in closed form, one could either rewrite the program in closed form to begin with, or translate the program into closed form by removing control flow structures [BPRS17]. Symbolic differentiation which acts on the mathematical expression using classical differentiation rules by substituting each term with a premade derivation, can then be applied to get the functions derivative. If mathematically possible both steps can be performed automatically, however the resulting term might grow exponentially in size, a problematic property called "expression swell". Moreover, not every program can be rewritten as closed form, as some implicit expressions lack such a representation. In contrast approaches from the field of numeric differentiation do not require any transformation of the initial function. Instead, the finite difference approximation is used for example, by computing the function's value at two points in close proximity and assuming a linear equation. This premise is valid thanks to a function's local linearity on very small intervals, where it can be approximated by its tangent. Due to the limited precision of computing hardware, and the computational effort required for repeated function evaluations for higher order derivations, this approach is often unsuitable especially for highly dimensional problems that result in many inputs like NN back propagation. There are also other methods which alternatively rely on concepts like Fourier transformation [KBM+20] or Spline fitting [Die75].

Finally, automatic differentiation of program code can be utilized, which transforms the program's expressions to yield gradients by converting the program into a data flow graph. It is as accurate as symbolic differentiation, supports control flow, can be automated and does not create unreasonable amounts of overhead. Symbolic derivation is performed on an expression by expression basis in the program's data flow, which are combined based on the chain rule. Since the program does not need to be converted to a closed form, each derivation remains short, thanks to computation results being stored in memory and getting reused. But as many implementations rely on dynamically tracking the data flow, some execution performance is lost.

Symbolic derivation in general is a mechanized process than can be fully automated, as described before. The conversion of program code expressions constituting a mathematical function into their derivatives to find the function's derivative can therefore also be automated. While there were multiple such systems that worked on restrictive languages specific for this purpose, autograd was the first implementation for the Python programming language [Mac16]. The goal of autograd was to be able to formulate a loss function and have the derivative automatically generated, even when using branches, loops, recursion, higher-order functions, data container objects, and more. Further, higher-order derivatives, taking the gradient of a function containing gradients, are also important features. The implementation of autograd constructs a data flow graph of a function to derive, by recording its operations when executing. This way is simpler than other options like trying to work with the source code directly, an AST representation of the code or hooking the interpreter to detect function and operator calls. Instead, all parameters passed to a function to differentiate are boxed as nodes, which are Python objects that wrap the data values. All called functions are wrapped with inspection code, that unboxes the data by unwrapping it, performs the function call and again boxes the result. This final box contains the value, the identity of the function called and references to the parameter boxes, creating a graph. The graph depends on the execution of the function's program code, and therefore on the parameters it is called with. It only contains taken branches and loops in unrolled form, which means, that the graph needs to be dynamically generated on every function call.

While autograd is no longer actively developed [MDJ+16] its successor JAX is more powerful and integrates a JIT to execute functions and their derivatives on a graphics processing unit (GPU) [FJL18]. When the JIT is enabled traces of functions are cached based on the shape of matrices, data types and tuple members used as call parameters. Furthermore, the JIT limits control flow structures to be statically analyzable, enforcing loops with known iteration count or ones depending on the parameter properties used for caching like matrix shape. Functional alternatives for if-else branches exist, that allow tracing both branches [LJ23].

Building upon autograd and Lua Torch the developers of PyTorch implemented their own version of automatic differentiation [PGC+17]. Since the focus of the project is ML the main application of gradients is the optimization of NN which utilizes some form of stochastic gradient descent algorithms in many cases. Due to Python's low execution

speed and the internal overhead of operator calls, core functionality was built in native C++ code instead of Python itself. TensorFlow is another well-known ML library for Python facing similar requirements and challenges, and therefore also comes with its own flavor of automatic differentiation [AAB+15]. However, it uses a gradient tape to record computation similar to autograd, while PyTorch follows a different approach of only storing computation graph subsets per intermediate result node.

2.4 Differentiable Rendering

The process of generating an image from scene parameters such as positions of lamps, colors and the geometry of models displayed, is called "rendering" in computer graphics. There are many algorithms to compute a 2D image depiction of a 3D scene using rasterization and path tracing for scenes represented as triangle meshes, voxels, point clouds or signed distance functions. However, due to their complexity there are no straightforward ways to compute gradients for the scene parameters defining the resulting image. A major problem are discontinuities in the rendering functions that stem from effects like occluding objects with hard edges, which let the function's output value jump in sudden ways [KBM+20]. But thanks to advances in differentiable rendering there are now multiple recent projects and approaches in this area of research.

The ability to compute gradients for scene parameters allows the integration of such rendering algorithms into optimization driven problem spaces. For example, ML based on neural networks that processes image data for applications like machine vision greatly benefit from differentiable algorithms. When using NN to estimate parameters of a scene based on an input image, a differentiable renderer offers a way to create feedback during training. There the estimated parameters are fed into a differentiable renderer, allowing a comparison of the images and computation of a loss value [ID18]. Thanks to the differentiable nature of the renderer, the backpropagation process can then reach through the renderer back to the network weights to adjust them. In this setup the rendering is only part of the training phase of the network. But, it is also possible to forgo NN entirely and just use the renderer in combination with a gradient based optimizer to estimate scene parameters based solely on an input image. Here, the optimizer does not alter weights of a NN, but the scene parameters directly to lower the loss function in an iterative manner.

PyTorch3D is a Python library built on top of PyTorch providing a differentiable mesh and point cloud rasterizer with a focus on performance and modularity [RRN+20]. To achieve the latter, the rendering pipeline is broken up into multiple stages. A regular rasterization renderer has to select the face closest to the camera for each fragment before deciding its color. Due to the non-differentiability of this z-sorting arising from discontinuities when one face is occluded by another one, a different approach is necessary. Instead, multiple faces are considered for each fragment, colored and later blended together. The same problem appears when moving faces in x-y-screen space, suddenly revealing another face or the background behind it with a different color. This discrepancy

is solved by employing a blur radius around each face preventing sharp changes. For performance reasons only a small number of nearest faces is considered, which is possible as the influence during blending sharply declines with each additional face.

The rasterization process is implemented in CUDA and detects the faces required for each fragment, which are then passed to a fragment shader. The shader programs are implemented in regular PyTorch code and do the lighting, coloring and blending. They use PyTorch's automatic differentiation during the gradient computation.

Alternatively to rasterization, rendering can be facilitated by light or path tracing. The Mitsuba framework allows the creation of differentiable rendering applications using Python based on Monte Carlo tracing [JSR+22a]. It leverages the "Dr.Jit" just-in-time compiler for native machine code generation of rendering functions and their derivatives [JSRV22]. By tracing the whole rendering process optimization opportunities are found to create more efficient machine code to be either run on a central processing unit (CPU) or GPU. The interface to "Dr.Jit" is language agnostic, allowing functions written in Python and C++ to be traced and then compiled.

2.5 Neural Radiance Fields

Neural rendering is an emerging field of research in computer graphics that uses NN to generate images from scene encodings. While regular volumetric rendering produces images from volumes, neural radiance fields learn volumetric information from images. They are a way to encode a scene into a network by training it on images from known view directions. The resulting network is then able to synthesize novel views of the scene which it was not provided with during the training phase, by interpolating between the images [MST+21]. The network is fed a five dimensional viewing ray to a spatial position consisting of a 3D location as a xyz-vector and spherical coordinates as two viewing angles, to compute a radiance and volume density value. The hypothetical rendering function fully describes the scene by computing the radiance and volume density for every point in the scene. Sending such rays through the scene and summing the values along their way yields the rendered image. The neural approach utilizes the function approximation abilities of NN to recreate this function based on the limited number of training images. Performance of high frequency parts in color and geometry improves when a special input encoding is employed. To this end, the five dimensional representation is transformed to a positional encoding in a higher dimensional space by introducing high frequency components. This adjustment is accomplished by simply taking multiple sine and cosine of the original encoding with exponentially increasing frequencies yielding exponential harmonics called a Fourier embedding.

Several improvement on the original NeRF paper have been proposed and implemented. One example is BARF which relaxes the requirements on the provided images used as training data. It allows for the viewing directions and positions of the training images to be imprecise or even unknown [LMTL21]. The system is based on the observation, that simpler and smoother signals are easier to re-align after being shifted. Hence, images can

be blurred before performing re-alignment operations, for example. Fidelity can then be stepwise increased for more accurate aligning. This principle is utilized with NeRF by applying a dynamic low-pass filter on the positional encoding. First the influence of the higher order harmonics is heavily reduced and then slowly increased during the training phase. With this so-called coarse-to-fine learning approach, time is first spent locating the view positions, after that graphical details are started to be reproduced.

Other projects like NeRF-W [MBRS+21] further extend and enhance the capabilities of neural rendering by introducing support for varying lighting conditions and imperfections of physical photography. Research also focuses on increasing training and computational performance with new methods like grid encodings, where trainable weights are contextualized with the location. This way the size of the required NN and therefore its evaluation cost can be cut down by dynamically re-parameterizing it with the consequence of a larger memory footprint [MESK22]. Recent work combining NeRFs with such grid encodings in a two branch staged setup yields visually better results that circumvent over-smoothing and noisy artifacts, which are common to NeRF and grid encodings respectively [XXP+23].

There is rapid development in the field, showing encouraging results. Already there are projects emerging that try to streamline the use of neural renderings for end users. Nerfstudio is one of these applications, which is a Python based framework that enables the creation of neural scene representations from videos and photographs. In addition to the actual ML and neural rendering, it incorporates COLMAP for view direction recovery from the training images [SF16]. Novel views can then be rendered based on the trained scene.

2.6 Surrogate modeling

One thing common to many of the algorithms discussed is the high computational effort they require. Especially, iterative optimization can be costly, when a complex model and its derivative needs to be evaluated again and again for thousands of times. A common remedy is the utilization of surrogate models that replace the original costly ones [BHB+19]. They are statistical approximations trained on the real model that can be used to drive the optimization algorithm for a large portion of the iterations ultimately reducing computation time. Interpolating models are particularly useful as they become more accurate in a specific area when additional data points are added there [BM19]. This capability allows letting an optimizer steer into a subspace which is then enhanced with new data points, before repeating the process.

The field of application for surrogate modeling is as broad as the field of numerical analysis and simulation. Usage of models ranges from the simulation of groundwater flow [ACJP15], chemical process engineering [MS19], to the optimization of airfoil shape [LBM18]. The latter uses the gradient-enhanced kriging with partial least squares (GEKPLS) model [BM19] which is an advanced derivative of the Kriging algorithm, which has its origins in geo-statistics where it was developed for the mining industry.

There it was used to estimate mineral concentrations over a large area when only drilling a small number of bore holes. GEKPLS is a gradient enhanced version of Kriging where each data point is also associated with a local gradient. To prevent problems with computational complexity and local linearity when data points are closely grouped, the training data set is reduced to a smaller set of principal components using the partial least squares algorithm. These elements are linear combinations of the original parameters to represent the data set and are fed into the model instead.

GEKPLS and various other modeling algorithms are implemented in the SMT module in Python. The library focuses on gradient based surrogate models that are useful for many applications. In the beginning shortly after its release simulations for aerospace engineering were the first adopters of the project.

As described in this chapter, Tamashii enables researchers to implement computer graphics algorithms including differentiable ones. These programs involve heavy computation and therefore rely on C++ to be realized in a way that is even efficient enough for real-time applications. But due to the complexity associated with the development in C++, experimental test setups could be improved if a second more simple and dynamic programming language was available. The Python language fits these requirements, allows for many different programming paradigms, provides facilities to be integrated with native software and therefore is a common choice for projects that offer script bindings. Thanks to its simple but powerful and extensible design, capable standard library, its large thriving ecosystem of third-party libraries, and cross-platform compatibility it is the proper choice for Tamashii. Additionally, Python's prevalence in the field of machine learning and statistics also makes it a fitting choice that opens access to many modern projects in these areas. Nerfstudio and SMT are examples of software that we can leverage effectively with the help of Python.

CHAPTER 3

Background

The following chapter introduces the Tamashii graphics framework, its use-case, what technologies it employs and current limitations faced. Furthermore, we discuss the Python programming language and its interpreter CPython. We summarize and compare different approaches to extend the language through the interpreter and the needed technologies to do so. In this regard, we describe C++'s approach to handling owned and unowned values, lay out the differences in resource management between the CPython interpreter and native C++ code, and present ways how to solve them. Finally, we highlight Nanobind's method to decide when and how to hand over or share value ownership, and explain some facilities it provides the developer with for additional control.

3.1 Tamashii

Tamashii ("soul"/"spirit" in Japanese) is a scientific rendering framework currently under development by the department of computer graphics at TU Wien. Its main goal is to simplify the creation of research applications in the field of computer graphics by providing a fundamental structure in the form of libraries implementing resource loading, input handling, user interface creation, a complete rendering framework and graphics API abstraction. This way it serves as a toolbox of reusable components to researchers who want to experiment with new algorithms and approaches while allowing them to focus on their novel work instead of having to re-implement the necessary underlying technologies again and again, like scene loading, shader compilation and queuing render commands. Although Tamashii provides a command line interface (CLI) only mode, it sets itself apart by aiming for an interactive workflow through a responsive graphical user interface (GUI) that allows the inspection of scenes and shows how running algorithms interact with them in real time.

While the project is not yet publicly released, several researchers at TU Wien use it internally complementing their research. Recently, it was part of a project to develop a

performant temperature simulation for urban spaces [FHR+23]. In this work, researchers leverage Tamashii's capabilities in computer graphics to reproduce heat transfer via thermal radiation by introducing a method for hardware-accelerated photon tracing.

The project is written in C++ and is managed using CMake for compilation and native project creation, as portability to the different systems Windows, Linux and MacOS is crucial. For the API to the graphics hardware Vulkan was selected to handle the rendering and computation tasks. The system is structured around the idea of so-called implementations which are modular computational backends written as C++ classes, that implement virtual methods called by the Tamashii framework. An implementation is able to react to events and access the rendering infrastructure, letting users of the framework build custom applications. We provide an overview of the structure of the framework in Figure 5.2 that shows how implementations, Tamashii itself and dependencies are compiled and linked.

Tamashii's GUI uses the Dear ImGUI [Cor14] library to draw interactive user interface elements. Implementations are also able to extend the GUI with their own elements. Events are handled through a queue that has its contents dispatched every frame. Actions like mutating the active scene, or loading a new one, taking screenshots and triggering a shutdown are realized this way, having the GUI or an implementation queue an action. Global configuration values and constants are available through a variable sub-system that automatically maps each value to a CLI option according to its name, type and value range.

However, implementing a certain algorithm that performs graphical computations as a Tamashii implementation is only a single step in the research process. Experimenting with multiple configurations, constant values, and creating comparisons with other approaches is an integral activity. Additionally, results of experiments need to be logged and often visualized with the help of other software packages. Furthermore, when working within larger codebases writing test cases to detect regressions is common practice. But due to the compiled nature of C++ and its inherently more complex usage than other languages, integrating all of these further requirements into the C++ code can become cumbersome and straining on the user, the problem Tamashii tries to solve.

