TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

D I P L O M A R B E I T

Pivy - Embedding a Dynamic Scripting Language into a Scene Graph Library

ausgeführt am Institut für
Interaktive Mediensysteme
der Technischen Universität Wien

unter Anleitung von

Univ.Prof. Dipl.-Ing. Dr.techn. Dieter Schmalstieg

und

Mag. Dr.techn. Hannes Kaufmann

als verantwortlich mitwirkendem Universitätsassistenten,

sowie unter Mitbetreuung von

Dipl.-Ing. Dr.techn. Gerhard Reitmayr,

durch

Tamer Fahmy

Urbarialgasse 1
A-2422 Pama

Matrikelnummer: 9526976

Datum

Unterschrift

Abstract

This thesis presents the design and implementation of *"Pivy"*: a *Python* language binding for the *Coin* scene graph library. Pivy allows for development of Coin applications in Python, interactive modification of Coin programs from within the Python interpreter at runtime and incorporation of Scripting Nodes – capable of executing Python code and callback functions – into the scene graph. Coin is a high-level 3D graphics toolkit for developing cross-platform real-time 3D visualization and visual simulation software; Coin's properties and features are e.g. suitable for application development in the Augmented Reality (AR) domain.

We differentiate between *extending* and *embedding* the Python interpreter. To create a Python extension, a C wrapper needs to be written and built as a shared library; Python then imports and makes use of this shared library at runtime. Different Python wrapping techniques and approaches – from manual wrapping to automatic wrapper generators such as SWIG – with a special focus upon large C++ libraries/frameworks applicable for Python are compared. The opposite direction is called *embedding*, where already existing Coin applications or libraries (written in C++) are given direct access to the Python interpreter.

Both use cases are showcased and their distinction explained through Python applications using Pivy and the special *SoPyScript* scene graph node which has been created to allow Python code to be embedded into a regular scene graph and executed during traversal of the same. The *SoPyScript* scene graph node is making use of both *extending* and *embedding* techniques; it is based upon ideas of the *VRML JavaScript* node and can be used from either Python or C++ applications.

Furthermore, the suitability and benefits of dynamically typed "scripting" languages over statically typed "system programming" languages such as C++ for *Rapid Application Development* (RAD) and *Rapid Application Prototyping* (RAP) are analyzed and demonstrated.

Keywords: 3D Graphics, Augmented Reality, Scripting Language, C++, Python, Coin, Open Inventor, Rapid Application Development, Rapid Application Prototyping

Kurzfassung

Die vorliegende Arbeit stellt das Design und die Implementierung von *Pivy* vor, einer *Python*-Anbindung für die Szenengraph-Bibliothek *Coin*.

Coin ist eine plattformunabhängige 3D-Graphik-Bibliothek, deren hoher Abstraktionsgrad die Entwicklung von Echtzeit-Visualisierungssoftware, z.B. im Bereich von Augmented Reality (AR), erleichtert. Pivy erlaubt die Entwicklung von Coin-Applikationen in Python und macht es möglich, Coin-Programme aus dem Python-Interpreter zur Laufzeit interaktiv zu modifizieren sowie Skript-Knoten, die Python-Code und Callback-Funktionen ausführen können, in den Szenengraph einzufügen.

Grundsätzlich sind zwei verschiedene Arten der Anbindung zu unterscheiden: die *Erweiterung ("extension")* des Python-Interpreters und seine direkte *Einbindung ("embedding")*. Eine Python-Erweiterung benötigt einen C-*Wrapper* in Form einer gemeinsam genutzten Bibliothek, die von Python zur Laufzeit geladen und verwendet wird. Verschiedene Techniken und Ansätze – von der manuellen Generierung bis zur automatischen Erstellung durch Generatoren wie *SWIG* – werden vorgestellt und verglichen. Das Hauptaugenmerk liegt dabei auf der Erweiterung von Python durch grosse C++-Bibliotheken. Umgekehrt ist es auch möglich, bestehenden Coin-Applikationen oder -Bibliotheken, die in C++ geschrieben sind, durch *Einbindung* des Python-Interpreters direkt Zugang zu diesem zu geben.

Beide Anwendungsfälle werden beschrieben und die Unterschiede zwischen ihnen erläutert. Dies geschieht anhand von Python-Applikationen, die Pivy verwenden, sowie insbesondere am Beispiel des *SoPyScript*-Knoten, der es erlaubt, Python-Code in einen regulären Szenengraph einzubetten und bei der Traversierung auszuführen. Der *SoPyScript*-Knoten verwendet beide Techniken (Erweiterung und Einbindung); er basiert auf der Idee und dem Design des *VRML JavaScript*-Knoten und kann wie dieser sowohl von Python- wie auch C++-Applikationen verwendet werden.

Des weiteren enthält die Arbeit auch eine Betrachtung der grundlegenden Einsatzmöglichkeit von dynamisch typisierten Skriptsprachen zur schnellen Entwicklung von Applikationen und/oder Prototypen – *Rapid Application Development (RAD), Rapid Application Prototyping (RAP)* – und den Vorteilen von Skriptsprachen gegenueber statisch typisierten "Systemprogrammierungssprachen" wie C++.

Acknowledgements

I am full of love, so here is whom I'd like to thank:

Mervat and **Mohamed**, *my parents*, for all their support and love in all those years and without whom I could have never achieved anything.

Dalia and **Rascha**, *my two sisters*, who always assisted me in difficult times and greatly enriched my life.

Prof. Dieter Schmalstieg, *my supervisor*, for giving me the opportunity to work as an undergraduate research assistant in his group at *Vienna University of Technology*; there I had the unique chance to learn the fundamentals of scientific work and research in the exciting field of Augmented Reality and was allowed to work on the creation of Pivy.

Gerhard Reitmayr, *guru and supervisor*, for his invaluable help, suggestions for improvements, many insightful discussions which greatly influenced the design of Pivy, memorable and very enjoyable coding sessions and in general for allowing me to watch and learn from a brilliant researcher at work.

The former **VRGroup** at *Vienna University of Technology*, namely **István Barakonyi, Hannes Kaufmann, Florian Ledermann, Joseph Newman, Thomas Pintaric, Thomas Psik, Daniel Wagner** for making our group an interesting and enjoyable workplace.

Systems in Motion for all the support over all those years, without Pivy would not have been what it is today; especially **Lars J. Aas**, **Peder Blekken**, **Morten Eriksen** and **Marius Kintel**.

Michael Kalkusch, who apart from being a marvelous friend over all those years, introduced me to computer graphics, making it accessible to me through his unique skill of explaining even the most complex topic in a very simple and graspable fashion (sometimes by abusing my cigarette packet and Zippo lighter to enhance his 3D explanations...).

Karin Kosina, for being such a lovely friend during all those years, making my life interesting and exciting again and who showed me that there seems to be vivid life and still sunlight on this planet outside the sphere of computer screens by opening the jalousie of my window (and forcing me to look out of my then monitor-lit room...).

Omar Medani, for dragging me with him through our study, making it so much more enjoyable and for sharing some of the most memorable and unforgettable moments in my life (from killing ourselves side by side in *hopeless* and *lost* soccer matches to taking part in *mission impossible* exams ;)) and being the splendid friend he was and is for so many years.

Andreas Pongratz, who helped me through troubled times in life, making math accessible to me (and showing me that there is a whole universe full of "exact definition" pitfalls hiding behind unsuspiciously simple equations such as $\mathbf{a} + \mathbf{b} = \mathbf{c}, \forall \mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{C}$) and for being such a wonderful, reliable, altruistic friend always willing to help (and never minding to plainly name things the way they are).

Very special thanks to **Oyvind Rideng** for his enduring patience, providing me with all the time and support I needed to finish this thesis and pushing me into the right direction when my motivation was wearing off.

I'd like to thank **Pål-Robert Engnæs**, who read my thesis while it was in progress and provided me with printouts containing valuable comments, feedback, suggestions for improvements and most importantly very constructive criticism.

Of course, I cannot leave this section without at least mentioning **Herbert** and **Gerhard Aguinaga**, **Maresa Dormann**, **Robert Fuchs**, **Elham Hedayati Rad**, **Karyn Laudisi**, **Barbara Langmüller**, **Herbert Schneeweiß**, **Terje Wiesener** and all of my other friends and supporters (who got not mentioned here, because of my supervisors impatiently waiting for me to finish this thesis… *shrug*).

Anyways, you know who you are!
Thanks so much to all of you!

Pivy - Embedding a Dynamic Scripting Language into a Scene Graph Library

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Different computer programming languages have been created for a variety of purposes and tasks. They range from the lowest level human-readable notation for a specific computer architecture: assembly languages; higher level languages following the procedural, functional and object oriented paradigm: C, C++, Java, Haskell, Lisp, OCaml; to so-called dynamic scripting programming languages: Lua, Python, Ruby, Scheme, Tcl.

1.1 System programming and dynamic scripting languages

The introduction of higher level languages – most notably C[1], which was developed in the early 1970s by Dennis Ritchie for use on the Unix[2, 3] operating system – has brought a fundamental change in how programmers write computer programs and allowed applications to be developed in much lesser time. Higher level languages are less efficient than assembly languages as the former generally need more machine instructions. Nevertheless, they allow programmers to achieve the same or greater functionality in lesser time by providing simple keywords such as *if* and *while* for control structures, by handling register allocation and generating procedure calling sequences automatically[4]. In a study of numerous languages Capers Jones observes that for a given task, assembly languages require about 3-6 times as many lines of code as system programming languages[5].

Certainly, the efficiency impact of higher level languages compared to assembly languages is not an issue and negligible on modern computing systems and architectures; one reason is that the design of modern computing architectures are heavily influenced by the attributes of higher level languages and vice versa higher level languages compilers generate highly optimized code through the provided specialized instructions and architecture specific mechanisms and extensions in charge[6].

So how does this relate to dynamic scripting languages and what makes them different?

To quote John K. Ousterhout – creator of Tcl/Tk[7] – in his paper *"Scripting: Higher Level Programming for the 21st Century"*[4]:

Modern computers are fundamentally typeless: any word in memory can hold *any kind of value, such as an integer, a floating-point number, a pointer, or an instruction. The meaning of a value is determined by how it is used: if the program counter points at a word of memory then it is treated as an instruction; if a word is referenced by an integer add instruction then it is treated as an integer; and so on. The same word can be used in different ways at different times.*

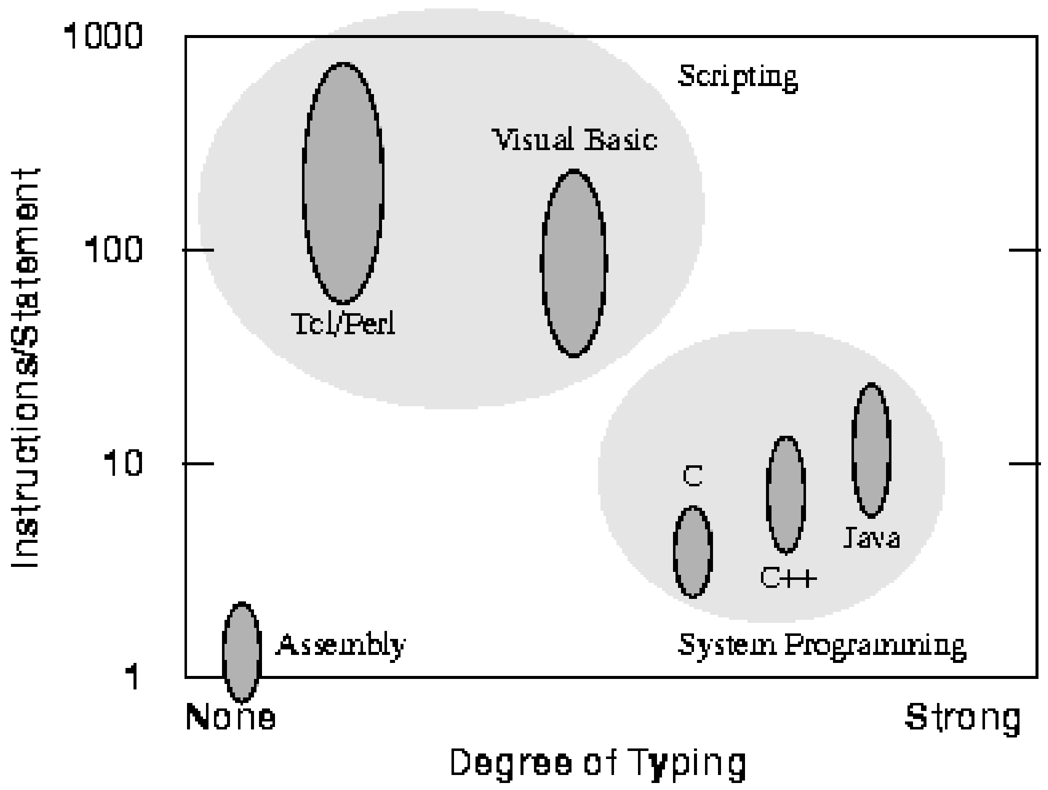


Figure 1.1: A comparison of various programming languages based on their level (higher level languages execute more machine instructions for each language statement) and their degree of typing. System programming languages like C tend to be strongly typed and medium level (5-10 instructions/statement). Scripting languages like Tcl tend to be weakly typed and very high level (100-1000 instructions/statement).

He claims that because of system programming languages are strongly typed and each variable must be declared with a particular type, it becomes difficult to impossible to develop new code quickly as data and codes is segregated. Furthermore, the required strict type checking mechanisms in place do not allow to make use of an object of one type where an object of a different type is expected. However, strong typing has the advantage, that compilers can use type information for error detection and improving performance, where compilers are able to generate specialized and optimized code.

He follows this dissection by stating that scripting languages are particularly suited to “glue together” components created in system programming languages; he argues that the key factor for this capability can be accredited to the general type-lessness of dynamic scripting languages and that there are no a priori restrictions on how components can be used. As any component or value can be used in any situation; components designed for one purpose can be used for totally different purposes never foreseen by the designer.

Figure 1.1[4] compares various programming languages and puts their degree of typing in relation to the instructions/statement needed for these. Python can be assigned to the same area as Tcl/Perl as it shares similar properties and an equally dynamically typed approach.

System programming languages such as C/C++ usually make use of a compiler that translates each statement in the source program into a sequence of binary instructions. Compilers also exist for dynamic languages and are usually provided in the form of *Just in Time compilers*[1]. They tend to not optimize code as well as the ones available for system programming languages.

The typeless nature of dynamic languages, especially with those that are dynamically typed, does not make it easily possible to deduce the involved and correct types (some dynamic languages allow to "hint" and specify allowed types to the compiler for optimization purposes, though). Therefore, they defer type decisions until runtime, where once the involved variables are encountered, their current type can then be determined. This requires (*Just in Time*) compilers to implement runtime type deduction and interference mechanisms in order to operate efficiently.

A good summarization on the different *types of scripting languages* can be found in the Wikipedia, The Free Encyclopedia, article on scripting languages[8]:

Application-specific languages *Many large application programs include an idiomatic scripting language tailored to the needs of the application user. Likewise, many computer game systems use a custom scripting language to express the programmed actions of non-player characters and the game environment. Languages of this sort are designed for a single application and, while they may superficially resemble a specific general-purpose language (e.g. QuakeC, modeled after C, Figure 1.2) they have custom features which distinguish them.*

Text processing languages *The processing of text-based records is one of the oldest uses of scripting languages. Many, such as Unix's awk and, later, Perl, were originally designed to aid system administrators in automating tasks that involved Unix text-based configuration and log files. Perl is a special case – originally intended as a report-generation language, it has grown into a full-fledged applications language in its own right. PHP was originally developed as a specialized language for creating dynamic web content, but is now used by some for general system administration tasks as well.*

Job control languages and shells *Another class of scripting languages has grown out of the automation of job control – starting and controlling the behavior of system programs. Many of these languages' interpreters double as command-line interfaces, such as the Unix shell or the MS-DOS COMMAND.COM. Others, such as AppleScript, add scripting capability to computing environments lacking a command-line interface.*

[1]compilers that translate code to machine code at runtime

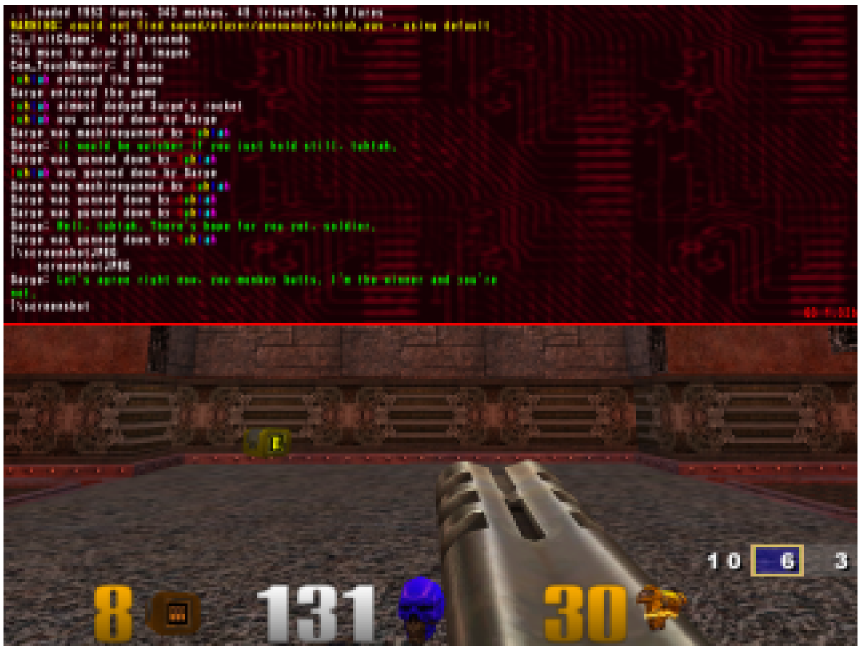


Figure 1.2: In-game Quake 3 Arena game console

General-purpose dynamic languages *Some languages, such as Perl, have begun as scripting languages but developed into programming languages suitable for broader purposes. Other similar languages – frequently interpreted, memory-managed, dynamic – have been described as "scripting languages" for these similarities, even if they are more commonly used for applications programming.*

Extension/embeddable languages *A small number of languages have been designed for the purpose of replacing application-specific scripting languages, by being embeddable in application programs. The application programmer (working in C or another systems language) includes "hooks" where the scripting language can control the application. These languages serve the same purpose as application-specific extension languages, but with the advantage of allowing some transfer of skills from application to application.*

1.1.1 Contribution

The thesis statement is as follows:

> *Dynamic scripting languages can provide efficient Rapid Application Development environments for existing low level 3D graphics systems. The application and integration of dynamic scripting languages in such systems combines the computational efficiency of a 3D graphics implementation in a system programming language and the development flexibility of dynamic scripting languages.*

Python, the programming language used and presented in this thesis, can be characterized as a general-purpose dynamic languages with a special suitability as an extension/embeddable language. The reasoning for this characterization will be presented in the next section of this thesis.

1.2 Deficiencies of system programming languages

Some of the general issues that accompany higher level statically typed languages such as C++ and which the author experienced during development, were:

Compile-time related issues such as simple typos in the code; to forget to provide corresponding definitions and declarations in the header file; correct application of pointer arithmetic; to overlook a const declaration in the API documentation or header file; to forcibly use explicit casts to help the compiler – which has to conduct strict type safety checks – decide which data type, structure or class is in command at a given context and time, etc.

So-called Integrated Development Environments (IDE's) such as XEmacs' *c++-mode* tend to help and remedy some of the problems above and therefore ease the development cycle. However, neither are they able to solve all of the aforementioned problems, nor do they solve the fundamental problem with system programming languages: system programming languages are statically and strongly typed[4, Scripting languages], which is one of the reasons why the development time needed to design, write and test a program in those languages is very high[9].

Link-time related issues such as unresolved symbols by missing libraries; to have to provide and determine the correct and right libraries and dependencies to link with; platform specific linker issues and idiosyncrasies: among those ABI (Application Binary Interface) issues due to the way of how C++ mechanisms such as name mangling are implemented on different systems and compilers; to have to deal with intermixing different libraries accidentally compiled with different compilers and last but not least to have to provide the correct magical linker flags (a not well documented science by itself).

Run-time related issues such as not being able to introspect and manipulate the current scene graph that help to immediately see and gain experience from the effects of the applied modifications.

Obviously, to use a dynamic general purpose language such as Python – which is interpreted – solves the compile- and link-time related problems. Furthermore, to utilize the provided introspection features of the Python interpreter and Open Inventor/Coin library makes it possible to inspect and manipulate the scene graph within the Python interpreter at runtime. This is very helpful and beneficial for the developer's learning experience and allows to solely focus, place the emphasis and spent effort on the Open Inventor/Coin related tasks.

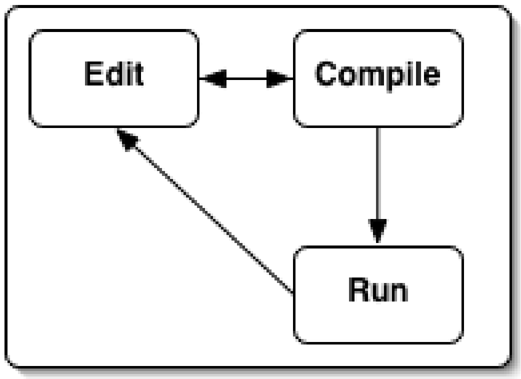


Figure 1.3: Development cycle with compiled languages

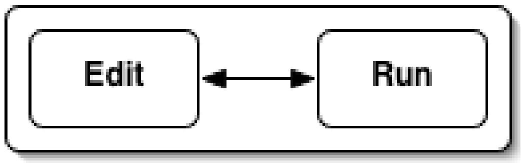


Figure 1.4: Development cycle with interpreted languages

1.2.1 RAP/RAD and interpreted environments for interactive 3D applications

The main intention to create Pivy is to provide a tool that allows Open Inventor/Coin newcomers to quickly learn, grasp and explore Coin and for experienced programmers to use it for *Rapid Application Development (RAD)* and *Prototyping (RAP)*. The write-compile/link-run cycle (Figure 1.2) needs to be replaced with a much faster write-run cycle (Figure 1.4) and at the same time allows to develop in a language with a much easier to use syntax, which has less to no declaration and memory handling overhead.

The developer is now able to increase productivity and turn around cycle; he is able to skip the compile/link step, where he solely needs to focus on the actual development process of his application. The tradeoff of this flexibility is very often decreased execution speed and higher memory consumption of the application. This is due to the interpretative nature of such solutions and because of many decisions, which compilers optimize away at compile time, have to be deferred until runtime.

1.2.2 The benefits of a Python binding

The scripting language targeted and chosen and treated in this thesis is Python, a popular dynamic programming language. Alternatives such as Ruby, Tcl/Tk, Perl or Lisp (dialects) lack popularity, versatility and flexibility in the way they allow to be extended or embedded or are syntactically very different to C++, which makes it hard to achieve the goals of:

1. transforming existing programs and applications easily forth and back between the source (C++) and target language (in our case Python)
2. reusing the existing documentation available for the C++ library.

Some of Python's main benefits are:

Extensive Standard Library Python comes bundled with a large and extensive collection of modules commonly referred to as the *standard library*. The Standard library provides a wide range of easy to use services and functionality, which ranges from various tools and modules useful for Operating System interaction, Internet protocols, tools for Web service applications, support for XML parsing and generation, string and regular expression processing to modules useful for the construction of Graphical User Interfaces.

Portability and Extensibility Python is available for a variety of platforms, which ranges from embedded systems to special-purpose processors. Python code runs unmodified, wherever the Python interpreter is available.

Furthermore, Python exposes its well designed C API which makes it easy to extend the interpreter or embed the interpreter into existing applications.

Elegant, Simple and Well Designed Syntax Python possesses a very elegant, clear and simple syntax designed with readability in mind. This fact helps to make existing code easy to understand, design, modify and maintain.

Easy-to-use High-level Data Structures Python features very powerful built-in data structures such as lists and tuples, dictionaries and sets. They allow the Python programmer to use them as building blocks for other more complex data structures.

Dynamic Typing and Binding Python uses dynamical typing and binding, which frees the programmer from the necessity to declare the type of every variable and makes the code less verbose and error-prone. Python checks and allows to inquire the type of every variable at runtime which makes it actually a strongly typed language.

Introspection Python allows to introspect and access any type, class, method or even the inline documentation at runtime which comes handy during experiments and is commonly used by an IDE to display the documentation of modules, classes or methods.

The Open Inventor/Coin design incorporates a runtime type system in order to gain introspection facilities and the required flexibility to hande the basic types. In fact, the design of Open Inventor/Coin relies and heavily makes use of the runtime type system which makes it overall dynamic in nature.

However, a dynamically typed language such as Python provides a more natural and fitting interface. Consequently, to have a Python binding available, which interfaces with Open Inventor/Coin, preserves the performance advantage of C++ and simultaneously allows to greatly improve development time and speed.

1.3 Development history

The development of Pivy started out as a project conducted as part of the authors computer science studies at Vienna University of Technology in 2002. The main intention to start the creation of Pivy back then was to allow scripting in Python from within Studierstube[10] applications through the addition of a scripting node facility with Pivy as its foundation.

Pivy was presented in a talk at the PyCon 2004 (a main Python community conference) in Washington D.C.

The first official stable Pivy release (version 0.3.0) in August 2005 marked the end of Pivy's existence as a development tool for early adopters and opened the doors for an interested wider public audience.

Since then, Pivy got used and incorporated in a variety of different applications and projects. Those make either use of the scripting node facility, in order to manipulate the scene graph through Python in an embedded fashion, or rather even write complete new Coin based applications from scratch.

Chapter 2

Related work

2.1 Existing 3D graphics APIs

Currently OpenGL is the primary choice for cross platform 3D graphics application development. OpenGL provides so-called immediate mode access to the frame buffer where the application itself has to maintain the data that describes the model.

OpenGL, designed as a low-level API, therefore provides no out of the box facilities for user interaction such as to move objects to a different location or to select them for further manipulations. Additional complicated code needs to be implemented by the programmer to fulfill these tasks.

2.1.1 Immediate vs. retained mode APIs

Higher-level libraries such as Open Inventor, Coin or Performer built on top of OpenGL have been developed to facilitate and speed up the development process. They allow the creation of complex 3D graphics applications.

Unlike OpenGL these libraries focus on the creation of 3D objects. Object information such as shape, size, location in 3D space, is stored in a scene database. In contrast to OpenGL they provide the necessary functionality to interact with objects and to change the objects in the scene.

Those libraries are referred to as operating in retained mode where all the data that describe a model needs to be specified in advance through the usage of predefined data structures. They internally organize the data in a hierarchical database, as can be seen in Figure 2.1.

Application- and data-driven scene graph APIs

Another important distinction is made in this context between application- and data-driven scene graph APIs. Data-driven toolkits only re-render when something changes in the scene, for example if the user changes the viewpoint of the scene. Application-driven toolkits re-

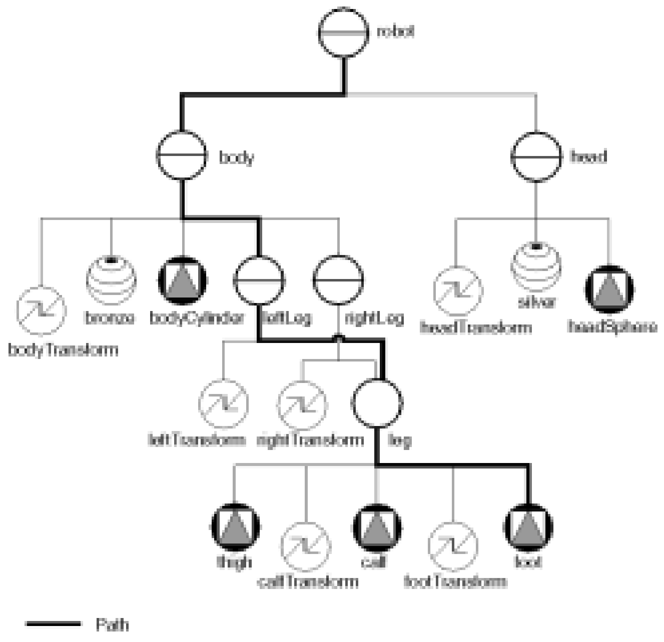


Figure 2.1: Open Inventor scene graph layout diagram from the Inventor Mentor robot example where the thick line represents the so-called path to a node

render the scene continuously in an application loop, which uses up all CPU resources available. The latter case is used for games and simulation software such as flight simulators where high and constant frame rates are desirable. In general a data-driven approach fits better for a general purpose 3D API where constant frame rate is not the main concern. More importantly resources should be available for other computational tasks. Typical examples that benefit from this approach are applications that visualize results of numerical simulations or 3D editors (level editor for games). Examples of application-driven APIs are Performer or OpenSG, whereas Open Inventor or Coin offer a data-driven API.

Performance considerations

Performance is a key problem, hence these libraries are usually implemented in a compiled language such as C++. However, the use of C++, a statically typed language with a heavy and complicated syntax, tends to be error-prone and cumbersome. A dynamically typed and bound language with an intuitive syntax like Python provides a more natural interface.

Examples for the first category of libraries that operate in immediate mode are *OpenGL, DirectX, Panda3D, Ogre3D*, whereas *Open Inventor, Coin, Performer, OpenSG, OpenScenegraph, Java3D* offer retained mode APIs.

For Python bindings this differentiation and classification is crucial due to the overall calling overhead on the Python side. Dr. Roman Geus in his talk *Python Wrapper Tools: A Performance Study*[11] at the EuroPython2004 finds that the interpretative and dynamic nature of Python induces a natural performance bottleneck. For example, a function call in Python takes ∼90 times longer than in C/C++. Furthermore, through the necessity to create wrappers for the type checking and conversion around the actual C/C++ functions and methods the calling overhead is increased again with various penalties due to the different approaches and tradeoffs chosen for the available wrapper generators such as SWIG, SIP, Boost.Python.

Suitability of retained mode libraries for dynamic languages

This makes retained mode libraries a perfect fit for interpreted and dynamic languages such as Python. When one has to deal with a retained mode scene graph library such as Coin, the calling overhead, which occurs during the creation of the scene graph in Python, is not really the most important issue, as the construction of such happens usually once at startup time and scene graph changes tend to be only a few instructions later on. Once the scene graph is constructed and the application reaches the main loop, all further scene graph handling is conducted on the C++ side.

The common use case with retained mode libraries is to use the provided facilities for serialization in order to load a scene graph description from a file through a couple of calls. Therefore, to optimize the calling overhead does not really gain much apart from an improvement in the startup time of the application.

When it comes to callback handlers, the execution of Python code, it is indeed beneficial to have those evaluated as fast as possible. Still, the typical use case in retained mode libraries is to change a couple of the involved data structures such as the fields of a scene graph node. This allows one of Python's most significant properties and main purposes, namely as a controlling language, to be used. One of the reasons is that the relevant CPU bound calculations and operations, such as action scene graph traversals or matrix calculations, are always performed on the C++ side. Therefore, the calling overhead, implicitly caused by dictionary lookups through variable references by the interpreter and through type conversions/checks induced by the wrapper, gets negligible as those CPU bound operations take significantly longer.

Suitability of immediate mode libraries for dynamic languages

If in comparison we look at immediate mode libraries, we find that Python code has to be constantly and repeatedly evaluated and invoked in a tight main loop of the application, where most of the execution time is then spent in the Python interpreter. We can easily deduct how this has a negative effect on the performance and how the calling overhead quickly becomes a significant bottleneck[1]

Another observation is that scalability is in favor towards retained mode libraries. It does not make a difference on the Python side if the application has to deal with 100 or 10000 nodes in the scene graph. On the other hand, it can be observed that overall application performance with libraries, that operate in immediate mode, linearly decreases with increasing lines of code. The reason is that significantly more time has to be spent on the Python side, where the calling overhead and available execution speed become severe limiting factors and restrictions.

[1]This can be easily seen, when the frame rates of typical PyOpenGL applications are compared to natively written C OpenGL applications.

2.2 Open Inventor/Coin - a high-level 3D graphics toolkit

Open Inventor was originally developed by Silicon Graphics, Inc. as the IRIS Inventor library. It has long since become the de facto standard graphics library for 3D visualization and visual simulation software in the scientific and engineering community. It has proved its value over a period of more than 10 years, its maturity, which contributed to its success as a major building block in thousands of large-scale engineering applications around the world.

Open Inventor is an object-oriented 3D toolkit that offers a comprehensive solution to interactive graphics programming problems. It presents a programming model based on a 3D scene database that dramatically simplifies graphics programming. It includes a rich set of objects such as polygons, text, materials, cameras, lights, track balls and handle boxes.

Open Inventor also defines a standard 3D file format (ASCII and binary) for scene data interchange. This allows the construction of scene graphs in ASCII files without the need to program. Those ASCII files can then be viewed through the provided viewers from Open Inventor or any common modeling tool.