A major drawback of using C++ is the quickly accumulating time spent repeatedly compiling and linking when performing experiments with different configurations. While the former can be tackled with smaller compilation units by optimizing included headers, enabling pre-built headers, and limiting the use of templates, the latter is a long standing weakness with languages utilizing a compiler-linker setup caused by linker program's often limited use of multi-threading. In recent years there have been projects trying to tackle this issue with promising levels of success [Uc20]. While build times for Tamashii range from only a few seconds for single compilation units to a few minutes for a full rebuild with linking on capable modern hardware, the process is highly disruptive to creative work flows, especially when required in frequent succession.

Moreover, the integration of C++ libraries can be an involved manner due to the lack of

a prevalent package manager or any common policy for build system design. Quickly trying to employ external code in an experimental fashion is therefore out of the question in many cases. For these reasons, we extend Tamashii to feature a scripting interface that allows for rapid development and integration with external projects to support researchers in their endeavors. With such an interface users are empowered write testing code in a more flexible, dynamic and more expressive scripting language while still being able to implement computationally taxing sections in C++.

3.2 Python

As previously described, Python is a general purpose programming language that offers many high level features as part of the core language design such as dynamic typing, garbage collection and reflection. It relies on an interpreter to be executed, with CPython being the reference implementation. The default installation distributed comes with an extensive standard library providing functionality for filesystem access, high- and low-level networking like an email client and sockets, GUI support, compression, file format support, cryptography and much more. Also the package manager "pip" is installed by default, which allows accessing packages registered in the index [Fou23i]. Further, it is complimented by a thriving ecosystem of other projects that are written in Python itself or contribute bindings to libraries written in other languages.

The design of Python follows an object-oriented strategy, but also incorporates other paradigms making for example functional programming possible. Numbers, lists, tuples, dictionaries are fundamental part of the language. Custom types can be defined using classes that support multi-inheritance and reflection. In addition to defining methods, classes may overload operators to alter the way objects are treated. Thanks to garbage collection no manual memory management is necessary [Fou23f].

Python enjoys wide adoption in many fields of programming including for-profit development and academic work. As extensibility was one of Python's main goals and strengths from its very inception, Python has been a popular choice as a porting and bindings target [Fou23f]. The way Python code is structured into modules, its dynamic typing system, openness to different programming paradigms and the fact that CPython and therefore its bindings API are written in plain C make Python attractive as a bindings target. Additionally, Python can be written and executed interactively in a Read eval print loop (REPL) environment which can be valuable when experimenting. Having the codebase structured into reusable modules introduces clear boundaries in terms of code factoring and mentally in terms of functional responsibility. Hence adding a module not written in Python itself does not create any unnatural fault lines in the code. Further, the lack of explicit typing gives Python code a certain malleable quality that allows it to remain functional even when values types handed through interfaces change as long as they can be used interchangeably. Although Python's design is fundamentally object oriented, it is flexible enough to easily accommodate libraries written in a pure procedural or functional way, also without requiring to build a pseudo-object-oriented

wrapper class around the library's original interface as known from Java [aia23]. Last but not least, C often being the lowest common denominator in respect to language interoperability, particular in the world of systems languages like C++, Rust and Zig, enables creating Python bindings in most programming languages [Dev23] [con23]. This advantage is further supported by the resulting large ecosystem, which attracts even more projects. Therefore, it has become the prevalent language in certain areas, like data analysis and automation tasks. Especially the ML community has adopted Python as a de facto standard with PyTorch and TensorFlow as the two dominant libraries in use.

Due to Python's reliance on an interpreter, any libraries written in languages that are compiled to native machine code need to extend the interpreter itself before they can be used in Python. The interpreter is an application that executes the Python code and constitutes the runtime environment. CPython is the reference implementation which is written in C, loads the Python code from disk, parses it and compiles it to a more efficient bytecode representation before executing it. Hence, the interpreter acts as a the CPU of a virtual machine that executes the bytecode. Its instruction set architecture (ISA) is a stack machine that pushes and pops values from a virtual stack instead of using a finite number of registers like the majority of CPUs implemented in hardware. Moreover, the instructions executed are dynamically typed and require multiple real CPU instructions to be evaluated [Fou23b] [Fou23c]. This abstraction makes Python highly portable, removes the need for ahead of time compilation and keeps the interpreter reasonable compact in contrast to modern JavaScript engines featuring fully-fledged optimizing compiler toolchains for JIT-compilation. However, a major drawback is Python's slow execution speed, which can be problematic if computation intensive algorithms are written in pure Python without resorting to native libraries like NumPy for example, which offers fast vectorized math operations.

Python's performance is also hindered by its memory model which requires everything to be a heap allocated object leading to a lot of indirection induced reference hopping. It also results in a larger memory footprint due to space needed for the reference address, the reference count, type information and additional meta information for the GC in case of complex objects that can create reference cycles. The dynamically typed nature of Python further strains the interpreter to efficiently execute the bytecode as the types of variables and temporaries have to be repeatedly checked and converted where necessary [GO20] [SRS+23]. Finally, one of the most significant design choices holding back CPython is the global interpreter lock, that mostly rules out any kind of real multi-threading from within Python. The GIL helps to simplify the implementation of memory management, especially atomic reference counting, and interoperability with other native libraries [PWc22]. While there has been ongoing work to replace the GIL, currently nothing manages to fully replace it that also fully offers the same guarantees that a lot of existing code relies on.

Apart from performance there are more potential obstacles to consider when planning to create Python bindings for an application or library. When extending an existing project with a subsystem, added code volume and complexity are to be expected. However, the

CPython interpreter is a very complex and large piece of code, that requires a lot of internal understanding of the Python language and CPython's specific implementation of the language to be able to use its C API. Furthermore, even though C is well suited as a language for creating such low level interfaces and has proven its worth in this regard over time, its use in codebases that are built on other technologies like C++ or Rust can introduce a second unnatural language crossover. This mismatch forces developers to deal with complexities in the build process and require additional glue code. Also due to CPython's automatic memory management, a lot of thought has to be put into handling ownership of memory handed over or shared with the interpreter. Moreover, functions written in a statically typed language like C++ cannot be simply exposed through CPython, which requires them to be as dynamically typed as any other Python function. This means that a considerable amount of obligatory C boilerplate code is needed to do type checking and reference count updating [Fou23g]. Due to this, a link to CPython cannot be simply added on top of an existing library in many cases, and a more thorough refactoring or even re-architecting of the implementation is necessary. Finally, splitting an application into two separate languages can complicate debugging across the boundary.

3.3 Interfacing with and extending the CPython interpreter

Python code is structured into modules, where each Python source code file constitutes its own module. A module can import the contents from other modules by their name and by declaring the names of the objects to import. There is no explicit way of exporting symbols from a Python module, instead every name not prepended with an underscore is made available. Although, the global `__all__` property of the module can also be set to change this default behavior. Modules are located by the interpreter by searching for a file with the same name in the current directory or one on the `sys.path` variable [Fou23j]. The file might contain Python source code or compiled machine code if it is a native Python module. In case of the former the code is loaded and executed before returning to the importing module, and in case of the latter it is loaded and hooked into the interpreter.

For the interpreter to be able to load a library containing machine code dynamically during runtime OS level functionality is required. An application's source code written in a compiled language like C, C++ or Rust is first translated into object files that contain optimized machine code, but miss address values required to reference symbols in one of the other object files like global variables or functions. These symbols are patched as part of the linking step, when the object files are assembled into a single large executable. However, in most cases the application's executable does not contain all machine code necessary to be executed in a standalone fashion. For example, the C runtime library is usually provided by the system and when the OS is instructed to run an application it dynamically links it to any libraries specified. This setup has several benefits to the

developer and user: By sharing library code across multiple applications memory and disk space can be saved, updates to system critical code in important libraries like the C runtime library can benefit all applications at once, and code shims for compatibility, emulation or inspection can be inserted. Dynamic linking mechanisms are not exclusive to OS libraries, as user applications can also be structured into multiple shared libraries during compilation that are linked together by the OS on startup. Furthermore, the dynamic linker cannot only be invoked by the OS when preparing an executable for execution, but it can also be directly utilized by the user program, which is called runtime linking. By calling certain OS specific functions the user program can let the OS run the dynamic linker which provides access to the contents of a library [Fre23a] [cWS+21]. This is the mechanism CPython uses when it detects that an imported item is a native module instead of a textual source file.

After CPython has loaded the native module's dynamic library via the OS' dynamic linker, it looks for an exported function called `PyInit_name`, where "name" is the module's name. This function creates the module object by providing a C structure with information describing the module. It includes an array of all functions exported by the module, that should be made available in Python. All exported functions have the same signature that takes a self-arguments, a parameter tuple and returns a Python object. The data consumed and returned is boxed within opaque Python containers and objects, which means some helper methods need to be called to determine their types and extract their values. Unpacking data and building objects is facilitated by functions like `PyArg_ParseTuple` that are called with a format string similar to `scanf` to describe the expected data format. When handling any values also held by the interpreter their reference count has to be manually updated to prevent memory leaks and use-after-free bugs [Fou23e].

Hence, creating Python bindings comes with a few challenges. A lot of boilerplate code is required to map dynamic types to statically typed function interfaces and data structures. Furthermore, manually and automatically managed memory has to be sent and received across the language boundary, while having to explicitly declare every reference count change. This task again requires many lines of code that need to be written, but also a lot of care has to be put into the possible flows of data, to ensure no memory bugs are introduced. Also errors are handled with exceptions in Python, a concept completely unavailable in C. Again, code has to be added that translates errors received from either side into their appropriate counterpart.

3.4 Pybind11 & Nanobind

As nearly all projects that want to add bindings face similar challenges, solutions to these common problems have been created in the form of third-party libraries. These packages can provide several benefits to the developer, for example, by wrapping the C based CPython API with a different higher-level language better fitting the binding target. They can also present ways to create more convenient and safer abstractions on top the

original API by utilizing said language's features. Depending on the implementation's factoring, this simultaneously helps to reduce the required boilerplate code for each function, type or variable binding, allowing for more concise, expressive and simpler code. Another advantageous functionality is a simplified build process, when the library provides integrations to build systems like CMake.

Two such libraries are Pybind11 and Nanobind, which are written in C++ and provide an object oriented API to CPython fully concealing the original C one. The development of both was initiated by Wenzel Jakob, who is a computer graphics researcher at EPFL[1], for use in the Mitsuba rendering framework. After being unhappy with technical decisions made when designing Pybind11 in 2015, he started development on Nanobind in 2022. Still, Pybind11 is currently well established and used in many large projects for ML and rendering. However, due to many additional features and remedies for edge cases contributed by the large number of users, Pybind11 has grown into a large project itself. Furthermore, its design as a header-only library, while making the build process for users rather simple, comes with many downsides with the increasing size of the codebase. Most and foremost, build times and the size of the resulting binaries have suffered due to this property of Pybind11, which needs to rely on link time optimizations to bring down binary sizes, further increasing build times [Wc23d].

Nanobind fixes these issues by focusing on a smaller set of features and instead heavily on performance, dropping support for much legacy code, switching to C++17 to simplify many parts, reducing support to only a small set of compilers and toolchains, and most importantly splitting the codebase into headers and regular source files. It therefore now requires compilation of an additional library file which users have to link against. Better performance is enabled by utilizing more modern CPython APIs like the recently added vector call protocol for functions, enforcing better data locality by storing meta information in the same memory as object fields, and replacing the STL `unordered_map` with a robin hood hashing map.

But although Nanobind supersedes Pybind11, their general usage and interfaces are very similar. Pybind11 itself heavily takes from the Boost.Python library which is part of the larger C++ Boost ecosystem [Wc23c]. They model the CPython API in an object oriented way and use C++ functionalities like RAII, operator overloading and user-defined literals to abstract error prone and lengthy interactions with CPython away. Memory management, for example, is handled through the STL smart pointer types, which clearly communicate intended ownership and also conveniently use reference counting. The libraries make heavy use of the concept of metaprogramming to automatically generate type specific code. Code required to map dynamic Python types to concrete static ones, loading and returning values for function calls depending on a function's signature and de-virtualizing method calls by class is all written as C++ templates and specialized on a per bindings basis. This design choice leaves Nanobind and Pybind11 with a very declarative, readable, modern and natural feeling interface that relieves the user from

[1] École polytechnique fédérale de Lausanne/Swiss Federal Institute of Technology in Lausanne

knowing detailed information about the internal workings of CPython in the majority of cases.

When taking a look at existing software, there are libraries that use bindings libraries such as the ones previously described, others create their own abstractions on top of CPython, while others again just use the plain C API. Numpy, which is a module part of Pythons default installation, provides vectorized math algorithms and data structures like multi-dimensional matrices and is written in C and Fortran. It does not rely on any bindings library and calls functions of the CPython interpreter directly [HOvcm23]. TensorFlow and PyTorch on the other hand define their bindings using Pybind11. However, PyTorch also uses the C API in some cases [CCD+23]. Mitsuba 3 also uses Pybind11, but its main developer Wenzel Jakob stated that it will transition to Nanobind in the future [JSR+22b] [Spe22].

In summary, there are several good reasons for CPython's functionalities being exposed as an C API. But even though the codebase of Tamashii, like many other libraries, is written in a language that is compatible with this API on a technical level without the need for alterations, there are many downsides to using it directly. While there are some that limit themselves to communicating to CPython without the help of additional libraries, these can offer many improvements to the development experience, code quality and maintainability of the project. They enable the usage of high-level, more complex and expressive language concepts not found in C, reduce the length of bindings code, represent shared resource management with simpler and safer abstractions, and better mentally decouple the bindings code from technical details of CPython's implementation. With this in mind, we decided to follow this path for Tamashii, similar to other modern and established libraries that offer hardware accelerated computation within Python, like PyTorch or Mitsuba 3.

3.5 C++ data ownership and transferal

In the previous chapter we highlighted the usefulness of a bindings library to support the interfacing of a C++ codebase with the CPython interpreter. Especially the management of resources can be clarified and simplified with the functionalities provided by Pybind11 and Nanobind. These libraries heavily rely on the ownership model represented by STL smart pointers to automatically infer how to handle values passed and received via the CPython API. Thus, to fully leverage these abilities a codebase has to embrace modern C++ memory management to be able to properly integrate bindings. In the following section we explain the three main types of smart pointers for handling ownership, describe how values can be passed according to the C++ core guidelines for sound interface design, and finally give details on Nanobind's way of data exchange with CPython.

The C++ programming language does not provide facilities to automatically manage dynamically allocated memory, like Python or Java which both feature a runtime environment with a GC. However, it is structured around the concept of statically determined lifetimes of values and variables. The STL builds on top of this to implement dynamic

memory management based on the notion of ownership of data. In the context of object oriented design, where data and related functionality are represented together as objects, this means that variables and objects, or respectively their member fields, may own another object. As explained before in Chapter 3.5, special pointer types are used to convey this concept of ownership and implement memory management. There are owning pointers such as `unique_ptr` and `shared_ptr` that keep strong references to heap allocated objects and free them whenever there are no more owners are left. These types were introduced to replace the `auto_ptr` class in the STL that was designed before the existence of C++11 move-semantics [Hel04]. It therefore relies on its copy-constructor to transfer ownership, which results in an unintuitive behavior and makes its use error prone.