The significant characteristics of Open Inventor:

- built on top of OpenGL
- defines a standard file format for 3D data interchange
- introduces a simple event model for 3D interaction
- provides portable animation objects called Engines
- is window system and platform independent
- is a cross platform 3D graphics development system
- encourages programmers to create new customized objects

Pivy is bound against Coin, which implements the SGI Open Inventor 2.1 API. Coin is portable over a wide range of platforms (any UNIX/Linux/*BSD platform, all Microsoft Windows operating systems, and Mac OS X) and adds additional features missing in the original SGI Open Inventor API such as VRML97 support, 3D Sound, 3D Textures, multi-threading and parallel rendering. Additionally GUI bindings implementing viewer widgets for several GUI toolkits (Qt, GTK+, Xt, Cocoa, Win32) are available. Coin is Open Source and has an active and growing community.

2.2.1 Overview of the Open Inventor/Coin API

As mentioned in the previous section the Open Inventor/Coin programming model is based on a 3D scene database (**SoDB**), which allows to define so-called *"scene graphs"* in order to provide the information that represents one or more 3D scenes. A *scene graph* itself contains one or more nodes of different types and varying functionality. Predefined nodes exist for grouping, that specify the property (to manipulate the appearance and the qualitative characteristics of a scene) or geometry (to define the shape) of objects. A hierarchical scene graph is constructed through the addition of nodes to group nodes such as **SoSeparator** or **SoGroup**; those hierarchies are *directed acyclic graphs*. Additionally so-called **engines** are provided to allow the animation of parts of the scene and in order to be able to constrain one part of a scene in relation to some other part of the scene.

Fields in Open Inventor/Coin

Each node contains a set of so-called *fields* that allow to manipulate the parameters of and state information in a node. Fields themselves are again containers for predefined basic types such as **SbVec3f**, which represent a 3 dimensional vector, or regular C/C++ data types such as **float**; they provide a consistent interface to set and access those contained basic types, allow Open Inventor/Coin to detect changes to the database and let fields to be connected to fields contained in another node. Two types of fields exist: (a) single-value fields such as **SoSFVec3f**, which contain exactly one basic type, and (b) multi-value fields such as **SoMFVec3f**, which contain arrays of the basic types and provide the necessary functionality to specify for example coordinate points or normal vectors. The contained basic types can be manipulated through the use of the **setValue()** and **getValue()** methods found in the C++ API regarding *fields*; correspondingly **setValues()** or **getValues()** are used to manipulate and access multi-value fields.

Scene graph traversals through actions

The scene graph is traversed by so-called *actions* from left to right in depth-first search order. The *actions* are implemented through the use of the *visitor pattern*, where each node implements its own action handler, such as **SoNodeName::GLRender** for a **SoGLRenderAction**, which gets invoked when the action reaches the node during a traversal. For example shape nodes, such as **SoCone**, implement and issue OpenGL calls to draw their geometry, whereas property nodes such as **SoMaterial** manipulate the collected traversal state in an **action** and set its corresponding elements such as the *emissiveColor* to the requested values.

Therefore, the order of the nodes in the scene graph is important. The upper graph of Figure 2.2 shows a basic scene graph of four nodes. The traversal starts at the **SoSeparator** node, then traverses the **SoTransform** node, which manipulates the position of the objects in the scene. The next encountered node is the **SoMaterial** property node, which for example sets the drawing color **diffuseColor** for consecutive nodes to the color blue. Finally, the **SoCone** shape node gets traversed. The output is a blue cone positioned in the 3D scene according to the values specified in the **SoTransform** node.

If we now flip the positions of the **SoTransform** and **SoCone** nodes, as shown in the lower graph of Figure 2.2, the result is different behavior. The cone is rendered in the default color of the **SoCone** as the SoMaterial node has not been traversed yet and cannot influence its appearance. Furthermore, the position of the cone is not influenced by the **SoTransform** node as the **SoTransform** node appears to the right of the **SoCone** node in the lower scene graph.

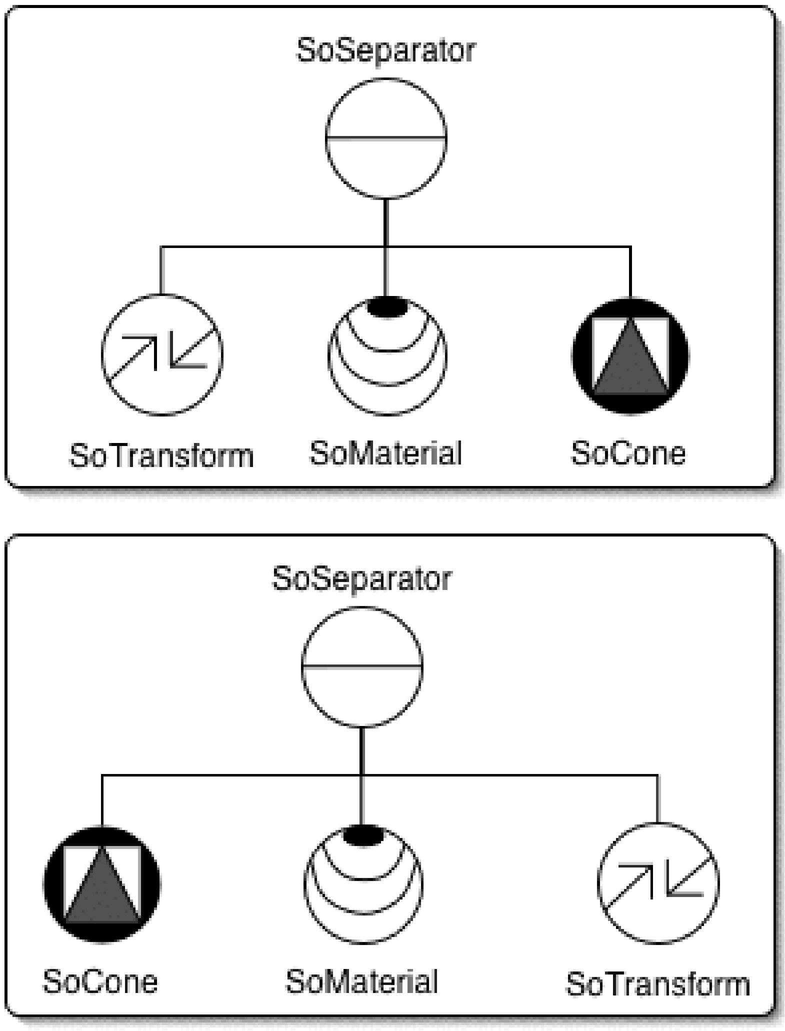


Figure 2.2: Flipping the position of the SoTransform and SoCone nodes

Subgraphs and paths in a scene graph

Another important notion is the one of a *path.* A *path* contains references to a chain of nodes, each of which is a child of the previous node, that represents a subgraph of a scene graph. They allow to isolate particular objects in the scene graph and can be either constructed by the developer or are returned by **actions**, such as a **SoSearchAction** or **SoRayPickAction**. Further actions can not only be applied on nodes but also on *paths* and allow for example to calculate the bounding box of an object in a 3D scene. Figure 2.3 shows the path to the **SoCone** node in our sample scene graph denoted as a thick line.

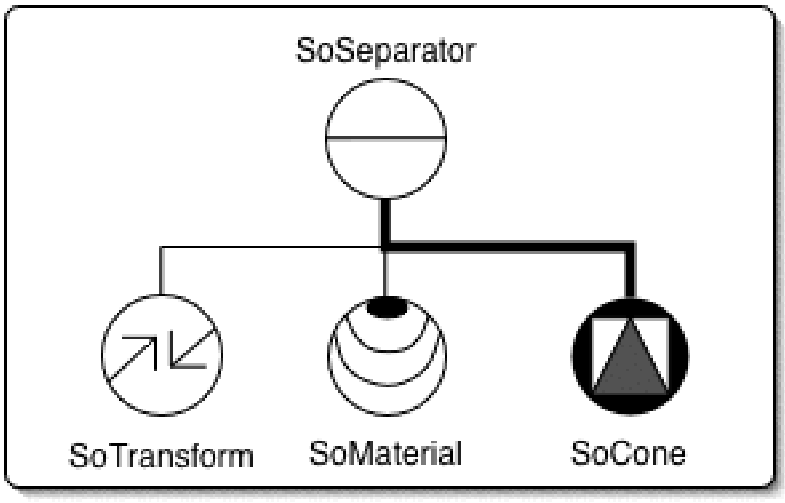


Figure 2.3: The path to a node in a scene graph

The Open Inventor/Coin file format

Open Inventor/Coin defines its own standard 3D (ASCII and binary) file format, that allows to serialize/store the scene graph contained in the scene database to for example a file or a memory buffer and vice versa allows a scene graph to be loaded from one. Listing 2.1 shows the Inventor file of our sample scene graph:

```
#Inventor V2.1 ascii

Separator {
  Transform {}
  Material {}
  Cone {}
}
```

Listing 2.1: The sample scene graph as a standard Inventor file

Note that for the built-in nodes the *"So"* prefix can be left out in the Inventor file. The provision of an own standard file format has an additional benefit: it allows for scripting. Applications can be rapidly developed through the construction of the scene graph solely out of Inventor files and the use of its elaborate syntax, which is fairly well described in chapter 11 of the *"The Inventor Mentor"*[12].

Reference counting in Open Inventor/Coin

Open Inventor/Coin uses *reference counting* for the memory management of the nodes, where each node keeps track of the number of references made to it. Once a node is added to a grouping node or referenced through a *path* its reference count gets incremented. Once nodes are removed from a grouping node or a *path* its reference count will be decreased and if the reference count drops to 0 the destructor of this node invoked. Reference counts can be manually increased through the **ref()** and decreased through the **unref()** method. Special care has to be taken when an *action* gets applied on a node as the action creates a *path* which references the node and increments its reference count. Once the action finished the traversal and the *path* is removed the reference count decreases, which is a problem for nodes that had originally a reference count of 0 when the *action* got applied as they will now be deallocated. Therefore, the reference counts of nodes, that an *action* will create a *path* on, should always be at least 1 in order to avoid memory errors. Open Inventor/Coin further provides a method for special cases to decrease the reference count, which does not trigger the nodes destructor: **unrefNoDelete()**.

Table 2.1 lists all reference and deletion cases as found in the according table in the *"The Inventor Mentor"*.

Increments Reference Count by 1	**Decrements Reference Count by 1**
Adding a node as a child of another node increments child's reference count	Removing a node as a child of another node
Adding a node to a path	Removing a node from a path
Applying an action to a node or path increments reference count of all nodes that are traversed	When traversal for the action finishes, all nodes that were traversed are unreferenced
Adding a node to an SoNodeList node	Removing a node from an SoNodeList
Setting an SoSFNode or SoMFNode value to point to a node	Changing an SoSFNode or SoMFNode value to point to a different node or to NULL, or deleting the value
Connecting an output of an engine to a field in a node or engine increments the engine's reference count	Disconnecting an engine's output from the field decrements the engine's reference count

Table 2.1: Reference counting rules

Directly referring to nodes in a scene graph

Finally, Open Inventor/Coin allows to specify names for nodes through the **setName()** method in order to retrieve the desired nodes by their name through the corresponding **getName()** method later on; it also provides an extensive runtime type system that can be used for example to gather the type of a node (**SoType::getTypeId()**). Furthermore, the Open Inventor/Coin API allows to query the parent class (**SoType::getParent()**, check if it is inherited from a type (**isOfType** or **SoType::derivedFrom()**) or to create a new instance of a known type (**SoType::createInstance()**). This runtime type system is a very powerful (introspection) mechanism, which we will use to our advantage for the creation of our binding.

2.2.2 Scripting in Open Inventor/Coin

As mentioned before, Open Inventor/Coin provides its own extensible text-based file format. The combined usage of engines, fields and its domain-specific syntax can be used for scene graph scripting and rapid construction of new scenes.

The Open Inventor syntax allows to describe *what* the scene graph should look like. This can be achieved by specifying the **nodes** and relationships of their **fields**, where the values of **fields** can again be constrained by **engines** with regard to the contents of one or more other fields in the scene graph. Additionally, **engines** can be used to animate parts of the scene graph.

Listing 2.2 demonstrates, how the combination of **engines** and **field** connections can yield interesting results. The purpose of this script is to load a model from an external file source (line 21-23) and rotate this model around the 3 main axis: X, Y, Z. The axis, which the model is rotated around, is changed in a specified frequency over time (line 8). The current axis is displayed in its vector form on the screen as a debugging aid, where for example $(1, 0, 0)$ represents the X axis (line 17-19).

```
 1  #Inventor V2.1 ascii
 2
 3  Separator {
 4     Transform {
 5        rotation = ComposeRotation{
 6           axis = DEF calc Calculator {
 7              a = TimeCounter {
 8                 min 0 max 2 step 1 frequency 0.1
 9              }.output
10              expression ”oA = (a==0) ? vec3f(1, 0, 0) :
11                 ((a==1) ? vec3f(0, 1, 0) : vec3f(0, 0, 1))”
12           }.oA
13           angle = ElapsedTime {}.timeOut
14        }.rotation
15     }
16
17     # debug output
18     Font { name ”Arial” size 20 }
19     SoText2 { string = USE calc.oA }
20
21     File {
22        name ”dolphins.wrl.gz”
23     }
24  }
```

Listing 2.2: Rotating dolphins script

The interesting part of the script[2] is the usage of the engines in the **SoTransform** node (line 4-15). There we manipulate the *rotation* field of this transformation node in order to set the desired axis. To set the frequency in which the axis changes should trigger and in order to set the current axis, we make use of the **SoTimeCounter** engine, which allows us to stepwise update the *output* field. The **SoTimeCounter** *output* field is again connected to another engine: a **SoCalculator**.

This engine offers the functionality of a general purpose calculator for *float values* and *3D float vectors*. It allows to evaluate simple expressions, has a variety of input and output fields. It features a single control flow statement in the form of a ternary operator (**expression ? true : false;**), which we make use of to determine which axis should be returned. Furthermore, we make use of the provided **vec3f()** function in order to convert our result into a suitable 3-dimensional vector.

We connect the output field *oA* of the **SoCalculator** engine to the *axis* field of the **SoComposeRotation** object. The **SoComposeRotation** serves the purpose of an adaptor that creates the necessary *rotation* field, that can be connected to the *rotation* field of the **SoTransform** node. The *angle* field of the **SoComposeRotation** is connected to the ElapsedTime engine's *timeOut* field, which is a controllable time source engine and which will update the **angle** field so that the model will be actually able to spin around its axis.

Figure 2.4 shows various states in the dolphin animation; the axis vector text string in the middle of the window indicates the axis the dolphins rotate around.

This style of programming is commonly referred to as *declarative programming*, where the main purpose of this paradigm is to describe *what* and not *how* something should be created. However, domain-specific declarative languages, such as HTML, SQL, regular expressions or in our case the Open Inventor/Coin file format, are not *Turing-complete*. In order to be able to make use of control structures, implement algorithms or write programs that have access to the object oriented Open Inventor/Coin API, we need to make use of imperative programming languages such as C, C++, Python or functional programming languages such as Scheme that again are used in an imperative procedural fashion[13][14]. Through the combination of both models, which is referred to as *embedding*, we are able to unite the benefits of both paradigms and create a very effective mechanism for a variety of previously not easily solvable tasks.

We will discuss embedding of an imperative language and its implications in chapter 5.

[2] which can be loaded by Open Inventor/Coin programs as we will see later

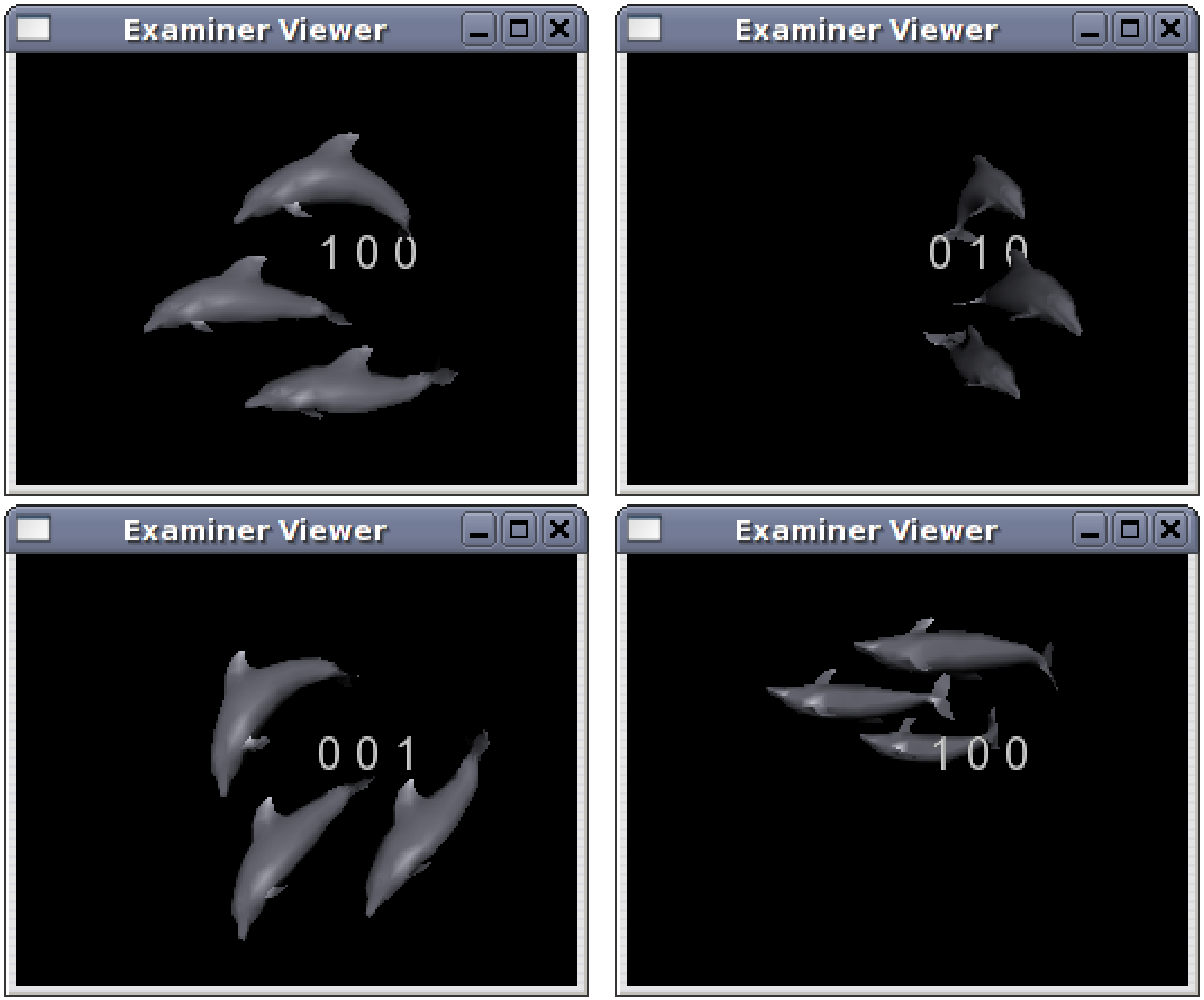


Figure 2.4: Examiner viewer executing the dolphin script

2.3 Python - a dynamic general purpose programming language

Guido van Rossum has begun the development of Python in 1990 at Stichting Mathematisch Centrum (CWI) in the Netherlands as a successor of the ABC programming language. Python – designed to be a highly readable language – is a dynamically typed interpreted programming language available on many different platforms and operating systems, that gained a lot of popularity in the recent years.

Alongside conventional integer and floating point arithmetic, it transparently supports arbitrarily large integers and complex numbers. Sequential types such as lists, sets, strings, tuples and dictionaries[3] serve as the data structure foundation. The combined usage of the data structures allow to express otherwise complicated problems in a few lines of code.

In Python everything is an object and Python has comprehensive support for object orientation: from polymorphism, meta classes to multiple inheritance and mix-ins. However, different to other programming languages information hiding (e.g. through mechanisms such as protected or private) is done by convention and not enforced. Furthermore, a mechanism for exception handling is provided, which is extensively used throughout the language and library modules to test for error conditions and other exceptional events in a program.

The following 6 major reasons can be accredited for Python's continued and ever increasing popularity:

1. Python is developed as an open source project with a big and friendly community and uses a completely unrestricted and truly free license (different from the GPL). This allows to embed the Python interpreter for any purpose (even for commercial use).
2. Python combines remarkable power and scalability with a very simple, elegant easy and quickly to learn syntax; Python provides facilities for runtime introspection at the same time.
3. Python's rich collection of standard modules (from ready to use modules to create web servers to Inter Process Communication modules.
4. Python bindings are available for nearly every popular library or framework that can be used immediately with a much steeper learning curve of the same API (written in another language)
5. Python is a multi-paradigm language; it permits several styles of coding paradigms or techniques, rather than to enforce one particular. Object orientation, structured programming, functional programming, aspect-oriented programming or design by contract are all supported.
6. Python's well designed and easy to understand straight forward C API allows to embed and extend Python or the application in any way one may desire.

[3]a mapping data structure, that makes use of a hash function

2.3.1 Overview of the Python C API

Most of the time it is sufficient to solely use functionality provided by the standard Python language as Python is a general purpose programming language, which comes with a broad range of useful modules in its standard distribution. However, if new functionality is required, that does not perform sufficiently through solely using the Python interpreter, then Python allows for the creation of new modules through its extension API and library. The same extension API allows to interface to existing libraries written in another language or integrate Python in mixed-language systems.

There are different implementations of the Python interpreter, that target different purposes and platforms that are in use today:

The first and most widely used Python interpreter is implemented in the *C* programming language, which is commonly referred to as *CPython*[4]. Other implementations of the Python interpreter exist for *Java*, *.NET* and Python[5].

We will discuss the Python C API through different implementations that solve the *Greatest Common Divisor* and compare the different approaches regarding their implications.

The mathematical definition for the *Greatest Common Divisor* reads as follows:

Definition 1 *Let $a, b \in \mathbb{Z}$. If $a \neq 0$ or $b \neq 0$, we define $gcd(a, b)$ to be the largest integer d such that $d|a$ and $d|b$. We define $gcd(0, 0) = 0$.*

To compute $gcd(a, b)$ we need an appropriate algorithm: *The Euclidean Algorithm.*

We will now quickly present 2 Lemmas for the *The Euclidean Algorithm.*

Lemma 1 *If $a > 0$, then $gcd(a, 0) = a$.*

Lemma 2 *Let $a > b > 0$. If $a = bq + r$, then $gcd(a, b) = gcd(b, r)$.*

Greatest Common Divisor in Python

An elegant Python implementation of the Euclidean algorithm is shown in Listing 2.3 and is saved in a file named *gcd.py*:

```python
def gcd(a, b):
    while b:
        a, b = b, a % b
    return a
```

Listing 2.3: gcd.py - Python implementation of the gcd() function

[4]and which is the one we use and focus upon in this thesis

[5]a Python implementation written in Python itself

We can invoke the python interpreter from the command line as follows in order to compute a result for *gcd*(15, 3) (line 2 shows the result):

```
$ python -c "from gcd import gcd; print gcd(15,3)"
=> 3
```

Listing 2.4: Invocation of the gcd.py module from the command line

Greatest Common Divisor as a Python extension

We will now have a look in how to create the same functionality exposed as a C module through usage of the Python C API.

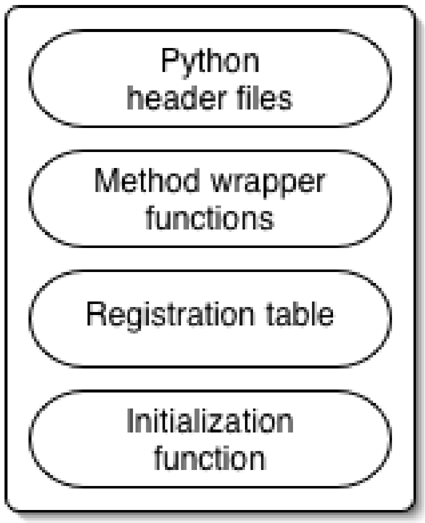


Figure 2.5: Python extension module structure common to all C modules

First, Listing 2.5 presents the C implementation of the same *gcd()* function:

```
#include <stdio.h>

int
gcd(int a, int b)
{
  int tmp = b;
  while (b) {
    tmp = b;
    b = a % b;
    a = tmp;
  }
  return tmp;
}
```

```

int
main(int argc, char * argv[])
{
  printf("%d\n", gcd(15, 3));
  return 0;
}
```

Listing 2.5: gcd.c - C implementation of the of the gcd() function

We need to extend the current code in order to create a manual so-called wrapper around the *gcd()* function so that it can be invoked from Python.

```
#include <Python.h>

extern int gcd(int a, int b);

static PyObject *
gcd_wrapper(PyObject * self, PyObject * args)
{
  int a, b, result;

  /* get Python arguments */
  if (!PyArg_ParseTuple(args, "ii", &a, &b)) {
    return NULL;
  }

  /* call the gcd() C function */
  result = gcd(a, b);

  return Py_BuildValue("i", result);
}


static struct PyMethodDef gcdmodule_methods[] = {
  {"gcd", gcd_wrapper, METH_VARARGS},
  {NULL, NULL}
};


void
initgcdmodule() {
  Py_InitModule("gcdmodule", gcdmodule_methods);
}
```

Listing 2.6: gcd_module.c - Python C module implementation of the gcd() function

The basic structure of every extension module is outlined in Figure 2.5.

We include the *Python.h* header file in line 1, which gives access to all Python definitions and declarations of the Python C API.

```
1 #include <Python.h>
```

Line 3 provides the extern declaration for the gcd() function which is contained in the gcd.c file.

```
3 extern int gcd(int a, int b);
```

From line 5 our actual Python wrapper for the gcd() function starts. The wrapper function, where one wrapper function has to be declared and implemented for each single C function that one desires to expose.

The wrapper function itself requires either 2 arguments (**self** and **args**) or 3 arguments (**self** and **args** and **kwargs**). All the involved types are of the generic PyObject type which is the base type of the Python C API. This function should further return a PyObject. To signal an error (e.g. when a type conversion fails because of wrongly passed arguments) it has to return NULL. If it does not return a value and no error has occured a special type **Py_None** has to be returned instead.

```
5 static PyObject *
6 gcd_wrapper(PyObject * self, PyObject * args)
```

Tasks of a wrapper

A wrapper has to solve and deal with the following 3 tasks (Figure 2.6):

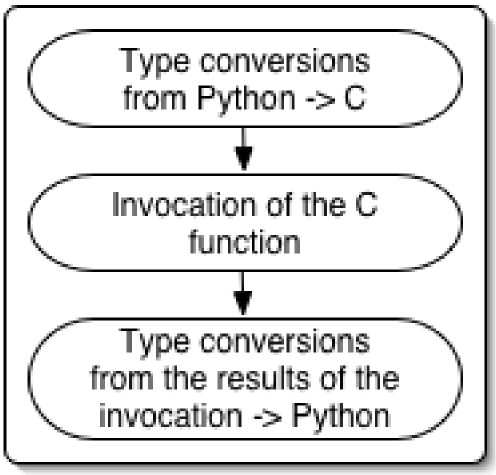

Type conversions from Python -> C
Invocation of the C function
Type conversions from the results of the invocation -> Python

Figure 2.6: Wrapper structure and tasks

1. The wrapper has to translate/convert the Python types to corresponding C ones. The Python C API provides convenience functions for this common data conversion operation: **PyArg_ParseTuple()** or **PyArg_ParseTupleAndKeywords()**. Those functions take the passed argument tuple (usually **args**) as their first parameter and provide a variable-length argument facility in the form of a format string, that contains conversion specifications and specifies how subsequent arguments should be converted. The results from such conversions, if any, are stored in the locations pointed to by the pointer arguments that follow the format. Each pointer argument must be of a type that is appropriate for the value returned by the corresponding conversion specification.

```
/* get Python arguments */
if (!PyArg_ParseTuple(args, "ii", &a, &b)) {
  return NULL;
}
```

2. The wrapper has to invoke the actual function or code that was meant to be wrapped and executed.

```
/* call the gcd() C function */
result = gcd(a, b);
```

3. The wrapper has to translate/convert the results of the function invocation back from C types to corresponding Python ones. The Py_BuildValue() functions takes similar conversion specifications as in Table 2.2.

Format string	**Python Type**	**C Type**
“s”	String	const char *
“s#”	String	const char *, int
“i”	Integer	int
“l”	Integer	long int
“c”	String of length 1	char
“f”	Float	float
“d”	Float	double
“O”	Object	PyObject *
“O&”	Object	converter, anything
“(items)”	(tuple)	matching-items

Table 2.2: Common Format specifiers for PyArg_ParseTuple() and Py_BuildValue()

```
return Py_BuildValue("i", result);
```

The **self** parameter is used when the wrapper function implements a built-in method to be applied to an instance of some object. In this case, the instance is placed in the **self** parameter; otherwise, **self** is set to NULL. **args** is a Python tuple that contains the function arguments passed by the interpreter. **kwargs** is a Python dictionary that contains keyword arguments.

Building the extension

In order to build the module Python provides the *distutils* package which provides support to build and install additional modules into a Python installation.

Listing 2.7 presents the setup.py build program for our example module. Distutils will invoke the right compiler and linker installed on the system with the correct platform specific flags provided without any further user intervention or specification needed.

```python
from distutils.core import setup, Extension

m = Extension('gcdmodule',
              sources = ['gcd_module.c', 'gcd.c'])

setup(name = 'gcdmodule', version = '0.1',
      description = 'Greatest Common Divisor',
      ext_modules = [m])
```

Listing 2.7: setup.py - a distutils build script for the Python C module

We can now similarly invoke the python interpreter from the command line as follows in order to compute a result for $gcd(15, 3)$ (line 2 shows the result):

```
1 $ python -c "from gcdmodule import *; print gcd(15,3)"
2 => 3
```

Listing 2.8: Invocation of the gcdmodule from the command line

The Python C API further provides functions for each of the basic Python objects. This allows to use them directly with much finer control than offered through PyArg_ParseTuple and Py_BuildValue, which are sufficient for most of the use cases. The functions for each of the basic Python objects are especially useful for the required flexibility and control, when the Python interpreter is to be embedded.

Reference counting in Python

The Python interpreter uses reference counting to manage memory (and in order to implement garbage collection). Each Python object has a reference count associated with, which tracks the number of places other objects reference to it. When the reference count reaches zero, Python reclaims the object's memory space automatically. This is no concern when pure Python programs are written as reference counting is handled automatically. However, the developer of such an extension is responsible for handling the reference counts manually, if he extends or embeds through the Python C API, as otherwise memory leaks or crashes will occur.

The Py_BuildValue() increases the reference count automatically and therefore the reference count must not be increased manually as wrapper functions are expected to return either an object with an incremented reference count or NULL to signal an error.

Python macro	**Description**
Py_INCREF(obj)	increments the reference count
Py_DECREF(obj)	decrements the reference count
Py_XINCREF(obj)	increments the reference count, but ignores a NULL object
Py_XDECREF(obj)	decrements the reference count, but ignores a NULL object

Table 2.3: Python reference counting macros

Table 2.3 lists the Python C API macros that allow to manipulate the reference count of Python objects.

Error handling

The last topic, regarding the Python C API, are the mechanisms in place for Error Handling. Errors can occur at multiple places and on both sides of the languages involved. Python uses exception handling for error detection and resolution. As mentioned above a wrapper functions should return NULL to signal an error, which is exactly what all the Python functions do with a couple of exceptions such as **PyArg_ParseTuple()**. Prior to returning NULL, an exception should be raised or cleared using the **PyErr_SetString()** and **PyErr_Clear()** functions. An example for raising an index out of bounds exception is: **PyErr_SetString(PyExc_IndexError, "index out of bounds");**

2.4 Wrappers and available approaches for Python

We have seen the basic tasks of a wrapper in section 2.3.1 "Overview of the Python C API". A more general description of a wrapper reads as follows: *"A wrapper is a piece of code that allow different pieces of code to work together that normally cannot because of incompatible interfaces"*. Therefore, a wrapper acts as an interface between its caller and the wrapped code. There are different purposes that a wrapper can fulfill:

Compatibility: When the wrapped code is in a different programming language or uses different calling conventions, as is the case with our Pivy Python binding for the Coin C++ library.

Emulation: For example, the OpenGL API which hides the functions of the video card driver or Chromium, which provides a wrapper around the OpenGL API in order to intercept OpenGL calls. The implication is that the wrapped code can only be accessed via the wrapper.

Security: Where the wrapped code and exposed interface prevents the calling program to execute certain functions or exposes them depending on the authorization level.

The key problems with wrapping a source language such as C++, which is to be used out of another target language such as Python, are that language features might be different. In the worst case no equivalent or similar feature exists in the target language, which can make it difficult or impossible to map certain functionality to an equivalent one. For example, protected class members do not exist in Python and private class members were just introduced a couple of versions ago. Python has no notion of true access restrictions for attributes, where every member and each method is public and attribute hiding is a matter of convention.