The simplest and most performant modern smart pointer type is the `unique_pointer` which ensures, that only one owner exists at every point in time, by disabling copying and only permitting move-semantics instead. A `unique_pointer`'s destructor, hence can just deallocate any memory currently held by the instance. On the other hand, for situations where shared access via multiple owners is needed the `shared_ptr` exists, which utilizes reference counting. It is supplemented by the `weak_ptr` type that is used to circumvent reference cycle induced memory leaks. Whenever an owning `shared_ptr` runs out of scope the counter associated with the referenced object is atomically decremented. When the counters value reaches zero, no owners are left, and the memory can be freed safely.

But there are many cases where ownership should not be transferred or shared when handing over the object to an interface. For this reason, there are also other ways to move and share values. While objects could be simply duplicated on the heap to prevent changes to the current ownership state, this is undesirable for memory consumption and performance reasons, and struggles with the problem of deep-copying for complex or recursive data structures. Therefore C++ offers C-style pointers and C++-style references to reference objects via their memory address. These variants however, do not clearly communicate any intended ownership of the referenced memory, as they do not implement any memory management mechanisms. This fact enables them to reference any kind of memory including globally declared or stack allocated variables. Hence, no assumptions can be made about the ownership and allocation strategy of the memory referred to by them. With the exception that C++-style references are enforced to always refer to a value, and therefore do not possess an empty state similar to null-pointers.

Interfaces taking references and pointers should thus never assume ownership of an object provided to them [SSvcm23c] [SSvcm23d]. In such cases they are supposed to require the caller to either provide a strong owning smart pointer per value or reference. The latter can be beneficial for reference counting `shared_ptr` if a transfer of ownership is not guaranteed to happen, to prevent unnecessarily updating the reference count, which needs to happen atomically and is therefore non-trivial on multi-core machines. In general, smart pointers for resource management are preferred over the use of manual memory management [SSvcm23e]. Although due to large amount of legacy code, there

are still many cases where smart pointers have to be relaxed to plain C-style ones.

Pybind11 and Nanobind follow these recommendations closely and translate between the reference count held by CPython and smart pointers received from and returned to the library. Users can declare intended ownership sharing or transfer by returning the appropriate smart pointer type. However, Tamashii's codebase on the contrary did not utilize STL smart pointers and employed more outdated practices, which necessitated wide ranging changes that are described later in Chapter 5.4.

In addition to inferring ownership transferals based on types alone, Nanobind provides facilities to make data transfer more flexible, expressive and safer by providing an interface to specify how ownership should be handled for return values of functions [Wc23e]. The bindings definition of a function or method may state a return value policy as an extra field, which can be one of a selection of supported values, as can be seen in Table 3.1. If none is set, the automatic policy is used which tries to find a safe default based on the type of the returned value. Moreover, when a function returns an owning pointer type like `unique_ptr` or `shared_ptr` their referenced objects are automatically adopted. However, when taking a `unique_ptr` as a function parameter, some limitations apply as Python does not have anything similar to move semantics and uses its own memory allocator. Furthermore, custom deleters are also not supported [Wc23b].

For large data matrices Nanobind comes with support for `ndarrays` which are based on the CPython buffer and DLpack protocols for efficient data exchange. They act like handles that view into an underlying buffer based on templated constraints that define the data type, shape and memory layout of the matrix. A `ndarray` keeps track of its buffer's life time through reference counting, which makes copying it cheap and safe. When creating such an `ndarray` a copy of the data is created by default when the `automatic` return value policy is active. This presumption makes matrices that are based on local or static arrays safe to return from a function. However, for large heap allocated buffers this is undesirable for performance reasons. Hence, the constructor of an `ndarray` can be supplied with an owner capsule object, which wraps the pointer to the memory block and a callback function. When the matrix and consequently its associated memory buffer expires the responsibility of freeing the memory is delegated to the function.

Policy	Effect
automatic	Depending on the type of the return value, one of the policies below is selected. When a pointer is returned, `take_ownership` is used. For data returned by value or right value (rvalue) reference, `move` is used. Finally, for left value (lvalue) references, the `copy` policy is selected. The `automatic` policy is the default.
take_ownership	Based on this policy ownership of a provided pointer is taken and transferred to CPython to manage the memories life time. Only a lightweight Python wrapper object needs to be created.
copy	A new instance of the C++ object is copy-constructed inside a Python object, which is handed over to CPython to be managed.
move	Similar to the `copy` policy, a new object is created which is move-constructed in this case. C++ might therefore be left with the remaining empty object.
reference	Similar to the `take_ownership` policy, a lightweight wrapper object is created, which however does not assume ownership of the data. When CPython deletes the wrapper, nothing happens.

Table 3.1: A selection of the available Nanobind return value policies with their effect on the returned data

CHAPTER 4

Method

The previous chapter laid out the reasons, why it is common to use a bindings library for interfacing with the CPython interpreter and making functionality available in Python. As described, there is a multitude of benefits compared to solely relying on the CPython API. Therefore, we also choose to employ Nanobind to develop the bindings library for Tamashii, after selecting it from the other available options. The following chapter discusses the selection process and reasoning behind it. Furthermore, we list the requirements for the bindings and explain the according fundamental design decisions. First, there are user and project related needs, that are rooted in the way the codebase is currently used and structured. Secondly, we have to consider the required capabilities of the bindings code itself and future even tighter integration with Tamashii. Based on these aspects, we form the design that guides the realization of the Python interface and define the methods needed to accomplish this goal.

4.1 Selection of a bindings library

There are two foundational approaches to structure a scripting interface that relies on an interpreter executing the script language. Either the interpreter and language runtime environment become part of the application, or alternatively the application is packaged into a module loaded by the interpreter. Both scenarios are well established ways to integrate scripting with large codebases and come each with benefits and drawbacks.

Embedding the interpreter keeps the entry point of the program as a part of the original application. The interpreter is only a sub-system that is used to occasionally execute script code in addition to the application's default control flow. For example, script code can be executed to handle predetermined events or in repeated intervals to change or extend the behavior of the application. This method can be compared to plugins that can be installed for an application, in fact plugins are often created using scripting languages for their ease of use, platform independence and added security provided

through sandboxing. For example, most modern web browsers allow installation of addons which are written in JavaScript. But also many other well-known programs like Autodesk AutoCAD and the Emacs text editor include support for extensions. Both implemented a customized version of LISP as a way to extend them [Inc23] [Fre23b]. Video game engines also often come with a built in scripting framework for a simpler and more high-level access to the engine's capabilities. The Unity engine is tightly integrated with its C# interface and the Godot engine allows either scripting in C# or its custom Godot-Script language [Tec23] [LMtGc23]. Many applications and especially games also rely on Lua as their scripting framework [IdFC23] [Wik22].

In contrast to that, packaging the application as a module to be loaded into the scripting language's environment, follows a different philosophy. Here the interpreter establishes the start of the program by executing a script that loads modules as dependencies. The application is now only one of multiple libraries that gets instructed to perform actions by the script. While it can do operations in the background, the roles are now reversed, and the control flow of the script is the focus of the developer in this setting. The same way an application becomes a part of a larger process when called through its CLI in a shell script which accomplishes its goal chaining operations of different programs, packaging the application as a language module allows it to be integrated into a scripted program. But instead of using a text based CLI a more powerful and efficient interface directly through the interpreter is used. The whole concept of extending the CPython interpreter follows this principle and modules like Numpy are good examples for this approach.

CPython however, supports both modes of operation [Fou23d]. Pybind11 also provides a convenient interface to start an interpreter instance and interact with it. Python source code is provided as a text string and can be executed by calling the `pybind11::exec()` function. Furthermore, modules can be loaded, their attributes accessed and functions executed from within the C++ application [Wc23a]. Nanobind on the other hand does not support this use-case. Table 4.1 illustrates the supported operation modes of both libraries as a matrix.

	Packaging	Embedding
Pybind11	Y	Y
Nanobind	Y	N

Table 4.1: Feature matrix comparing Pybind11 and Nanobind

To make an informed decision, when selecting the type of binding to create and library to use, we made prototypes of all three possible combinations. Embedding the interpreter is only supported by Pybind11 but comes with a few advantages. The existing code of Tamashii would need less restructuring when Python code was only intended to react to events emitted by the framework. Moreover, this plugin model meshes well with the GUI first approach of Tamashii. Furthermore, this form of Python integration mimics the way Blender allows user scripting likewise using Python [Fou23a]. However, it also

deviates from most other scientific software available in Python which usually exists as installable modules like Mitsuba3 for example. Also this method is less flexible, as the script needs to follow the control flow of Tamashii more strictly. Additionally, scripts would be centered around Tamashii itself, which might not be desirable in cases, where Tamashii is only a part in greater research project.

Packaging the C++ code as a native module loadable by the interpreter is a capability of both Pybind11 and Nanobind, which meant the creation of two separate prototypes. Packaging Tamashii as a Python module requires more design work and more drastic changes in the codebase, as the framework is structured as a standalone desktop GUI application. Since some areas would ideally need a full rewrite to properly accommodate Python bindings, some workarounds are necessary. Additionally, the Python module introduces another build target for each Tamashii implementation, as the module is a shared library which needs to be compiled and linked differently than a standalone application. Hence, precautions have to be taken to ensure that the project structure and build scripts of existing Tamashii implementations are still compatible or only require minimal changes. However, this also simplifies the separation of the bindings code from the remaining application which is only necessary for the Python module build target. Plus, there is already existing test code that interfaces with Tamashii's CLI ultimately using it rather similarly to the way a Python module would be used. Furthermore, future public distribution of Tamashii could be simply done as an installable Python module via the Python package index, which can be easily accessed through "pip".

All three prototypes emulate the project structure of Tamashii's codebase which is split into multiple components which get compiled into separate statically linked libraries. The build system is a simplified version of the setup used by Tamashii and generates four component libraries, a standalone application and python bindings for each prototype. One of the components implements a simple GUI based on OpenGL and ImGui [Cor14], while the others only provide simple placeholder functions. Each prototype implements an interface to call code from the components, in particular they start and run the GUI. The three prototypes show that there are no observable downsides or obstacles when implementing one approach over the other. Thanks to extensive documentation and available examples, both can be realized in a reasonable amount of time. Due to the relatedness of Nanobind and Pybind11, their interfaces are also nearly identical. One of Nanobind's main goals is the reduction of build times, which we could not verify on the basis of the small test case created. In Listing 7.3 and Listing 7.5 we include the Python code of one packaging and one embedding prototype for comparison.

We opt for packaging Tamashii as a Python module to achieve a maximum of flexibility, easier integration with other software written in Python, follow established scientific software, and allow for a cleaner separation of bindings code from the rest of Tamashii. For the library helping with the creation of bindings we determine Nanobind, due to its smaller footprint, expected better compilation performance, better runtime performance and ability to use more modern C++ features. Furthermore, as Pybind11's creator has moved on to Nanobind, it is presumed that future development attention will shift away

from Pybind11, making Nanobind the way forward.

4.2 Design of the bindings creation workflow

Tamashii's codebase is structured into multiple C++ libraries that get linked together. The main utilities are part of the core library, that contains many of the important features such as the GUI and scene graph, but also the fundamental infrastructure like the event loop, configuration variables and CLI. We separate the code needed for bindings defined with Nanobind out from the core library of Tamashii. Therefore, we name this new library "core-bindings" and link it to "core" and Nanobind. In the following sections we lay out the requirements in three categories.

Compatibility Tamashii is a framework used by several researchers to implement and test algorithms in the field of computer graphics. This circumstance however results in some specific requirements to any Python bindings that should be added to Tamashii. Most and foremost, we seek to avoid breaking compatibility with the code of existing Tamashii implementations when introducing the scripting bindings. Since some changes to the core of Tamashii are necessary, some breaks are inevitable, but we aim to keep them to a minimum. These changes are two-fold, on the one hand changes to the C++ source code have to be made, on the other hand the set of build scripts has to be adapted to support additional build targets.

A major requirement to the design of the bindings is to make their use optional to Tamashii implementations. Python bindings are intended to be an additional feature in the toolbox of researchers instead of a burdensome necessity even to projects that do not need them. Therefore, we want Python to only be a build dependency, when bindings are created. Hence, the link phase also needs to further exclude any unnecessary code when bindings are disabled. Additionally, we target build scripts to remain functional after the introduction of the bindings code to the core codebase of Tamashii and when choosing to use bindings, the required changes to the build scripts are to be kept minimal.

Convenience & Safety Moreover, creating bindings should be convenient for the developer of a Tamashii implementation. Thus we plan to provide premade bindings for core features of Tamashii that can be instantiated with little code. Furthermore, any code necessary for initialization of the system is ideally accessible through a single function call. There is also a template necessary that shows how to structure the source code and write a build script. For consistency, installation of dependencies needs to be automated and follow the rest of Tamashii. The code facing developers adding bindings to their Tamashii implementation requires an expressive and safe design. The same goes for the predefined bindings, which we intend to clearly communicate errors and prevent undefined behavior, when they are used in Python code.

Extensibility The design has to additionally focus on extensibility. This prospect means, that implementation developers shall be easily able to create bindings for their own code and add them to the predefined ones. Furthermore, we determine that it is required to possibly modify and add to the predefined bindings, for customizing the Python interface of core components. As Tamashii is in active development, alterations to the predefined bindings has to be easy and extending the bindings to expose more of Tamashii's infrastructure needs to be straightforward.

For these reasons the "core bindings" library is an additional component, that is only optionally linked to. This library contains bindings to core components of Tamashii, provides required initialization code and offers ways to alter the bindings after their instantiation.

4.3 Design of the core bindings interface

Before specifying any concrete parts of the interface, we need to determine and lay out its relevant features first. These requisites, in combination with the requirements described previously guide the design of the core bindings. The scope of our work is limited to a minimal viable product (MVP), that is limited to cover the most important parts of Tamashii. Our goal is to enable the realization of experiments and show the viability of such an interface to improve the experience using Tamashii. In the future, it is possible to add more functionality to the MVP and extend it to more of Tamashii's components, as Tamashii itself is under continued development.

Since the initial use case of the Python interface is to enable writing of test and experimentation code more quickly and simpler, its development focused on the needs of the existing experiments. The extent of the MVP is therefore determined by the demands of this code, such that it can then be ported to Python. The tests are implemented in C++ as part of the application itself with the help of shell scripts and some Python code to process textual output created by writing to files or the console output. They load scenes, access parts of the scene graph, interact with the camera, perform rendering operations and take screenshots of the resulting image. These items are the general features that the bindings need to provide to accommodate the old tests.

Global state A Python script depending on Tamashii needs to be able to import it as a module, to access the bound functions and variables. Also, the Tamashii framework itself needs to be instantiated and the data stored in its memory constitutes a compound state, that has to be represented in a Python script. This distinction between the module and the state of Tamashii ought to be clearly visible, as the former denotes the available functionality in the form of code, and the latter describes the resources and state currently in use.

Scene graph The entities that make up the actively rendered scene are stored in a nested data structure called the scene graph. It contains for example the lights and

models. These entities need to be accessible from within Python and their properties have to be mutable. Such parameters are the position, facing direction and intensity of lights in the scene. The configuration of scenes can be serialized and stored to disk in various formats. Tamashii supports the `gltf` format and is able to hot load scenes during runtime, which is supposed to be also possible through the bindings.