Another key problem is argument handling where a C/C++ function/method might interpret an argument as an output value or even input/output value. Examples for functions, where the wrapper has to take care about the arguments and handle them specially, are presented in the next listing:

```
/* returns result in the double * c argument */
void add(double a, double b, double * c) {
  *c = a+b;
}

/* double * x is as an input/output argument */
void negate(double * x) {
  *x = -(*x);
}
```

Listing 2.9: Example listings for input output parameters

The last and most important key problem that such wrappers have to deal with are type conversions which are responsible to map and translate data types from the source language to corresponding ones in the target language.

Various different and interesting approaches in order to create a Python wrapper are in existence. They range from manual wrapping, runtime system based wrapping, foreign function interfaces to semi- and automatic wrapper generators. The next sections will present some of the available approaches and provide a comparison of some wrapper generators and the reasoning for the approach that has been chosen for our Coin binding.

2.4.1 Manual wrapping

We have presented the manual wrapping approach in our discussion in section 2.3.1 “Overview of the Python C API”. To create a wrapper manually has the benefit of the finest control and allows for the generation of a very tight and optimized wrapper.

Unfortunately, this approach quickly becomes unfeasible and quickly cumbersome for larger interfaces as a lot of boilerplate code for the different interfaces we deal with has to be written. Even worse, every single function, class, method and data type in such an interface has to be registered and wrapped, which is a very tedious and error prone task.

Furthermore, if the library changes or adds new interfaces in future versions the wrapper code has to be adapted and extended accordingly manually again, which makes maintenance of such a solution very difficult if not impossible to achieve efficiently. This is especially true for C++, which offers language features such as function overloading, virtual inheritance, const interfaces, static members, templates..., where it becomes very difficult to provide a complete manual wrapper and which makes the manual wrapper code that has to be written very involved and complicated.

Manual wrapping through runtime system usage

On the other hand, if the source library or language offers an extensive runtime system - such as Objective C - a manual wrapper becomes actually feasible again. The introspective mechanisms offered by the target language's runtime system allow to create a manual wrapper with minimal lines of code. This code then has to deal with data type conversion, whereas calling methods or creating instances of objects can be achieved through a very generic code layer which makes heavy usage of the runtime system. Furthermore, if the API has been very consistently designed the problem of determining the input, output or input/output arguments of a method becomes less involved and can be handled by a self-written special wrapper generator in order to cover the whole API.

In fact this approach was considered for our binding and where the author is quite sure now that this approach will have worked very well. A problem is that certain parts of Coin are not covered by the runtime system[6]. However, those can be made accessible by extending the Coin runtime type system on the C++ side respectively. The alternative is to wrap them manually or again through the usage of an automatic wrapper generator. It is also unclear how a solely Coin runtime system based approach will affect performance; it can be expected that more Python code than in the wrapped case will have to be evaluated by making more intense access of the runtime system for type deduction and checking. Again this can be optimized by identifying and factoring common calling idioms out to the C++ side or optimizing the type system for such usage. The performance of such a solution also depends heavily on the performance of the Coin runtime system implementation, which again might not be fast enough for certain cases. Furthermore, the SoGui toolkits are not runtime based and have anyways to be wrapped in a regular approach.

The main reason why the author decided not to go this route back then, was the lack of good and deep understanding of the Coin internals. This makes it necessary to spend a lot of time in order to gain a better understanding of the limits of such an approach, which was deemed as a too risky and dangerous route to take. Additionally, not many projects were at hand using such an approach from which one could have learned. On the other hand, the use of automatic wrapper generators was proven and safe and which many projects make use of.

2.4.2 Automatic and semi-automatic wrapper generators

Due to the aforementioned shortcomings of the manual wrapping approach several wrapper generators have been developed. Wrapper generators create the necessary wrapper and boilerplate code automatically by inspecting the existing sources (typically the header files in case of C/C++). Furthermore, they allow to specify which functions/methods/classes should get wrapped and how this has to be done. A special so-called *interface* file is used from those

[6]for example the sensor nodes

specifications. Wrapper generators provide a specially designed syntax for type conversion, interface extension, error, input/output argument and ownership handling.

Another very important task of a wrapper generator is to conduct and help with correct memory management. Here the question of ownership, i.e. if it is the responsibility of the generated Python wrapper or the C++ side to delete the underlying C++ object. Most of the time memory management issues remain sufficiently hidden by wrapper generators but there are special situations, where the question of ownership is not clear[7] and manual intervention is required to resolve this question.

We distinguish between automatic and semi-automatic wrapper generators. Automatic wrapper generators allow to solely provide the header files of the library that has to be wrapped. Those header files get parsed by the automatic wrapper generator to generate the wrapper and through a flexible, general and versatile wrapper generator syntax any part of the generated wrapper output can be controlled. In contrast, semi-automatic wrapper generators require the developer to create an interface file for each structure and class that needs to be wrapped. They need to be specified in a header file alike declaration syntax or used with a special wrapper generator syntax out of which the wrapper code is generated.

The difference in those approaches range from the amount of language features covered for the source language(s), focus on a tighter integration with a specific target language or library, wrapper size to overall flexibility and maintainability of those solutions.

Here is a short presentation and description of the currently available Python automatic/semi-automatic wrapper generators:

Boost.Python [8] is an open source C++ library which enables seamless interoperability between C++ and the Python programming language and provides a concise IDL-like interface in order to bind C++ classes and functions to Python. Boost.Python's rich set of features and high-level interface make it possible to engineer packages from the ground up as hybrid systems, that give programmers easy and coherent access to both the efficient compile-time polymorphism of C++ and the extremely convenient run-time polymorphism of Python. This is achieved entirely in pure C++ through leveraging the full power of C++ compile-time introspection and of recently developed meta-programming techniques without the introduction of a new syntax. Because it leverages template meta-programming to introspect about types and functions, the user never has to learn a third syntax: the interface definitions are written in concise and maintainable C++.

Boost.Python can be classified as a semi-automatic wrappers as an interface file in an IDL-like language has to be written for every method and class. However, a code generator named Pyste[9] has been developed that allows Boost.Python to be used as an automatic wrapper generator. Pyste uses GCC_XML[10], an XML output extension to the GNU compiler collection, to parse all the headers and extract the necessary information to automatically generate C++ code.

[7]such as when transferring the ownership of an object wrapped in one wrapper generator to another structure wrapped by another wrapper generator, as is the case with bridging

[8]http://www.boost.org/libs/python/

[9]http://www.boost.org/libs/python/pyste/

[10]http://www.gccxml.org/

SIP [11] SIP is a semi-automatic wrapper generator that has been originally developed to create a Python binding for the Qt GUI toolkit but can in the mean time be used for wrapping other C++ libraries as well.

SIP comprises a code generator and a Python module. The code generator processes a set of specification files and generates C or C++ code which is then compiled to create the bindings extension module. The SIP Python module provides support functions to the generated code.

The specification files contains a description of the interface of the C or C++ library, i.e. the classes, methods, functions and variables. The format of a specification file is almost identical to a C or C++ header file.

SWIG [12] (Simplified Wrapper and Interface Generator) is an automatic wrapper generator that connects programs written in C and C++ with a variety of high-level programming languages.

SWIG is used with different types of languages including common scripting languages such as Lua, Perl, PHP, Python, Ruby and Tcl. The list of supported languages also includes non-scripting languages such as C#, Common Lisp (CLISP, Allegro CL, CFFI, UFFI), Java, Modula-3 and OCAML. Also several interpreted and compiled Scheme implementations (Guile, MzScheme, Chicken) are supported.

SWIG is most commonly used to create high-level interpreted or compiled programming environments, user interfaces, and as a tool to test and prototype C/C++ software. SWIG can also export its parse tree in the form of XML and Lisp s-expressions. SWIG may be freely used, distributed, and modified for commercial and non-commercial use.

2.4.3 Other alternatives and approaches

Additional interesting alternatives exist in order to create extension modules or embedding Python. Their usage can be combined with existing wrapper generators in order to improve performance or further ease usage.

Pyrex [13] is a programming language developed to aid in the creation of Python extension modules and features a Python-like syntax. The Pyrex compiler will convert the Pyrex code into C code. Code, which manipulates Python values and C values, can be freely intermixed and where conversions occur automatically wherever possible.

Reference count maintenance and error checking of Python operations is also automatic, and the full power of Python's exception handling facilities, including the try-except and try-finally statements, are available to the developer – even in the midst of manipulating C data. This allows different to wrapper generators, whose main focus is to wrap existing libraries, for rapid development of new Python types without the need to know nor create them manually through the Python C API.

[11] http://www.riverbankcomputing.co.uk/sip/
[12] http://www.swig.org/
[13] http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex/

ctypes [14] is an advanced ffi (Foreign Function Interface) package for Python. ctypes allows to call functions exposed from dlls/shared libraries and has extensive facilities to create, access and manipulate simple and complicated C data types in Python - in other words: wrap libraries in pure Python. It is even possible to implement C callback functions in pure Python.

The benefit of the ctypes approach is that the calling overhead for functions is heavily minimized as it operates directly on the dynamic link library. Still, in/inout/out argument handling has to be handled manually and due to name mangling (and different name mangling schemes for different compilers) and various other language features in C++ this approach is most only feasible for libraries written exclusively in C.

Elmer [15] is a tool which embeds a Python module into a C, C++, or Tcl application. The embedded Python module is used just as if it was written in the same language as the application itself. Therefore, it does not require any knowledge of the Python/C API.

Elmer generates "glue" code that embeds an interpreter for one type of language into an extension for another. Elmer allows function calls and various data types, both native and custom, to pass transparently between the two languages.

2.5 3D graphics in Python

A broad range of 3D modeling tools and 3D graphics libraries exist that take advantage of the power of Python. This section presents some from the sheer number of available and interesting Python based 3D projects.

Many of the presented projects can be used and combined with our binding as they are exposed to the same language, which allows them to be easily integrated. For example, *PyOpenGL* can be used unmodified with our binding and allows the Pivy developer to issue OpenGL commands. *Blender* allows to create 3D models and export them to VRML or Inventor files through ready to use Python export scripts for later reuse in scenes created in Pivy. Others, such as *VTK* or *Open Scene Graph*, are a similar or the same domain, where it is interesting to learn which wrapping strategies and approaches those projects have chosen in order to create a binding for a large C++ library. Then again others are listed for completeness and to show the versatility of available and readily usable 3D related projects that can be combined with Pivy, such as *SciPy* or the *cgKit*.

2.5.1 Existing Python tools and bindings

PyOpenGL is a Python binding for OpenGL developed by Mike C. Fletcher.

PyOpenGL includes support for OpenGL v1.0, OpenGL v1.1, GLU, GLUT v3.7, GLE 3, WGL 4, and Togl (Tk OpenGL widget). It also includes supports for dozens of extensions (where supported in the underlying implementation).

[14] http://starship.python.net/crew/theller/ctypes/
[15] http://elmer.sourceforge.net/

Additionally, it offers a retained mode library called OpenGLContext which features a partial implementation of VRML97.

PyOpenGL is perfectly suited for Rapid Application Development of OpenGL applications in Python. Apart from that it is used by a variety of Python projects to add OpenGL support.

As mentioned before, this mature binding can be made use of directly in Pivy, as all that is needed for the integration is an OpenGL context. The OpenGL context can be either created in Pivy or PyOpenGL. The converted *"Inventor Mentor"* contain examples that demonstrate PyOpenGL's usage in the Pivy.

PyOpenGL uses SWIG to create the binding. Currently, the use of ctypes, a Foreign Function Interface package for Python, is investigated for the creation of the binding.

URL: `http://pyopengl.sourceforge.net/`

PyFX is an interesting Python framework which makes it easy to use graphics effects based on Cg shaders.

PyFX[15] is implemented on top of OpenGL and allows to write shader code in Python that gets executed on graphics processors (GPUs).

The PyFX framework is written in pure Python, uses SWIG to wrap additional required C libraries and similarly to PyOpenGL can be used together with Pivy.

URL: `http://graphics.cs.lth.se/pyfx/`

cgKit (the Python Computer Graphics Kit) is a generic 3D package that can be useful in any domain where you have to deal with 3D data of any kind, be it for visualization, creating photorealistic images, Virtual Reality or even games.

It is a collection of Python modules that contain the basic types and functions to be able to create 3D computer graphics images. The kit mainly focuses on Pixar's RenderMan interface, but some modules can also be used for OpenGL programs or non-RenderMan compliant renderers such as POV-Ray.

Again this mature project can be combined with Pivy. In fact it provides very broad and rich functionality and can be used to import various file formats through its plugins such as X3D, that Coin does not offer support for, yet. This is achieved, because cgKit wraps various C/C++ libraries, which provide for example the aforementioned X3D support. As a variety of libraries are wrapped, the cgKit further provides support for spezialized input devices, such as data gloves.

URL: `http://cgkit.sourceforge.net/`

Blender is a very mature and professional cross platform 3D modeller which has a very long feature list comprising a fully featured gaming engine. Blender exposes all its internals through a Python API which allows Python to be used as a scripting facility.

Blender can natively export and import DXF, Inventor and VRML files but additional exporters and importers already exist as Python scripts or can easily be written in Python and called from within Blender.

URL: `http://www.blender3d.org/`

Chromium is a system for interactive rendering on clusters of graphics workstations. Various parallel rendering techniques such as sort-first and sort-last may be implemented with Chromium. Furthermore, Chromium allows filtering and manipulation of OpenGL command streams for non-invasive rendering algorithms.

The configs are created through Python scripts which allows to specify different SPUs (stream processing unit). A SPU is the part of a node that intercepts OpenGL calls and performs actions on the OpenGL stream.

Chromium can be used for clustering solutions and connecting multiple displays to a big one. Most importantly it allows to intercept OpenGL calls which can be used together with PyOpenGL and lets one further exploit its capabilities from within Python.

URL: http://chromium.sourceforge.net/

Open Scene Graph is a open source high performance 3D graphics toolkit, used by application developers in fields such as visual simulation, games, virtual reality, scientific visualization and modeling.

Open Scene Graph has a Python binding, called PyOSG[16]. PyOSG uses the Boost.Python library, is in alpha state and appears to be unmaintained now.

URL: http://www.openscenegraph.org/

SGI OpenGL Performer is a an application-driven library used in real-time visual simulation and other performance-oriented 3D graphics applications for which a Python binding exists: *pyper*[17]. The *pyper* binding makes use of SWIG to create the wrapper.

Performer simplifies development of applications used for visual simulation, manufacturing, simulation-based design, virtual reality, scientific visualization, interactive entertainment, broadcast video, architectural walk-through, and computer aided design.

URL: http://www.sgi.com/products/software/performer/

The Visualization ToolKit (VTK) is a software system for 3D computer graphics, image processing, and visualization used for scientific purposes. It also features a variety of visualization algorithms. In contrast to the scene-graph based APIs VTK uses a pipeline approach.

The Python binding has been created using GCC_XML, CableSwig[18] and CSWIG, a modified version of SWIG.

ITK makes heavy use of C++ features, where SWIG was not able to cope with all of them. Therefore, CableSwig is used to create wrappers to interpreted languages such as Tcl and Python. It was created to produce wrappers for ITK because the toolkit uses C++ structures that SWIG cannot parse (deeply nested template instantiations). CableSwig is a combination tool that uses GCC_XML as the C++ parser. The input files are Cable style input files. The XML produced from the Cable/GCC_XML input files are then parsed and feed into CSWIG.

[16]http://sourceforge.net/projects/pyosg/
[17]http://oss.sgi.com/cgi-bin/cvsweb.cgi/performer/src/pyper/
[18]http://www.itk.org/HTML/CableSwig.html

Furthermore, a scientific data visualizer, called MayaVi[19] for VTK and which is implemented in Python, exists.

URL: http://www.vtk.org/

OGRE (Object-Oriented Graphics Rendering Engine) is a scene-oriented, flexible 3D engine written in C++ designed to make it easier and more intuitive for developers to produce games and demos utilizing 3D hardware. The class library abstracts all the details of using the underlying system libraries like Direct3D and OpenGL and provides an interface based on world objects and other intuitive classes.

OGRE comes with Python bindings, named PyOgre[20] and is focused on game development. PyOgre is currently undergoing a rewrite using SWIG.

URL: http://www.ogre3d.org/

Panda3D (Platform Agnostic Networked Display Architecture) was developed by Disney VR Studio for its massively multiplayer online game Toontown Online. The engine contains a strong networking architecture that allows to create shared experiences quickly and easily. The studio then released the engine as open source and began to coordinate with universities in order to prepare it for the open source community. In 2003, Panda3D made its debut through Carnegie Mellon University.

Panda3D is an application-driven library, which makes it very well suited for games and simulations.

Panda3D operates and has been created with a combination of C++ and Python. Developers may script in Python, and Panda3D's special *interrogate* system converts the C++ code to Python.

To quote Panda3D's *Introduction to Scenegraph Design*[16] on how the *interrogate* system works:

> *Interrogate works like a compiler, scanning and parsing C++ code. Instead of creating object libraries, it creates an "interrogate database" of objects, methods, global variables, etc. that are contained in the corresponding C++ library. This database may be queried to discover these functional elements and their interfaces.*
>
> *To make use of this database, one creates an automatic code generator in the scripting language that scans the interrogate database and generates scripting language wrapper calls that execute the C library calls, via the scripting language's foreign function interface. A benefit of this system is that interrogate imposes no restrictions on exactly how a scripting language interfaces with the C libraries, allowing them to interface in whatever way best "fits in" with the natural environment of a particular language.*

Panda3D makes heavy usage of C++ and where SWIG was not found to cope with templates, nested classes and function overloading[17]. Further, Panda3D was originally

[19] http://mayavi.sourceforge.net/

[20] http://www.ogre3d.org/wiki/index.php/PyOgre/

designed to work with Squeak[21], an open source implementation of Smalltalk, where the *Interrogate* system was already in place.

URL: http://www.panda3d.org/

Jython3D Rob Smallshire has put up a page providing examples and descriptions in how to make use of Java3D[22] from Jython[23]. Java3D is a low level 3D scene-graph based graphics programming API for the Java language.

Jython is an implementation of the high-level, dynamic, object-oriented language Python written in Java, and seamlessly integrated with the Java platform. It thus allows you to run Python on any Java platform.

URL: http://www.smallshire.org.uk/jython3d.htm

Crystal Space is a portable 3D Game Development Kit written in C++ using OpenGL. It offers a large set of features; among them shader support (CG, vertex programs, fragment programs, …), 3D sprites (frame based or with skeletal animation using Cal3d[24] animation library), procedural textures, scripting (using Python) and physics plug-in based on ODE[25].

The Crystal Space 3D Engine uses Python for scripting through a plug-in mechanism. Python scripts can be dynamically invoked from within the engine depending on game specific actions. Python can also be used to program the main game core. It has full access to the C++ Crystal Space API.

Crystal Space uses SWIG as the basis of its language bindings. The technology has been generalized enough that the supported languages share a common SWIG interface definition. Each language also has its own specific definition files. Because the binding definitions are generalized, it should be possible to create additional language bindings for Crystal Space (any supported by SWIG) with a minimal amount of effort, as compared to creating new bindings from scratch.

URL: http://www.crystalspace3d.org/

SciPy is a library of scientific tools for Python. SciPy supplements the popular Numeric and numarray modules, gathering a variety of high level science and engineering modules together as a single package.

SciPy includes modules for graphics and plotting, optimization, integration, special functions, signal and image processing, genetic algorithms, ODE solvers, and others.

SciPy is not 3D related in itself, but because 3D graphics programming typically involves a lot of mathematical manipulation, SciPy is often a useful tool.

URL: http://www.scipy.org/

[21] http://www.squeak.org/
[22] https://java3d.dev.java.net/
[23] http://www.jython.org/
[24] http://cal3d.sourceforge.net/
[25] http://www.ode.org/

Chapter 3

Creating Pivy

In order to create a Python wrapper for Coin through SWIG the following tasks had to be tackled:

Build system The creation of a suitable cross-platform build system is crucial in order to assist with the compilation of the generated wrapper code into a dynamic link library that can then be imported by the Python interpreter at runtime.

The build system allows for an easy, convenient, consistent and correct creation of the aforementioned dynamic link library by a developer or user. It gathers available information from the operating system, support libraries and Python installation on the system in order to provide the compiler and linker with the correct flags and runs a couple of checks to ensure and assert a correctly setup development system.

A good build system helps the developer with maintenance tasks such as packaging, deployment and by further handling installation on the user's system. By sticking to common expectations, conventions and standards in how the build system should behave, eases the experience from the user's point of view and aids in the reuse of proven existing mechanisms and tools.

Therefore, the decision has been made to use the distutils[18] package found in the default Python installation for the creation of the Python extension. Other tools such as SCons[19], which do not introduce any further dependencies, were brought in where distutils was felt to be not appropriate or not designed to handle the task at hand well.

SWIG typemaps for type conversions As we have mentioned in the previous chapter one of the wrapper's responsibilities is to convert and translate data types from one language to the other. A good mapping is one of the decisive criteria that make bindings either integrate well with common expectations found in the usage of a target language or makes the bindings overall feeling alien more oriented towards the source language which might follow totally different design principles and patterns. Furthermore, a clear and well chosen mapping aids to smoothen the learning curve and makes the binding more convenient to use. The keywords regarding type conversions for Python are coherency, consistency and *pythonic*.

A lot of thought went into how to map the Coin basic types to corresponding Python representatives in order to create a good, easy to learn and convenient binding. Some choices were natural and obvious (for example, the standard types such as integer, string, float) others were tricky and at times limited to the target language options available at hand such as the inconspicuous “unsigned char *”. Furthermore, the overall design was kept as similar as possible to the one found in C++ to (a) allow easier translation of code from one language to the other and to (b) allow the reuse of existing extensive documentation, where the small differences are pointed out.

Conversion of existing C++ code to Python In order to test the functionality of the binding and in order to determine what are the most important aspects of the binding that need to be tackled, already existing examples in the source language need to be converted to the target language. They serve for testing purposes and additionally demonstrate how to use the binding.

“The Inventor Mentor” comes with a lot of C++ examples, which were converted to Python. The book serves as a guide to new users and covers a large part of the library. This helped to identify the most important functionality and the missing coverage of the wrapper very quickly. A desired side effect of the conversion of the examples is that they serve as a self-documenting entry and reference point for new Pivy users. Apart from teaching how to convert C++ programs to Python, the examples showcase certain aspects of the Coin API in Python.

Integrate unit tests to verify correctness and help with maintenance Unit tests serve the purpose to assure and verify the proper and correct operation of a binding. Covering a large part or most of the API through unit tests allows to identify problems early. Furthermore, through the modification of parts in the binding or addition of new functionality, unit tests indicate breakage of originally working functionality very early in the development cycle. They serve as a quality insurance mechanism and allow to test for working operation on different platforms very easily and quickly. Unit tests can be applied effectively through the addition of a test case for a bug, that has been identified and fixed. This ensures that the same bug cannot “sneak back” into the tree unnoticed at any later time after larger modifications have take place. A nice side effect of an extensive collection of scripts, that touch various parts of the binding, is, that they can be used for profiling purposes in order to fine tune certain facets of the wrapper.

The unit tests turned out to be an absolutely crucial and very helpful feature. During upgrades of the *SWIG* wrapper generator to newer versions they ensure that already existing functionality continued to work and they help to identify problematic areas, which have to be examined and addressed from the very start.

3.1 Why SWIG?

Before the creation of the binding started, an evaluation of the available existing wrapper generator solutions, namely *Boost.Python, SIP, SWIG*, has been conducted. In order to find the most suitable for the task, a list of goals and properties, the desired wrapper generator should be able to offer, has been compiled:

Cross-platform *Coin* is available for a multitude of platforms, therefore the wrapper generator should be available on all platforms *Coin* supports. Furthermore, the actual generated wrapper should be usable without any modifications on all platforms for easier deployment management.

SWIG and *Boost.Python* are cross-platform, available natively on all *Unix* platforms, *Mac OS X* and *Windows*. *SIP* was not; it lacked *Mac OS X* and build system support for a variety of Unix platforms and moreover a *SIP* Windows binary was only available under a commercial license.

Free unrestricted license The wrapper generator should not put any restrictions or dictate the license choice (as is the case with the GPL or commercial licenses) on the outputted wrapper or its usage in order to be able to use it for any non-commercial and commercial usage. *SWIG* and *Boost.Python* have free BSD-like licenses, whereas *SIP* Windows usage was restricted[1].

Python version support If possible no specific Python version should be enforced in order to be able to use the wrapper with. *SWIG* works with older Python versions and can make use of newer features (esp. new style classes) found in the Python interpreter through optional flags, *Boost.Python* works with Python 2.2 upwards (new style classes) where an older version is available to work with older Python versions. An older *SIP* version works with older versions through the use of Python's *classic* classes to wrap C++ classes; new style classes are supported in the newer SIP versions.

Proven, wide-spread, good documentation The wrapper generator should be proven, well understood and in wide-spread usage. Furthermore, having a community, mailing lists and good documentation in order to be able to get support if any questions or problems arise is a big advantage.

SWIG is the longest standing project of the compared wrapper generators and has been publicly available since February 1996. It is in wide-spread usage today and many projects that created a Python binding for their own purposes such as Subversion, PyOgre, wxPython, Performer, PyOpenGL use *SWIG* to create wrappers. Among other companies and research institutes *Google* and *NASA* make use of *SWIG*. *SWIG* has extensive documentation. *Boost.Python* was still quite new, when first encountered and *SIP* had a rather small community which was anyways mainly focused on the Qt GUI toolkit. Apart from that there was little to no *SIP* documentation available.

Easy to use, extend, maintain, integrate/bridge The interface code that needs to be written should be easy to write, extend and maintain. Furthermore, it should be possible to integrate and bridge the generated wrapper with/to other Python bindings.

Boost.Python is a C++ library, which makes extensive usage of C++ templates. C++ templates tend to be hard to read and error messages are generated at link time, when all types are resolved, which makes it a bit hard to maintain and develop with. It was prefered to develop directly in the clear Python C API instead of having to use involved C++ mechanisms, which results in an additional level of complexity and compiler issues regarding the varying grade of C++ support.

[1]this seems to have changed now and it is licensed under the Python BSD-like license

SWIG and *SIP* allow to directly access the actual C++ pointers of the wrapped structures and classes, which allows for integration with different bindings. However, the well structured wrapper code generated by *SWIG* was found to be easier to read and clearer, which helps a lot in debugging problems. Furthermore, the *SWIG* generated wrapper code is kept very general and allows for a lot of flexibility in tweaking, which is achieved by providing several layers, such as a Python proxy module; any part in the type conversion code of any function or method can be overridden by self written code.

Moreover, the type conversions can be specified outside of the actual classes through a flexible naming scheme for any type in the overall main interface, in contrast to SIP where it has to be specified in the class after the method itself. This is a very helpful feature when it comes to maintenance and minimizes the overall lines of code that are necessary. The interface for a class itself can be extended and any method signature can be redefined into a more *"pythonic"*[2] interface. *SIP*'s lack of such flexibility typically results in a binding with a more C++ like API.

Additionally, *SWIG* is able to directly parse the C++ header files and therefore makes it sufficient to just specify them in a main interface. The developer can focus on the actual type conversion code and only create special interface files if more control is desired. *SIP* more or less requires to create an interface for each and every single class. *SIP* tries to solve the problem of drifting header files for different versions through the addition of additional conditional syntax for versioning. This works for single self-contained libraries such as Qt, but is not efficiently manageable with different development models or libraries, such as *Coin*. The reason is that apart from a couple of different development branches, various support libraries, such as the different SoGui libraries and additional visualization libraries, based on *Coin*, exist.

It is cumbersome to add new classes with *SIP* as again complete tweaked header files have to be added for new classes, whereas with *SWIG* and a well laid out interface structure and foundation, new classes are simply added through specifying header files again.

And last but not least, the wrapper generator should have a stable API. The *SIP* interface and internals tend to drastically change from release to release, driven by the desire and requirements to provide a better wrapper for the Qt GUI toolkit. It is unmaintainable to have to rewrite major parts of the interface files in order to be able to make use of newer *SIP* versions. *SWIG* changes too with newer versions, especially the C++ side, but backwards compatibility is prioritized. This is a necessity, as many Python bindings in existence depend on *SWIG*.

Additional language bindings Another big advantage of *SWIG* is that different to *Boost.Python* and *SIP* the wrapper generator allows to create bindings for various supported languages such as C#, Java, Ruby, Perl, Tcl, etc. out of the same code base. *SWIG* is also able to create a wrapper for C libraries and not only C++ as is again the case with *Boost.Python* and *SIP*.

Covers all C++ features in Coin The wrapper generator should be able to handle all encountered C++ features in *Coin*, which does not make heavy usage of them to be

[2]a term coined in the Python community to refer to certain syntactic idioms expected by Python programmers

able to stay cross-platform and work with a large variety of different compilers.

SWIG was quite lacking, when it came to C++ support at the time it was first evaluated. However, support for C++ improved over the last years as wrapping C++ libraries with *SWIG* was requested in the *SWIG* community. As *Coin* does not make use of advanced C++ features such as templates, exceptions and follows an *"has-a"* relationship rather than an *"is-a"* model, this did not pose a big problem. The lack of operator and function overloading handling back then was handled through writing our own dispatch mechanism which checked for argument length and types in order to differentiate between and call the right methods.

Acceptable performance As we have seen in the previous sections the calling overhead for Python is already high. However, when wrapping a retained mode library this is not a big concern. Both *Boost.Python* and *SIP* have an advantage over SWIG in their generated wrapper performance. For example, *SWIG* uses a proxy class written in Python for additional gains in flexibility and more suitable target language API exposure, but the trade off is additional calling overhead. However, *SWIG's* flexibility always offers the possibility to optimize for certain parts in the wrapper, which makes this issue not a serious restriction and concern again. We need to especially optimize the basic types in *Coin*, as their code is the one that is invoked most frequently.

No additional dependencies The wrapper generator should not add additional library or module dependencies.

SIP requires a sip module and therefore can only be installed on the system if the wrapper is generated for the same *SIP* version. *Boost.Python* provides shared libraries, which either need to be linked with the Python interpreter or alternatively the runtime linker needs to be instructed where to find the libraries, if they are not supposed to be installed on the system. *SWIG* creates a simple self contained Python extension without further additional library dependencies, which allows the straightforward usage of different *SWIG* versions on the same system.

Modularization As mentioned before, *Coin* has different support libraries. Each of them should be independently wrapped into a Python module in order to minimize resource usage.

SWIG provides an extensive runtime mechanism for type queries and the creation or conversion of new types at runtime. *SWIG* allows external access to this runtime mechanism, which provides the necessary means to allow the various parts to be split and placed into separate modules as the modules have shared access to the same type information.

As we have seen, *SWIG* turned out to be our best option and its few shortcomings can be easily dealt with. Most importantly, as Coin does not make heavy usage of C++ features and its extensions are created through usage of macros and not through an overriding inheritance model, it was not required to use *Boost.Python*, which had better support for advanced C++ features. Furthermore, *SWIG* provides the necessary infrastructure and foundation for future support of additional target languages without having redetermine the parts of the API that need to be handled specially and relearn a new syntax for the wrapper generator.

3.2 The SWIG interface and glue code

This section will cover some of the so-called *SWIG typemaps* that have been used for the creation of Pivy. It will provide a discussion about the most important *SWIG* constructs applied in Pivy, namely:

- how to apply the *typemaps* for type conversions,
- how to tweak the provided API to make it more *pythonic* and let *SWIG* handle C++ operators,
- and how to handle the Coin callbacks.

3.2.1 SWIG typemaps and type conversion/typechecks

What are typemaps and what are they used for? To answer that question we quote the introduction paragraph of section *10.1.1* in the *Typemaps chapter* of the *SWIG* manual[20]:

> *One of the most important problems in wrapper code generation is the conversion of datatypes between programming languages. Specifically, for every C/C++ declaration, SWIG must somehow generate wrapper code that allows values to be passed back and forth between languages. Since every programming language represents data differently, this is not a simple of matter of simply linking code together with the C linker. Instead, SWIG has to know something about how data is represented in each language and how it can be manipulated.*

The basic idea of the **typemap** handling mechanism in *SWIG* is to allow the *SWIG* user to change any part or let him specify parts differently for any generated wrapper function.

Typemaps allows to handle argument, handle return values, assign and read a global variable, assign member variables and the creation of constants.

Once defined, a typemap remains in effect for all of the declarations that follow. A typemap may be redefined for different sections of an *SWIG* interface file. *Typemaps* have scoping rules that allow to tie a *typemaps* to a specific class or define them globally.

One exception to the typemap scoping rules pertains to the %extend declaration. %extend is used to attach new declarations to a class or structure definition. Because of this, all of the declarations in an %extend block are subject to the typemap rules that are in effect at the point where the class itself is defined.

Furthermore, *typemaps* come with an extensive pattern matching mechanism, which allows the user to define general *typemaps* that will be applied for the method signatures matching the specified arguments pattern. Those patterns matches allow for argument type matches or argument type and argument name matches, which makes them incredibly powerful and is extremely well suited for APIs that follow the same design pattern as through a couple of global *typemap* specifications a wide range of the API can be covered.

3.2.2 SbName typemap example

For example, in the *interfaces/pivy_common_typemaps.i SWIG* interface file, we define typemaps with global scope that should be applied to any matching signature:

```
287 %typemap(in) SbName & {
288   if (PyString_Check($input)) {
289     $1 = new SbName(PyString_AsString($input));
290   } else {
291     SbName * tmp = NULL;
292     $1 = new SbName;
293     SWIG_ConvertPtr($input, (void**)&tmp, SWIGTYPE_p_SbName, 1);
294     *$1 = *tmp;
295   }
296 }
```

Listing 3.1: SbName global input typemap

In fact this is a very interesting input *typemap*, which allows any method accepting an **SbName** argument to optionally be provided by a regular Python string from Python. After the typemap declaration, where we tell *SWIG* what kind of *typemap* we are dealing with, we specify the datatype this *typemap* should be applied to when encountered. In our case this is **SbName &** (line 287). Note the ampersand *(&)* at the end. This typemap is not going to be applied for any signature that does not take a **SbName &** reference as an argument but for example a plain **SbName** instead. However, it is going to get applied for pointer arguments, such as **SbName ***, as *SWIG* treats C++ references as pointers internally.

What this *typemap* then does is to check if the provided input argument **$input** is a Python string (line 288) upon which it will create a new **SbName** class and assign it to **$1** (line 289), where **$n** which is the general substitute or a special variable referring to a local C/C++ variable corresponding to type **n** in the typemap pattern. In our case we only have a single **SbName &** variable, so it is the first an only one and therefore **$1**.

If the input argument turns out not to be a Python string, then in the *else* branch of the conditional we extract the contained **SbName** class through the provided *SWIG* runtime system function **SWIG_ConvertPtr()** (line 293).

Why do we create a *tmp* variable (line 291) and copy over the contents from the object (line 294) to a newly allocated **SbName** instance (line 292) instead of simply assigning **$1** directly in the **SWIG_ConvertPtr()** function, which is perfectly legitimate? The reason for that is, that we create a problem with freeing the **SbName** instance later on as we cannot free the instance contained in **$1**. The ownership of the object might be assigned to the Python side and therefore we are then not allowed to free it. Furthermore, checking the ownership is not an option in this particular case, as the *freearg* typemap, which allows to add typemap code for freeing arguments, only passes the C/C++ values and not the *SWIG* wrapped Python object anymore. If we decide not to free it we create a memory leak. Therefore, we copy it and can be sure that the code can be safely freed for both cases as we always create a new **SbName** instance.

```
%typemap(freearg) SbName & {
  if ($1) { delete $1; }
}
```

Listing 3.2: SbName global freearg typemap

Listing 3.3 presents the generated wrapper code for

```
static SoNode * getByName(const SbName &name);
```

which takes an **SbName &** input argument and where we can see our *typemaps* getting applied (line 9-16, 24-26 and 31-33).

```
 1  static PyObject *_wrap_SoNode_getByName__SWIG_0(PyObject *,
       PyObject *args) {
 2      PyObject *resultobj = NULL;
 3      SbName *arg1 = 0 ;
 4      SoNode *result;
 5      PyObject * obj0 = 0 ;
 6
 7      if(!PyArg_ParseTuple(args,(char *)"O:SoNode_getByName",&obj0))
           goto fail;
 8      {
 9          if (PyString_Check(obj0)) {
10              arg1 = new SbName(PyString_AsString(obj0));
11          } else {
12              SbName * tmp = NULL;
13              arg1 = new SbName;
14              SWIG_ConvertPtr(obj0, (void**)&tmp, SWIGTYPE_p_SbName,
                   1);
15              *arg1 = *tmp;
16          }
17      }
18      result = (SoNode *)SoNode::getByName((SbName const &)*arg1);
19
20      {
21          resultobj = autocast_base(result);
22      }
23      {
24          if (arg1) {
25              delete arg1;
26          }
27      }
28      return resultobj;
29      fail:
30      {
31          if (arg1) {
```

```
32                  delete arg1;
33              }
34          }
35          return NULL;
36  }
```

Listing 3.3: The SWIG generated wrapper code for SoNode::getByName()

3.2.3 SWIG's shadow classes

In addition to the C module extension wrapper code, *SWIG* creates an accompanying Python proxy module (also known as shadow modules) in order to provide a more natural API. These proxy classes are typically implemented in the target language itself.

Proxy classes are used to keep track of the ownership in the **thisown** flag; they contain and expose the C++ pointer to the actual wrapped C++ object in a special **this** instance in each class. The *"shadow"* class is implemented in the target language, which allows to attach new Python methods to the class and allows to further inherit from it. Again any part in the proxy class can be overridden or extended through a **%feature("shadow")** directive.

In order for *SWIG* to provide a flexible and very general mechanism, the *"shadow"* class methods takes an variable number of arguments ***args** by default. When the Python interpreter is used in interactive mode the method signature of such methods are not very descriptive to the developer as it cannot be seen what kind of parameters are allowed. To remedy that *SWIG* provides an **autodoc** feature. The **autodoc** feature can be enabled in order to create according Python docstrings which provide the necessary information as can be seen in the next Listing 3.3 for the generated proxy code of the **SoNode.getByName()** method:

```
class SoNode(SoFieldContainer):
    """Proxy of C++ SoNode class"""
    ...
    def getByName(*args):
        """
        getByName(SbName name) -> SoNode
        getByName(SbName name, SoNodeList l) -> int
        """
        return _coin.SoNode_getByName(*args)

    getByName = staticmethod(getByName)
```

Listing 3.4: The SWIG generated Python proxy SoNode.getByName()

3.2.4 Autocasting

In Listing 3.3 of the *SWIG* generated wrapper code for **SoNode::getByName()** we can see another interesting function call in line 21:

```
resultobj = autocast_base(result);
```

Pivy features a mechanism to automatically cast pointers to their right type. This is done through the combined usage of the dynamic runtime introspection mechanisms found in the *OpenInventor* API and the *SWIG* runtime features.

In order to *"autocast"* for example Coin nodes inherited from SoBase, we create the following output typemap. It will invoke the specified code before the wrapper returns its argument:

```
%typemap(out) SoBase * {
  $result = autocast_base($1);
}
```

Listing 3.5: Output typemap invoking the autocast_base function

The *SWIG* variable **$result** contains the Python return value. We pass the **$1** input argument of type **SoBase *** to our own **autocast_base** function, that will conduct the *"autocasting"*.

In Listing 3.6 we first make sure, through Coin runtime system usage, if the **base** object passed to our function is actually inherited from **SoFieldContainer** through the method **isOfType()** (line 78). We then gather the real type of the node through the invocation of the **getTypeId()** method on the passed instance (line 81). Afterwards we create a new *SWIG* object of type **SoBase** (line 85) and create an Python argument object, that contains the new *SWIG* object in the first item and the typename of the object, which we gathered from the Inventor runtime system, as a string in the second item (line 86). This **cast_args** argument is then passed to our **cast** function with the first **self** instance that contains NULL (line 88). The reason we factor the actual cast out to another function is twofold:

1. The **cast** function gets exposed to Python, where it allows the Pivy user to initiate manual casts if that should be found necessary. This also explains why we have to create a Python argument first in order to be able pass it to the **cast** function.
2. autocast_base() is not the only function that is used for *"autocasting"*. We further need to provide *"autocasting"* functions for paths, fields and events. In order to collect all common code in one function to make the code-base more readable and maintainable, we simply reuse the common part through the **cast** function.

The code is embedded in a while loop (line 84) that serves the purpose to cast back to the first built-in node that is known to the Coin runtime system in order to be able to *"autocast"* and therefore make use of unknown and unwrapped nodes in Pivy. The loop either terminates if no such node can be found or a result has been acquired. We ask the Coin runtime system to provide the parent type of the inheritance hierarchy if the **cast** function was failing so far (line 93).

```
71 /* autocasting helper function for SoBase */
72 SWIGEXPORT PyObject *
73 autocast_base(SoBase * base)
74 {
  PyObject * result = NULL;

  /* autocast the result to the corresponding type */
  if (base && base->isOfType(SoFieldContainer::getClassTypeId())) {
    PyObject * cast_args = NULL;
    PyObject * obj = NULL;
    SoType type = base->getTypeId();

    /* in case of a non built-in type get the closest built-in
        parent */
    while (!(type.isBad() || result)) {
      obj = SWIG_NewPointerObj((void*)base, SWIGTYPE_p_SoBase, 0);
      cast_args = Py_BuildValue("(Os)", obj, type.getName().
          getString());

      result = cast(NULL, cast_args);

      Py_DECREF(cast_args);
      Py_DECREF(obj);

      if (!result) { type = type.getParent(); }
    }
  }

  if (!result) {
    Py_INCREF(Py_None);
    result = Py_None;
  }

  return result;
}
```

Listing 3.6: The autocast_base() helper function

The **cast** function is found in the *pivy_common_typemaps.i* interface file as well. In order to tell *SWIG* that it should include it into the generated wrapper we make use of the **%native** directive:

```
%native(cast) PyObject * cast(PyObject * self, PyObject * args);
```

Listing 3.7: Exposing the cast() function to SWIG

As mentioned above the **cast** function is implemented as a C Python function. After the variable declaration block, where we also declare the *SWIG* runtime structure of type *swig_type_info* (line 25), we extract the passed arguments (line 31). The *SWIG* runtime type system maps the Coin types to strings which contains their name with a pointer sign at the end. So a "**SoSeparator**" should be provided as "**SoSeparator ***" to let the *SWIG's* runtime **SWIG_TypeQuery()** work. In order to at least provide a more natural interface to the

Python developer for the *“unpythonic”* **cast** function, we only require the class name itself without a pointer and insert it in the function automatically (line 40-44). We try to query the type in the swig type system information (line 46), should this invocation fail, we prefix the typename with **So** and try again (line 48-58). This is necessary as the *OpenInventor* API allows the built-in types to be referred to without the **So** prefix. We then extract the C++ pointer of our Coin instance out of our original *SWIG* object through **SWIG_ConvertPtr()** (line 63) and proceed then with the actual cast-a-like through the creation of a new *SWIG* object with the *SWIG* type information gathered previously from the type system (line 66).

```
21  /* a casting helper function */
22  SWIGEXPORT PyObject *
23  cast(PyObject * self, PyObject * args)
24  {
25    swig_type_info * swig_type = 0;
26    void * cast_obj = 0;
27    char * type_name, * ptr_type;
28    int type_len;
29    PyObject * obj = 0;
30
31    if (!PyArg_ParseTuple(args, "Os#:cast", &obj, &type_name, &
        type_len)) {
32      SWIG_fail;
33    }
34
35    /*
36     * add a pointer sign to the string coming from the interpreter
37     * e.g. "SoSeparator" becomes "SoSeparator *" - so that
        SWIG_TypeQuery()
38     * can do its job.
39     */
40    if (!(ptr_type = (char*)malloc(type_len+3))) { SWIG_fail; }
41
42    memset(ptr_type, 0, type_len+3);
43    strncpy(ptr_type, type_name, type_len);
44    strcat(ptr_type, " *");
45
46    if (!(swig_type = SWIG_TypeQuery(ptr_type))) {
47      /* the britney maneuver: "baby one more time" by prefixing 'So'
          */
48      char * cast_name = (char*)malloc(type_len + 5);
49      memset(cast_name, 0, type_len + 5);
50      cast_name[0] = 'S'; cast_name[1] = 'o';
51      strncpy(cast_name+2, ptr_type, type_len+2);
52
53      if (!(swig_type = SWIG_TypeQuery(cast_name))) {
54        free(cast_name); free(ptr_type);
55        SWIG_fail;
```

```
    }

    free(cast_name);
  }

  free(ptr_type);

  SWIG_ConvertPtr(obj, (void**)&cast_obj, NULL,
     SWIG_POINTER_EXCEPTION | 0);
  if (SWIG_arg_fail(1)) { SWIG_fail; }

  return SWIG_NewPointerObj((void*)cast_obj, swig_type, 0);
  fail:
  return NULL;
}
```

Listing 3.8: The cast() helper function

3.2.5 Autorefcounting

As we have seen in the previous chapter Coin uses reference counting for memory management. It is quite cumbersome if we have to deal with and think about memory management on the Python side as one of the strengths of a dynamic language is that the developer does not have to worry about low-level memory management required.

So can we automatize the reference counting process and couple it to the Python allocation and deallocation routines that use reference counting themselves?

If we have a look again at the reference counting rules presented in the previous chapter in table 2.1 (page 16) we see that if a node is added to a node as a child of another node this increments the child's reference count.

We need to ensure the correct handling of the following two situation:

1. The reference count should never drop to zero as long as a Python instance "owning" the Coin node is allocated as the *SWIG* structure otherwise points to a deallocated memory area and will either cause memory corruption or a segmentation fault.
2. Should a Python object fall out of scope and therefore get deallocated, but a reference to the Coin node is still needed, the reference count should not drop to zero again. In this case the ownership additionally needs to be transferred to the C++ side.

An example for the second case is the following:

```python
def addCone(sep):
  sep.addChild(SoCone())

addCone(SoSeparator())
```

Listing 3.9: Reference Counting and Python scoping rules

The *addCone()* function is invoked with a newly instantiated **SoSeparator** as its argument. A newly instantiated **SoCone** instance is added to the **SoSeparator**. Now the instantiated **SoCone** object gets out of scope and therefore deallocated; **unref()**'ing the object now causes the reference count drop to zero with the undesired side effect, that the C++ node gets deallocated too.

The answer to all those considerations is surprisingly simple. Every Python object will **always** increase the Coin node's reference count on allocation through the invocation of **ref()**. Upon deallocation it will invoke an **unref()** causing the reference count to drop to *zero*. If the Coin node has been added as a child to a grouping node its reference count will be *two* and therefore drop to *one*, which at the same time solves the ownership question very elegantly: the Python object has been deallocated and apart from the reference count being correct on the C++ side, the Coin C++ instance is correctly deallocated once it is removed from the grouping node.

SWIG offers a ready to use feature for the reference count handling as it is commonly encountered in various projects.

```
/* let SWIG handle reference counting for all SoBase derived
   classes */
%feature("ref") SoBase "$this->ref();"
%feature("unref") SoBase "$this->unref();"
```

Listing 3.10: Reference Counting and

Through usage of the **%feature("ref")** and accordingly **%feature("unref")**, we are able to tell *SWIG* to call the **ref()** and **unref()** methods for all **SoBase** derived nodes. This works well for the allocation of Python objects. Unfortunately, this causes a memory leak. The reason for that is that by default the destructors of the nodes are declared in the protected section to prohibit exposure of the destructors as the reference counting mechanism should be used. This leads to:

1. *SWIG* not creating wrapper code for destructors and therefore the **%feature("unref")** directive will never be able to add the **unref()** invocation to any node inherited from **SoBase**.
2. a memory leak.

Again the solution for this problem is simple: We **%extend** the **SoBase** *SWIG* interface file with a destructor that invokes **unref()**. This causes *SWIG* to generate a wrapper for a destructor for every **SoBase** derived node, which then calls **unref()** on deallocation of the Python object.

```
%extend SoBase {
  /* add a public destructor - otherwise refcount of new SoBase
   * derived instances, raised by the autoref feature, never gets
   * decreased */
  ~SoBase() { self->unref(); }
}
```

Listing 3.11: SoBase %extended with a destructor

3.2.6 Method/operator overloading

The Coin C++ API heavily uses method and operator overloading. Method overloading is handled through *SWIG* where it implements its own dispatching mechanism to determine which method needs to be invoked based on the following two rules[3]:

1. Number of required arguments, where methods are sorted by an increasing number of required arguments.
2. Argument type precedence, where all C++ datatypes are assigned a numeric type precedence value.

The dispatch mechanism can always be manually refined in case of border cases. The *SWIG* **%rename** directive, which allows to rename certain functions, and **%ignore** directive is used to achieve that.

In order to wrap overloaded operators a combination of the *SWIG* **%extend** and Python's special methods can be used. The following listing shows the overloaded operator handling for the **SbVec3f** basic type class, where **self** is a pointer to the **SbVec3f** instance:

```
%extend SbVec3f {
SbVec3f
__add__(const SbVec3f &u) { return *self + u; }
SbVec3f
__sub__(const SbVec3f &u) { return *self - u; }
SbVec3f
__mul__(const float d) { return *self * d; }
SbVec3f
__rmul__(const float d) { return *self * d; }
SbVec3f
__div__(const float d) { return *self / d; }
int
__eq__(const SbVec3f &u ) { return *self == u; }
int
__nq__(const SbVec3f &u) { return *self != u; }
```

[3]The exact rules are found in chapter 6 "SWIG and C++" of the SWIG manual

```
// add a method for wrapping c++ operator[] access
float
__getitem__(int i) { return (self->getValue())[i]; }
void
__setitem__(int i, float value) { (*self)[i] = value; }
}
```

Listing 3.12: Overloaded operators handling through %extend

3.2.7 Python field assignment operator handling

However, **operator=** cannot be handled in this fashion as there is no direct synonym for that in Python. In order to still provide a similar syntax and to provide similar functionality to overloading the assignment operator, which are specially useful for field assignments, we can make use the **__getattr__** and **__setattr__** special Python methods dealing with attribute access.

The current implementation in Python is shown in the next listing which is found in the *fields/SoFieldContainer.i SWIG* interface file:

```
%pythoncode %{
  def __getattr__(self, name):
    try:
        return SoBase.__getattribute__(self, name)
    except AttributeError, e:
        field = self.getField(name)
        if field is None:
            raise e
        return field

  def __setattr__(self, name, value):
    if name == 'this':
        return SoBase.__setattr__(self, name, value)
    field = self.getField(name)
    if field is None:
        return SoBase.__setattr__(self, name, value)
    field.setValue(value)
    return field
%}
```

Listing 3.13: Python attributes special methods for assignment operator overloading

This solution has however one major drawback: it is currently very slow! This code needs to be run for every single field access invocation and Python itself adds calling overhead.

We can profile the code snippets in the next listings through usage of the standard Python profiler. The first code snippet is using the direct **setValue()** interface and the second snippet the field assignment operator.

```python
## First code snippet using setValue()
from pivy.coin import SoMaterial

s = SoMaterial()
d = s.diffuseColor
for i in range(100000):
  d.setValue(0.1,0.2,0.3)

## Second code snippet using field assignment
from pivy.coin import SoMaterial

s = SoMaterial()
for i in range(100000):
  s.diffuseColor = (0.1,0.2,0.3)
```

Listing 3.14: Profiling assignment operator overhead

Listing 3.15 shows how we invoke the Python profiler for both code snippets:

```
$ for snippet in snippet{1,2}.py; do
> python -m profile -s time $snippet
> done
```

Listing 3.15: Invoking the Python profiler ordered by internal time

As we can see in Table 3.1 and 3.2, the second code snippet, which uses our assignment operator overload, takes ~**25** seconds with most of the time spent in the concerningly slow **getField()** method invocation, compared to the one that uses the direct **setValue()** invocations which needs ~**1.5** seconds. Therefore heavy usage of the field assignment operator is currently discouraged unless used for interactive sessions or performance uncritical parts such as the construction of the scene graph at startup time.

The current solution for the assignment operator overload handling needs to be optimized as the current **getField()** implementation takes too much time. This also exemplifies the type of main problems we need to deal with, if we had chosen a fully runtime based approach as discussed in the *"Manual wrapping through runtime system usage"* section on page 29.

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
100000	0.800	0.000	0.800	0.000	coin.py:33555(setValue)
1	0.360	0.360	1.450	1.450	snippet1.py:2(?)
1	0.150	0.150	0.210	0.210	coin.py:10(?)
1	0.060	0.060	0.280	0.280	__init__.py:17(?)

Table 3.1: Snippet 1: 113258 function calls in 1.450 CPU seconds

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
100006	18.570	0.000	21.000	0.000	coin.py:3659(getField)
100000	1.470	0.000	1.470	0.000	coin.py:33555(setValue)
100006	0.970	0.000	23.440	0.000	coin.py:3777(__setattr__)
1	0.940	0.940	24.970	24.970	snippet2.py:2(?)

Table 3.2: Snippet 2: 913251 function calls in 24.970 CPU seconds

3.2.8 Callback handling

A very important part of the OpenInventor API is callback handling. It should be possible to create callback handlers in Python and tell Coin to invoke those. In order to make that work, we need a kind of *proxy* function that gets registered on the C++ side and which takes care to forward invocation requests to the right *callable* Python object.

To solve that problem we make use of *SWIG* **typemaps**, the **%extend** directive and a *proxy* function, which acts as our callback handler.

We will demonstrate how to handle callbacks considering the **SoCallback** case as example. The **SoCallback** class defines a method named **setCallback** that has the following signature:

```
void SoCallback::setCallback(SoCallbackCB * function,
                             void * userdata = NULL);
```

The **SoCallback::setCallback()** methods sets up the **function** to call at traversal of this node. **userdata** will be passed back as the first argument of the callback function.

The callback **function** should have the following signature:

```
void mycallback(void * userdata, SoAction * action);
```

We extend the SoCallback interface with our own **setCallback()** method. This one will provide the original Coin **setCallback()** methods with our *proxy* function callback handler (line 38) and construct an *argument tuple* that contains the actual Python callable object *pyfunc* and *userdata* arguments (line 39-41) that the user provided. This will be looped through the *userdata* argument of our original Coin **setCallback()** method to our *proxy* function callback handler.

```
36 %extend SoCallback {
37   void setCallback(PyObject *pyfunc, PyObject *userdata = NULL) {
38     self->setCallback(SoPythonCallBack,
39      (void *)Py_BuildValue("(OO)",
40                   pyfunc,
41                   userdata ? userdata : Py_None));
42   }
43 }
```

Listing 3.16: Extending the SoCallback::setCallback method

The proxy function callback handler, named **SoPythonCallback**, will convert the **action** argument passed to a new *SWIG* structure through usage of the **SWIG_NewPointerObj()** function (line 8). It extracts the Python callable object (line 10) out of our previously created tuple that we looped through and are now provided with in the **userdata** argument. It constructs the argument list for the Python callable object (line 11) that will follow the same signature as in the C++ **mycallback** callback handler case above. We pass the **userdata** as the first argument, gathered from the original **setCallback()** invocation in Python, and pass the recently converted original **action** object as the second argument. We then evaluate our Python callable object (line 13) at which point our Python callback handler gets invoked.

```
2  static void
3  SoPythonCallBack(void * userdata, SoAction * action)
4  {
5    PyObject *func, *arglist;
6    PyObject *result, *acCB;
7
8    acCB = SWIG_NewPointerObj((void *)action, SWIGTYPE_p_SoAction, 0);
9
10   func = PyTuple_GetItem((PyObject *)userdata, 0);
11   arglist = Py_BuildValue("(OO)", PyTuple_GetItem((PyObject *)userdata, 1), acCB);
12
13   if ((result = PyEval_CallObject(func, arglist)) == NULL) {
14     PyErr_Print();
15   }
16
17   Py_DECREF(arglist);
18   Py_DECREF(acCB);
19   Py_XDECREF(result);
20 }
```

Listing 3.17: The proxy function callback handler

It is now very important to make sure that no memory leaks occur in this process by proper and correct deallocation (line 17-19) as callback handlers are likely to be invoked many times in the lifetime of a Coin application and leaking can render an application useless.

```
%typemap(in) PyObject *pyfunc {
  if (!PyCallable_Check($input)) {
    PyErr_SetString(PyExc_TypeError, "need a callable object!");
    return NULL;
  }
  $1 = $input;
}
```

Listing 3.18: The proxy function callback handler

The last task that remains to be done is to specify a typemap (Listing 3.18) which checks if the user provided an callable object and raises an corresponding exception in the case it was not.

As the **Sensors** are not **SoBase** derived classes and therefore do not provide the required **SoBase::getTypeId()** to query the type from the Coin runtime type system, casts through the provided **cast()** function have to be provided still manually in the callback handlers. A simple solution that has been implemented, is to provide a third item, which represents the type string (e.g. **SoNodeSensor**, **SoAlarmSensor** etc.) in the Python tuple that gets passed in the *"void * data"* argument of the callback handlers. This allows the callback handlers to register and invoke the casting function with the right type as we always know where the actual invocation came from. At the same time this permits to implement a generic solution, which works for all **Sensor** nodes.

3.2.9 Mapping C++ data types to Python

As stated in the introduction of this chapter data type conversions are a very important part of the binding. However, in order to make a binding more suitable, certain data types or classes found in the Coin C++ API should be mapped on to Python datatypes. Especially when they allow programs written in the target language to integrate better with the overall syntax.

The decision has been made to keep the original classes with their C++ API exposed in order to allow easier conversion of Python programs to C++. This is helpful for usage of Pivy as a prototyping tool or for use cases where performance is a concern, as those additional type conversions require additional memory copies, which have an impact on overall execution performance.

Table 3.3 shows some of the mapped basic types. As we have seen in the "SbName typemap example" section global checks are available, which allow those type conversions to take place transparently without changing the API. Extending the API in such a non-intrusive way offers further flexibility and choice on the user side through providing an explicit and simple pattern to follow. Most importantly: the original documentation for C++ stays valid, which smoothes the learning curve.

C++ basic type	**Python**
SbColor	(fff) tuple
SbMatrix	((ffff)(ffff)(ffff)(ffff)) tuple
SbName	String
SbRotation	(ffff) tuple
SbString	String
SbTime	float tuple
SbVec2d	(ii) tuple
SbVec3f	(fff) tuple

Table 3.3: Some examples of Python mapped Inventor basic types

3.2.10 Special treatment for unsigned char *

The **“unsigned char *”** data type deserves special mentioning as at first sight it might be assumed that it can be simply handled as a C character string. This particular datatype gets for example used in the **SbImage** basic type which is an abstract datatype for 2D and 3D images. To handle **“unsigned char *”** as a string and therefore using the Python String functions **PyString_FromString** and **PyString_AsString** to process them will not work as they can contain **’\0’** characters[4]. The memory areas pointed to by **“unsigned char *”** pointers contain rather C arrays with a certain length.

Python lacks a designated Array data type in the Python C API. However, various efficient numerical array implementations exist, such as NumPy[5] and numarray[6] exist but have not yet been integrated into the standard Python distribution and it remains uncertain when and if that will happen. They provide their own Python data types and C APIs that can be used but create additional library dependencies.

As data still is required to be copied in memory forth and back from the C++ side to the Python side, even if their respective C APIs are used[7], the decision was therefore made to use the **PyString_AsStringAndSize()** function of the Python C API. The **PyString_-AsStringAndSize()** function allows to treat a memory region as a string but copies the region specified by the **size** argument disregarding any **’\0’** characters. Additional usage of *SWIG* **typemaps** and the Coin API allow to extract and gather the length for their transformation. The numerical extensions can be still made use of from the Python side[8] and no additional library dependencies are created. This helps in deployment and allows Pivy to stay general and to integrate with a wider variety of use cases. Should an array implementation get part of the standard Python distribution the typemaps can be quickly adjusted to make use of this new array facility. If it is required for special purposes[9], the involved typemaps can always be adjusted to interface to one of the currently existing array implementations.

3.3 Bridging to the SoGui bindings

The final section of this chapter demonstrates how the different toolkit bindings, among other tasks responsible for OpenGL context setup, have been wrapped. Furthermore, the approach of creating a bridge to existing Python toolkit bindings, such as PyQt[10], will be discussed. The examples will focus on the SoQt integration as a bridge to the aforementioned PyQt Python binding has been developed.

[4] which are interpreted as string terminators in C
[5] `http://numeric.scipy.org/`
[6] `http://www.stsci.edu/resources/software_hardware/numarray`
[7] which is a big performance concern for very large arrays
[8] as they still appear as strings on the Python side
[9] in fact it has been considered to be done for the SIM Voleon Python module
[10] a Python binding for the Qt GUI toolkit

3.3.1 SoGui modules

To allow for modularization and better resource usage all satellite libraries such as the SoGui libraries are contained in their own extension module. The *SWIG runtime support* allows external access to the runtime and allows to gather the type information through the **%import** directive, which will not create any wrapper code but instead provide the *SWIG* type system with the necessary type information.

```
%import(module=”pivy”) coin.i
```

Listing 3.19: Importing the pivy main interface in soqt.i

The main *SWIG* interface file for the SoQt toolkit is located in *interfaces/soqt.i.* Apart from the necessary *SWIG* specification it contains additional **typemaps** that deal with the conversion and routines for the bridge of the Qt classes to structures suitable for PyQt.

The next listing shows the line that specifies the name of the module and which package it should be located in; in our case: **pivy.gui**.

```
%module(package=”pivy.gui”) soqt
```

Listing 3.20: The SWIG module declaration in soqt.i

3.3.2 PyQt bridging

To allow Pivy users to make use of the native wrapped *Qt* classes such as **QWidget** or **QEvent** for further manipulation through the **PyQt** Python binding, we need to create a bridge that translates our *SWIG* structures to *SIP*[11] structure.

The bridge to PyQt really just deals with two aspects:

1. Direct access to and extraction of the C++ *Qt* pointer contained in PyQt.
2. Forth and back wrapping of the encapsulated pointers into corresponding *SWIG* or *SIP* structures.

Fortunately, both wrapper generators provide functions for direct access to the wrapped C++ pointers, which point to the actual memory locations of the actual *Qt* instances and allow the creation of new wrapper structures at runtime.

As *SIP* can be used as a Python module, which we import into our namespace to look up the necessary *SIP* **unwrapinstance(obj)** and **wrapinstance(addr, type)** functions. Through this manner we avoid the creation of an additional library dependency to the *SIP* library. Should the *SIP* import fail, we assume it has not been installed and *PyQt* support is not desired. We then simply continue to use our own *SoQt SWIG* structures.

The next listing shows the input typemap for the **Qevent** class, where we can see the import attempt of the **sip** module (line 175), where we first check if it can be been found in the

[11]as SIP is the wrapper generator that has been used to create the PyQt binding

global namespace (line 174). Should the import work we continue and try to get hold of the **unwrapinstance()** function (line 181) and check if it is a callable Python object (line 183). We then invoke it with our **$input** argument, which is then a *SIP* structure created by *PyQt* in order to get hold of the actual pointer address to the wrapped C++ instance (line 186). This address contained in a Python long object, we forcibly cast to a **QEvent *** (line 226) and the result of this operation is our **QEvent** class instance that we can pass to *SoQt* in **$1** for further processing.

Should the import fail, it will raise an **PyExc_ImportError** exception in which case we know that no **sip** module can be imported and that **$input** is a *SWIG* and not a *PyQt SIP* structure. We therefore proceed normally, clear the exception (line 235) and extract the pointer to the **QEvent *** instance through **SWIG_ConvertPtr()** (line 236).

```
169 %typemap(in) QEvent * {
170   {
171     PyObject *sip;
172
173     /* check if the sip module is available and import it */
174     if (!(sip = PyDict_GetItemString(PyModule_GetDict(
          PyImport_AddModule("__main__")), "sip"))) {
175       sip = PyImport_ImportModule("sip");
176     }
177
178     if (sip && PyModule_Check(sip)) {
179       /* grab the unwrapinstance(obj) function */
180       PyObject *sip_unwrapinst_func;
181       sip_unwrapinst_func = PyDict_GetItemString(PyModule_GetDict(
            sip), "unwrapinstance");
182
183       if (PyCallable_Check(sip_unwrapinst_func)) {
184         PyObject *arglist, *address;
185         arglist = Py_BuildValue("(O)", $input);
186         if (!(address = PyEval_CallObject(sip_unwrapinst_func,
              arglist))) {
187           PyErr_Print();
188         } else if (PyNumber_Check(address)) {
189           $1 = (QEvent*)PyLong_AsLong(address);
190         }
191
192         Py_DECREF(arglist);
193       }
194     }
195   }
196
197   if (PyErr_ExceptionMatches(PyExc_ImportError) || !$1) {
198     PyErr_Clear();
```

```
    if ((SWIG_ConvertPtr($input, (void **)(&$1), SWIGTYPE_p_QEvent,
        SWIG_POINTER_EXCEPTION | 0)) == -1) SWIG_fail;
  }
}
```

Listing 3.21: The QEvent * input typemap

3.3.3 Decoupling the main loop for interactive sessions

We need to *"decouple"* the main loop of the SoQt binding in order to allow interactive usage out of the Python interpreter, when the **SoQt::mainLoop()** is invoked as it otherwise will not allow the user to type in any commands for further evaluation.