Screenshots To inspect, compare and document results, saving the rendering output as an image is crucial. For this reason, Tamashii has a feature to create screenshots with and without showing the GUI. However, to include output images in plots or process them within Python, returning image data directly through the Python interface is superior compared to writing it as a file to disk in terms of ease-of-use, efficiency and portability.

Variables As described before, global configuration parameters of Tamashii are stored in globally accessible variables. Each has a type, value and optional modifier flags. These flags define whether its value is mutable, should be serialized to a configuration file or whether it should be made available as a CLI parameter. Python code needs to be able to read and write to these variables to have proper control over Tamashii.

Initialization of Tamashii is a non-trivial process, which involves access to the filesystem to read resources like configuration files and shader code, parsing of the configuration, compilation of the shader programs, initialization of graphics hardware and communication with the window manager. Furthermore, a considerable amount of resources are allocated during this time like access to the GPU, file handles and memory. This task has to happen before Tamashii can be used by any Python functions, and there are several ways to perform it. One option is to handle the initialization implicitly and therefore hiding it from the user. This route might activate during loading of the Python module when CPython invokes the instantiation of the module in C++. Alternatively, every function call and variable lookup could feature a check whether the system is ready, and else perform initialization. On the other hand the user could be required to setup Tamashii explicitly themselves, by calling an init-function or constructing an object. Such an approach has multiple benefits over the implicit one. First and foremost, performing non-trivial tasks in an obtuse and unexpected manner is considered a side-effect, which is generally discouraged, as operations are supposed to only do what they appear to do [vRWC01] [Mar09]. Furthermore, in this specific case, implicitly starting Tamashii results in code that is less expressive and robust, harder to maintain and forgoes common Python design patterns.

Booting Tamashii on importing can be undesirable for example, when importing it as a third-party dependency from a different script, even though it is not actually required. A situation like this can be illustrated by a module A which needs functionality from a module B unrelated to Tamashii, but module B also offers functionality using Tamashii and therefore imports it. Tamashii will then be started in all cases. Furthermore, hooking into the interface to lazily initialize Tamashii has its own problems, in addition to violating the generally advised separation of concerns [Mar09]. Firstly, when the first interface

access is responsible for starting a module, in a script that features more complex control flow, it becomes difficult to reason about when resources are allocated. Moving unrelated code while refactoring a script has then the potential to introduce unexpected behavior like race conditions. A straightforward alleviation is to perform a no-operation once by calling the interface at a known point in time, which is equivalent to an init-function. Secondly, if hooking is used, it needs to be consistently implemented for each and every interaction that can be executed with the interface, as otherwise undefined behavior on the C++ side can be expected. If for example variables where globally accessible, could be accessed immediately after loading the module and one was not hooked but triggered a C++ callback into the Tamashii framework, there are no C++ language features that would guarantee anything better than a segmentation fault at some point.

Hence, explicit initialization is the clearer, safer, widely encouraged and in the context of Python libraries the more intuitive method. As Python and C++ are both object oriented languages, it is natural to represent Tamashii's state as an object, that is initialized through a call to its constructor method. Thereby the similar question of cleanup operations is also solved, as the object's lifetime dictates when deinitialization happens. This approach also solves the problem of initialization order, as a piece of code that relies on a valid Tamashii instance being present, can simply request to have the corresponding object handed over as a call parameter, also causing the dependence to be self-documenting. In case Tamashii is only one of several resource intensive operations run after each other, it might be important, that for example memory is freed at a deterministic point during execution. This method also simplifies a possible way into the future where multiple concurrent Tamashii instances would be supported, which enables the ability to to parallelize isolated operations on different dedicated GPUs.

The same arguments apply to the other parts of the Python interface to Tamashii. The currently loaded scene, the lights, camera and variables all are represented as objects. To modify an object in Python code, publicly accessible properties are preferred over getter and setter methods. These functions are hooked with callbacks in case more complicated mutation logic needs to be implemented [vRWC01]. The interface means to mimic this behavior, only using getter or setter methods to imply that computation of the result or the mutation of the object incurs considerable cost. For example, moving the camera position is cheaper than changing the currently loaded scene. Thus, the former is a simple read/write property, while the latter is an explicit method called `openScene()`.

Using objects to represent the live state of Tamashii, like a light that is part of the actively rendered scene, however comes with potential downsides. Data dependencies that can result in dangling references have to be considered. When the active scene is replaced by loading a different one, the previously mentioned light object becomes invalid. Precautions have to be taken to prevent possible undefined or unexpected behavior. Instead, clear errors have to be raised, which is facilitated by runtime checks that make sure referenced objects are still valid before they are used, or else an exception is thrown.

CHAPTER 5

Implementation

Based on the design laid out in the previous chapter we created an implementation that extends Tamashii with an interface to the Python language. To this end, we modified the build system and integrated it with Nanobind. Moreover, we created a new version of Tamashii's variable system that improves on the old one in several ways. This work lead to the addition of the core module, that contains and structures many premade bindings for the main components of the Tamashii framework. Finally, we made custom bindings on top of the core module for the Interactive Adjoint Light Tracer (IALT) Tamashii implementation to demonstrate the process and how it can be emulated by other implementation developers. These modifications and extensions enable the realization of three of the use cases envisioned for the MVP as shown later.

The following chapter shows how we realized the described concepts in code and highlights selected details. The alterations we made to the build system are explained and the new variable system is discussed, elaborating why its creation was necessary and how it improves on the old one. Furthermore, details of the core bindings module and their motivation are reviewed. Then, the data exchange with CPython via Nanobind is explained and we state the changes to Tamashii's memory management required to properly interface with Nanobind. Finally, we summarize the decisions and techniques that went into adding custom bindings for IALT.

5.1 Build system

The process of compiling and linking the C++ source code of Tamashii is automated using CMake, which is a script based build system. Each directory contains a `CMakeLists.txt` file that consist of a shell-like script language that creates and describes build targets. Source files can be found using glob-operations and whole libraries can be loaded as packages. However, CMake does not drive compilation itself, instead it generates build files, like simple make files or platform specific IDE project structures [Ic23]. This way

a codebase using CMake can be edited and built using Visual Studio in a Windows environment or using Xcode in a MacOS one.

Tamashii's repository has a base directory for all source code files. There, sub-directories for Tamashii's different components exists: "core", "cuda_helper", "renderer_vk" and "rvk". We place our newly created "core_bindings" library also in the same location. The resulting directory hierarchy is displayed in Figure 5.1. Finally, the implementations are also situated in an appropriately named subdirectory. During build time, each component is compiled to a static library and each implementation builds its executable, which can depend on one or more components.

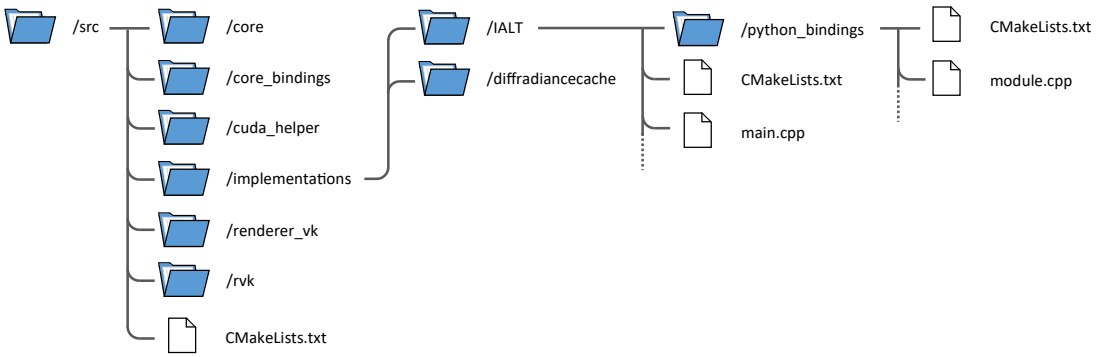


Figure 5.1: Each Tamashii implementation has its own subdirectory. The code specific to the Python bindings is located in another subdirectory.

Nanobind also supports CMake and provides several functions that can be called within the script language to configure the way Nanobind and the bindings library are built. Nanobind is loaded as a CMake package and its source files can either be compiled to a static or dynamic library. Then one or more targets can be defined by adding a Nanobind module in the script. The target creates a regular shared library with a specific name under the hood that automatically gets linked to CPython and the Nanobind library. As it is a regular CMake target, the script can use it as such and specify any other library dependencies, target properties or install instructions. This target compiles the implementation and bindings code as a Python module that statically links to all of Tamashii's components, as illustrated by Figure 5.2.

To allow Tamashii implementations to define their own bindings using Nanobind, they need to separate the code only needed for the use as a standalone application from the remaining logic. In most cases this only refers to the application entry point implemented in the `main` function. Furthermore, the code that creates the Python module, also instantiates the Tamashii core module before defining any custom bindings. For this reason, a template was created, that can be used to structure the code of the implementation so that the relevant parts are included for each kind of target. An implementation needs to create a subdirectory called `python_bindings` for all bindings code and move the application entry point into a `main.cpp` file in its base directory, as shown in Figure 5.1.

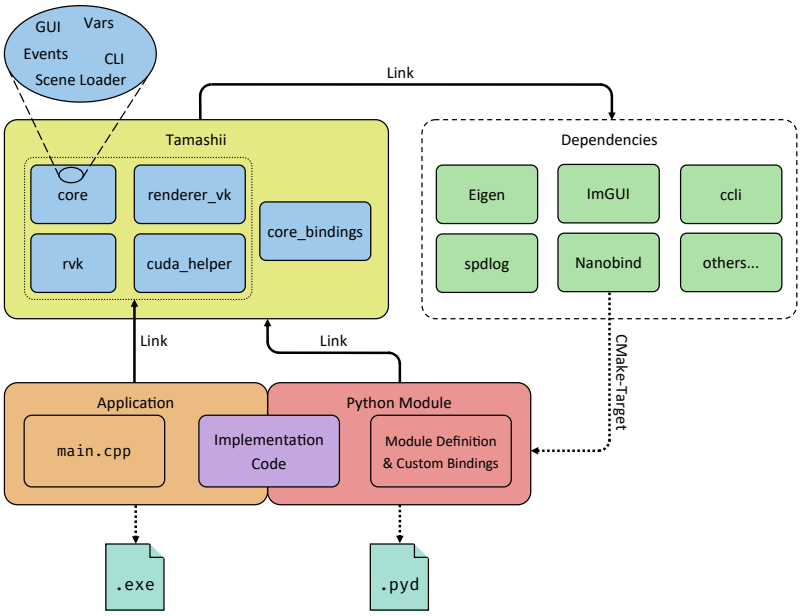


Figure 5.2: The code of the implementation is shared with the application and Python module compilation targets. Nanobind's CMake script creates a shared library compilation target, which also links to the core bindings.

Then a few lines are added to its `CMakeLists.txt` script that control whether the bindings should be included in the build process. The resources required for the bindings themselves consist of a standardized build script, where only a name constant needs to be updated, and a C++ source file that defines the Nanobind module and instantiates the core module. We list the necessary additions to the base CMake script in Listing 5.1 and give an example of a minimal module definition in Listing 5.2. The full CMake script template that needs to be included in the `python_bindings` directory is provided in Listing 7.

Thanks to this setup only minor changes to existing code are necessary for implementation developers to add Python bindings. Further, the required changes are exclusively additions of mostly premade code section, with the only exception being the refactoring of the entry point function as a separate file. Since the prototypes from Chapter 4.1 are modeled after Tamashii's codebase structure, we had the chance to experiment and find the least invasive approach to adapt the build system in a smaller and simpler environment. With this strategy we developed and tested the integration with Nanobind before needing to modify the CMake scripts of Tamashii.

```
# The line below or something similar already exists
file(GLOB_RECURSE SOURCES "*.h" "*.hpp" "*.c" "*.cpp")

# Addition 1: Exclude files only used for bindings
list(FILTER SOURCES EXCLUDE REGEX "python_bindings/*.*")

# [...] Regular build target definition

# Addition 2: Add optional build target for the bindings
if(BUILD_PYTHON_BINDINGS)
    add_subdirectory(python_bindings)
endif()
```

Listing 5.1: Two sets of patches have to be added to the CMake script to enable building of custom Python bindings.

5.2 Configuration variable system

The Tamashii codebase makes heavy use of globally accessible configuration parameters. These attributes define for example, whether the application should start in windowed mode, rendering background color and the implementation to load. In the legacy system, each variable is a C++ object that stores the variable's value, name, description and modifier bits. They conveniently provide limited reflection capabilities for automatic command line argument parsing and configuration persistence. This way, adding a new configuration value, that can be set through the CLI and be saved to a file is as simple as writing a single line of code that declares the variable object.

However, the way they were realized came with a few downsides. First, the values stored by variables were dynamically typed, which was a never applied feature and allowed for potential type errors during runtime. Moreover, the intended usage of specific variables and the data they contained was not self-documenting in code for this reason. Second, instead of a unified internal storage for all potential data types, they relied on a private member field for each kind of assignable type, which wasted memory and increased the complexity of state management code. Third and finally, their reflection abilities were an implementation detail, that could not be employed by outside code.

As the last shortcoming was of special interest to the creation of the Python bindings, we developed a new implementation that tries to solve the problems with the original implementation, while keeping all the functionalities used by the Tamashii codebase. Additionally, we created it as a library that is separate from Tamashii named CCLI to facilitate reusage in other projects [LP23]. To address the first issue, the design of CCLI makes heavy use of C++ template mechanisms. Variables are now strongly typed and may therefore only store one kind of value. They can be defined as scalar or vectorized variables that store an array of values. With the help of modern C++'s deduction capabilities most template parameters can be omitted for simple variable declarations. As a variable now only support one data type, the second issue also goes away. Last but

not least, all template specialized variable classes inherit from a common base class that allows for runtime type erasure. In combination with an interface that permits iteration over all active variables, the name, value and data type of each variable can be found.

This capability also enables to emulate the old implementation's dynamically typed variables to a certain extend. While their value type cannot be modified anymore, it is still possible to hand them over as untyped opaque data containers when base class pointers are used. They give access to a set of virtual methods, that can be used to read and write typed data in a fallible manner, in case the types do not match.

5.3 Core Module

The core module class represents the Python bindings on the C++ side. It is part of the core bindings of Tamashii and gives implementation developers access to the premade bindings, to extend and modify them. However, to enable a maximum of flexibility, so that a developer can still leverage premade bindings when creating their own ones even if the concept of such a core module does not fit their needs, we made the core module purely optional. It is implemented as a C++ class which is instantiated at the start of the Nanobind module function and registered to the system. Any bindings created by the core module can then be retrieved through the so called exports object, that holds all binding handles exposed to Python.

As the core module initializes the Tamashii framework it needs to create the implementation instances which are user defined types. Therefore the core module is a template, that takes a parameter pack list of all available implementation classes. But to prevent the whole class from needing to be templated, only the user facing part is, while the bulk of code is part of a plain base class, so it can be implemented out-of-line.