We do that through the creation of our own **mainLoop()** which overrides the original one. We then check if the Python interpreter is an interactive one by inspecting the value contained in **sys.argv[0]** which is then be empty (line 6). We spawn a new thread through the Coin thread abstraction function **cc_thread_construct**, which will invoke our **Pivy_-PythonInteractiveLoop()** helper function (line 7). Right after that we start the real **SoQt::mainLoop()** (line 8) and once this exits stops and destructs the allocated thread (line 9-11) and exits the Python interactive loop (line 12) at which point we return to our original Python interactive loop in the main thread. In the other case where a Pivy program has been regularly invoked from the command line, i.e. non interactively, we just call the **SoQt::mainLoop()** as usual.

```
 1 %extend SoQt {
 2   static void mainLoop() {
 3     PyRun_SimpleString("import sys");
 4     PyObject *d = PyModule_GetDict(PyImport_AddModule("__main__"));
 5     PyObject *result = PyRun_String("sys.argv[0]", Py_eval_input, d
         , d);
 6     if (!strcmp(PyString_AsString(result), "")) {
 7       cc_thread *py_thread = cc_thread_construct(
           Pivy_PythonInteractiveLoop, NULL);
 8       SoQt::mainLoop();
 9       void *retval = NULL;
10       cc_thread_join(py_thread, &retval);
11       cc_thread_destruct(py_thread);
12       Py_Exit(0);
13     } else {
14       SoQt::mainLoop();
15     }
16   }
17 }
```

Listing 3.22: The extended SoQt mainLoop()

As we mentioned above, we need to provide a helper function instead of directly invoking the **PyRun_InteractiveLoop()** function provided by the Python C API. The reason for that is that the function signature does not match with the one needed for the Coin **cc_thread_construct()** thread handler signature.

```
static void *
Pivy_PythonInteractiveLoop(void *data) {
  PyRun_InteractiveLoop(stdin, "<stdin>");
  return NULL;
}
```

Listing 3.23: The Pivy_PythonInteractiveLoop() helper function

This solution has however a major pitfall: thread safety. As neither Qt nor Coin is thread safe without further adoing, certain invocations in the Python interpreter can corrupt data structures and lead to crashes. Therefore, a better solution needs to be implemented, where the Python interactive loop is invoked in the same thread as the *SoQt* application. This can be done through waiting for and reading the next interactive input line from the Pivy user in a thread. Once the line is complete and the Python code can be evaluated a flag is raised and *SoQt* asynchronously notified that it should execute the ready Python code in its own thread.

Chapter 4

Programming with Pivy

This chapter explains how to write Coin programs with Pivy, names some of the syntactical differences to the C++ API, shows how Pivy can be used for debugging and Rapid Application Development and how new Coin extensions nodes can be integrated and made use of.

4.1 Pivy differences reg. the C++ API

Apart from the obvious syntactic differences between Python and C++ the differences regarding the API are rather small. The only differences usually are in the way arguments are passed and values are returned, where the generated wrapper was tweaked in a way to provide a more *"pythonic"* approach and API.

As we have seen in the previous chapter, the basic types found in Coin have been mapped to corresponding suitable ones in Python. The **::setValue()** methods accept mapped Python types as input arguments, such as a Python list containing 3 items. Furthermore, mapped Python types can also be assigned to the Coin fields as demonstrated in the next listing for the **SoMaterial.diffuseColor** field:

```
>>> m = SoMaterial()
>>> m.diffuseColor.setValue((0,0,1))
>>> m.diffuseColor
<pivy.coin.SoMFColor; proxy of C++ SoMFColor instance at
    _a4bd0908_p_SoMFColor>
>>> m.diffuseColor.getValues()
[<pivy.coin.SbColor; proxy of C++ SbColor instance at
    _a8be0908_p_SbColor>]
>>> m.diffuseColor.getValues()[0].getValue()
(0.0, 0.0, 1.0)
>>> m.diffuseColor = (1,0,1)
>>> m.diffuseColor.getValues()[0].getValue()
(1.0, 0.0, 1.0)
```

Listing 4.1: Field value assignment in an interactive session

Callback handlers for classes such as **SoTimerSensor** do not need to be functions. In fact they can be any Python object that is callable, such as methods or lambda expressions.

```
>>> from pivy.coin import *
>>> class Handler:
...   def callback_handler(self, node, sensor):
...     print "callback_handler called!"
...
>>> handler = Handler()
>>> ts = SoTimerSensor(handler.callback_handler, None)
>>> ts.setInterval(1.0)
>>> ts.schedule()
```

Listing 4.2: A Python method as a callback in an interactive session

Methods, where the arguments are either treated as output or input/output arguments, are returned as result tuples. Method signatures, which require the length of data types contained in an argument to be specified, have been changed when the length can be deducted from the passed Python type itself. Methods, that return a **SbBool** to signal failure or success after the execution of a method and further return results in the case of success, will return a result tuple containing the output parameters or a Python None object. Example of such methods are discussed in the following listings:

The **SoFieldContainer::getFieldName()** in C++ reads as follows:

```
SbBool getFieldName(const SoField *const field, SbName &name) const
```

Listing 4.3: SoFieldContainer::getFieldName() C++ signature

SoFieldContainer::getFieldName() finds the name of the given field and returns the value in the **name** argument, it returns **TRUE** if the field is contained within this instance, and **FALSE** otherwise.

In order to make this invocation more *pythonic* the signature of this method has been changed to the following for Pivy:

```
PyObject * getFieldName(SoField * field);
```

Listing 4.4: SoFieldContainer::getFieldName() Python signature

The method returns the **name** argument directly as a Python string, in order to avoid the second indirection and additional namespace lookup caused by returning it as an SbName, and signals the error case through returning the Python **None** type, resulting in the following usage:

```
>>> cone = SoCone()
>>> cone.getFieldName(cone.height)
'height'
```

Listing 4.5: SoFieldContainer::getFieldName() invocation in Python

Another example where multiple output arguments are returned as a Python tuple can be found with **SoSFImage::getValue()**.

```
const unsigned char * SoSFImage::getValue(SbVec2s & size, int & nc)
    const;
```

Listing 4.6: SoSFImage::getValue() C++ signature

SoSFImage::getValue() returns a pixel buffer and sets the **size** argument to contain the image dimensions and **nc** to the number of components in the image.

```
>>> s = SoSFImage()
>>> s.setValue(SbVec2s(1,1), 3, "abc")
>>> s.getValue()
('abc', <pivy.coin.SbVec2s; proxy of C++ SbVec2s instance at
    _f0e10908_p_SbVec2s>, 3)
```

Listing 4.7: SoSFImage::getValue() Python usage

As we can see in the above listing, Pivy returns the result arguments in a 3-item tuple, starting with the original output parameter, which is followed by the **size** and **nc** output arguments.

Another notable difference is that types in Pivy get automatically casted to the right type; a special **cast(object, 'castname')** function is exposed to handle bordercases.

4.2 Converting a simple example from C++ to Python

This section demonstrates, how an existing C++ source file can be converted to Python with very minimal effort as the syntax of both languages is sufficiently similar. The example source is an examiner viewer that loads a scene showing a caffeine molecule from a gzip compressed VRML file (Figure 4.1).

The following section will explain the C++ sources for this example by dissection.

In lines 1-6 we include the necessary C++ header files for the nodes and C++ classes, that we are going to use.

```
1 #include <Inventor/SoDB.h>
2 #include <Inventor/SoInput.h>
3 #include <Inventor/nodes/SoSeparator.h>
4 #include <Inventor/@Gui@/So@Gui@.h>
5 #include <Inventor/@Gui@/viewers/So@Gui@ExaminerViewer.h>
```

Listing 4.8: The readfile.cpp.in header includes

The *@Gui@* are substitution macros for the Coin *sogui-config* shell script, which allows to write generic code for usage with the different SoGui bindings provided by Coin. They free the developer from having to adapt the code for each GUI toolkit. The substitution is handled automatically during the build procedure in the *sogui-config* script as shown in the next listing.

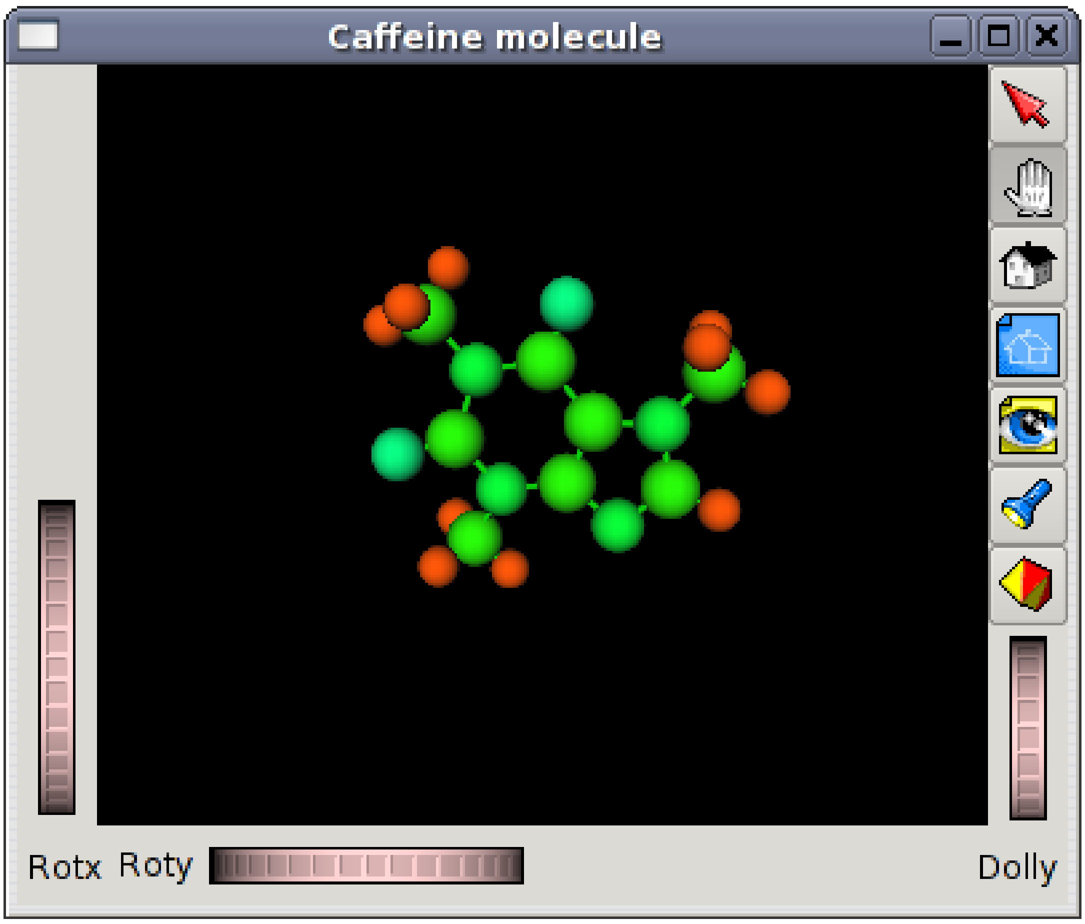


Figure 4.1: Examiner viewer showing a VRML caffeine molecule

```
$ soqt-config --build readfile readfile.cpp.in
soqt-config --ac-subst readfile.cpp.in
g++ -I/usr/local/include/Inventor/annex -D_REENTRANT -I/usr/qt/3/include \
    -g -O2 -c readfile.cpp -o readfile.o
g++ -I/usr/local/include/Inventor/annex -D_REENTRANT -I/usr/qt/3/include \
    -g -O2 -L/usr/local/lib -L/usr/qt/3/lib -o readfile readfile.o -lSoQt \
    -lqt-mt -lXmu -lXi -lCoin -lGL -lXext -lSM -lICE -lX11 -ldl -lpthread \
    -lm
```

Listing 4.9: Invocation of the soqt-config shell script

The next lines define a function *readFile()* which takes a **filename** to a Coin supported file format such as an *Inventor* or *VRML* file. The *readFile()* function creates an *SoInput()* instance on the stack and tries to open a file provided in **filename** through the SoInput::openFile() method. In line 16 the **SoDB::readAll()** method reads all graphs from the **SoInput** instance and returns them under an **SoSeparator** node. If the file contains only a single graph under an **SoSeparator** node[1], no extra **SoSeparator** root node will be made, but the returned root node will be the top-most node from the file. The file is closed[2] (line 22) and the function returns the allocated scene graph.

```
 7 SoSeparator *
 8 readFile(const char *filename)
 9 {
10     SoInput mySceneInput;
11     if (!mySceneInput.openFile(filename)) {
12         fprintf(stderr, "Cannot open file %s\n", filename);
13         return NULL;
14     }
15 
16     SoSeparator * myGraph = SoDB::readAll(&mySceneInput);
17     if (myGraph == NULL) {
18         fprintf(stderr, "Problem reading file\n");
19         return NULL;
20     }
21 
22     mySceneInput.closeFile();
23     return myGraph;
24 }
```

Listing 4.10: The readFile() function

[1]which is the most common way to construct and export scene graphs

[2]this could be left out, as this method will be triggered automatically by the **SoInput** destructor when the **SoInput** instance falls out of scope

In the main function of the program we initialize the SoGui binding and implicitly the Coin System at line 29.

```
29 @WIDGET@ myWindow = So@Gui@::init(argv[0]);
```

Listing 4.11: Initializing Coin

Note again the usage of the substitution macros that allow for a generic piece of code regarding the different SoGui libraries.

The next line invokes the *readFile* function and passes the gzip compressed VRML filename, which denotes the file that contains the caffeine molecule.

```
31 SoSeparator * scene = readFile("caffeine.wrl.gz");
```

Listing 4.12: Invoking the readFile() function

The rest of the program creates a so-called *"Examiner viewer"* and passes the parent window of the toolkit as the first parameter (line 33). Line 35 specifies the root of the scene graph that Coin should traverse, adds a title for the window in the next line and instructs it to show the widget as well as the parent widget in the next line (line 37-39). We finalize the setup by invoking the application's *mainLoop()*, which allows the user to begin interaction with the application.

```
33 So@Gui@ExaminerViewer * myViewer = new So@Gui@ExaminerViewer(
       myWindow);
34
35 myViewer->setSceneGraph(scene);
36 myViewer->setTitle("Caffeine molecule");
37 myViewer->show();
38
39 So@Gui@::show(myWindow);
40 So@Gui@::mainLoop();
```

Listing 4.13: Creation of the Examiner viewer

To convert this example to Python, we need to carry out some syntactical adjustments. Python uses the *semicolon (;)* to separate statements in one single line of code, but does not require it for the termination of a single statement in a line. A simple *search* and *replace"* operation allows to remove all *semicolons* within the source file. The next step is to replace all → C pointer dereferenciations symbols with the namespace delimiting *dot (.)*, as Python does not make use of pointers in its syntax. The same is done for any static class member function, where the *double colon (::)* is equally replaced by a namespace delimiting *dot (.)*.

So for example the line:

```
29 @WIDGET@ myWindow = So@Gui@::init(argv[0]);
```

is translated to:

```
22     myWindow = SoGui.init(sys.argv[0])
```

in Python and the line:

```
33 So@Gui@ExaminerViewer * myViewer = new So@Gui@ExaminerViewer(
     myWindow);
```

gets:

```
26     myViewer = SoGuiExaminerViewer(myWindow)
```

Python is a dynamically bound language and therefore does not require any type declarations for variables. Variables are simply introduced in the code through assignment, whenever they occur and are needed. For example, in the previous listing, the *@WIDGET@* type declaration is left out. Pivy features an SoGui abstraction layer, which is implemented as a proxy. The proxy uses runtime introspection to map the corresponding calls down to the native binding. Substitution macros are not required and Pivy code can be written irrespectively of the underlying installed GUI toolkit binding on the system, which allows the code to be run without modification or adaption to a specific toolkit.

Finally, the header file section is replaced by Python **import** statements, that will load the Pivy *coin core* and the *sogui proxy* modules.

```
4 from pivy.coin import *
5 from pivy.sogui import *
```

Listing 4.14 shows the whole C++ source code fully converted to Python. The separation into a main function is not really required and the following lines

```
34 if __name__ == "__main__":
35     main()
```

show a very typical and common practice Python idiom. Line 34 checks if *__name__* equals *__main__*; the *__name__* variable contains the name of a module, class, function/method in Python and if invoked "standalone" from the python interpreter - *__name__* is set to *__main__* in the global namespace rather than its module name. This allows to differentiate between "standalone" execution and the code imported as a module, where this line evaluates to false and where the main function does not get executed.

The remaining task is to remove (or comment out) all curly braces, that are used to delimit blocks in C/C++ as Python uses indentation for block delimiting. Optionally, a Python *"shebang"* (line 1) can be provided, which allows direct invocation of the script, when the execution bit is set, without the need to prepend it with the python interpreter on POSIX compliant systems.

```
1 #!/usr/bin/env python
2
3 import sys
4 from pivy.coin import *
5 from pivy.sogui import *
6
```

```python
def readFile(filename):
    mySceneInput = SoInput()
    if not mySceneInput.openFile(filename):
        print >>sys.stderr, "Cannot open file %s" % (filename)
        return None

    myGraph = SoDB.readAll(mySceneInput)
    if myGraph == None:
        print >>sys.stderr, "Problem reading file"
        return None

    mySceneInput.closeFile()
    return myGraph

def main():
    myWindow = SoGui.init(sys.argv[0])

    scene = readFile("caffeine.wrl.gz")

    myViewer = SoGuiExaminerViewer(myWindow)
    myViewer.setSceneGraph(scene)
    myViewer.setTitle("Caffeine molecule")
    myViewer.show()

    SoGui.show(myWindow)
    SoGui.mainLoop()

if __name__ == "__main__":
    main()
```

Listing 4.14: The converted Python readfile application

As can be clearly seen, the conversion and translation process is really straightforward, uncomplicated and inexpensive. The whole process can be reversed and therefore encourages the use of Python for Rapid Application Prototyping, even if the final application has to be developed and deployed in C++.

4.3 Rapid Application Development (RAD) and Prototyping (RAP) with Pivy

All too often during the development of an application it is necessary to quickly test out an idea or explore a solution through experimentation. Rapid Application Development (RAD) and Prototyping (RAP) is greatly hindered if immediate turnaround is not possible - as having to compile and restart the whole application after each modification forms a very time consuming obstacle that greatly reduces productivity.

This section will show how to make use of one of Python's most important and highly valued features: the interactive interpreter facility.

When the Python interpreter is started without passing any arguments, such as a file containing Python source code, the interpreter starts in *"interactive mode"*. Commands are read from a tty and it prompts for the next command with the primary prompt, usually three greater-than signs (">>> "); for continuation lines it prompts with the secondary prompt, by default three dots ("... "). The interpreter prints a welcome message stating its version number and a copyright notice before printing the first prompt:

```
$ python
Python 2.4.2 (#1, Dec 16 2005, 19:26:35)
[GCC 3.4.4 (Gentoo 3.4.4-r1, ssp-3.4.4-1.0, pie-8.7.8)
] on linux2
Type "help", "copyright", "credits" or "license" for
more information.
>>>
```

Listing 4.15: An interactive Python interpreter session

The interpreter can be exited by pressing the *Ctrl-D* key combination and features *readline*[3] support for convenient line editing on most platforms.

All lines, that are typed into an editor to form a Python program, can be run out of the interpreter without modification. Even more so, Python code can be organized into modules, which can be imported into the running interactive Python interpreter.

4.3.1 Interactive scene graph introspection

We can modify the program from the previous section to make it more suitable as a module and in order to allow the inspection and manipulation of the scene graph through its exposure. The easiest approach to get the scene graph exposed, is to change the scope of the *scene* variable, which references the root node, from local scope in the *main* function to a global scope in the *readfile* module. Through usage of the **global** keyword we let the Python interpreter know, that the variable encountered after the **global** keyword should be put into the global namespace. (Listing 4.16).

```
def main():
    global scene

    myWindow = SoGui.init(sys.argv[0])
    scene = readFile("caffeine.wrl.gz")
```

Listing 4.16: Telling the Python interpreter to put the scene into the global namespace

Now that everything is in place, we can proceed with the interactive session. We import Pivy's *coin* core module (line 1) into the global namespace in order to be able to make use of

[3]a free and unencumbered BSD-licensed readline replacement, named libedit, with readline emulation is available from the NetBSD (http://www.netbsd.org/) project.

the Coin nodes. This is followed by the import of the *readfile* module, which we keep in its own namespace (line 2). We call the *main()* function of the *readfile* module, which will let the window from Figure 4.1 pop up (line 3). Special measurements in Pivy's SoGui bindings have been added - as described in the previous chapter - in order to avoid blocking the Python interactive mode when the application enters the *mainLoop*. We then get hold of the root node, which we previously exposed globally in the *readfile* module namespace (line 4). After verifying that we got hold of the right node (line 5), we wish to dump and introspect the current scene graph. To achieve this, Coin provides a special *action*: the **SoWriteAction**. After instantiating an **SoWriteAction** (line 7) we apply the *action* on the scene graph's root node (line 8), which results in the current scene graph to be written out.

```
 1 >>> from pivy.coin import *
 2 >>> import readfile
 3 >>> readfile.main()
 4 >>> scene = readfile.scene
 5 >>> scene
 6 <pivy.coin.SoSeparator; proxy of C++ SoSeparator instance at
      _e0a23e08_p_SoSeparator>
 7 >>> wa = SoWriteAction()
 8 >>> wa.apply(scene)
 9 #Inventor V2.1 ascii
10
11
12 Separator {
13
14   Separator {
15
16     Material {
17       ambientColor 0.2 0.2 0.2
18       diffuseColor 0 1 0.1875
19       specularColor 0 0 0
20       shininess 0.2
21       transparency 0
22
23     }
24     Separator {
25       ...
```

Listing 4.17: Interactive scene graph introspection

In order to demonstrate the powerful capabilities, we continue the interactive session and add another object into the scene, that gets loaded from another file without the need to leave the interpreter or to re-edit existing code.

The next listing starts by the instatiation of an **SoInput** class, where we instruct it in the next line to open another gzip compressed VRML file in the same fashion as we did in the *readfile* module example from the previous section (line 1-2). However, the major difference is that we do it interactively by typing it into the running interactive session and are able to experience the effects of our actions immediately. We load the file through the *SoDB.readall()* invocation (line 3) and can reuse the already instantiated **SoWriteAction** to inspect the contents of this new object.

```
 1 >>> i = SoInput()
 2 >>> i.openFile(”mug.wrl.gz”)
 3 >>> mug = SoDB.readAll(i)
 4 >>> wa.apply(mug)
 5
 6 Separator {
 7
 8   Separator {
 9
10     Info {
11       string ”Object Creator: unknown”
12
13     }
14     ShapeHints {
15       vertexOrdering COUNTERCLOCKWISE
16       shapeType SOLID
17
18     }
19     Coordinate3 {
20       point [ -0.0300752 -0.0125313 -3.7593999e-09,
21           -0.0294179 -0.0125313 -0.0062529799,
22           -0.0274751 -0.0125313 -0.0122327,
23           ...
24     }
25   }
26 }
```

Listing 4.18: Interactive scene graph introspection cont. - adding a new object

We then add the new *mug* node to the existing scene graph (line 1) and notice that the object is too small, which causes the instantiation of an **SoScale** node (line 2) to remedy that. We insert it as the first node into the new object (line 3) and experiment with the *scaleFactor* field until we like the result (line 4-6). The final result is shown in Figure 4.2.

In order to save the result persistently, we write it to a new Inventor file by using an **SoOutput** class, where we set the filename the output should saved to (line 7-8). Finally, we apply an **SoWriteAction** on the entire *scene* again, but this time by specifying an output object the **SoWriteAction** should write its results to. We close the file of the **SoOutput** instance in order to flush the buffers and the result is a new scene, which we just created **interactively**.

```
1 >>> scene.addChild(mug)
2 >>> scale = SoScale()
3 >>> mug.insertChild(scale, 0)
4 >>> scale.scaleFactor = (10,10,10)
5 >>> scale.scaleFactor = (30,30,30)
```

```
>>> scale.scaleFactor = (40,40,40)
>>> o = SoOutput()
>>> o.openFile("coffemug.iv")
1
>>> SoWriteAction(o).apply(scene)
>>> o.closeFile()
```

Listing 4.19: Interactive scene graph introspection cont. - saving the results

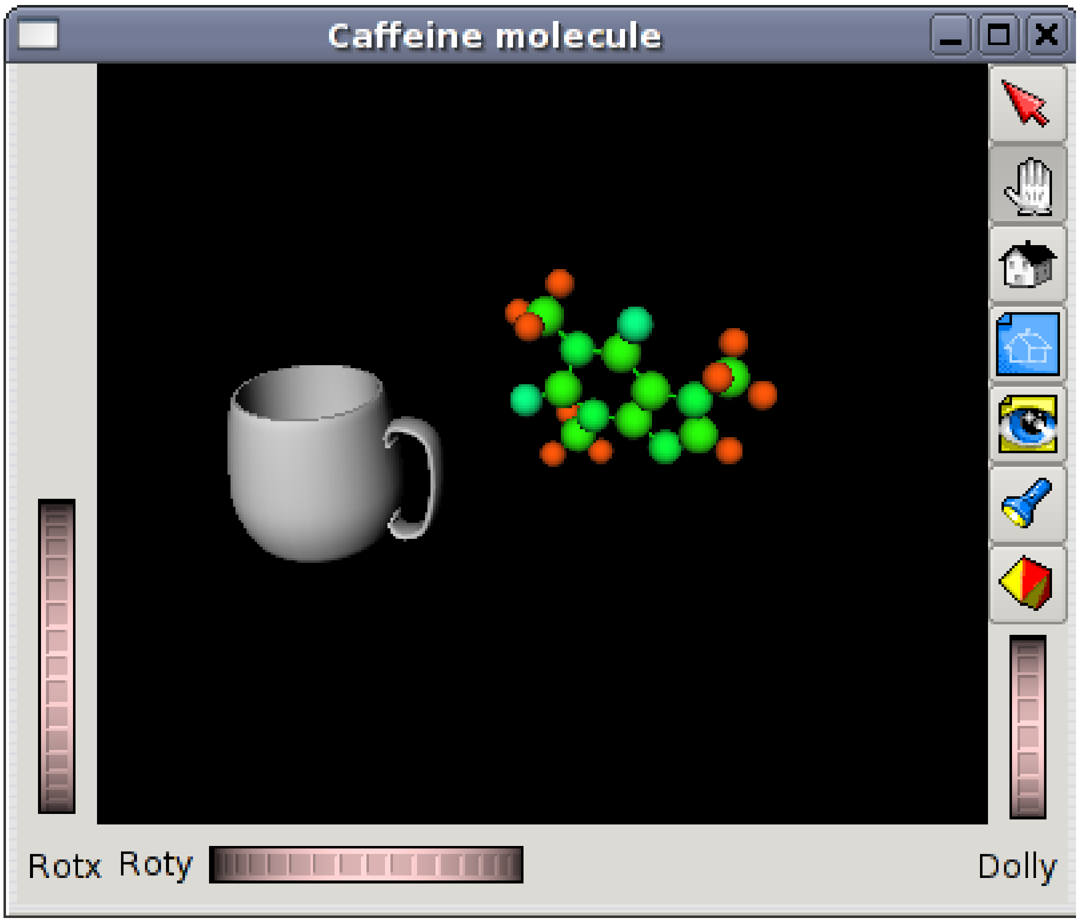


Figure 4.2: The result of the interactive session

4.3.2 Pivy as a debug and testing tool

The previous section outlined Pivy's usage as an interactive scene graph introspection and modification tool. In the same lines the whole available repertoire of the Coin library can be used to debug problems in the application itself.

A simple way to accomplish this, is presented in the next listing. The usage of the **SoNodeSensor** *sensor* node allows to track field and state changes in the nodes, that we are interested in. Their state can be written to the standard output or dumped into a logfile. This allows for a lot of additional flexibility, as filters or other behavior can be easily implemented in the callback handler; behaviors, such as interactive debugging of the application through the combination of various available *actions*, for example the **SoRayPickAction**.

```python
def rootChangedCB(void, mySensor):
  changedNode = mySensor.getTriggerNode()
  changedField = mySensor.getTriggerField()

  if changedNode:
    print "The node named '%s' changed" % (changedNode.getName().
        getString())

  if changedField:
    fieldName = changedNode.getFieldName(changedField)
    print " (field %s)" % (fieldName)
  else:
    print " (no fields changed)"

sensor = SoNodeSensor(rootChangedCB, None)
sensor.setPriority(0)
sensor.attach(scene)
```

Listing 4.20: Tracking node and field changes

Another interesting Pivy use case, is the creation of **black-box-** and **white-box-** testing frameworks for Coin applications. The test cases can be written in a similar fashion as the Pivy unit tests and where they test out different code execution paths of the application and verify the results. Through the availability of immediate turnaround, the simple syntax and flexibility of a *dynamic* language, the tedious task of writing test cases becomes much easier compared to the same task in a *statically typed* system programming language.

4.4 Extending Pivy with new Inventor nodes or nodekits

Sometimes, the functionality offered through the available and provided nodes is simply not sufficient and new specialized functionality needs to be implemented. Coin has an extension architecture built around convenience macros and the runtime type system, which allows for the creation of new nodes in C++.

There are two ways to gain access to the new functionality of this new Coin extension nodes in Pivy:

1. Gaining access to and control the new nodes through the fields, which can be gathered and queried through Coin's runtime type system.
2. Creating *SWIG* interface files in order to create a wrapper contained in a separated module for them.

The simplest way to make use of new extension nodes that have been designed to expose and control their functionality through their fields is to use the information provided by the runtime type system. The nodes can then be linked in or loaded dynamically[4] where node instances will then be autocasted/typed to the first class that is known (wrapped) in the inheritance hierarchy through Pivy's autocasting mechanism. For example, if we have to access a new shape node, named **SoMouse**, which inherits from **SoCone**, then once encountered, Pivy will autocast it automatically to an **SoCone()**. Through the methods in **SoFieldContainer** introspection/querying/manipulation of the nodes fields through the Inventor runtime type system is possible. Should the node **SoMouse** be inherited as follows: **SoMouse→SoHemiSphere→SoCone** then Pivy takes care again to make sure that **SoMouse** will be casted to SoCone(). If a *SWIG* wrapper has been created for **SoHemiSphere** and imported before the **SoMouse** shape node is encountered, then the **SoMouse** node will be autocasted to a **SoHemiSphere** node instead.

If new functionality needs to be accessed through methods in the extension node classes and field access on its own is not sufficient, then it becomes necessary to create a complete wrapper for these kind of basic types, fields, nodes, nodekits or libraries. This is especially true, when new field and basic types have been developed or when it is a Coin based library altogether for which access is desired.

In order to create a wrapper - the same principles apply and similar measurements have to be undertaken as has been outlined in the previous chapter with the important distinction that the wrapper interface file will be much simpler and quite minimalistic, as it can reuse a lot of the already available *SWIG* typemaps for the basic types and fields.

We demonstrate the creation of a new type on the basis of the provided example from the Pivy source distribution, which can be found in the *examples/extend/* directory.

The extension node kit consists of a header file and source file of which both need to be available as the extension node needs to be compiled into the wrapper in order for the linker to be able to resolve the symbols from the extension node.

The *shapescale.i* interface file starts with a global module *docstring* definition, which is then used in the *%module* directive which tells *SWIG* about the module name and passes the *docstring* definition as an parameter (line 7). We include the header files required to compile the wrapper and can reuse the *coin_header_includes.h* header file from Pivy's *interfaces/* directory. Furthermore, some platform specific tweaks have to be applied for the proprietary Windows platform in order to allow the wrapper to compile there as well.

We then *%include* the *pivy_common_typemaps.i* *SWIG* interface file, which collects *typemaps* that are common to and should be present in every Pivy interface file (line 24). The next line *%import*'s (note: do not use *%include SWIG* directive here) (line 27) the Pivy coin module in order to get access to all Coin types. The difference between the *%import* and *%include SWIG* directive is that *%import* does not generate actual wrapper code but rather collects information about the involved types and inheritance hierarchy of the classes. If we make use of *%include* instead, it will generate a full Coin wrapper again, which is not what we want and will result in a very large wrapper module.

[4]Dynamic Loading of Extension Nodes - http://doc.coin3d.org/Coin/dynload_overview.html

The last line *%include*'s the *ShapeScale.h* header file (line 29) which contains all the declarations for the *ShapeScale* node out of which the wrapper generator will create the auto-generated wrapper code.

```
 1 %define SHAPESCALE_MODULE_DOCSTRING
 2 "The ShapeScale class is used for scaling a shape
 3 based on projected size.
 4 ..."
 5 %enddef
 6
 7 %module(docstring=SHAPESCALE_MODULE_DOCSTRING) shapescale
 8
 9 %{
10 #if defined(_WIN32) || defined(__WIN32__)
11 #include <windows.h>
12 #undef max
13 #undef ERROR
14 #undef DELETE
15 #endif
16
17 #undef ANY
18
19 #include "ShapeScale.h"
20 #include "coin_header_includes.h"
21 %}
22
23 /* include the typemaps common to all pivy modules */
24 %include pivy_common_typemaps.i
25
26 /* import the pivy main interface file */
27 %import coin.i
28
29 %include ShapeScale.h
```

Listing 4.21: The ShapeScale interface file shapescale.i

In order to build it, some kind of build system has to be used. The decision was made to use the cross-platform *SCons* build system for these purposes. The build description of *SCons* is written in Python, which is why we can easily query the compiler and linker flags necessary to build the Python extension module out of *distutils.sysconfig*. The build system should then invoke *SWIG* with the required *SWIG* flags[5] and compile then the wrapper. Listing 4.22 shows the most relevant parts of the *SConstruct* build description file, which has to be written by the developer and where this one can be used as a template.