Only a single core module instance can be created and registered. Hence, it is realized as a singleton object, that is dynamically allocated by the user and handed over for registration. Listing 5.2 shows how an implementation instantiates and registers the core module with its renderer backend. Dynamic allocation by the user has a few upsides over a global static instance: No resources are needed when the core module is not in use to make it fully optional. Moreover, any possible side effects of the constructor happen at a known point in time, as it would be otherwise executed at some point during dynamic library loading before user code. Handling and communication of errors is an issue for this kind of setup. Furthermore, by having the user provide the instance they can write their own sub-classed version of the core module to extend or modify its behavior. Also custom memory allocation strategies can be employed and possible resource acquisition failure can be handled directly at the allocation site. Finally, conceptual separation is strengthened with the instantiation of the specialized core module template being part of the user's concern, and passing it over for registration marking the boundary to the core bindings inner workings.

Since implementation developers might want to add to the binding handles created

```cpp
// [...] Includes

T_USE_PYTHON_NAMESPACE
T_USE_NANOBIND_LITERALS

// The module name has to match the name set in the CMakeLists.txt
NB_MODULE(pymashii, m) {
  // Create and define the core module infrastructure
  using BackendList = RenderBackendTypeList<VulkanRenderBackendDefault>;
  auto coreModule = std::make_unique<CoreModule<BackendList>>();
  CoreModuleBase::defineCoreModule(m, std::move(coreModule) );

  // Modify parts of the core module
  CoreModuleBase::the().exports().light().def("customMethod", [](python::
    Light& self) {
    return "This string was returned from a custom method";
  });
}
```

Listing 5.2: Defining a Nanobind module named “pymashii” and instantiating the core module. The premade bindings can be modified by querying the exports object.

by the core module, they need to be accessible after their definition. Hence, they are held by the exports object that can be queried through the core module’s interface. Inside, each Nanobind handle is stored in a member of type STL optional, due to their lack of default constructability and to forgo the need of a heap allocation for each one. The optionals are populated by the core module during registration using an emplace operation. By accessing the handles through the exports object, the bindings of a Tamashii implementation can call methods on the Nanobind handles to modify them. Thereby, they can for example add custom methods and properties to classes exported by the core module. As a demonstration we extend the `Light` class with a custom method in Listing 5.2.

Access to Tamashii’s configuration variables is necessary to have control over many of Tamashii’s behaviors. Hence, making them available in Python by creating bindings to all of them is a required part of the core module. However, as a naïve implementation of this addition would result in a lot of code duplication that is hard to maintain, we selected a more sophisticated and scalable approach. Due to the switch to CCLI for the definition of Tamashii’s configuration variables, we are able to automatically generate all Python bindings for them in the core module. They are realized as read-write properties of the Tamashii instance, that transfer the value to and from the underlying variables when accessed. During the registration phase we use CCLI to iterate over all active variables and add them to the bindings. With the help of C++ template meta programming the necessary getter and setter callback lambda functions are passed to Nanobind. As CCLI supports vectorized variables that hold multiple values, extra specializations are necessary that convert them to Python lists. Moreover, we introduced special cases that handle

variables with two and three values by wrapping them as Python tuples instead. With this addition paired values like 3D vectors, window size or RGB colors can be exchanged as tuples.

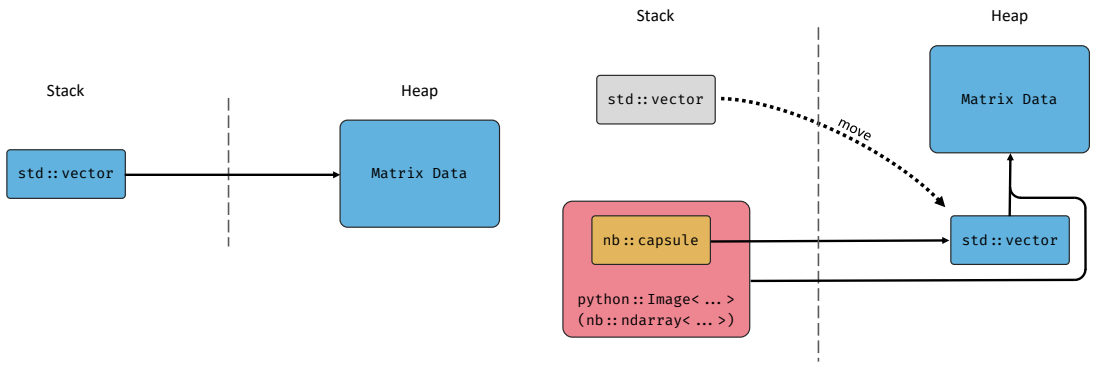
5.4 Cross-Language memory management

Exchanging data with CPython works exclusively through heap allocated objects, as the interpreter stores every kind of value as a reference counted object including simple numbers. Furthermore, CPython tries to take strong ownership of any object handed to it and manages its life time like any other object created from within Python. Also, values that are sent to a native module as function call parameters could be held by other Python objects. Therefore, when dealing with data communicated over the CPython API boundary, shared ownership has to be assumed. Thus, any C++ code that deals with these kinds of values, cannot simply move the object's contents, deallocate it, or create references that might outlive it, as ultimately CPython is in control of its lifetime. Moreover, to safely reference any values received from Python, shared ownership has to be indicated by modifying the reference count.

Due to the technical difficulties and limitations of requiring `unique_ptrs` as function parameters, we decided to exclusively expect objects to be provided as `shared_ptr` type to the interface [Wc23b]. This way the deallocation of objects remains simpler, as Nanobind can hide the call to CPython with type erasure and no special deallocator types need to be introduced into the code. Furthermore, the interface behaves more like any other Python function, as unique ownership is out of the question there and any different behavior would be confusing. On the other hand, the interface has to deal with the fact, that Python code might still be in possession of any object that it receives. Hence, calls to methods on said object might occur at any point in during the program's execution. For example, when the Python script attaches a new light to the scene, it might hold on to the object, even after the scene is unloaded.

Handing over values that were created on the C++ side to the Python one does not have to deal with this issue, which therefore makes the process simpler for bindings authors. A `unique_ptr` can be returned without specifying any return policy, and Nanobind automatically transfers ownership of the memory over to CPython without the need for copying. This feature allows for a performant exchange of data from C++ back to Python. The interface returns objects requested from the scene graph like the lights this way.

Additionally, the mechanism of capsules explained in Chapter 3.5 for heap allocated memory is an important part of the data exchange for screenshot data from Tamashii to the Python environment. When the script schedules a screenshot, the resulting pixel data is transferred back to it as a Numpy `ndarray` of the shape $W \times H \times 3$, where W and H are the width and height of the window's viewport. No data has to be copied, as the Tamashii renderer provides it contained in a STL vector whose contents are moved into a heap allocated vector instance managed by a Python capsule. This transferal can



(a) Tamashii's renderer returns the screenshot data as a STL `vector`.

(b) The data is moved into a heap allocated `vector` which is referred to by the capsule and the image's `ndarray`.

Figure 5.3: Transferring ownership of screenshot data to a Nanobind capsule to prevent copying of the data when returning it to CPython.

be seen in Figure 5.3. The capsuled vector serves as the backing of the matrix returned through Nanobind. The Python script receives a Numpy `ndarray` than can be used like any other one, but when it is freed, the capsule handles the destruction of the STL `vector` instance. The code responsible for screenshots that get returned to Python is very compact and displayed in Listing 5.3.

```
python::Image<uint8_t> TamashiiInstance::takeScreenshot() const
{
  auto screenshot= Common::getInstance().screenshot();

  // Move the data into a heap allocated vector than can be held by CPython
  const auto vectorHandle = new std::vector<uint8_t>{
    std::move(screenshot.data)
  };
  nb::capsule owner(vectorHandle, "Screenshot capsule", [](void* ptr)
    noexcept {
    delete static_cast<std::vector<uint8_t>*>(ptr);
  });

  size_t shape[3] = { screenshot.height, screenshot.width, 3 };
  return { vectorHandle->data(), 3, shape, owner};
}
```

Listing 5.3: Screenshot data is moved into an object managed by a Nanobind capsule

To realize a live representation of scene graph components like the lights or the camera, references to the equivalent C++ objects are necessary. However, this presents two challenges for guaranteeing sound life time dependencies. First, the Python object has to be able to take informed ownership of the scene graph component, to prevent a use-after-free scenario, when the graph changes due to a scene reload or a modification

through the GUI. Second, the referenced component has to be part of a currently active scene, so that any changes can take effect when rendering. The Python bindings classes for these items therefore use `weak_ptr` to reference into the scene graph. Hence, they do not take any strong ownership, but have the ability to check whether their reference is still valid to be dereferenced. Furthermore, they check the currently active scene, if their component is still in use. This way unintentional accesses of retired scene components can be reported to the script developer by throwing an exception that is accompanied with an error message.

Implementing this kind of shared ownership model into the existing codebase of Tamashii required considerable alterations. Including, but not limited to the scene graph, rendering scene and scene loader classes underwent in-depth refactoring of their memory and ownership management. These classes relied on completely manually managed allocations with `new` and `delete` featuring complex hand-written destructors that free vectors storing pointers. Since the interface's MVP focuses on the interaction of the Python script with selected parts of the scene graph like the scene, lights and camera, most changes originated in the classes related to these facilities. However, due to the interconnected nature of the code, most modifications have a propagating effect on other parts of the program. In this instance, the main focus was on switching the ownership model of scene graph entities from raw C pointers to C++ `shared_ptr` in the render scene class. Even though the Python script mostly interacts with this class, patches were also necessary in the code responsible for producing, handling and consuming the entities, as they also take shared or sole ownership of these objects at least for a short period of time.

For this reason, for example, the scene importer and exporter infrastructure of Tamashii needed to be updated to emit objects with automatically managed life times. While the latter only supports the `gltf` format, the importer also loads `bsp` and `ies` files, which enlarges the surface area for refactoring. But in addition to replacing the owning pointer types and updating the allocation cites accordingly, we had to modify parameter passing by reference for internal methods to use C++ rvalue references. In consequence to these changes, also helper data structures to transfer the imported scene data and to mark entity selection in the GUI editor, plus the methods that consume them required transformation. The consumers of scene graph entities are the GUI and the rendering infrastructure in Tamashii. Although the GUI never takes ownership of any entities it is heavily involved with the scene graph reading and modifying it. Therefore, there are many places where we replaced raw pointer accesses with rvalue references. Besides changes to the virtual render backend interface that notifies implementations when entities are added or removed, we revised the render command objects which get queued. They are realized as a sum type that can hold differently typed parameters as payload depending on the command type. Their implementation previously relied on a union, but introducing members with non-trivial destructors like `shared_ptr` is problematic, as it requires hand-written custom clean up code. Instead we opted to switch to a STL `variant`, that internally stores the currently held type, provides type safe accessors during runtime and automatically runs destructors.

5.5 Adding custom bindings

In addition to creating the bindings that reside in the core module available to all Tamashii implementation developers, we set out to add basic custom bindings to one of the implementations for differential rendering algorithms which is currently in development. As laid out before in Chapter 2.4, a renderer that utilized differentiable algorithms to compute a resulting image can be used to optimize specific scene parameters to match its result to a desired target. In case of the IALT Tamashii implementation the renderer offers two main functions for the purpose of rendering and computing according gradients called `forward` and `backward`. Furthermore, it comes with a loss algorithm that is initialized with a target scene state to then calculate a scalar value that describes the deviation to the current scene state. The implementation also allows selecting the scene parameters which should be considered in the gradient computation. Test cases for these operations existed previously only in C++ directly integrated into the implementation codebase.

To access the features of the IALT implementation from within Python, we added custom bindings on top of the core module, that export the rendering, targeting and de-/serialization functions. The latter are necessary to interface with numeric optimizers. The whole state of the scene based on the selected parameters is serialized into a large vector, which is accompanied by a similar such vector of the according gradients for each parameter. These two items can then be provided to optimizers which are agnostic to the specific problem like ADAM and output the next optimization step again as a vector. To convert between the scene representations as the scene graph held on the C++ side and the parameter vectors handled on the Python side, we export two methods that implement the `parameterVectorToLights` and `lightsToParameterVector` functions. As IALT uses the C++ Eigen library for many of its computations and stores its matrix data in `EigenVectorXd` objects [GJ+10]. Similar to the way we transfer screenshot data in Chapter 5.4, we again employ CPython capsules to move the ownership of the underlying memory block to the Python context without having to reallocate and copy its contents. For receiving data from Python, we utilize Eigen's `Map` class, that allows wrapping an unowned memory block as a read-only matrix instance without the need of duplicating the data in memory. Listing 5.4 lists the code implementing the `parameterVectorToLights` method where we wrap the `ndarray` as a variable called `mappedVec`. With the help of the exports objects, we can attach the methods to the scene class predefined by the core bindings.

We represent the definition of parameters per light instance to include during computation as its own class. This way it is possible to add convenience methods for common operations like enabling or disabling special patterns of parameters. For example methods exist for the default configuration, all parameters, no parameters, enabling rotation or position parameters. These methods are useful when an optimization problem is initialized. In Listing 5.5 we write two demonstrative Python functions that setup a scene to either optimize for all available parameters or only the light's intensities. For debug purposes a Python `__repr__` method for stringifying the configuration was also introduced. The

```
CoreModuleBase::the().exports().scene().def(
  "parameterVectorToLights",
  [](python::Scene& scene, python::Array<double> vecArray) {
    if (vecArray.ndim() != 1) {
      throw std::runtime_error{
        "Invalid parameter vector. Expected only a single dimension."
      };
    }

    const Eigen::Map<Eigen::VectorXd> mappedVec{
      vecArray.data(), static_cast<unsigned int>(vecArray.shape(0))
    };
    auto& ialt = getIALTInstance();
    ialt.getOptimizer().parameterVectorToLights(mappedVec);
  }, "parameterVector"_a);
```

Listing 5.4: Implementation of `parameterVectorToLights` utilizing `Eigen::Map` to prevent copying of data

optimization parameters are accessible for each light on the scene graph through a method added to the core module light class.

```
def optimizeLightsAll( t ):
  lights= t.scene.getLights()
  for l in lights:
    l.optimize().setAll()

def optimizeLightsOnlyIntensity( t ):
  lights= t.scene.getLights()
  for l in lights:
    opt= l.optimize()
    opt.reset()
    opt.intensity= True
```

Listing 5.5: Selecting different parameter configurations for gradient calculation

CHAPTER 6

Evaluation

In the following chapter we describe the process of creating experimental setups that use the Python interface to interact with Tamashii and perform different kinds of tests, and discuss the viability of this method to experiment. Three scenarios are evaluated based on two Tamashii implementations with Python bindings. The first two scenarios concern the optimization of scene parameters using the IALT Tamashii implementation developed by Lipp and Hahn [HL22]. The final arrangement deals with the training of a neural rendering framework with images automatically rendered by Tamashii utilizing the default implementation. We base these settings on experimental work already done employing Tamashii in combination with other technologies, to approximately represent the activities in the graphics research process that typically involve Tamashii.

6.1 Scene parameter optimization

Based on the algorithms contained in the IALT Tamashii implementation and the custom bindings created for it, different kinds of scene parameter optimization can be realized in a Python script. By applying the gradients to the scene parameters with a damping factor, we can make a simple form of gradient descent for example. For this to work, we need the single serialized vector with all the targeted scene parameters packed into it and the accompanying gradient vector. This basic approach can be greatly enhanced with the help of the ADAM optimization algorithm when we feed it the parameter and gradient vectors instead.

In the Python script we first place a light in the scene and save the illumination produced as the target state of the scene. Then, the light is moved to a random starting position. The `lightsToParameterVec` method converts the state of the scene into a Numpy vector that is to be updated by ADAM and fed back into `forward`. Calling `backwards` computes the loss value and gradients, that are needed by ADAM to perform the next optimization step. We perform these operations for a preset number of iterations, which

```
# [...] Imports, init Tamashii, open scene

def setLightPos(light, pos):
  m = light.modelMatrix
  m[3][0:3] = pos
  light.modelMatrix = m

light = t.scene.getLights()[0]    # Prepare target
setLightPos(light, target)
t.frame()
t.currentRadianceAsTarget(True)

setLightPos(light, init)          # Set light to init position
t.frame()

loss_hist = []
params = t.scene.lightsToParameterVector()
adam = Adam(params, 0.25)

for i in range(iterations):       # Iterate with ADAM
  t.forward(params)
  t.frame()
  phi, grads= t.backward()
  loss_hist.append(phi)

  adam.step(grads)
```

Listing 6.1: Optimizing a light position with ADAM to find a target configuration

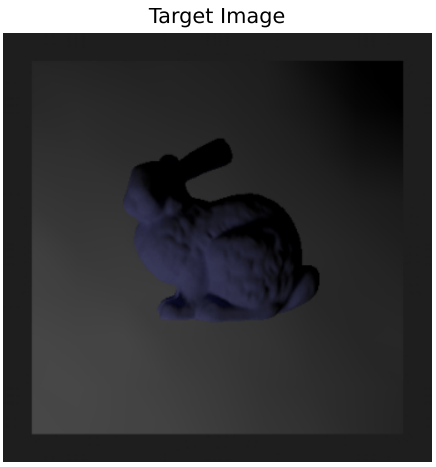
is presented in Listing 6.1. By logging the loss value and taking screenshots before and after, the results can be visualized by the script using "matplotlib" [Hun07]. When activating the GUI, we can view the optimization process live as the light source moves into the target position.

The graphs generated show that both the simple gradient descent, as well as ADAM find optimal light positions, as can be seen by the very small residual loss value. We can also observe that the final rendered image matches the original target image, which are displayed in Figure 6.1. However, ADAM operates very efficiently and only takes a fraction of the rendering iterations compared to the selected gradient descent approach. The comparison of the graphs in Figure 6.2 makes this especially apparent. Nevertheless, it should be noted that no effort was made to optimize the damping parameter.

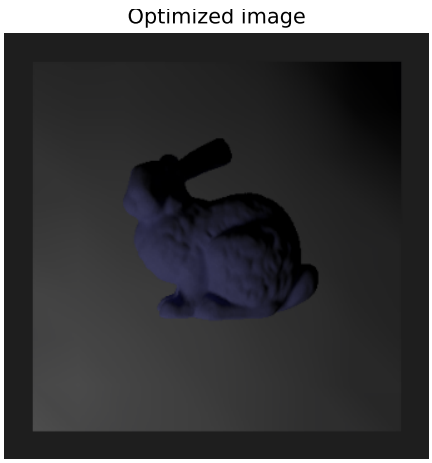
6.2 Optimization supported by surrogate modeling

In addition to using Adam as an optimizer, we can employ surrogate modeling to further improve the performance of the experiment. By first training a GEKPLS [BM19] model, as first mentioned in Chapter 2.6, with positions and gradients from the scene it can be utilized by the optimizer instead of asking Tamashii to render and calculate gradients for

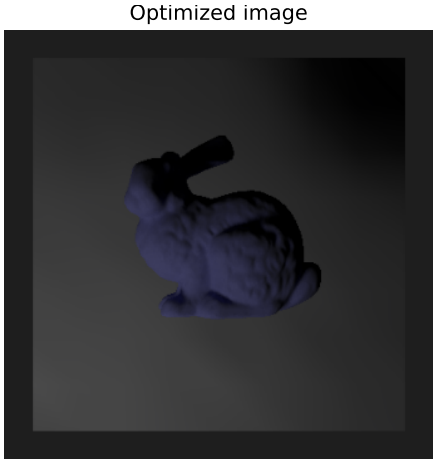
(a) Initial light location before optimization.



(c) Target light location.



(e) Light location optimized using gradient descent with $\gamma = 0.98$ after 2500 steps.



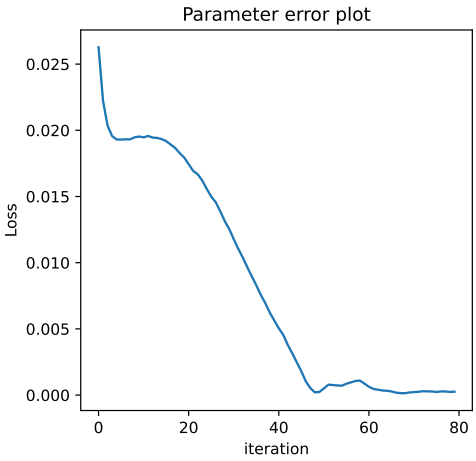
(g) Light location optimized using ADAM after 80 steps.

Figure 6.1: Images rendered from scene states before and after optimization with either simple gradient descent or ADAM.

each step. Thanks to the SMT library a configurable implementation of the modeling algorithm is available. This test case was already written in Python, but communicated with Tamashii via its CLI and executed regular expressions on the output text to locate and parse the output values. Due to Tamashii having to be started and run to completion for each simulation step, the test script suffered from a prolonged runtime.

By porting the script to make use of the bindings created for the previous test, the same Tamashii instance can be kept active and instructed to perform all the simulation steps.

(a) Loss value over 2500 steps of gradient descent optimization.

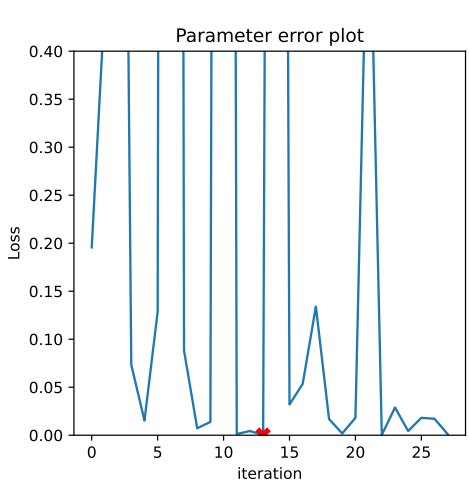


(c) Loss value over 80 steps of ADAM optimization.

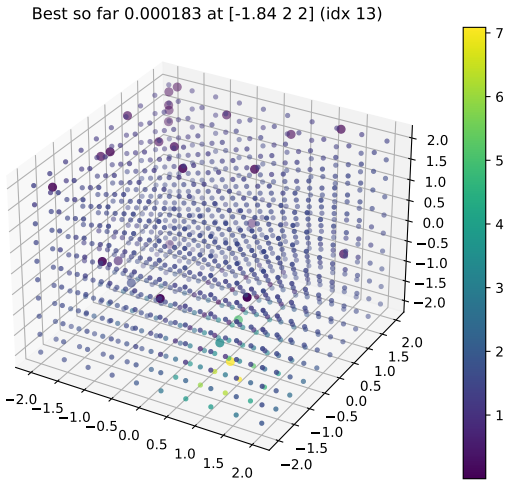
Figure 6.2: Comparison of the plotted loss values when optimizing with gradient descent or ADAM.

The data exchange can also be done directly by receiving the gradients as Numpy arrays from `backward`. Training the GEKPLS model happens with a list of lamp position and associated gradient pairs. These tuples are computed by Tamashii by sampling the loaded scene using Latin Hypercube sampling (LHS) in a three-dimensional pattern. The Latin Hypercube sampling (LHS) algorithm is provided by SMT as well [Hwa17] [BHB+19]. With the help of the model a larger number of approximated value pairs is generated in a 3D grid pattern that is then sent to the optimizer. This algorithm then tries to minimize the error within the model's values. When a minimum is found, the parameter vector of the scene is updated and Tamashii re-renders it with a resulting loss and gradients vector. The new gradients are appended to the training data set and the GEKPLS model is re-trained. This procedure improves the accuracy of the model in the area of interest. Here, "matplotlib" also visualizes the optimization progress and result.

Again, the optimization algorithm is able to find the optimal light position to illuminate the mesh like in the target configuration. But compared to the previous approach of only using ADAM, this method only needs to perform less than a quarter of the rendering iterations for this arrangement. Still, the actual runtime is longer than pure ADAM. However, both scripts were not tuned for runtime performance in any form. From the graph plotting the loss value in Figure 6.3a we can see comparatively erratic behavior, but also a continuous downtrend of the best minimum value. Like for many other fields that rely on optimization algorithms, this method also shows to be a promising way for this kind of problem to reduce computational cost. For more complex scenes where the



(a) Loss values of the 28 rendering steps to train and update the GEKPLS model. The lowest value (index 13) is marked in red.



(b) Light locations used for rendering.

Figure 6.3: Optimizing the light position with GEKPLS enhanced ADAM.

cost of repeated rendering becomes too expensive, surrogate modeling could be a valuable remedy.

6.3 Training of a neural renderer

The final setup shows Tamashii's capabilities as a programmatically controlled renderer, which we demonstrate by creating images for a ML training dataset. A set of images is made that depicts a 3D model from many different perspectives, by moving the camera around it. We provide the Nerfstudio software with the images to train a NN which learns to compute illumination values for queried locations, using the NeRF approach explained in Chapter 2.5. When the training process is finished, the network is able to approximate values for viewing positions that were not part of its known data, allowing the synthesis of new images. For the script driving the image generation the base functionalities provided by the core module are sufficient.

A scene gets loaded into Tamashii containing a single 3D model. We then rotate the camera around the scene's center while facing inwards in the shape of a sphere. The code required for this task is listen in Listing 6.2. The rendered images are imported into Nerfstudio through its CLI, which prepares the image data by calculating down-sampled versions and view direction reconstruction. Even though the exact locations of the camera are known, the latter step is a fixed part of the Nerfstudio pipeline. The finished NN can be loaded into Nerfstudio which locally hosts a browser based user interface that shows

```
# [...] Imports

def lerp(a, b, x, y, i):
    return a+ (b- a)* (i / (y- x))

outDir= 'img'
# [...] Clear output directory

t= pymashii.Tamashii()
t.bg= (255,255,255)
t.window_size= (550,550)

t.openScene("dozer.gltf")
t.frame()
c= t.scene.getCamera()

radius= 3.5
alpha= acos(-2.5/radius)
beta= acos(2.1/radius)
center= np.array([20,0,0])
position= np.array([0,0,0], dtype= float)

ctr= 0
for i in range(0, 10):                              # 10 parallels
  angle= lerp(alpha, beta, 0, 10, i)
  localRadius= radius* sin(angle)
  position[1]= radius* cos(angle)
  for j in range(0, 40):                            # 40 meridians
    position[0]= localRadius* sin(j * 0.15) + 20
    position[2]= localRadius* cos(j * 0.15)
    c.lookAt(position, center)                      # Move and rotate camera
    t.frame()
    t.saveScreenshot(f'{outDir}/shot-{ctr}.png', False, False)
    ctr+= 1
```

Listing 6.2: Capturing training data with automated rendering

the training data images in 3D space and superimposing the resulting neurally rendered image.

The images generated by Nerfstudio confirm that the process works in principle. We can check how the software assigns the images with locations and viewing directions using the browser based GUI shown in Figure 6.4a. However, the quality of the renderings is very poor compared to experiments in Figure 1 where regular photographs taken with a smartphone were used as training data. The output clearly shows the 3D shape of the depicted object when rotated, however they lack any kind of shading as illustrated with the sample in Figure 6.4c. This defect could be due to the lack of texture on the 3D model, the absence of a background, the distance between the viewing locations or the simplicity of the model. Still, the test confirms, that the Python scripting interface is a viable

(a) Depiction of the camera positions matched by Nerfstudio for training.

(b) Image rendered and exported from Tamashii.

(c) Image rendered by Nerfstudio.

Figure 6.4: Rendering a 3D model with Nerfstudio trained on images exported from Tamashii.

means of workflow automation and enables Tamashii to be used in a programmatically controlled manner.

CHAPTER 7

Discussion & Conclusion

In this thesis we show, that the Tamashii rendering framework can successfully be programmatically controlled, integrated into data processing workflows and implementations automatically tested by adding Python bindings. The work we covered, includes the definition and realization of a MVP, that offers the most important functionalities of Tamashii. We selected the scope according to the needs of existing test code, easy expansion and potential requirements in the very near future. Hence, we could port over experiments to a new Python implementation, and demonstrated how new ones can be added. To facilitate the interaction between Tamashii and the CPython interpreter, extensive changes in the memory management and formalizing object ownership were necessary. In addition to moving to modern C++ smart pointers in critical places, we extended Tamashii with a component that contains Python bindings for the other facilities of the core component, such as the scene graph and renderer. Its design is highly modular to allow for easy customization by implementation developers, who integrate their own code and need to alter the predefined interface. In case the Python bindings are redundant altogether, they can be fully disabled during the build phase. Moreover, we built a new library handling the global configuration variables, that offers better type-safety, a lower memory footprint, support for callbacks and public access to its reflection capabilities. All these changes and more, were moved into Tamashii's main codebase in the end, making bindings creation available to its users.

In conclusion, we met the main goals outlined in the beginning and the result satisfies the targeted feature requirements. The addition of bindings to existing Tamashii implementations is a quick and simple process that involves none or only a few changes to the code, and a small set of extensions of the build scripts, that can be taken from the examples provided. This arrangement makes for a seamless experience when switching to a more automated workflow with Tamashii. Additionally, we lay out a clear direction for porting existing test and experimentation code from various shell and Python scripts which are limited to Tamashii's CLI, to a more flexible, expressive, performant and maintainable

format. Users are also given the option and necessary tools to replace, modify or extend the provided bindings to fit their needs. The ability to access the large ecosystem of Python libraries makes including dependencies simple and repeatable on distribution. This way researchers can rely on established modules for effective visualization, modern ML algorithms, diverse optimization strategies and convenient data processing functions, only to name a few. All of these aspects are in line with Tamashii's spirit of providing powerful pre-built infrastructure to developers of computer graphics algorithms.

However, there are also prominent limitations of the interface in its current form. It exposes only a fraction of Tamashii's full capabilities. Furthermore, it repurposes code originally written for the GUI, which does not offer ideal functions in some areas. As Python interacts with a Tamashii in a single threaded manner, some by default parallelized operations have to be executed in the main thread, to simplify the bindings. Due to the limited scope of this thesis, only the parts of Tamashii's resource management were modernized, that were required for the MVP.