[5] it is important to keep them in sync with the *SWIG* flags found in the main *distutils*-based *setup.py* build script

```
53 PIVY_ROOT = '../..'
54
55 env.Append(CPPPATH = ['.', PIVY_ROOT + '/interfaces', distutils.sysconfig.get_python_inc()])
56 env.Append(LIBPATH = [PIVY_ROOT, distutils.sysconfig.get_python_lib()])
57 env.Append(LINKFLAGS = distutils.sysconfig.get_config_vars().get('LINKFORSHARED', '').split())
58
59 SWIG_SUPPRESS_WARNINGS = '-w302,306,307,312,389,361,362,467,503,509,510'
60 env.Append(SWIGFLAGS = '-c++ -python -includeall -modern -D__PIVY__' +
61                        '-I. -I%s/interfaces -I%s/fake_headers -I/usr/local/include -I%s %s' %
62                        (PIVY_ROOT, PIVY_ROOT, INCLUDE_DIR, SWIG_SUPPRESS_WARNINGS))
63
64 lib = env.SharedLibrary('_shapescale', Split('shapescale.i ShapeScale.cpp'),
65                         SHLIBPREFIX = '',
66                         SHLIBSUFFIX = distutils.sysconfig.get_config_vars()['SO'],
67                         SWIGCXXFILESUFFIX = '_wrap.cpp')
```

Listing 4.22: The SConstruct build description for the ShapeScale extension module

The first part of the build description deals with adding the correct compiler and linker flags (line 53-57). The next part creates the option list that should be passed in the *SWIG* invocation (line 59-62). The final part tells *SCons* to create a shared library, which represents our Python extension (line 64-67). *SCons* features direct support for *SWIG*, where it is sufficient to provide the name of the *SWIG* interface file and any additional *SWIG* options that the *SWIG* binary should be invoked with. In our case *SCons* will invoke *SWIG* to generate the wrapper file called *shapescale_wrap.cpp* first and then will link the result of the compilation of *shapescale_wrap.cpp* and *ShapeScale.cpp* into a shared library, called *_shapescale*, with the platform specific suffix, that has been gathered from the *distutils.sysconfig* package.

Once the Python extension module is built, it can be imported as shown in the next listing:

```
from pivy.coin import *
from pivy.sogui import *
from shapescale import *
```

Listing 4.23: The Python import statements for the new extension module

Different to the regular Pivy core modules, but similar to Coin extension nodes, the node has to be initialized (for registration with the Coin runtime type system) through the invocation of its **ShapeScale.initClass** method. The **ShapeScale.initClass** is best conducted right **after** the **SoGui.init()**invocation.

```
    window = SoGui.init(sys.argv[0])
    ShapeScale.initClass() # init our extension nodekit
```

Listing 4.24: The Python extension module node initialization

After this has been successfully done the extension node can be used in the same fashion as with the other wrapped Coin nodes.

4.5 When not to use Pivy?

We conclude this chapter by naming some domains, where Pivy might not be the best choice to use as some of those tasks may be outside of Pivy's scope.

Python is a dynamic interpreted language; therefore like all other dynamic languages the execution speed can be an issue. As we have reasoned in chapter 2 due to the properties of a retained mode library Python matches such a paradigm quite well.

For most applications Python's execution performance will be more than sufficient, especially when it is used as a controlling language for application logic, where intensive computations are conducted on the C++ side.

Still, for applications where the biggest part of the execution time is spent in for example the callback handlers and large amount of data has to be processed in tight loops, Pivy and Python might be too slow. Furthermore, to factor out most of the parts to C/C++ Python extensions might be too much work and too time-consuming.

Another problem could arise, if the startup time of an application takes up most of the time in an applications life time. In this case the time saved by the lack of the recompilation step, is lost again through lengthy startup times and overall development time is negatively impacted in addition to the loss of immediate turnaround. Sometimes, this can be remedied by redesigning the platform to a more runtime based system featuring a component framework, where each component can be independently reloaded or restarted at runtime. Still, such a framework takes a lot of planning and time to develop and could not be feasible at all due to new constraints in regard to execution speed; a runtime system will have the tendency to execute more introspection related code that could significantly raise the amount of executed Python code.

If an application depends on threading for the necessary performance on multi-core or multi-processor systems or the need for asynchronous event handling, then Pivy currently does not offer proper thread support, Python further uses a *Global Interpreter Lock*, which limits threads to the execution on one CPU. This can be worked around through the implemention of the thread related parts on the C/C++ side or through usage of a forking process model combined with IPC (Inter Process Communication) facilities for synchronization and execution of the code out of multiple interpreters. The problem with this solution is that a forking process model works great on Unix systems, but due to the process creating overhead on other platforms, such as Windows, this approach might not be an option again because of cross-platform constraints. However, threading is a tricky business and if not really necessary should be avoided at best. For example, most of the involved libraries are not thread safe by themselves and locking/unlocking for the critical sections can become a very error-prone and tedious task by itself, very hard to debug without the necessary defensive coding techniques in place and could end up in dramatic performance losses and slower execution of the code altogether, when locking/unlocking is overdone or applied unwisely and incorrectly.

As we have seen, all of the problems mentioned above have the same problem in common: *execution speed*. Still, Python can be used for application prototyping or in an embedded fashion in the scene graph, as we will see in the next chapter. This allows developers to benefit from an overall improvement in development time through the provision of immediate turnaround. After the prototype has been implemented and the application deemed to slow or hard to optimize, the application can always be rewritten in C/C++, where the difficult and time consuming parts of the application logic and design have already been solved.

Chapter 5

Embedding Pivy

In the previous chapters we explained, how to make use of Pivy in applications solely written in Python. But what if we have an already existing application written in C++ and still desire to quickly test out an idea or extend the current application with code written in Python? Here is, where another prominent feature of Pivy comes in handy: *SoPyScript* - the Python scripting node.

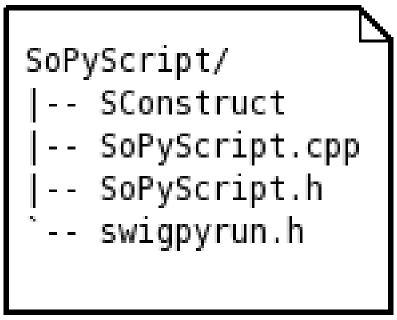
Through the combination of the extension mechanisms of Coin, that allow us to create a new scene graph node or node kit, and usage of both extending and embedding techniques of Python, we find the necessary foundation ready for the creation of a very flexible and powerful special purpose Coin node. The whole process is further simplified through the additional availability of introspection facilities, both in the Coin and Python libraries.

A few ideas and examples of what can be achieved through such a node in the scene graph, that just indicate the wide range of possible usage scenarios, are:

- embedding a web server, that outputs the current state of fields and nodes in a scene graph.
- use of remote procedure calls for the purpose of synchronization or feeding the scene graph state from external sources.
- writing parts of the scene graph, logging state changes or action traversals to a file for debugging or profiling purposes.

Additionally, nodes with new functionality can be created through the simple addition of the *SoPyScript* node into the scene graph. Furthermore, this can be achieved without the declaration of any macros, the creation of a corresponding header file or a compilation and linking step, which is unavoidable for newly created nodes in the C++ case.

Furthermore, scripts contained in the *SoPyScript* node can be reloaded in the application at runtime and override previous declared classes, methods, functions or variables, due to its interpretative nature, which provides a very flexible and powerful prototyping environment with immediate turnaround times.



```
SoPyScript/
|-- SConstruct
|-- SoPyScript.cpp
|-- SoPyScript.h
`-- swigpyrun.h
```

Figure 5.1: The file system hierarchy of the SoPyScript node

5.1 Design and implementation of the SoPyScript node

The design and implementation of the *SoPyScript* node is based on ideas found in the *VRML Script node* of the *VRML97* specification. However, there are some fundamental differences between the VRML Scripting and the *SoPyScript* node implementation regarding the approach and intended usage, which led to a couple of overall different design decisions:

The first and most obvious difference is that JavaScript - a special purpose embedding language - has been chosen as the VRML Scripting node language. This comes as no big surprise as VRML was also targeted to be used for the Web, which makes the decision to use JavaScript a natural one - as JavaScript was and is a popular and a readily available language in the domain of the World Wide Web.

Secondly, JavaScript is used in the VRML Scripting node to allow the definition and creation of simple application logic in order to allow for example event handling.

Finally, VRML does by itself not feature an API and was not intended to be exposed as a library. This is reflected in the design in how the scene graph can be modified, where the fields of a node are the single most important notion. Furthermore, nodes are added through accessing arrays which makes it a bit cumbersome to work with.

In contrast, Pivy uses Python - a fully fledged general purpose language. This allows to use the node for more than just simple event handling logic, as we:

1. have full access to the Coin and now additionally to a VRML API - as Coin is a library that provides and exposes an API for VRML,
2. can specify and override handlers for actions,
3. can use the Python interpreter module namespace to store state information,
4. are able to make full use of the other available Python modules installed.

Figure 5.1 shows the file system hierarchy of the SoPyScript node implementation, which consists of the following components:

SConstruct is the cross-platform SCons build description that allows to build the *SoPyScript* as a dynamic link library on any of the supported platforms.

SoPyScript.h and SoPyScript.cpp are the files containing the actual implementation of the *SoPyScript* node in the form of a regular Coin extension node. This node makes use of the Python C API to embed Python. The Coin extension code is heavily based on the Coin template of the VRML Scripting Node implementation.

swigpyrun.h is a header file containing the *"SWIG runtime"* code and declarations for required functions such as **SWIG_TypeQuery()** or **SWIG_NewPointerObj()** that allows to query and therefore make use of the Pivy SWIG types.

swigpyrun.h is SWIG version dependent, as it heavily relies upon the internal SWIG runtime code, that gets constantly improved and changed. This requires this header file to be regenerated for newer or different SWIG versions.

The external runtime mechanism was introduced after several iterations of SWIG runtime code improvements in SWIG version 1.3.25, where it was previously required to use tricks such as linking against a so-called *"libpivy_runtime"* dynamic link library, generated and built through an empty SWIG interface file, to get the linker resolve the symbols of the involved *"SWIG runtime"* functions.

The SWIG runtime code, which is automatically generated for SWIG wrapper code, can be created through the following invocation:

```
$ swig -python -external-runtime swigpyrun.h
```

Listing 5.1: SWIG invocation to create the external runtime header file

This discussion will now be followed with a dissection of the most important source code parts found in the *SoPyScript.cpp* Coin extension node implementation[1] and which explains the reasoning of their existence.

After we have included all the required Python, Coin and swigpyrun.h header files we will encounter a PYTHON_URLLIB_URLOPEN C processor macro definition (Listing 5.2). This Python code snippet is responsible for loading in a file through a given URL as the VRML specification allows.

```
// Python code snippet to load in a URL through the urllib module
#define PYTHON_URLLIB_URLOPEN "\
import urllib\n\
try:\n\
  fd = urllib.urlopen(url.split()[0])\n\
  script = fd.read()\n\
  fd.close()\n\
except:\n\
  script = None\n\
del url"
```

Listing 5.2: The PYTHON_URLLIB_URLOPEN macro

[1]the complete listing can be found in the Pivy source distribution...

The way it works is to inject the requested URL into an *url* variable into the global Python dictionary of the *SoPyScript* node when the script gets invoked in **SoPyScript::executePyScript()** (Listing 5.3).

```

// add the url to the global dict
PyDict_SetItemString(PRIVATE(this)->local_module_dict, "url", url);

PyObject * result;
result = PyRun_String(PYTHON_URLLIB_URLOPEN,
                      Py_file_input,
                      PRIVATE(this)->local_module_dict,
```

Listing 5.3: Executing the PYTHON_URLLIB_URLOPEN macro

A different solution is to write an URL parser and open a socket in C/C++ or to use an external library such as libcurl[2] for these purposes. However, apart from considerably complicating the code or adding another library dependency[3], we simply reuse functionality contained and provided in the standard Python implementation to our advantage. This is a very nice example, that demonstrates how an embedded Python interpreter allows to unleash Python's rich facilities for our own purposes and permits for the creation of advanced functionality in no time through the simple execution of Python scripts on the C++ side.

The next lines we encounter, are a *GlobalLock* class declaration (Listing 5.4). The *GlobalLock* class deals with Python's *Global Interpreter Lock (GIL)*, which needs to be acquired and released appropriately when Python code is to be executed. If this is not done, segmentation faults are likely to occur as the critical sections are not protected, especially when multiple Scripting nodes are in the scene graph or threads are used.

```
class GlobalLock {
  public:
    GlobalLock(void) : state(PyGILState_Ensure()) {}
    ~GlobalLock() { PyGILState_Release(state); }
  private:
    PyGILState_STATE state;
};
```

Listing 5.4: The GlobalLock class

This refactors the functionality into its own class and uses the implicit C++ scoping rules. Through the creation of a *GlobalLock* instance on the stack at the beginning of a block, the *GlobalLock* constructor will be executed and therefore retains the GIL. After the desired Python code is executed and the *GlobalLock* falls out of scope, the destructor gets invoked and releases the GIL again, which means that we cannot forget to release it. At the same time, it lessens the verbosity of the code and makes the code easier to maintain.

[2] http://curl.haxx.se/
[3] creating maintenance and deployment issues again

Next we implement the constructor of our private class *SoPyScriptP* (Listing 5.5) which follows Coin's Cheshire Cat/Bridge pattern technique in order to avoid problems in a library with a public API where binary compatibility (i.e. a stable ABI (Application Binary Interface)) between releases are important – as is the case with the Coin library. The pattern allows to hide all the private implementation details and data within an internal class, only visible to the C++ implementation file of the "real" class.

```
60 class SoPyScriptP {
61   public:
62     SoPyScriptP(SoPyScript * master) :
63       isReading(FALSE),
64       oneshotSensor(new SoOneShotSensor(SoPyScript::eval_cb, master)),
65       handler_registry_dict(PyDict_New()),
66       local_module_dict(PyDict_New())
67     {
68       if (!global_module_dict) {
69         Py_SetProgramName("SoPyScript");
70         Py_Initialize();
71         global_module_dict = PyModule_GetDict(PyImport_AddModule("__main__"));
72 
73         if (PyRun_SimpleString("from pivy.coin import *")) {
```

Listing 5.5: The SoPyScriptP private class constructor

In the *SoPyScriptP* constructor we initalize the Python C library (line 70) and get hold of Python's global dictionary (line 71) where we then import the main Pivy Coin module into our scripting node namespace (line 73). We initialize our *SoOneShotSensor* instance (line 64) and schedule a call for the execution of the **SoPyScript::eval_cb()** *SoOneShotSensor* callback handler through **PRIVATE(this)->oneshotSensor->schedule();**, which executes our Python scripts whenever **SoPyScript::notify()** notifies us about any field changes, so that the Python field handlers get evaluated. A second place, where our *SoOneShotSensor* gets scheduled and where we want our handlers to be invoked, is at the end of **SoPyScript::readInstance()**, which is responsible for parsing and initializing the Coin extension node once it is encountered when an Inventor file gets loaded in.

Another instance, which is worth mentioning, gets allocated and instantiated in the *SoPyScriptP* constructor: *handler_registry_dict*. For additional flexibility and in order to put no artificial restrictions regarding the mapping of the Coin field names to Python functions the *handler_registry_dict*, a regular Python dictionary, lets the user override the default field name mapping to any function name the developer desires[4]. The default field name mapping tries to find a Python function by the same name of the field, which is prepended with a **handle_** prefix, such as **handle_color** in order to avoid namespace problems.

We skip the rest of the private *SoPyScriptP* class implementation and mention the **SoPyScriptP::createPySwigType()** method, which works similar to the **cast()** function found

[4]an example in how this can be applied will be demonstrated in the next section

in Pivy. It is responsible for the conversion of the encountered C++ types into SWIG types usable by Pivy.

In Listing 5.6 we find the static **SoPyScript::initClass()** method which is responsible for registering the *SoPyScript* Coin type in the Coin runtime type system.

```
void
SoPyScript::initClass(void)
{
  if (SoType::fromName("SoPyScript").isBad()) {
    SoPyScript::classTypeId =
      SoType::createType(SoNode::getClassTypeId(),
                         SbName("SoPyScript"),
                         SoPyScript::createInstance,
                         SoNode::nextActionMethodIndex++);
```

Listing 5.6: The static SoPyScript::initClass() class method

An additional line has to be manually invoked in order to allow the *AudioRenderAction* to be handled in the *SoPyScript* implementation as it is not initialized and registered automatically by the default Coin type system creation method (reason: it is part of the VRML and not the original Open Inventor specification).

```
    SoAudioRenderAction::addMethod(SoPyScript::getClassTypeId(),
        SoNode::audioRenderS);
```

In order to allow Python handlers during *action* scene graph traverals to get invoked, we have to provide our own dispatch mechanism and routine, which is done and handled in the **SoPyScript::doAction()** method, which all virtual *action* handler methods will invoke.

It takes two arguments: **action** (of type *SoAction*) and **funcname** (an ordinary C character string).

```
SoPyScript::doAction(SoAction * action, const char * funcname)
```

Whenever an *action* now traverses the scene graph and invokes its virtual handler method according to the *visitor pattern*, our overridden handler method will then invoke our **SoPyScript::doAction()** method by providing the actual *action* instance and the name of the overridden method as shown in Listing 5.7, followed by the invocation of the original virtual method of the parent class[5].

```
void
SoPyScript::GLRender(SoGLRenderAction * action)
{
  SoPyScript::doAction(action, "GLRender");
  inherited::GLRender(action);
}
```

Listing 5.7: The overridden SoPyScript::GLRender() method

[5]in our case the one found in *SoNode*

In **SoPyScript::doAction()** we introspect our Python interpreter dictionary and check if a corresponding **funcname** is found and has been specified by the user.

```
188 PyObject * func = PyDict_GetItemString(PRIVATE(this)->
       local_module_dict, funcname);
```

If this is the case we convert the provided action from C++ to a SWIG Python object.

```
197 SbString typeVal(action->getTypeId().getName().getString());
198
199 PyObject * pyAction;
200 if (!(pyAction = PRIVATE(this)->createPySwigType(typeVal, action)))
       {
```

After we check that the found **funcname** is actually a callable Python instance, i.e. either a method or function, we construct the Python arguments (line 212) and invoke the function (line 214).

```
207 if (!PyCallable_Check(func)) {
208   SbString errMsg(funcname);
209   errMsg += " is not a callable object!";
210   PyErr_SetString(PyExc_TypeError, errMsg.getString());
211 } else {
212   PyObject * argtuple = Py_BuildValue("(O)", pyAction);
213   PyObject * result;
214   if (!(result = PyEval_CallObject(func, argtuple))) {
215     PyErr_Print();
216   }
217   Py_XDECREF(result);
218   Py_DECREF(argtuple);
219   Py_DECREF(pyAction);
220 }
```

We finalize that by invoking the parents node **SoNode::doAction()** method and are done.

```
229 inherited::doAction(action);
```

We then have to implement **SoPyScript::copyContents()**, **SoPyScript::notify()**, **SoPyScript::createInstance()**, **SoPyScript::getFieldData()** and **SoPyScript::initFieldData()** (where the *"static"* fields are initialized) methods as required for standard Coin node extensions.

The only exception is that in **SoPyScript::notify()** we are checking if the **script** field, which exposes our embedded Python script, has changed. This can occur whenever the developer decides to load a different script or modifies the embedded script at runtime. We then instruct the *SoPyScript* to re-evalute the whole script again, so that those requested changes take effect.

```
382 else if (f == &this->script) { this->executePyScript(); }
```

So how do we declare and allow new fields in Inventor files to be created dynamically at runtime when the *SoPyScript* is encountered and its format read, as fields typically have to be known and declared in advance at compile time through the provided Coin macros?

We make use of the **SoPyScript::readInstance(SoInput * in, unsigned short flags)** method which reads a definition of an instance from the input stream *in*. The input stream state points to the start of a serialized/persistant representation of an instance of this class type.

This exposure makes it basically possible to define and extend the Inventor file format for a node in any way we desire and allows to implement our own special parser. The decision was made to stick to the file format for *unknown Coin nodes*, which was an important decision. Those provide all the information (*fields*) we require and when saved in a standard format and later encountered in another application, lacking the *SoPyScript* node, the Coin library will still be able to parse it through the standard parser. Furthermore, external parsers such as the ones in syntax highlighters and Coin/VRML modeling tools will also be able to use the provided Coin file without having to do anything special.

```
SoPyScript {
  fields [ SoSFColor color ]
  color 1 0 0
  script "print 'color value: ', color.getValue().getValue()"
}
```

Listing 5.8: SoPyScript file format example

As demonstrated in Listing 5.8, we can specify our fields in fields section and can initialize and refer to them regularly afterwards.

```
422 // check for a comma at the end and strip it off
423 const SbString fieldname =
424   (name[name.getLength()-1] == ',') ?
425     name.getSubString(0, name.getLength()-2) : name;
426
427 // skip the static fields
428 if (fieldname == "script" || fieldname == "mustEvaluate") {
      continue; }
429
430 /* instantiate the field and conduct similar actions as the
431    SO_NODE_ADD_FIELD macro */
432 SoField * field = (SoField *)type.createInstance();
433 field->setContainer(this);
```

Listing 5.9: The SoPyScript::readInstance() method

The most important part of our overridden **SoPyScript::readInstance()** implementation is shown in Listing 5.9. First, we get hold of the fieldname (lines 422-425) and then check if the fieldname is one of the static fields we encountered, where we then just proceed in the **SoPyScript::readInstance()** and therefore skip them as they will be read in by the following

SoNode::readInstance() base class invocation at the end of our overridden method (line 441).

```
440 // and finally let the regular readInstance() method parse the rest
441 SbBool ok = inherited::readInstance(in, flags);
```

Finally, we dynamically create the fields and add them to the SoPyScript node (lines 432-433).

5.2 Embedding scripts in Inventor files

In the previous section we have seen a dissection of the code for the *SoPyScript* node. But how does one actually use it? This section will demonstrate along a simple example script what is possible and how the functionality provided can be used.

Listing 5.8 has already shown a simple script. In order to turn it into a valid Inventor file, we need to provide an Inventor header. We also extend it a little bit:

```
 1 #Inventor V2.0 ascii
 2 
 3 DEF root SoSeparator {
 4    SoPyScript {
 5       fields [ SoSFColor color ]
 6       color 1 0 0
 7       script "
 8 root = SoNode.getByName('root')
 9 cone = SoCone()
10 root.addChild(cone)
11 
12 print '== script loaded =='
13 "
14    }
15 }
```

In line 3 we specify an *SoSeparator* group node and give it a name in order to be able to refer to it later in the script. This is followed by an *SoPyScript* node declaration, where we again add a color field and initialize it to red. The script field contains our actual Python script, where we retrieve the root *SoSeparator* node through the invocation of **SoNode.getByName()**. We instantiate a cone shape node instance and append it as the second node to the root *SoSeparator* node (line 9).

In order to load and run this example, we need to create our own viewer and provide the *SoPyScript* dynamic link library in the search path for the *"dynamic loading of extension nodes"* mechanism[6]. We could create this viewer in Pivy, but the more common use case is to embed the *SoPyScript* in already existing C++ applications.

[6] http://doc.coin3d.org/Coin/dynload_overview.html

Listing 5.10 shows an implementation of such an ordinary *Examiner viewer* in C++, that expects an Inventor file to be passed as its first parameter.

```
#include <Inventor/SoInput.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoTransform.h>

#include <Inventor/Qt/SoQt.h>
#include <Inventor/Qt/viewers/SoQtExaminerViewer.h>

#include "SoPyScript.h"

int
main(int argc, char *argv[])
{
  if (argc != 2) {
    printf("Usage: %s file.iv\n", argv[0]);
    exit(1);
  }

  // initialize Inventor and Qt
  QWidget * window = SoQt::init(argv[0]);

  SoPyScript::initClass();

  SoInput * input = new SoInput();
  input->openFile(argv[1]);

  SoSeparator * root = new SoSeparator;
  root->ref();

  root->addChild(SoDB::readAll(input));

  // initialize an Examiner Viewer
  SoQtExaminerViewer * examinerViewer = new SoQtExaminerViewer(
     window);
  examinerViewer->setSceneGraph(root);
  examinerViewer->show();

  SoQt::show(window);
  SoQt::mainLoop();

  return 0;
}
```

Listing 5.10: C++ SoQt Examiner viewer that reads in a file specified in its first parameter

It is statically linked with the *SoPyScript* node implementation, which makes it necessary to include the *SoPyScript.h* header file and invoke **SoPyScript::initClass()** manually.

```
21    SoPyScript::initClass();
```

Line 23-24 load in the Inventor file:

```
23    SoInput * input = new SoInput();
24    input->openFile(argv[1]);
```

We add the resulting scene graph to our root node (line 29). As we already declared an encapsulating *SoSeparator* in the Inventor file, we could have chosen to leave out the instantiation of the additional *SoSeparator* in the viewer and directly assigned the results of the **SoDB::readAll()** invocation to the *SoSeparator* in the *Examiner viewer*. Another option is to skip the declaration in the Inventor file and to make use of a **root->setName("root");** invocation to give our root node a name, in order to be able to refer to it later. For our simple example this refinement does not matter and it is a bit safer to construct the scene graph this way.

```
23    SoInput * input = new SoInput();
24    input->openFile(argv[1]);
```

After successfully compiling and linking our *Examiner viewer*, where we in addition to the Coin and SoQt include path and compiler/linker flags have to provide the include path to the *Python.h* header file and the Python compiler/linker flags[7], we can finally invoke our script:

```
$ ./examin script.iv
== script loaded ==
```

The result of the invocation in the form of an *Examiner viewer* containing our added Cone is shown in the left picture of Figure 5.2.

Now this is actually still quite boring and we did not make use of the color field yet. So in oder to color the cone red, we prepend a *SoMaterial* node and feed its **diffuseColor** value with our color field. We then apply a *SoWriteAction* on the root node that will write out the current representation of our scene graph. This exemplifies the border-case for the autoref'ing mechanism in Pivy and the reason why it was decided against hiding the **ref()** and **unref()** methods of the SoNode in the wrapper, therefore keeping them exposed to the Pivy users. As the *SoWriteAction()* gets applied on a node that has been specified in the Inventor file and not created in Python, its reference count will be 0. We therefore have to **ref()** it manually as the root Separator will otherwise get deallocated after the *Action* traversal.

[7]both can be acquired through Python's distutils.sysconfig module

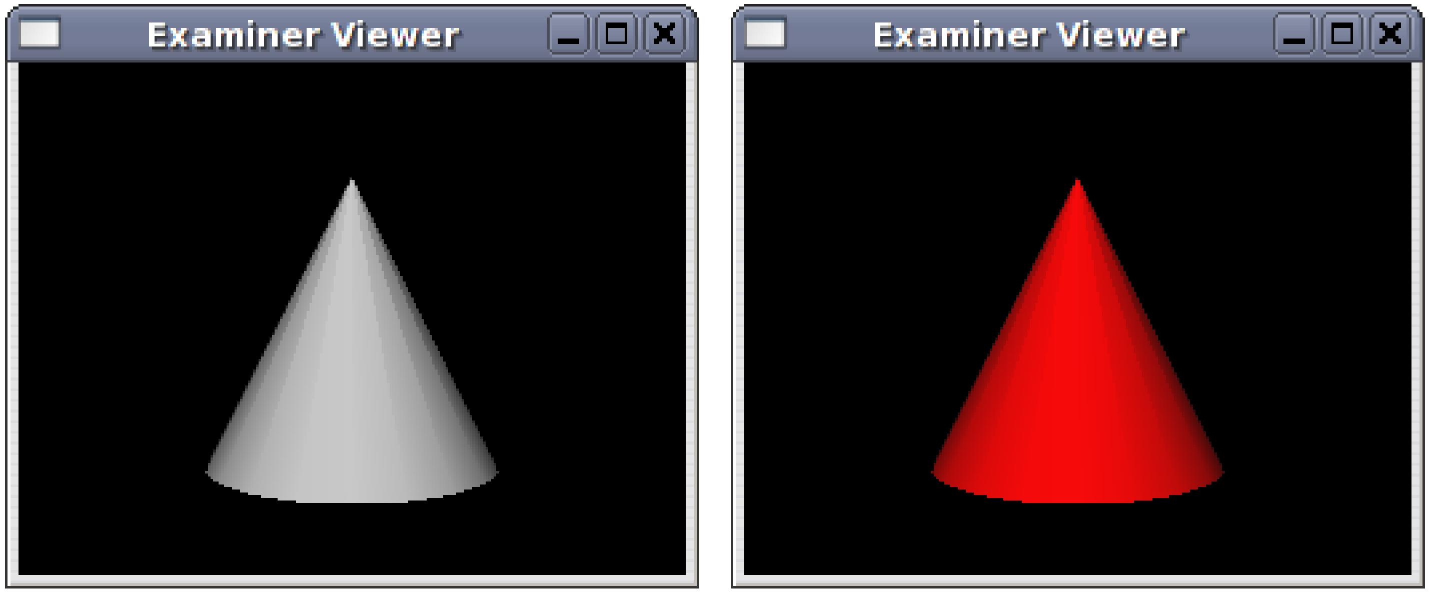


Figure 5.2: The results of the first and second SoPyScript invocation

```
#Inventor V2.0 ascii

DEF root SoSeparator {
   SoPyScript {
      fields [ SoSFColor color ]
      color 1 0 0
      script "
root = SoNode.getByName('root')
cone = SoCone()
root.addChild(cone)

mat = SoMaterial()
mat.diffuseColor = color.getValue()
root.insertChild(mat, 0)

root.ref()
wa = SoWriteAction()
wa.apply(root)

print '== script loaded =='
"
   }
}
```

The following listing shows the console output from the invocation of our second run of the modified script and the right picture in Figure 5.2 shows the result on the screen: a red cone. In the console output we can see, that there is no mention whatsoever yet about the *SoPyScript()* node as the execution of the script happens during the construction of the *SoPyScript()*.

```
$ ./examin script.iv
#Inventor V2.1 ascii

DEF root Separator {
  Material {
    diffuseColor 1 0 0
  }
  Cone {
  }
}
== script loaded ==
```

Python uses indentation for delimiting blocks of code. This not an issue at all in the *SoPy-Script* node as the script containing multiple lines can be written from the beginning of the line.

Another noteworthy fact is that the script can be factored out to its own file and referred to by an URL, which can be of any form the Python **urllib** module supports. For example:

```
# file reference back to the script
script "../SoPyScript/script.py"

# file reference using URL notation
script "file:./script.py"

# URL reference to a locally running web server
script "http://localhost/script.py"
```

The *SoPyScript* node checks if it can open the contents of the script through **urllib.urlopen()** in **SoPyScript::executePyScript()** before it proceeds and assumes it is a regular embedded Python script.

To factor out the actual script implementation has the benefit of enabling the usage of all features, such as syntax highlighting, offered in a Python IDE, while the scripts are developed. At the same time, this mechanism also makes the Inventor file less crowded, which improves readability. Moreover, the developer is able to create his own library of useful scripts for reuse in various nodes.

5.3 Using SoPyScript as a Coin node extension mechanism

We conclude the discussion about the *SoPyScript* node and this chapter by demonstrating another use case. The *SoPyScript* can be used to create new functionality, which can currently only be achieved through usage of the Coin extension mechanism in C++. The Coin C++ extension mechanism is unnecessarily involved, when everything we desire is just the manipulation of some element states in the scene graph during traversal. The author likes to refer to this desire as: *the creation of the* ***essence of functionality*** *of a complete Inventor Node by solely using the SoPyScript node.*

The following presentation will focus on the *SoPyScript* Pivy example, that can be found in the *examples/SoPyScript/* directory of the Pivy source distribution and is a direct adaption of the Glow example extension property node found in *"The Inventor Toolmaker"*[21].

The Glow property node modifies the emissive color of the current material in order to make objects appear to glow. A vector field called color represents the color of the glow; a float field called brightness, ranging from 0 to 1, indicates how much the object should glow. The actions that deal with materials need to be implemented: **GLRender()** and **callback()**. Since the actions perform the same operation the common code gets refactored into a **doAction()** method. The **doAction()** method updates the emissive color element based on the values of the color and brightness fields of the node.