In summary, there are still many hypothetical improvements to the Python interface and Tamashii in general that could be implemented. Future work that focuses on the latter has the opportunity to tackle the remaining parts of Tamashii that rely on manual resource management and nondescript pointer types. Possibly, we could employ an even safer alternative to STL smart pointers, that do not merely convey and control ownership, but also communicate and enforce other traits like potential null-ness, span length or borrowed life time. The changes we made to Tamashii's memory management were also done in a heavy handed way that makes very liberal use of `shared_ptr` which we could optimize if the relevant parts of the code made better guarantees about their usage of potentially heap allocated objects.

Moreover, we could expand the core bindings to feature more of Tamashii's functionality and better map the internal object hierarchy. Likewise, we currently do not leverage CCLI to its fullest, as nearly no variables are equipped with callback functions that would allow propagation of live updates of configuration values. Furthermore, if Nanobind and therefore the python bindings added support for asynchronous function calls [Fou23h], we could express multi-threaded and background operations in a way natural to the Python language. Possible additions to the functional range of the bindings could include more direct access to the GPU through its shaders, rendering queue or bindings to "rvk", facilities to add custom GUI elements, or the ability to interact with Tamashii's event loop.

There are also more drastic modifications possible, in case Tamashii were to put a stronger focus on Python scripting in the future. If we decide to distribute it as a Python package at some point, it would make sense for us to re-architect it in a more modular way, where each component becomes a reusable module accessible from within Python.

Overall, we successfully implemented an interface to the Python language that enables ergonomic workflows with the Tamashii framework. With the realization of the MVP we show the promising potential of Tamashii as an automated tool when controlled by

a scripting language. Thereby, we clearly mark a future way forward for the research enabled by Tamashii.

List of Figures

List of Tables

List of Code Listings

Acronyms

AOT ahead of time. 6

API application programming interface. 3, 10, 17–19, 21–24, 29, 43

CLI command line interface. 17, 18, 30–32, 34, 40, 51, 53, 57

CPU central processing unit. 14, 20

GC garbage collection. 5, 6, 8, 9, 20, 24, 61

GEKPLS gradient-enhanced kriging with partial least squares. 15, 16, 50, 52, 53, 61

GPL GNU General Public License. 7

GPU graphics processing unit. xi, xiii, 12, 14, 34, 35, 58

GUI graphical user interface. 17–19, 30–32, 34, 45, 50, 54, 58, 65, 82, 83

IALT Interactive Adjoint Light Tracer. 37, 46, 49

ISA instruction set architecture. 20

JIT just in time. 6, 12, 20

JVM Java virtual machine. 8

LHS Latin Hypercube sampling. 52

lvalue left value. 27

ML machine learning. 2, 3, 12, 13, 15, 20, 23, 53, 58

MVP minimal viable product. 33, 37, 45, 57, 58

NeRF neural radiance fields. 3, 14, 15, 53

NN neural network. 11–15, 53

RAII Resource acquisition is initialization. 9, 23

REPL Read eval print loop. 19

rvalue right value. 27, 45

SMT surrogate modeling toolbox. 3, 5, 16, 51, 52

STL standard template library. 9, 10, 23–26, 42–45, 58

Bibliography

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow, Large-scale machine learning on heterogeneous systems, November 2015. Repository.

[ACJP15] Michael Asher, Barry Croke, A.J. Jakeman, and L. Peeters. A review of surrogate models and their application to groundwater modeling. *Water Resources Research*, 51, 07 2015. Link.

[aia23] Oracle and/or its affiliates. Java native interface specification contents: Design overview. `https://web.archive.org/web/20230823134210/https://docs.oracle.com/en/java/javase/20/docs/specs/jni/design.html`, ©1993-2023. Online accessed: 23rd August 2023 Link.

[BHB+19] Mohamed Amine Bouhlel, John T. Hwang, Nathalie Bartoli, Rémi Lafage, Joseph Morlier, and Joaquim R.R.A. Martins. A python surrogate modeling framework with derivatives. *Advances in Engineering Software*, 135:102662, 2019. Repository Link.

[BM19] Mohamed Amine Bouhlel and Joaquim Martins. Gradient-enhanced kriging for high-dimensional problems. *Engineering with Computers*, 35, 01 2019. Link.

[BPRS17] Atılım Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: A survey. *J. Mach. Learn. Res.*, 18(1):5595–5637, jan 2017. Link.

[CCD+23] Maintainers: Soumith Chintala, Gregory Chanan, Dmytro Dzhulgakov, Edward Yang, and Nikita Shulga. Authored by com-

munity members. Pytorch python module definition using pybind11. https://web.archive.org/web/20230818105537/https://raw.githubusercontent.com/pytorch/pytorch/61b6b038b02b38e0d2758d86305fdc24edf76d00/torch/csrc/Module.cpp, May 2023. Online accessed: 18th August 2023.

[Chi21] Soumith Chintala. Growing open-source: From torch to pytorch. https://web.archive.org/web/20230818124028/https://soumith.ch/posts/2021/02/growing-opensource/, August 2021. Online accessed: 18th August 2023.

[CKF11] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS workshop*, number CONF, 2011. Repository Link.

[con23] Zig contributors. Zig language reference. https://web.archive.org/web/20230820095937/https://ziglang.org/documentation/master/, 2023. Online accessed: 20th August 2023 Link.

[Cor14] Omar Cornut. Dear imgui. https://web.archive.org/save/https://www.dearimgui.com/, July 2014. Repository.

[Cor23] Oracle Corporation. The garbage first garbage collector. https://web.archive.org/web/20230818170127/https://www.oracle.com/java/technologies/javase/hotspot-garbage-collection.html, ©2023. Online accessed: 18th August 2023.

[cWS+21] Microsoft Learn contributors, Tyler Whitney, Kent Sharkey, Mike Jones, Gordon Hogenson, Saisang Cai, and @nxt. Link an executable to a DLL. https://web.archive.org/web/20230820122619/https://learn.microsoft.com/en-us/cpp/build/linking-an-executable-to-a-dll?view=msvc-170, March 2021. Online accessed: 20th August 2023 Link.

[DEH+18] Ulan Degenbaev, Jochen Eisinger, Kentaro Hara, Marcel Hlopko, Michael Lippautz, and Hannes Payer. Cross-component garbage collection. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018. Link.

[Dev23] The Rust Project Developers. The rustonomicon: Foreign function interface. https://web.archive.org/web/20230820094936/https://doc.rust-lang.org/nomicon/ffi.html, 2023. Online accessed: 20th August 2023.

[Die75] Paul Dierckx. An algorithm for smoothing, differentiation and integration of experimental data using spline functions. *Journal of Computational and Applied Mathematics*, 1:165–184, 1975. Link.

[Eat88] John W. Eaton. Gnu octave: About. https://web.archive.org/web/20230910195254/https://octave.org/about, 1988. Online accessed: 10th September 2023.

[Fc23] OpenJS Foundation and Electron contributors. Electron: Desktop development made easy. https://web.archive.org/web/20230820142033/https://www.electronjs.org/, ©2023. Online accessed: 20th August 2023.

[FHR+23] Christian Freude, David Hahn, Florian Rist, Lukas Lipp, and Michael Wimmer. Precomputed radiative heat transport for efficient thermal simulation. *Computer Graphics Forum (Pacific Graphics 2023)*, 42, October 2023.

[FJL18] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 4(9), 2018. Link.

[Fou23a] Blender Foundation. Blender 3.6.3 release candidate Python API: Quickstart. https://web.archive.org/web/20230820135231/https://docs.blender.org/api/current/info_quickstart.html, August 2023. Online accessed: 20th August 2023.

[Fou23b] Python Software Foundation. The bytecode interpreter (3.11). https://web.archive.org/web/20230820093920/https://devguide.python.org/internals/interpreter/, August ©2001-2023. Online accessed: 20th August 2023.

[Fou23c] Python Software Foundation. dis - Disassembler for Python bytecode. https://web.archive.org/web/20230820094338/https://docs.python.org/3/library/dis.html, August ©2001-2023. Online accessed: 20th August 2023 Link.

[Fou23d] Python Software Foundation. Embedding python in another application. https://web.archive.org/web/20230820134802/https://docs.python.org/3/extending/embedding.html, August ©2001-2023. Online accessed: 20th August 2023.

[Fou23e] Python Software Foundation. Extending Python with C or C++. https://web.archive.org/web/20230820124531/https://docs.python.org/3/extending/extending.html, August ©2001-2023. Online accessed: 20th August 2023.

[Fou23f] Python Software Foundation. General python faq. https://web.archive.org/web/20230820093012/https://docs.python.org/3/faq/general.html, August ©2001-2023. Online accessed: 20th August 2023.

[Fou23g] Python Software Foundation. Parsing arguments and building values. https://web.archive.org/web/20230820093642/https://docs.python.org/3/c-api/arg.html, August ©2001-2023. Online accessed: 20th August 2023.

[Fou23h] Python Software Foundation. The Ptython standard library: Coroutines and tasks. https://web.archive.org/web/20230820141647/https://docs.python.org/3/library/asyncio-task.html, August ©2001-2023. Online accessed: 20th August 2023.

[Fou23i] Python Software Foundation. The python standard library. https://web.archive.org/web/20230820093347/https://docs.python.org/3/library/index.html, August ©2001-2023. Online accessed: 20th August 2023.

[Fou23j] Python Software Foundation. The Python tutorial: 6. Modules. https://web.archive.org/web/20230820122316/https://docs.python.org/3/tutorial/modules.html, August ©2001-2023. Online accessed: 20th August 2023.

[Fre23a] Free Software Foundation. Link an executable to a DLL. https://web.archive.org/web/20230820124026/https://man7.org/linux/man-pages/man3/dlopen.3.html, March 2023. Online accessed: 20th August 2023.

[Fre23b] Free Software Foundation. 49 Emacs Lisp packages. https://web.archive.org/web/20230820132109/https://www.gnu.org/software/emacs/manual/html_node/emacs/Packages.html, ©1993–2023. Online accessed: 20th August 2023.

[GJ+10] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010. Repository.

[GO20] Micha Gorelick and Ian Ozsvald. *High Performance Python: Practical Performant Programming for Humans*. O'Reilly Media, April 2020. Link.

[Hel04] David B. Held. A proposal to add a policy-based smart pointer framework to the standard library. *JTC1/SC22/WG21 - The C++ Standards Committee - ISOCPP*, September 2004. Document: SC22/WG21/N1681 J16/04-0121 Link.

[HL22] David Hahn and Lukas Lipp. Personal communication, October 2022.

[HOvcm23] Jim Hugunin, Travis E. Oliphant, and various community members. Numpy implementation of the numpy multiarray module. https://web.archive.org/web/20230818104757/https://raw.githubusercontent.com/numpy/numpy/

4b5ae68c78a222eba2e07cc1e34bd29d5cfeb30d/numpy/core/src/multiarray/multiarraymodule.c, May 2023. Online accessed: 18th August 2023.

[Hun07] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. Website.

[Hwa17] John Hwang. Latin hypercube sampling. https://web.archive.org/web/20230828144524/https://smt.readthedocs.io/en/latest/_src_docs/sampling_methods/lhs.html, © 2017. Online accessed: 28th August 2023.

[Ic23] Kitware Inc. and contributors. CMake Reference Documentation: Introduction. https://web.archive.org/web/20230820135856/https://cmake.org/cmake/help/latest/, ©2000-2023. Online accessed: 20th August 2023.

[ID18] Eldar Insafutdinov and Alexey Dosovitskiy. Unsupervised learning of shape and pose with differentiable point clouds. *Advances in neural information processing systems*, 31, 2018. Link.

[IdFC07] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of Lua. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, page 2–1–2–26, New York, NY, USA, 2007. Association for Computing Machinery. Link.

[IdFC23] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. About Lua. https://web.archive.org/web/20230818163646/https://www.lua.org/about.html, May 2023. Online accessed: 18th August 2023 Showcase.

[Inc23] Autodesk Inc. Introduction (autolisp). https://web.archive.org/web/20230820130854/https://help.autodesk.com/view/OARX/2023/ENU/?guid=GUID-A0E9D801-8BE9-4BF1-85E8-3807E15F3B71, ©2023. Online accessed: 20th August 2023.

[JSR+22a] Wenzel Jakob, Sébastien Speierer, Nicolas Roussel, Merlin Nimier-David, Delio Vicini, Tizian Zeltner, Baptiste Nicolet, Miguel Crespo, Vincent Leroy, and Ziyi Zhang. Mitsuba 3 renderer, 2022. Website Repository.

[JSR+22b] Wenzel Jakob, Sébastien Speierer, Nicolas Roussel, Merlin Nimier-David, Delio Vicini, Tizian Zeltner, Baptiste Nicolet, Miguel Crespo, Vincent Leroy, and Ziyi Zhang. Mitsuba3 python module definition using pybind11. https://web.archive.org/web/20230818110606/https://raw.githubusercontent.com/mitsuba-renderer/mitsuba3/

6a1644335d9e9564afa6858456e19970706cefb8/src/python/main.cpp, October 2022. Online accessed: 18th August 2023.

[JSRV22] Wenzel Jakob, Sébastien Speierer, Nicolas Roussel, and Delio Vicini. Dr.jit: A just-in-time compiler for differentiable rendering. *Transactions on Graphics (Proceedings of SIGGRAPH)*, 41(4), July 2022. Repository Link.

[KB14] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014. Link.

[KBM+20] Hiroharu Kato, Deniz Beker, Mihai Morariu, Takahiro Ando, Toru Matsuoka, Wadim Kehl, and Adrien Gaidon. Differentiable rendering: A survey. *arXiv preprint arXiv:2006.12057*, 2020. Link.

[LBM18] Jichao Li, Mohamed Amine Bouhlel, and Joaquim Martins. Data-based approach for fast airfoil analysis and optimization. *AIAA Journal*, 57:1–16, 11 2018. Link.

[LJ23] Anselm Levskaya and Matthew Johnson. Jax - the sharp bits. https://web.archive.org/web/20230820091658/https://jax.readthedocs.io/en/latest/notebooks/Common_Gotchas_in_JAX.html, August 2023. Online accessed: 20th August 2023 Link.

[LMtGc23] Juan Linietsky, Ariel Manzur, and the Godot community. Godot engine 4.1 documentation: Scripting languages. https://web.archive.org/web/20230820134204/https://docs.godotengine.org/en/stable/getting_started/step_by_step/scripting_languages.html, ©2014-2023. Online accessed: 20th August 2023.

[LMTL21] Chen-Hsuan Lin, Wei-Chiu Ma, Antonio Torralba, and Simon Lucey. Barf: Bundle-adjusting neural radiance fields. In *IEEE International Conference on Computer Vision (ICCV)*, 2021. Repository Website Link.

[LP23] Lukas Lipp and Matthias Preymann. ccli - cool command line interface, 2023. Repository.

[Mac16] Dougal Maclaurin. *Modeling, inference and optimization with composable differentiable procedures*. PhD thesis, 2016. https://dash.harvard.edu/handle/1/33493599 Link.

[Mar09] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009. Link Link.

[MBRS+21] Ricardo Martin-Brualla, Noha Radwan, Mehdi S. M. Sajjadi, Jonathan T. Barron, Alexey Dosovitskiy, and Daniel Duckworth. NeRF in the Wild:

Neural Radiance Fields for Unconstrained Photo Collections. In *CVPR*, 2021. Website Link.

[MDJ+16] Dougal Maclaurin, David Duvenaud, Matt Johnson, Jamie Townsend, and many other contributors. Hips autograd, 2016. Repository.

[MESK22] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Trans. Graph.*, 41(4):102:1–102:15, July 2022. Repository Website Link.

[MS19] Kevin McBride and Kai Sundmacher. Overview of surrogate modeling in chemical process engineering. *Chemie Ingenieur Technik*, 91, 01 2019. Link.

[MST+21] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. *Commun. ACM*, 65(1):99–106, dec 2021. Repository Website Link.

[PGC+17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017. Link.

[PGM+19a] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. Repository Link.

[PGM19b] Paula Pufek, Hrvoje Grgic, and Branko Mihaljevic. Analysis of garbage collection algorithms and memory management in Java. pages 1677–1682, 05 2019. Link.

[PWc22] The Python Wiki contributors. Globalinterpreterlock. `https://web.archive.org/web/20230823152536/https://wiki.python.org/moin/GlobalInterpreterLock`, Dec 2022.

[RPN20] Sebastian Raschka, Joshua Patterson, and Corey Nolet. Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *Information*, 11(4), April 2020. Link.

[RRN+20] Nikhila Ravi, Jeremy Reizenstein, David Novotny, Taylor Gordon, Wan-Yen Lo, Justin Johnson, and Georgia Gkioxari. Accelerating 3d deep learning with pytorch3d. *arXiv:2007.08501*, 2020. Repository Link.

[Rud17] Sebastian Ruder. An overview of gradient descent optimization algorithms. 09 2017. Link.

[Sal23] Pablo Galindo Salgado. Garbage collector design. https://web.archive.org/web/20230818165714/https://devguide.python.org/internals/garbage-collector/, ©2011-2023. Online accessed: 18th August 2023.

[Sca23] Rizel Scarlett. Why python keeps growing, explained. https://web.archive.org/web/20230819164521/https://github.blog/2023-03-02-why-python-keeps-growing-explained/, March 2023. Online accessed: 28th August 2023.

[Sco09] Michael L. Scott. *Programming Language Pragmatics (Third Edition)*. Morgan Kaufmann, Boston, third edition edition, 2009. Link.

[SF16] Johannes Lutz Schönberger and Jan-Michael Frahm. Structure-from-motion revisited. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. Repository Website Link.

[Spe22] Sebastien Speierer. Mitsuba3 announces its planned move to nanobind. https://web.archive.org/web/20230818110736/https://github.com/mitsuba-renderer/mitsuba3/discussions/121, July 2022. Online accessed: 18th August 2023.

[SRS+23] Ariya Shajii, Gabriel Ramirez, Haris Smajlović, Jessica Ray, Bonnie Berger, Saman Amarasinghe, and Ibrahim Numanagić. Codon: A compiler for high-performance pythonic applications and dsls. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, CC 2023, page 191–202, New York, NY, USA, 2023. Association for Computing Machinery. Article Link.

[SSvcm23a] Bjarne Stroustrup, Herb Sutter, and various community members. C++ core guidelines. https://web.archive.org/web/20230818112327/https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines, April 2023. Online accessed: 18th August 2023.

[SSvcm23b] Bjarne Stroustrup, Herb Sutter, and various community members. C.41: A constructor should create a fully initialized object. https://web.archive.org/web/20230818112327/https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines, April 2023. Link.

[SSvcm23c] Bjarne Stroustrup, Herb Sutter, and various community members. R.3: A raw pointer (a t*) is non-owning. https://web.archive.org/web/20230818112327/https://isocpp.

github.io/CppCoreGuidelines/CppCoreGuidelines, April 2023. Link.

[SSvcm23d] Bjarne Stroustrup, Herb Sutter, and various community members. R.4: A raw reference (a t&) is non-owning. https://web.archive.org/web/20230818112327/https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines, April 2023. Link.

[SSvcm23e] Bjarne Stroustrup, Herb Sutter, and various community members. R.smart: Smart pointers. https://web.archive.org/web/20230818112327/https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines, April 2023. Link.

[Tec23] Unity Technologies. Coding in c# in unity for beginners. https://web.archive.org/web/20230820133727/https://unity.com/how-to/learning-c-sharp-unity-beginners, ©2023. Online accessed: 20th August 2023.

[Uc20] Rui Ueyama and contributors. mold: A modern linker. https://web.archive.org/save/https://github.com/rui314/mold, September 2020. Online accessed: 11th September 2023.

[vRWC01] Guido van Rossum, Barry Warsaw, and Nick Coghlan. PEP 8 – Style Guide for Python Code. https://web.archive.org/web/20230820135551/https://peps.python.org/pep-0008/, July 2001. Online accessed: 20th August 2023 Link.

[Wc23a] Jakob Wenzel and contributors. Embedding the interpreter. https://web.archive.org/web/20230820135016/https://pybind11.readthedocs.io/en/stable/advanced/embedding.html, July 2023. Online accessed: 20th August 2023.

[Wc23b] Jakob Wenzel and contributors. Object ownership, continued. https://web.archive.org/web/20230820140631/https://nanobind.readthedocs.io/en/latest/ownership_adv.html, April 2023. Online accessed: 20th August 2023 Link.

[Wc23c] Jakob Wenzel and contributors. pybind11 — seamless operability between c++11 and python. https://web.archive.org/web/20230820125955/https://pybind11.readthedocs.io/en/latest/, July 2023. Online accessed: 20th August 2023.

[Wc23d] Jakob Wenzel and contributors. Why another binding library? https://web.archive.org/web/20230820124843/https://nanobind.readthedocs.io/en/latest/why.html, April 2023. Online accessed: 20th August 2023.

[Wc23e] Jakob Wenzel and contributors. Why another binding library? https://web.archive.org/web/20230820141209/https://nanobind.readthedocs.io/en/latest/ownership.html, April 2023. Online accessed: 20th August 2023.

[Wik22] Wikipedia contributors. Category:lua (programming language)-scripted video games — Wikipedia, the free encyclopedia. https://web.archive.org/web/20230820133318/https://en.wikipedia.org/w/index.php?title=Category:Lua_%28programming_language%29-scripted_video_games&oldid=1119515244, November 2022. Online accessed: 20th August 2023.

[Wil06] Paul Wilson. *Uniprocessor Garbage Collection Techniques*, volume 637. 04 2006. Link.

[Xu22] Haoran Xu. Understanding garbage collection in javascriptcore from scratch. https://web.archive.org/web/20230819102054/https://webkit.org/blog/12967/understanding-gc-in-jsc-from-scratch/, July 2022. Online accessed: 19th August 2023.

[XXP+23] Linning Xu, Yuanbo Xiangli, Sida Peng, Xingang Pan, Nanxuan Zhao, Christian Theobalt, Bo Dai, and Dahua Lin. Grid-guided neural radiance fields for large urban scenes, 2023. Repository Website Link.

Appendix

Template CMake script to build custom bindings

The following Listing 7.1 contains the CMake script code that loads Nanobind, defines the Python module as a shared library build target and links to the appropriate libraries. Furthermore, it defines copy commands, that automatically gather runtime shared libraries to copy them to the install location. A developer of a Tamashii implementation has to set a custom name in the global variable that has to match the name passed to Nanobind's `NB_MODULE()` macro, and potentially update the list of link libraries.

```
# This name has to match the module name set in the 'NB_MODULE()' macro
set(PYMASHII pymashii)

# Get the sources starting from the parent directory
file(GLOB_RECURSE SOURCES "../*.h" "../*.hpp" "../*.c" "../*.cpp")

# Exclude the entry point of the main application
list(FILTER SOURCES EXCLUDE REGEX "../main.cpp")

# Grouping
source_group(TREE "${CMAKE_CURRENT_SOURCE_DIR}/.." PREFIX "Source Files"
    FILES ${SOURCES})

# nanobind already included by core_bindings
nanobind_add_module(
  ${PYMASHII}
  NB_STATIC # Build static libnanobind (the extension module itself remains a
     shared library)
  ${SOURCES}
)
# The name of the nanobind library target depends on the config above (
    NB_STATIC, NB_SHARED, LTO, STABLE_API,...)
set_target_properties(nanobind-static PROPERTIES FOLDER ${
    FRAMEWORK_EXTERNAL_FOLDER})

# copy dlls to build directory
add_custom_command(TARGET ${PYMASHII} POST_BUILD
  COMMAND ${CMAKE_COMMAND} -E copy_if_different $<TARGET_RUNTIME_DLLS:${
    PYMASHII}> $<TARGET_FILE_DIR:${PYMASHII}>
  COMMAND_EXPAND_LISTS
```

```
26 )
27
28 # Dependencies
29 target_link_libraries(${PYMASHII} PRIVATE tamashii::core tamashii::
       core_bindings tamashii::vkrenderer tamashii::implementations)
30 add_dependencies(${PYMASHII} tamashii::core tamashii::core_bindings tamashii
       ::vkrenderer tamashii::implementations)
31
32 # Install
33 install(TARGETS ${PYMASHII} RUNTIME DESTINATION bin LIBRARY DESTINATION bin
       ARCHIVE DESTINATION lib)
```

Listing 7.1: Complete template for a CMake script to build custom bindings

Packaged bindings prototype with Nanobind

This prototype was used to test whether and how Nanobind can be incorporated into Tamashii's project structure. The following chapter shows the C++ and Python code that was created for this purpose. We do not include the rest of the project structure, which mostly exists to model the structure of Tamashii. Furthermore, we exclude the equivalent prototype built with Pybind11, which is nearly identical. The code of this method to creating bindings can be compared to the approach of embedding CPython, as shown in the next chapter.

```
1  // [...] includes
2
3  namespace nb = nanobind;
4  using namespace nb::literals;
5
6  std::optional<gui::DemoGui> demoGui;
7
8  NB_MODULE(pybinding, m) {
9    m.def("add", [](int a, int b) { return moduleA::addFunction(a, b); }, "a"_a
       , "b"_a);
10
11   m.def("guiCreate", []() {
12     if (demoGui) {
13       throw std::runtime_error("Cannot create multiple guis");
14     }
15
16     auto instance = gui::DemoGui::create();
17     if (!instance) {
18       throw std::runtime_error("Could not create gui");
19     }
20
21     demoGui.emplace(std::move(*instance));
22     }, R"pbdoc(
23     Create a simple gui in a native window.
24   )pbdoc");
25
26   m.def("guiBeginFrame", []() -> bool {
```

```
27      if (!demoGui) {
28          throw std::runtime_error("Missing gui instance");
29      }
30
31      return demoGui->frameBegin();
32      }, R"pbdoc(
33      Begin a gui frame.
34    )pbdoc");
35
36    m.def("guiEndFrame", []() {
37      if (!demoGui) {
38        throw std::runtime_error("Missing gui instance");
39      }
40
41      demoGui->frameEnd();
42      }, R"pbdoc(
43      End a gui frame.
44    )pbdoc");
45
46    m.def("guiAddButton", [](const char* name) -> bool {
47      if (!demoGui) {
48        throw std::runtime_error("Missing gui instance");
49      }
50
51      return demoGui->addButton(name);
52      }, R"pbdoc(
53      Add a labeled button to the gui.
54    )pbdoc");
55
56    m.def("guiAddText", [](const char* text) {
57      if (!demoGui) {
58        throw std::runtime_error("Missing gui instance");
59      }
60
61      demoGui->addText(text);
62      }, R"pbdoc(
63      Render text to the gui.
64    )pbdoc");
65
66  }
```

Listing 7.2: Module definition of the prototype that tests packaging with Nanobind.

```
1  import sys
2  import os
3
4  if __name__ == '__main__':
5    sys.path.append(os.path.abspath('../../_project/src/pybinding/Debug'))
6    import pybinding as pb
7
8    pb.guiCreate()
9
10   ctr= 0
11
```

```
12   while pb.guiBeginFrame():
13     if pb.guiAddButton('Click me!'):
14       ctr+= 1
15
16     pb.guiAddText(f'Clicks: {ctr}')
17
18     pb.guiEndFrame()
```

Listing 7.3: Python script of the prototype that tests packaging with Nanobind. It draws the GUI and draws a button.

Embedded CPython prototype with Pybind11

This prototype was used to test whether and how Pybind11 can be used to possibly embed the CPython interpreter into Tamashii. As this approach differs greatly to packaging Tamashii in its entirety as a Python module, the Python script features a different structure. Here, it only implements a callback that gets called by the main loop which is part of the C++ code. Again, as described before, we only show the parts of the prototype's code that is relevant for comparison. While this method of integrating Python code is popular among many applications, we ultimately decided to go the other route for reasons laid out in Chapter 4.1.

```
1  // [...] includes
2
3  namespace py = pybind11;
4
5  gui::DemoGui* guiPtr{ nullptr };
6
7  PYBIND11_EMBEDDED_MODULE(jintai, m) {
8    m.def("add", [](int i, int j) {
9      return moduleA::addFunction( i, j );
10   });
11
12   m.def("guiAddButton", [](const char* label) -> bool {
13     if (guiPtr) {
14       return guiPtr->addButton(label);
15     }
16     return false;
17   });
18
19   m.def("guiAddText", [](const char* txt) -> void {
20     if (guiPtr) {
21       guiPtr->addText(txt);
22     }
23   });
24 }
25
26
27 int main( int argc, const char** argv ) {
28   py::scoped_interpreter guard{};
29
```

```
30   try {
31     // A very dumb way to extend the sys.path to make the local gui script
       visible
32     std::string modulePath = argv[0];
33     auto pos= modulePath.find("_project\\src\\pybinding\\Debug\\pybinding.exe
       ");
34     assert(pos != std::string::npos);
35     modulePath.resize(pos);
36     modulePath += "src\\pybinding";
37
38     auto sysModule = py::module::import("sys");
39     sysModule.attr("path").attr("append").call( modulePath );
40
41     // Load the gui script
42     auto guiScript = py::module::import("gui_script");
43     auto drawFunction= guiScript.attr("drawGui");
44
45     // Create gui and set context pointer for python functions
46     auto demoGui= gui::DemoGui::create();
47     assert(demoGui, "Could not create gui");
48
49     guiPtr = &demoGui.value();
50
51     // Render loop
52     while (demoGui->frameBegin()) {
53       drawFunction();
54
55       demoGui->frameEnd();
56     }
57   }
58   catch (py::error_already_set& e) {
59     std::cerr << e.what() << std::endl;
60   }
61 }
```

Listing 7.4: Module definition of the prototype that tests embedding CPython with Pybind11. The main program loads the Python code, updates the GUI and executes a callback each frame.

```
1  import jintai as jt
2
3  ctr= 0
4
5  def drawGui():
6    global ctr
7    if jt.guiAddButton("Click me!"):
8      ctr+= 1
9
10   jt.guiAddText(f'Clicks: {ctr}'
```

Listing 7.5: Python script of the prototype that tests embedding CPython with Pybind11. It implements a callback that is called each frame to draw the GUI.

Training Nerfstudio on real photo data

(a) Real front view.

(c) Rendered front view.

(e) Real back view.

(g) Rendered back view.

Figure 1: Comparison of real photos used as training data and rendered output images created with Nerfstudio.