In order to create this node we have to define a header file for the declarations and a source file for the implementation. In Listing 5.11 we see the header file for this node (with the original comments stripped out).

```
 1 #include <Inventor/SbColor.h>
 2 #include <Inventor/fields/SoSFColor.h>
 3 #include <Inventor/fields/SoSFFloat.h>
 4 #include <Inventor/nodes/SoSubNode.h>
 5
 6 class SoColorPacker;
 7
 8 class Glow : public SoNode {
 9   SO_NODE_HEADER(Glow);
10   public:
11   SoSFColor color;
12   SoSFFloat brightness;
13   SoSFFloat transparency;
14   static void initClass();
15   Glow();
16   protected:
17   virtual void GLRender(SoGLRenderAction *action);
18   virtual void callback(SoCallbackAction *action);
19   virtual void doAction(SoAction *action);
20   virtual ~Glow();
21   private:
22   SoColorPacker * colorPacker;
23   float transpValue;
24 };
```

Listing 5.11: The header file of the Glow property extension node

The header file starts out by including the required header files for the referenced classes. In line 9 we see the use of the required **SO_NODE_HEADER(Glow);** convenience macro that fills out declarations such as methods dealing with the Coin type system, static class member methods such as **Glow::createInstance()** or methods dealing with field access. This is followed in line 11-13 by declarations for the public fields that should be used. Then the obligatory static **Glow::initClass()** and constructor are declared. In the protected section the Action handlers that should be overridden are specified. The header file ends by declaring the protected destructor[8], the private **colorPacker** class and a helper variable **transpValue** for setting the transparency value.

```
#include <Inventor/actions/SoCallbackAction.h>
#include <Inventor/actions/SoGLRenderAction.h>
#include <Inventor/elements/SoLazyElement.h>
#include <Inventor/elements/SoOverrideElement.h>

#include "Glow.h"
```

[8]the destructor should never be directly invoked; instead the reference counting mechanism will take care of invoking the destructor after an **unref()** invocation results in a Node's reference count to drop to 0

```
 7
 8 SO_NODE_SOURCE(Glow);
 9
10 void
11 Glow::initClass()
12 {
13   SO_NODE_INIT_CLASS(Glow, SoNode, "Node");
14 }
15
16 Glow::Glow()
17 {
18   SO_NODE_CONSTRUCTOR(Glow);
19   SO_NODE_ADD_FIELD(color, (1.0, 1.0, 1.0));
20   SO_NODE_ADD_FIELD(brightness, (0.0));
21   SO_NODE_ADD_FIELD(transparency, (0.0));
22   colorPacker = new SoColorPacker;
23 }
24
25 Glow::~Glow() { delete colorPacker; }
```

Listing 5.12: The implementation file of the Glow property extension node - part I

The C++ implementation of this extension node is shown in Listing 5.12 and 5.13. Again we have to make use of a convenience macro, this time named **SO_NODE_SOURCE-(Glow);** which fills out the implementation for the methods that got declared through the **SO_NODE_HEADER(Glow);** macro in the header file. The **Glow::initClass()** and constructor of the Glow class further utilizes macros to initialize and register the class and fields with the Coin run time type system. The destructor just frees up the heap instantiated colorPacker instance; the actual functionality of the node is implemented in **Glow::doAction()**.

The **emissiveColor** color is defined as the product of the **brightness** and **color** fields which gets computed in line 44. The emissive color state of the **SoLazyElement** is then set to the one contained in the computed **emissiveColor** *SbColor* value. Similarly, the transparency state is set through the **transparency** field.

The **colorPacker** has to be provided when the transparency is set, so that the transparency will be merged with the current diffuse color in the state.

```
27 void
28 Glow::GLRender(SoGLRenderAction *action)
29 {
30   Glow::doAction(action);
31 }
32
33 void
34 Glow::callback(SoCallbackAction *action)
35 {
36   Glow::doAction(action);
37 }
```

```

void
Glow::doAction(SoAction *action)
{
  if (!brightness.isIgnored() &&
      !SoOverrideElement::getEmissiveColorOverride(action->getState())) {
    SbColor emissiveColor = color.getValue() * brightness.getValue();
    SoLazyElement::setEmissive(action->getState(), &emissiveColor);
  }

  if (!transparency.isIgnored() &&
      !SoOverrideElement::getTransparencyOverride(action->getState())) {
        transpValue = transparency.getValue();
        SoLazyElement::setTransparency(action->getState(),
                                       this, 1,
                                       &transpValue,
                                       colorPacker);
  }
}
```

Listing 5.13: The implementation file of the Glow property extension node - part II

This example needs then to be compiled and statically linked into the application that uses it or alternatively dynamically loaded through the dynamic loading mechanism of Coin.

Listing 5.14 presents the exact same functionality implemented as a Python script. Line 2 uses **floatp**, a conversion utility provided by SWIG, in order to allow the creation and the passing of pointer values to methods that require it - as is the case with **SoLazyElement.setTransparency()** method.

```
1  colorPacker = SoColorPacker()
2  transpValue = floatp()
3
4  def doAction(action):
5      global transpValue
6
7      if not brightness.isIgnored() and not SoOverrideElement.getEmissiveColorOverride(action.getState()):
8          emissiveColor = color.getValue() * brightness.getValue()
9          SoLazyElement.setEmissive(action.getState(), emissiveColor)
10     if not transparency.isIgnored() and not SoOverrideElement.getTransparencyOverride(action.getState()):
11         transpValue.assign(transparency.getValue())
12         SoLazyElement.setTransparency(action.getState(), self, 1, transpValue, colorPacker)
```

```

def GLRender(action):
    action.setTransparencyType(SoGLRenderAction.SORTED_OBJECT_BLEND)
    doAction(action)

def callback(action):
    doAction(action)
```

Listing 5.14: The implementation file of the Glow property extension node in Python

Listing 5.15 shows how we declare all the involved fields **color**, **brightness** and **transparency**. We initialize those values where the **color** field gets connected to the output of the **translation** field of an *SoShuttle* (line 11). We specify our script "glow.py"... **done**!

```
 9    DEF Glow SoPyScript {
10        fields [ SoSFColor color, SoSFFloat brightness, SoSFFloat transparency ]
11        color 1 0 0 = USE shuttle.translation
12        brightness 0.5
13        transparency 0.3
14        script "./glow.py"
15    }
```

Listing 5.15: Python Glow property extension node script usage out of an Inventor file

No need to specify header includes, macros and implement initialization routines. Furthermore, the **exactly** same functionality is provided in **19** lines of code instead of **80** lines of code as is the case with the C++ implementation.

Most importantly, we did not have to go through iterative recompile and link cycles, which gave us immediate turnaround in developing and testing out our new extension functionality!

The time saved in implementing this extension in Python through the use of the *SoPyScript* node and therefore allowed to just focus on the implementation of the actual functionality, allows to quickly enhance the Python example even further. An additional textscroller script has been implementing, which demonstrates the usage of embedding an *SoTimerSensor()* sensor, that invokes our embedded *changeStringSensorCallback()* function (Listing 5.16 and 5.17). Figure 5.3 shows various states of the glow and textscroller animation.

```
    DEF TextScroller SoPyScript {
        fields [ SoSFString string, SoSFColor color ]
        string "I am Pivy powered!"
        color 0 1 0
        script "./textscroll.py"
    }
```

Listing 5.16: The usage of the textscroller script out of an Inventor file

```
idx = 0
text = string.getValue().getString()
text_length = len(text)

interval = 0.15

def changeStringSensorCallback(data, sensor):
  global idx
  string.setValue(text[:idx])

  if idx == text_length:
    sensor.setInterval(5.0)
  else:
    sensor.setInterval(interval)

  idx %= text_length
  idx += 1

timersensor = SoTimerSensor(changeStringSensorCallback, None)
timersensor.setInterval(interval)
timersensor.schedule()

string.setValue(text[:idx])

print '== TextScroller script loaded =='
```

Listing 5.17: The implementation file of the textscroller script

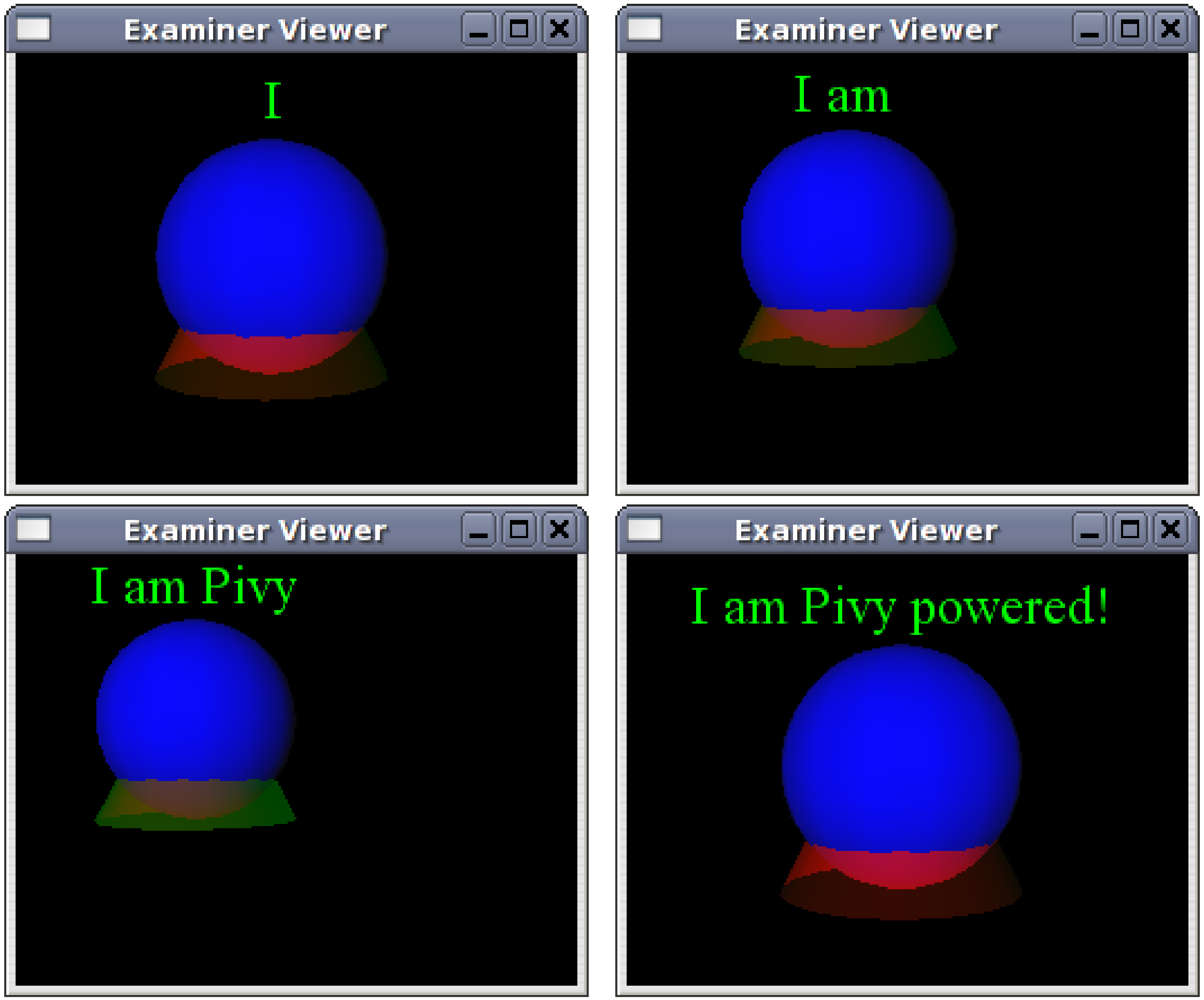


Figure 5.3: Examiner viewer executing glow and textscroller script

Chapter 6

Results

This chapter presents some of the projects and applications that have been accomplished through the use of Pivy. Four use cases will be presented, that show the versatility and variety of usage scenarios, that Pivy can be applied for:

1. The first use case presents the creation of a viewer for an "Ubiquitous Tracking Simulation Environment", developed in collaboration by the VRGroup's of the University of Technology Munich, Germany, and Vienna, Austria.
2. The second use case demonstrates, how the SoPyScript got used for "Authoring and User Interaction for the Production of Wave Field Synthesis Content in an Augmented Reality System" at the Fraunhofer Institute for Digital Media Technology IDMT in Ilmenau, Germany.
3. The third use case describes, how Pivy was made use of at Systems in Motion, Norway, to create a fully-fledged Python application for the Oil & Gas industry, while using PyQt for the GUI and Pivy for the 3D visualization part.
4. The fourth use case highlights, how D-Level, Italy, makes use of Pivy in their process control and technology integration platform in order to create custom visualization widget-plugins.

6.1 Use Case 1: TU Munich/Vienna - "Ubiquitous Tracking Simulation Environment"

During their collaborative research in the field of "Ubiquitous Tracking for Augmented Reality" [22] the VRGroup's of the University of Technology Munich, Germany, and Vienna, Austria needed to develop an simulation environment for ubiquitous tracking.

The program is called *"pyubitracksim.py"* and features the following components:

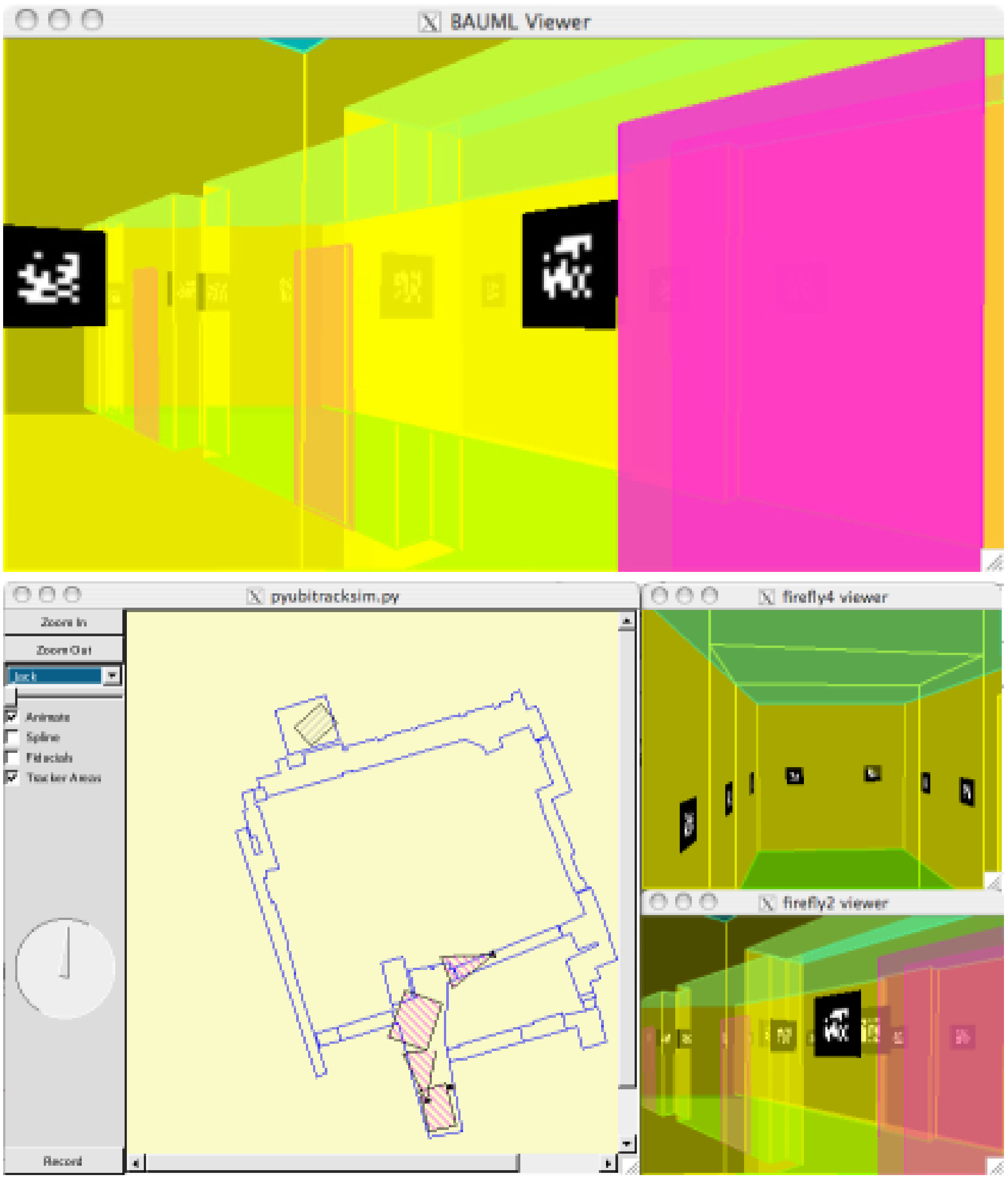


Figure 6.1: The Ubitrack Pivy based viewer running on Mac OS X

- Parsing of a BAUML[1] geometry file
- 2D Qt view of floor geometry (rather than walls, ceilings and other structures)
- Control points can be dropped onto the plan to define the path of a figure following a spline
- The splines are implemented using the SciPy library of scientific tools for Python
- A 3D view from the point of view of the figure in the plan implemented in Pivy

The BAUML specification represents a XML schema describing a XML language to build a world model used for Augmented Reality and Ubiquitous Computing research projects at the Vienna University of Technology. It provides the structures to describe geometry and spatial relations between buildings and rooms contained within building. in addition to that it allows to store locations of markers used for optical tracking within the building.

This project uses Pivy for visualizing the institute floor and the attached ARToolkit paper markers, that allow optical tracking to be used in *Mobile Augmented Reality* settings (Figure 6.1).

Thanks to **Joseph Newman** for providing the screenshots and information.

6.2 Use Case 2: Fraunhofer IDMT - "Wave Field Synthesis"

The Fraunhofer Institute for Digital Media Technology IDMT in Ilmenau[2] makes use of the SoPyScript facility in Pivy for "Authoring and User interaction for the production of Wave Field Synthesis Content in an Augmented Reality System"[23]. (Figure 6.2, and 6.3)

The Fraunhofer IDMT understands itself as technological and economic motor for the region Thuringia. With its innovation power our institute gives impulses for the economy and science in Germany and worldwide.

Fraunhofer IDMT uses the SoPyScript node in a Studierstube application in order to prototype the sending of UDP Open Sound Control[3] commands in order to control a *"Wave Field Synthesis"* audio system through a newly developed Augmented Reality GUI.

Thanks to **Frank Melchior** for providing the screenshots and information.

6.3 Use Case 3: Systems in Motion - "ConocoPhillips Onshore Operations Center for Ekofisk"

Systems in Motion AS[4], a Norwegian software development house for 3D visualization libraries and applications and who developed Coin3D which Pivy provides a binding for, is helping

[1]BAUML http://www.studierstube.org/projects/mobile/ocar/baumldoc.html
[2]http://www.idmt.fraunhofer.de/
[3]http://www.cnmat.berkeley.edu/OpenSoundControl/
[4]http://www.sim.no/

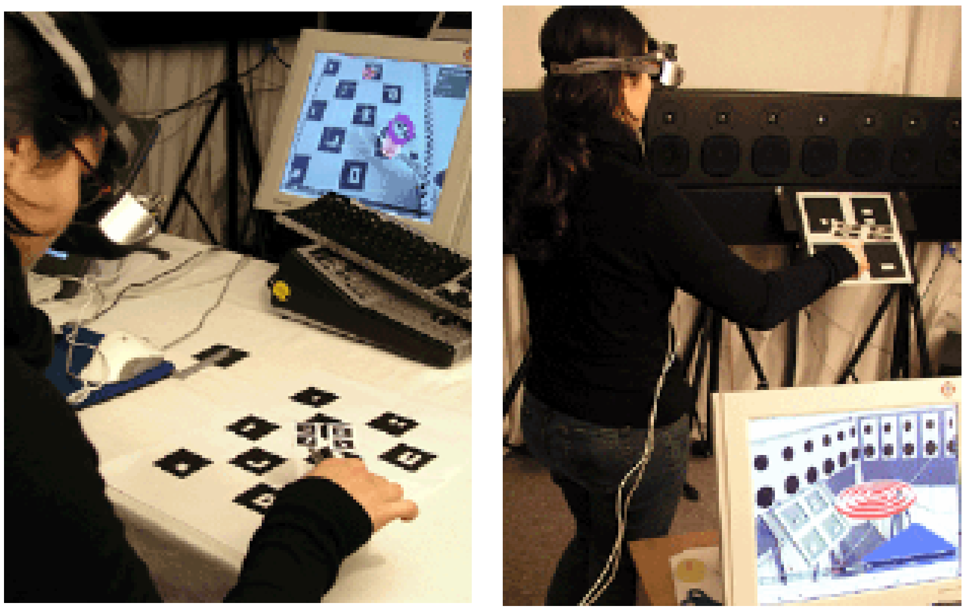


Figure 6.2: Exocentric and egocentric setups for sound source manipulation

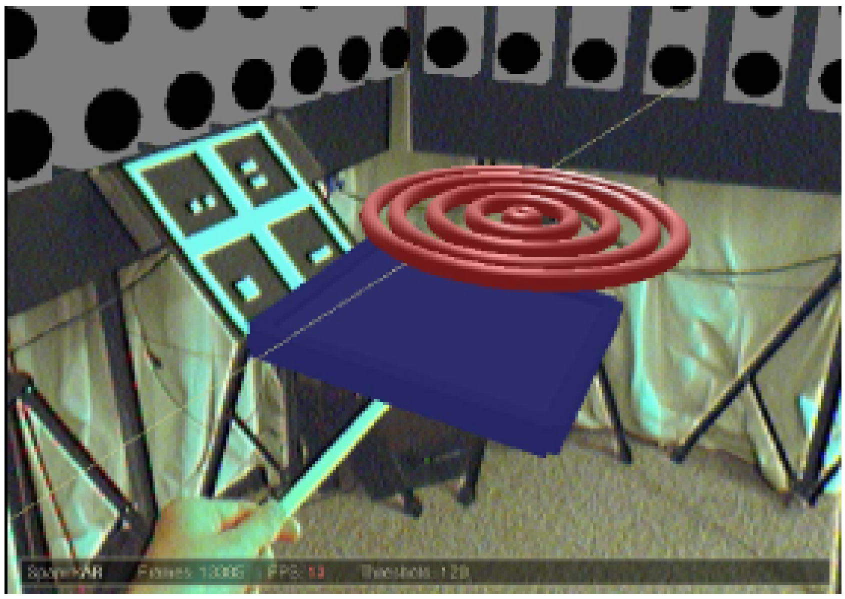
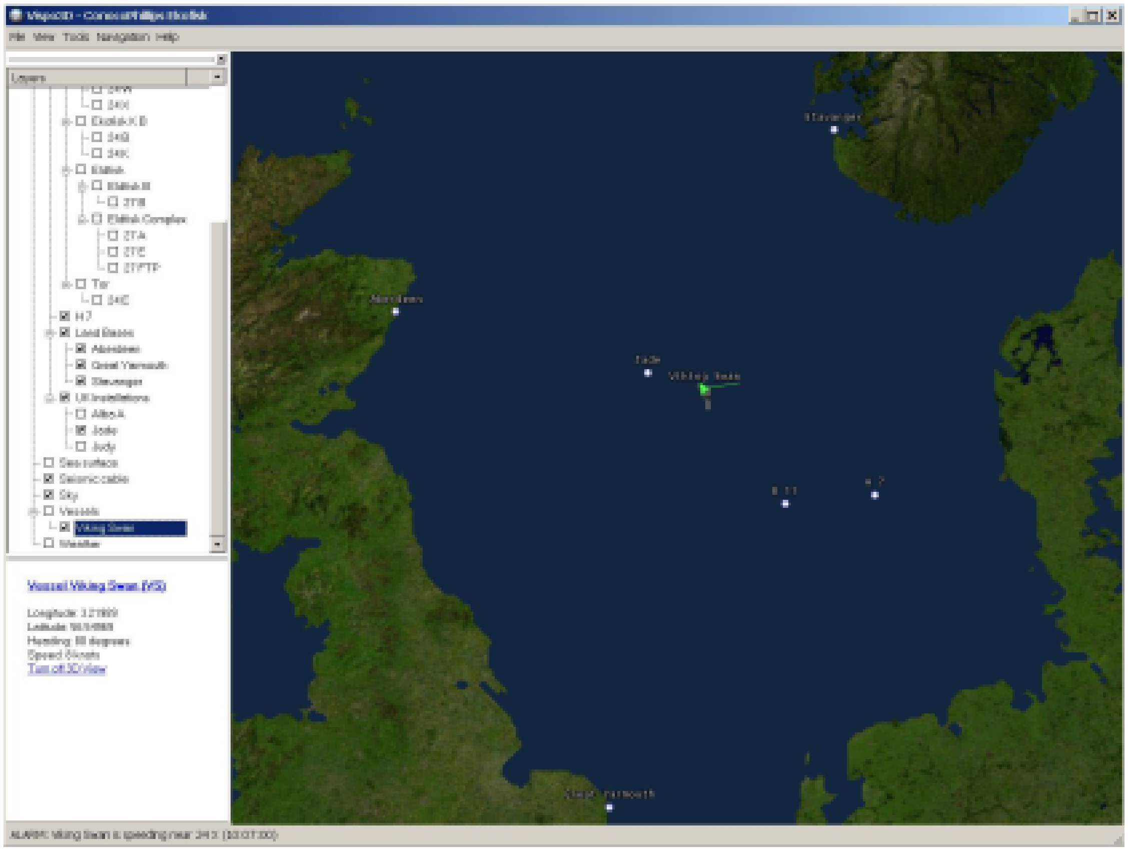


Figure 6.3: User View for the egocentric setup for sound source manipulation



Vessel Viking Swan (VS)

Longitude: [illegible]
Latitude: [illegible]
Heading: [illegible] degrees
Speed: [illegible] knots
Turn off 3D View

ALARM: Viking Swan is speeding near [illegible]

Figure 6.4: The Ekofisk application - North Sea overview

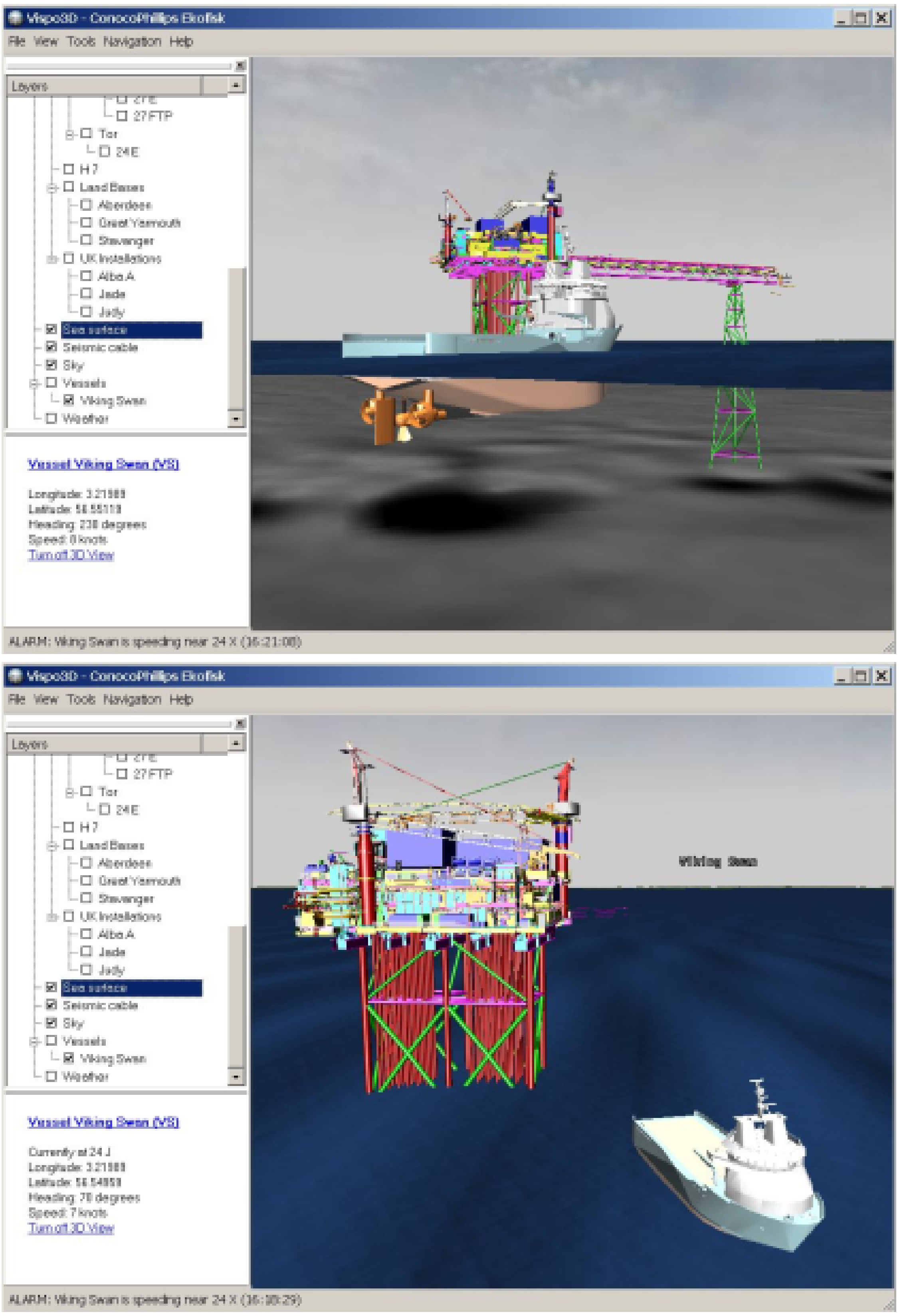
ConocoPhillips move personnel from offshore oil rigs to the Onshore Operations Center (OOC) in Stavanger, Norway, by visualizing installations, vessels, weather and related production data in real time.

The application is integrated with radar systems, weather stations and production databases. (Figure 6.4 and 6.5) In the project Systems in Motion has capitalized on 10 years of experience with real-time graphics, including the Coin3D scene graph toolkit and the SIM Scenery[5] terrain engine.

Application code has been written almost exclusively in Python, a dynamic programming language ideally suited for rapid application development (RAD) and Rapid Application Prototyping (RAP). Use of Python was made possible by the Pivy Coin3D binding.

Thanks to **Pål-Robert Engnæs** for providing the screenshots and information.

[5]http://www.sim.no/products/Scenery/



Vessel Viking Swan (VS)

Longitude: 3.21583
Latitude: 56.55118
Heading: 238 degrees
Speed: 0 knots
Turn off 3D View

ALARM: Viking Swan is speeding near 24 X (16:21:08)

Vessel Viking Swan (VS)

Currently at 24 J
Longitude: 3.21583
Latitude: 56.54858
Heading: 78 degrees
Speed: 7 knots
Turn off 3D View

ALARM: Viking Swan is speeding near 24 X (16:20:29)

Figure 6.5: The Ekofisk application - closeups

6.4 Use Case 4: D-Level - “Devil Framework”

D-Level[6] is a privately held company that develops and provides custom and general purpose advanced software systems for data acquisition, process control and systems automation. Founded in 2000, D-Level is headquartered in Rivignano (UD) - Italy, where they are developing the Devil Framework[7].

The Devil Framework is a system developed to fulfill the growing need to collect, integrate, correlate, control and visualize all information produced and consumed by the various hardware and software technologies involved in modern working processes. The framework is designed from the ground-up to be scalable, extendible, configurable and easily deployable but at the same time secure and reliable. Furthermore, the Devil Framework is conceived to inter-operate with as many devices as possible and to run on all major computer operating systems available today and is developed using open standards and open technologies, giving users total control on systems and data, and to developers a powerful open environment.

The Devil Framework screenshot in Figure 6.6 is showing the following components:

- the views manager/editor in edit mode (center-background),
- the main toolbox (top-right),
- the interactive distributed python console (bottom-right),
- the components browser (bottom-left) and
- the configs browser (top-left-back).

D-Level uses and integrates Pivy for custom visualization widget plugins and are also planning in integrating Pivy with their product’s Integrated Development Environment.

Thanks to **Alessandro Iob** for providing the screenshot and information.

[6]http://www.dlevel.com/
[7]http://www.dlevel.com/products/devil

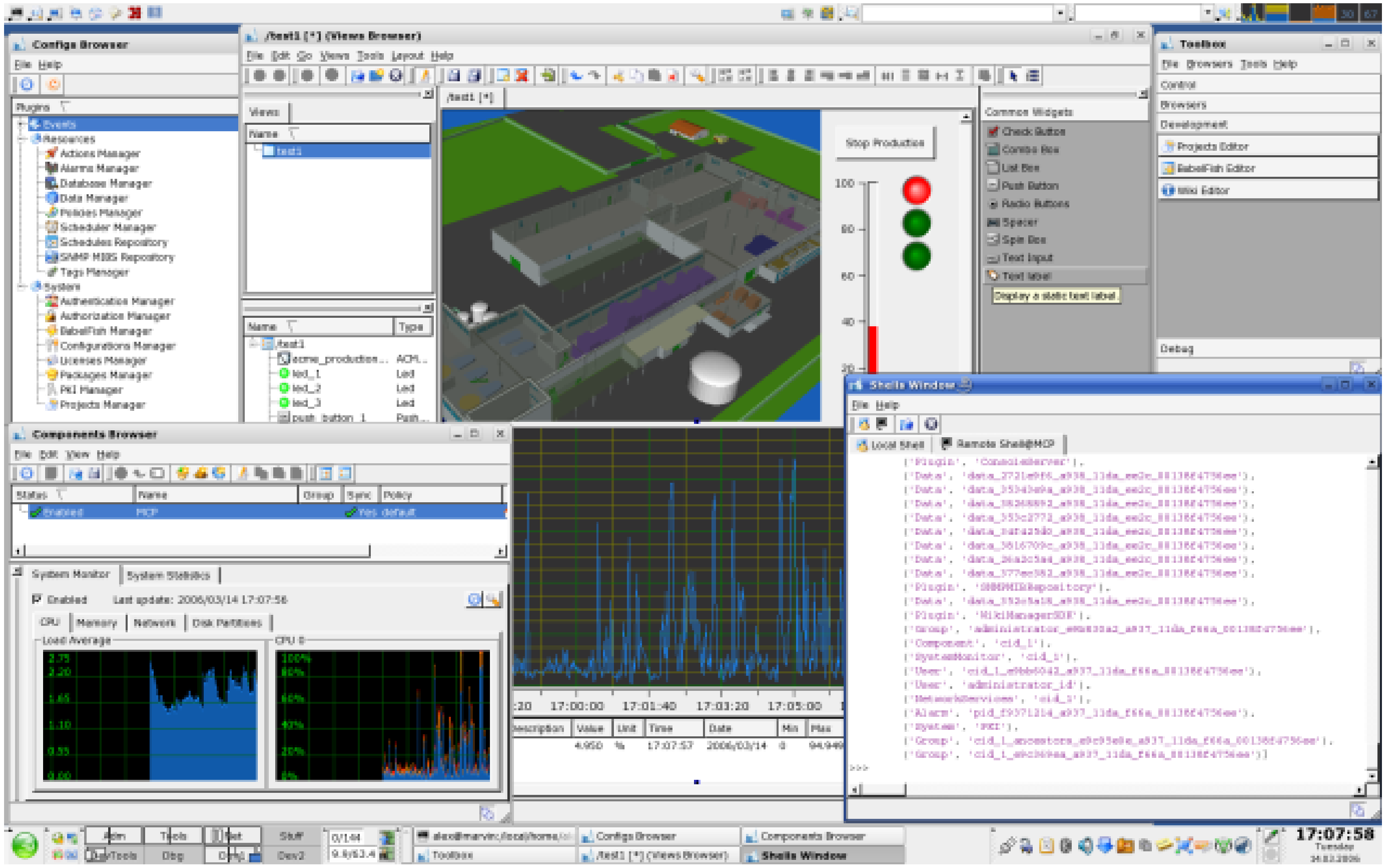


Figure 6.6: The Devil Framework

Chapter 7

Conclusions

The presented thesis showcased how we can benefit from the usage of a dynamically typed and bound language such as Python in a scene graph library as Coin which has been designed and implemented in a statically typed system programming language. Furthermore, we have demonstrated how the existence of a runtime system in the aforementioned scene graph library, implemented for the required and necessary flexibility, can be used to our advantage.

Through the possibility of interactive modification of Coin programs from within the Python interpreter at runtime and incorporation of scripting nodes into the scene graph, the usefulness as a debugging and Rapid Application Development and Prototyping tool is greatly expanded.

A very important decision, that needs to be made before the actual implementation starts, is whether to make usage of an automatic wrapper generator such as *SWIG* or not. Apart from gaining a quick and efficient mechanism to create a complete wrapper, which serves as a solid foundation for a variety of use cases, the usage of an automatic wrapper generator allows to extend our current binding quickly and easily. This permits the incorporation of newly created nodes and libraries, that have been developed in the system programming language. In order to reduce redundancy in newly developed extension modules and improve re-usability of the already existing wrapper interfaces, common typemaps and generic functionality has been factored out into a single wrapper interface file.

Through the usage of the available runtime mechanisms, that the wrapper generator provides, we can create independent modules. Therefore, special purpose facilities, such as the SoPyScript scripting facility, do not have to load all the involved and otherwise unused C/C++ dynamic link libraries and huge Python extension modules. This improves the maintainability, extensibility and the overall actual resource usage.

We have seen and argued, that due to the distinct properties of retained mode operated libraries, they present a better fit for usage out of interpreted languages. Libraries, that operate in immediate mode, are penalized through the calling overhead of interpreted languages. Moreover, one of the main benefits of dynamically interpreted languages, namely their use as a controlling language, can be exploited in this domain to its fullest. CPU intensive calculations continue to be conducted in the lower end system language.

The distinction between *extending* and *embedding* the Python interpreter has been explained and its close relationship witnessed. It can be observed, that once the time has been taken to fully wrap a library and the involved application interfaces, the effort quickly pays off. It provides an extension architecture, such as a plug-in system, which can be extended quickly in unexpected interesting new ways. Furthermore, the embedding part reveals itself afterwards just as a matter of combining and reusing the existing foundations, while writing few lines of integration *"glue code"* around the interfaces.

Finally, we have proven that it is nowadays possible to efficiently wrap libraries and interfaces written in C++, which is considered to be a complex and difficult language and task. Through the organization and modularization of the wrapper code base for reuse, where the focus lies upon the automation of as much as possible through the wrapper generator, we find ourselves in a position to tackle big to huge C++ libraries quite effortlessly. The provision of a good foundation, which fully wraps the base classes of the C++ library, additionally helps in this process.

An important benefit of the usage of a multi-lingual wrapper generator, such as *SWIG*, is, that once the organizational and core foundation aspects have been solved, we solely need to revisit all the type conversion and glue code in place and rewrite them for the new target language, as we make use of the same wrapper generator. All the required changes and code parts, that need to be tackled, are then already identified and neither the organization of the wrapper interfaces nor the *SWIG* type code definitions have to be changed. In contrast, a manual wrapping approach or the use of wrapper generators for a single target language requires the rediscovery and recreation of the same effort for each additional language.

7.1 Future work

Pivy has come a long way and proven to be a very stable and ready to use Coin binding. Pivy provides a suitable platform for advanced usage scenarios, which is viable for commercial production quality applications today. The usage scenarios of Pivy range from prototyping, Rapid Application Development to usage as a convenient, advanced and readily available debugging facility. Clearly, users and developers will come up with novel and interesting new ideas in how Pivy can be exploited through the combination and integration of the already provided facilities at hand.

Still, there is always something that can and needs to be improved. We close this thesis by naming some of the current ideas:

- Provide appropriate documentation and tutorials to give new and advanced Coin developers a quick and easy start and allow them to quickly see what Pivy offers and how to make use of these offerings and facilities.
- Adapt Pivy for newer *SWIG* and *Python* versions to make improvements such as better performance and additional language features available.
- Improve the performance through minimizing the calling overhead for the crucial Coin basic types by for example manually wrapping them. This will implicitly benefit all the additional bindings for the existing Coin based libraries.

- Make Pivy optionally fully thread-safe. Initial support through the *-threads* option has been introduced in *SWIG* version 1.3.28. An audit throughout the type conversion wrapper code base needs to conducted in order to identify the locations where Python's *global interpreter lock* needs to be acquired or released for the sections where Python code is executed. The usage of this feature currently results in a decrease of the overall wrapper performance by around 60%, which unfortunately constitutes a very significant impact. However, *SWIG* provides the necessary facilities to allow fine grained specification in which parts the thread handling should be enabled.

- Provide additional wrappers for Coin based classes and integrate Pivy into the Coin source code base and build system. To have Pivy integrated in the Coin source code base, allows the user to create the binding in one pass without the need to hop through several stages. Moreover, it further ensures that the binding stays updated with every consecutive release of the Coin library. The way Pivy's build system work is flexible enough to achieve such a goal quite easily. A autotools[1] m4 macro needs to be written, which tests for an installed Python base, and the given Python setup.py build, clean, install targets need to be issued accordingly in the Coin build system. However, what remains to be discussed is whether the Pivy interfaces should be merged into the Coin header files or kept in a separate subdirectory using the current approach of copying the required header files over to the build directory in order to prepend the required *%include interface.i* directives.

  Aside from that, the interface files and build system will need to be split up and tweaked accordingly for each Coin and support library to allow an independent Pivy code base for each submodule.

  Finally, wrappers and interfaces should be created for and integrated with the additional Coin support libraries in existence such as NutsnBolts or SIM Scenery.

- Allow the creation of extension actions, engines, fields, nodes and node kits in Pivy. SoPyScript allows the creation of new functionality, which solves most of the problems. When it comes to extending the Coin type-system itself with new nodes and node kits solely written in Python, Pivy does not offer a simple solution for that yet. It should be possible by closely mimicking the functionality of the macros contained in the SoSubAction/Node/Field/Engine header files to create this highly requested functionality. There are 2 possibilities and approaches to achieve that:

  1. Through the combined usage of the wrapped SoType::createType(), SoType::overrideType() and SoType::createInstance() methods, which are provided through the Inventor type system, and a small and convenient Python utility module.
  2. Through the creation of the required functionality in C++, which allows to create those new types. This functionality is then exposed through inclusion in the main Coin Pivy *SWIG* interface.

- Integrate with SciPy, a library of scientific tools for Python. As mentioned in chapter 2, SciPy includes among others modules for graphics and plotting, optimization, integration, special functions, signal and image processing, genetic algorithms and ODE

[1] http://www.gnu.org/software/autoconf/

solvers. However, SciPy currently lacks a 3D viewer component, where both Pivy and SciPy could mutually benefit from integrating with each other.

- A runtime based development system: One of the ideas that have been discussed in the past, was to create a runtime based development system and framework. This does not make it necessary anymore to restart the application, but rather just reload its components (using a provided component API) at runtime. This permits to improve the turnaround times drastically for applications that have a lengthy startup time.

  Apart from that, it allows to deploy and upgrade such a system without requiring a restart, which improves and allows for new maintenance procedures and models. However, this tends to be a rather tricky process, where special languages such as Erlang have been created to tackle exactly this problem domain. The problems range from fundamental questions such as *"What needs to be done if the underlying fundamental libraries written in C/C++ change?"* to *"How does one provide updates for Python modules efficiently to track things like variable/class removals and changes from and in the namespace etc.?"*. A solid and well-designed framework, which takes those and similar questions into account, appears to be a feasible and workable solution.

  Further ideas, such as the integration with existing IDE's or to embed a web server into such a system, which provides remote introspection and permits to update components at runtime, promises to quickly yield even more gains in productivity and maintenance.

- Provide and integrate bindings for other languages such as Ruby, Lua, Java and C#. The most straight forward binding to create, should be a Ruby binding, due to the similarities found in the Python and Ruby languages. Once tackled and finished, the code base will be adapted and abstracted well enough in order to serve as a generic platform for the more involved languages, such as Java and C#. It remains to be discussed and seen if a fully fledged binding for Lua, which is a special purpose language and popular in the gaming industry[2], should be created or rather be used as an alternative to JavaScript in an embedded fashion in the VRML scripting nodes. The latter option seems to be the better solution for a Lua binding.

- And last but not least, advertising Pivy to make it better known in the Python community as a viable platform for 3D applications.

[2]due to its lower memory overhead and easy to use C API for embedding purposes

Appendix A

The Pivy file system layout

Pivy's file system layout had to be refined during the development process a couple of times whenever the need for more flexibility arose or when the requirements for a more flexible, practical and suitable build process changed. Figure A.1 shows the top-level root directory of the Pivy source distribution. The file system layout should be simple, practical (for both command line and GUI-driven purposes), very clean, obvious and as much as possible flat and direct (i.e. not crowded or hidden in an endless and intrusive subdirectory structure). Furthermore, it should adhere and comply to common open source community practices regarding the location and naming conventions of certain files, such as *README* or *distutils' setup.py*.

Inventor: contains the *SWIG* interface files (*.i* suffix). It contains a subdirectory hierarchy matching the one found in Coin for the header files.

SoPyScript: contains the sources and *SCons* based build script for the *SoPyScript* scripting node.

VolumeViz: similar to the **Inventor** directory above, **VolumeViz** contains the *SWIG* interface files for the *SIM Voleon*[1] volume rendering library that is based on and to be used together with Coin.

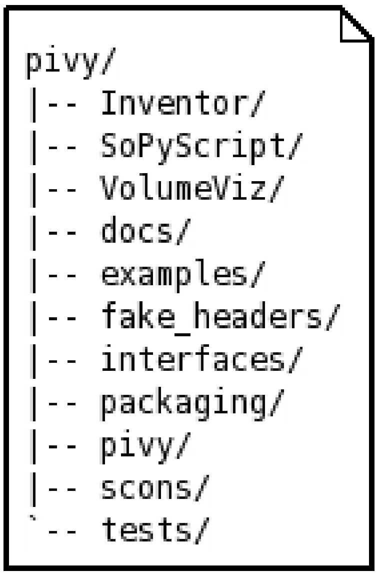
docs: contains older ChangeLogs and Pivy related documentation.

examples: contains various examples, demonstrating usage of various aspects found in Pivy.

fake_headers: contains empty *"fake"* header files in a subdirectory hierarchy they occur in in order to prohibit **SWIG** to generated wrappers for these header files.

interfaces: contains the main *SWIG* interface files for Coin and each Coin based library. Furthermore, it contains *pivy_common_typemaps.i*, which collects typemaps and support functions that are required in all extension modules, and *coin_header_includes.h*, which lists all Coin header *#include* statements and represent the Coin header files that should be inspected and wrapped.

[1] http://www.sim.no/products/SIM_Voleon/



```
pivy/
|-- Inventor/
|-- SoPyScript/
|-- VolumeViz/
|-- docs/
|-- examples/
|-- fake_headers/
|-- interfaces/
|-- packaging/
|-- pivy/
|-- scons/
`-- tests/
```

Figure A.1: The top-level file system hierarchy of the Pivy source code distribution

packaging: contains package file descriptions for various package managers and installers. (currently contains only a package description for the Mac OS X installer)

pivy: contains the Python package structure for the *pivy* package. Furthermore, *sogui.py* contains the *sogui proxy* mechanism and the *SWIG* generated wrapper source files and *SWIG* proxy classes will be located here as well during the build step.

scons: contains a local copy of the *SCons* build system to allow users to compile the SCons based parts of Pivy such as the *SoPyScript* scripting node, the extension and *SoPyScript* examples and in an earlier Pivy version the *SWIG* runtime library.

tests: contains the Pivy unit tests in *pivy_tests.py*.

Another top-level directory is created during the build step used by Python's *distutils* build mechanism: **build/**. However, **distutils** allows to specify a different location for this directory.

Appendix B

The Pivy build system

As mentioned, for the build system requirements a combination of two tools are made use of:

distutils is a suite of standard Distribution Utilities for Python with the goal to make building, distributing, and installing Python modules, extensions, and applications painless and standardized.

SCons is an open source next-generation software construction build tool. It can be thought of an improved, cross-platform substitute for the classic *make* utility with integrated functionality similar to *autoconf/automake*[1] and compiler caches such as *ccache*[2].

B.1 Distutils

Distutils is used to gather information about the compiler and linker flags the Python interpreter expects its extensions to be compiled with. Additionally, we make use of *distutils'* build framework and override certain behavior in order to gather system information and keep a log of activities to assist in user support or in order to gather additional compiler and linker flags required to link against the wrapped Coin C++ libraries.

Furthermore, distutils provides the possibility to register the Python project at the *Python Cheese Shop*[3] and to create source or binary package distributions where a description of what to package in addition to the extensions and Python modules (such as examples or documentation) can be provided in a separate file: typically *MANIFEST.in*.

B.2 The setup function in setup.py

Listing B.1 shows the distutils setup function for Pivy, which is contained in the *setup.py* build description file in the root of the Pivy source distribution. The **name** specifier in the

[1] http://www.gnu.org/software/autoconf/
[2] http://ccache.samba.org/
[3] http://cheeseshop.python.org/

function tells *distutils* the name of the whole package. This will be taken into account once *setup.py* is asked to create a *.tar.gz* or *.zip* source archive, where additional files apart from the actual extension and module that should be put into the package can be specified in a *MANIFEST.in* as mentioned above; *MANIFEST.in* provides its own simple syntax to express what to include and what not.

```
setup(name = "Pivy",
      version = PIVY_VERSION,
      description = "A Python binding for Coin",
      long_description = __doc__,
      author = "Tamer Fahmy",
      author_email = "tamer@tammura.at",
      download_url="http://www.tammura.at/download/",
      url = "http://pivy.tammura.at/",
      cmdclass = {'build'   : pivy_build,
                  'clean'   : pivy_clean},
      ext_package = 'pivy',
      ext_modules = pivy_build.ext_modules,
      py_modules  = pivy_build.py_modules,
      packages = ['pivy', 'pivy.gui'],
      classifiers = filter(None, PIVY_CLASSIFIERS.split("\n")),
      license = "BSD License",
      platforms = ['Any']
      )
```

Listing B.1: The distutils setup() function for Pivy in setup.py

The **version**, **description**, **long_description**, **author**, **author_email**, **download_url**, **url**, **classifiers**, **license** and **platforms** specifiers contain and provide meta information regarding the project. This meta information is for example used by distutils to provide the *Python Cheese Shop* with the required information in the project description listing. The **classifiers** contain additional meta information regarding the intended audience, related topics and operating system support.

The **cmdclass** specifier expects a dictionary that permits to override the built-in build rules. In this case, we are overriding the distutils default *build* (to use our own customized build instructions) and *clean* (to remove any additional files such as the wrapper sources generated by *SWIG*) we created during the build process) methods. In the build method, we are invoking the external *SWIG* processes and providing the *Extension()* directives that contain the information about the extension modules that should be built.

The **ext_package** specifier tells *distutils* which Python package the extensions live in; the **ext_modules** specifies the name of the C/C++ extension module list that should get built. This list is filled in the *pivy_build* class in the *swig_generate()* method, which is also responsible to invoke the *SWIG* executable with the right flags.

The same is done for the Python modules that should be installed and byte compiled in **py_modules**. To precompile the Python modules to Python bytecode is a very important step. If they are not precompiled, the rather large Python proxy modules, that are generated by *SWIG*, take a long time to import into the Python interpreter. The reason is that the Python interpreter will have to parse the whole Python file itself every time the module is imported the first time.

The **packages** specifier lists the Python packages that should be installed. In our case, we advise *distutils* to pick them up from *pivy/pivy* and *pivy/pivy/gui*.

B.3 Platform specific compiler and linker flags

In order to gather the compiler and linker flags necessary for Coin and the SoGui support libraries such as SoQt, we make use of the *coin-config* and *sogui-config* configuration helper shell scripts. The exception is Microsoft Windows, which currently has a big user-base and is a very differently and strangely designed platform in comparison with systems that are Unix based, such as Linux, Solaris, *BSD or Mac OS X; the *coin-config* configuration scripts are not provided for that platform and therefore required special treatment in the build specification by having to specify the compiler and linker flags ourselves. In the Windows case, we can however rely on the *COINDIR* environment variable being set and gather the information manually. For example:

```
if sys.platform == "win32":
  INCLUDE_DIR = os.getenv("COINDIR") + "\\include"
  CPP_FLAGS = "-I" + '"' + INCLUDE_DIR + '"' + " " + \
              "-I" + '"' + os.getenv("COINDIR") + "\\include\\Inventor\\annex" + '"' + \
              " /DCOIN_DLL /wd4244 /wd4049"
  LDFLAGS_LIBS = '"' + os.getenv("COINDIR") + "\\lib\\coin2.lib\" "
  if module == "sowin":
    CPP_FLAGS += " /DSOWIN_DLL"
    LDFLAGS_LIBS += '"' + os.getenv("COINDIR") + "\\lib\\sowin1.lib" + '"'
  elif module == "soqt":
    CPP_FLAGS += " -I" + '"' + os.getenv("QTDIR") + "\\include\" /DSOQT_DLL"
    LDFLAGS_LIBS += '"' + os.getenv("COINDIR") + "\\lib\\soqt1.lib\" "
else:
  INCLUDE_DIR = self.do_os_popen("coin-config --includedir")
  CPP_FLAGS = self.do_os_popen("%s --cppflags" % config_cmd)
  LDFLAGS_LIBS = self.do_os_popen("%s --ldflags --libs" % config_cmd)
```

Listing B.2: Special Windows treatment

The build script uses a dictionary mapping, named **MODULES**, which lists the modules that should be built. Furthermore, the **SOGUI** list is used to specify the available SoGUI bindings that should be probed and looked for in the build system. If the bindings are not found, they will be removed from the **MODULES** dictionary.

B.4 Module mapping and build flexibility

```
SOGUI = ['soqt', 'soxt', 'sogtk', 'sowin']
MODULES = {
'coin' : ('_coin', 'coin-config', 'pivy.'),
'simvoleon' : ('_simvoleon', 'simvoleon-config', 'pivy.'),
'soqt' : ('gui._soqt', 'soqt-config', 'pivy.gui.'),
'soxt' : ('gui._soxt', 'soxt-config', 'pivy.gui.'),
'sogtk' : ('gui._sogtk', 'sogtk-config', 'pivy.gui.'),
'sowin' : ('gui._sowin', 'sowin-config', 'pivy.gui.')
}
```

Listing B.3: MODULES dictionary used for mapping the configure scripts

The mapping reads as follows:

```
'modulename' : ('name of the extension without suffix',
                'name of the configuration script',
                'location in the package hierarchy')
```

If a new Coin based library should be added and it features its own configuration script, it is sufficient for the extension to be wrapped and built as a Python extension to create an entry in the **MODULES** table as the rest will be handled and taken care of by the generic build system mechanisms we created.

B.5 Dealing with the Coin header files

The last method we mention (as a complete discussion of the build system is way beyond the scope of this thesis), is the *copy_and_swigify_headers()*.

We need to tweak the Coin headers, partly because earlier *SWIG* versions had problems to parse them, partly because certain header files need refinements in order to produce a more *pythonic* interface and in order to add the type conversions. A constraint was that the Coin headers themselves could be only modified out of their repository. In earlier Pivy development this was handled by copying the current Coin headers manuall into the Pivy repository and extending them inline by adding the necessary *SWIG* directives and using them as *SWIG* interface files. This turned out to be quite cumbersome for four reasons:

1. Quite unsurprisingly the Coin header files had a tendency change by their developers over time (corrections, additions, refinements) and the *SWIG* changes needed to be reintegrated into the newer header files, which was a really tedious and error-prone task.

2. Apart from having to track the changes by keeping an eye on the commit messages such a solution made it also dependent on specific Coin releases and therefore quite unflexible. E.g., if some user decided to use the Coin development branch rather than a

release, he needs to take the new headers, which might have added new methods, take out the *SWIG* declarations from the Pivy provided Coin headers, merge them back into the Coin headers again and place them back into the Pivy directory hierarchy.

3. The header files got crowded and were hard to read and maintain, where the *SWIG* declarations and typemaps were not easily spottable.

4. Licensing considerations, as header files restricted by the GPL got incorporated. To avoid the arise of any unforeseen licensing issues[4] and in order to stay under control regarding any licensing decisions of Pivy[5], this problem needed to be solved.

So in order to keep the declarations away from the header files, stay Coin version agnostic, make the involved procedures more maintainable and in order to avoid any licensing issues, the refined and much better approach copy the header files, that need to be equipped with *SWIG* interface declarations, out of the Coin include path location on the system. After the copy process, a *SWIG* interface include is added after the copyright section and before the include block as shown in the next listing.

```
#ifndef COIN_SBVEC3F_H
#define COIN_SBVEC3F_H

/* Copyright (C) part */

/* Pivy preprocessor protection */
#ifdef __PIVY__
/* SWIG interface file inclusion directive */
%include Inventor/SbVec3f.i
#endif

/* first header file */
#include <stdio.h>
```

Listing B.4: Preprocessor macro protected SWIG %include directive in a Coin header file

B.6 The fake_headers and -includeall approach

The Pivy build system invokes *SWIG* with the **-includeall** option, which instructs *SWIG* to **not** ignore any encountered *#include* statements in the header file and works as if the *%include SWIG* directive is specified at the same location, which includes another file to be inspected by *SWIG* and therefore lets the header file be treated as if it were an additional *SWIG* interface file.

This is done for one practical reason: not having to specify any header files in the right order. Were we required to specify the header files in the right order, we repeatedly need to spend a

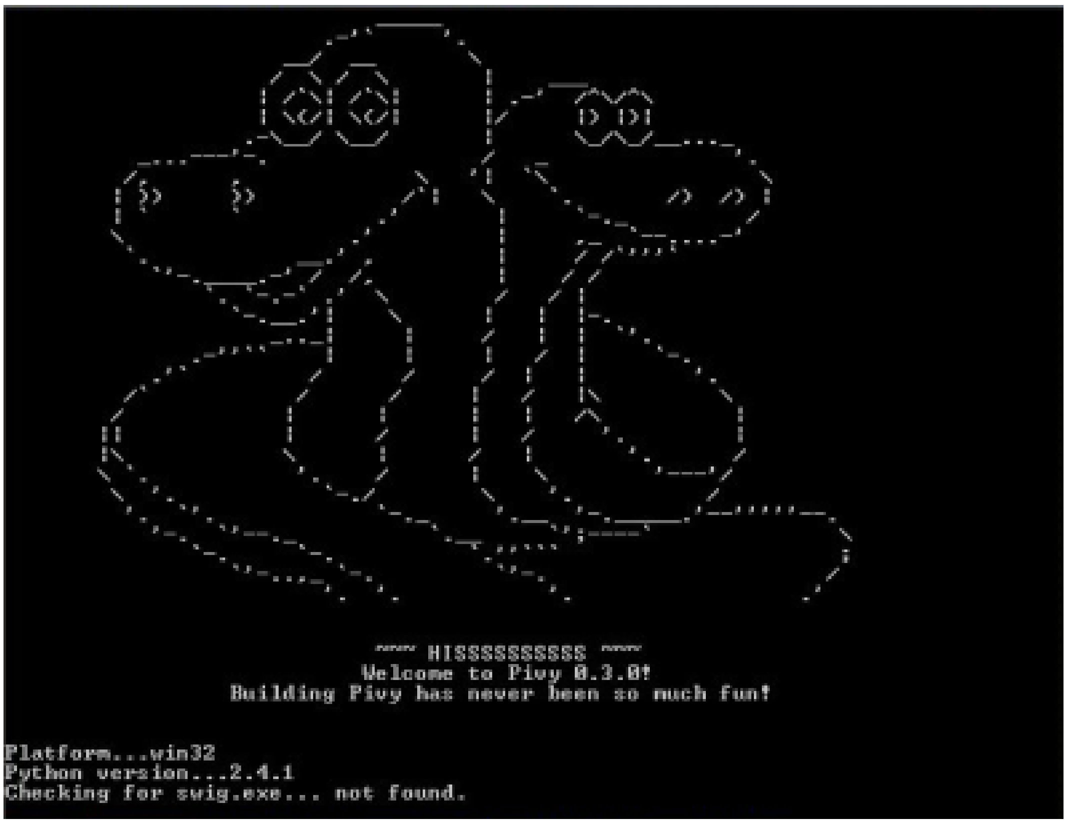
[4]by for example using Pivy together with a commercial Coin license

[5]the GPL requires any software that makes use of GPL licensed code to be put under the same GPL license

large amount of time in determining the correct order. Furthermore, it is quite cumbersome to add additional Coin support libraries as the same effort needs to be reiterated.

However, the *-includeall* approach causes another problem: *SWIG* will also wrap standard include system header files such as *stdio.h*, which are not relevant for the Coin wrapper, and which is going to unnecessarily enlarge the wrapper and possibly include other system files themselves, that again are unnecessarily wrapped. To remedy that, the **fake_headers** directory is in place, which contains empty header files with the names of any external header files that Coin *#include's*. If we now put the **fake_headers** directory as the first entry in the include search path for *SWIG*, *SWIG* will consult and look into the **fake_headers** directory first and if no header file has been found there, only then proceeds to consult the rest of the include search path.

This solves the problem very elegantly as *SWIG* will produce no wrapper code for the empty header files in the **fake_headers** directory and we therefore can provide the header files in any order we like as the whole and complete type information will be collected as the Coin header themselves had already to worry about specifying the right *#include's* for proper type declarations, which we now can take advantage of. Furthermore, to wrap all Coin classes, we just have to specify all header files, which can be nicely automatized by a simple shell or Python script again.



```
                    ~~~~ HISSSSSSSSSS ~~~~
                  Welcome to Pivy 0.3.0!
           Building Pivy has never been so much fun!

Platform...win32
Python version...2.4.1
Checking for swig.exe... not found.
```

Figure B.1: Build failure reported by a Pivy user

B.7 Importance of build system output and tests

The build system has to address platform specific idiosyncrasies and deal with various user setups that cannot be always anticipated or foreseen. It is therefore crucial to write out or log relevant info that could help in spotting and resolving a problem easily to the screen or a logfile. Furthermore, it is important to test the system for a proper build environment before we proceed in order to verify that the minimal system and dependency requirements for a successful build are met.

In Figure B.1 the build a stops as the required *SWIG* executable has not been found. If the Pivy build system is silent about the conducted steps, it will be many times harder and involve significantly more communication overhead in order to determine the cause of a failed build on a different user setup.

B.8 SCons usage

When it comes to building C++ shared libraries which are not to be used as Python extension modules as is the case with the *SoPyScript* scripting node or with Pivy wrapped extension nodes that involve a simple *SWIG* invocation and are not required to be distributed or installed into the Python system installation, *distutils* is not the most suitable tool for such tasks as it is tailored to building and distributing Python extensions.

As mentioned in the file system layout discussion, a local copy of the *SCons* build tool is provided as a convenience in the Pivy tree, not requiring users to get hold of it in order to build the parts of Pivy that require it.

Earlier *SWIG* versions required the creation of a *SWIG* runtime library, named **libpivy_-runtime** back then; for example to allow modules to be used independently or allow the scripting node to query the *SWIG* type system information contained in Pivy's **pivy._coin** core extension module. This led to a couple of complications as *distutils* was not very suited for the creation of a shared library[6] and where *SCons* was discovered in the process of researching a solution and actually used to create this library and handle the involved platform issues. Fortunately, later *SWIG* versions reworked the *SWIG* runtime mechanism and where this library was not needed anymore.

The following listing demonstrates the usage of *SCons* in order to build the *SoPyScript* scripting node:

```
.../pivy/SoPyScript $ scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
g++ -O2 -Wall -D_REENTRANT -fPIC -I/usr/local/include/Inventor/
    annex -I. -I/usr/include/python2.4 -c -o SoPyScript.os
```

[6]due to some cross-platform limitations regarding the linker flags, namely -rpath, which was needed to set the runtime library search path to include our *SWIG* runtime library, which could not be installed into the system

```
    SoPyScript.cpp
g++ -Xlinker -export-dynamic -L. -lpython2.4 -shared -o
    libSoPyScript.so SoPyScript.os -L/usr/local/lib -L/home/tamer/
    projects/pivy_release -L/usr/lib/python2.4/site-packages -lCoin
    -lGL -lXext -lSM -lICE -lX11 -ldl -lpthread -lm
scons: done building targets.
.../pivy/SoPyScript $ ls -1
libSoPyScript.so
SConstruct
SoPyScript.cpp
SoPyScript.h
SoPyScript.os
swigpyrun.h
```

Listing B.5: Building the SoPyScript node

Listing 4.22 on page 77 shows an *SConstruct SCons* build description for a Coin extension node.

What's left to be said is that *SCons* is very versatile and flexible tool that can be used for a variety of purposes, using Python as its only dependency and its language to write the build system description in; *SCons* is used as a fully-fledged build system[7] for various projects, such as Blender or Doom 3.

[7]In fact this thesis has been built with SCons by making use of its LaTeX support

Bibliography

[1] B. W. Kernighan and D. M. Ritchie, *The C Programming Language.* Prentice Hall, 1978.

[2] K. Thompson, "Unix time-sharing system: Unix Implementation," *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 1931–1946, 1978.

[3] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX operating system.* Addison-Wesley, 1989.

[4] J. K. Ousterhout, "Scripting: Higher-Level Programming for the 21st century," *IEEE Computer*, vol. 31, no. 3, pp. 23–30, 1998.

[5] C. Jones, "Programming languages table," *Release 8.2*, 1996.

[6] D. A. Patterson and J. L. Hennessy, *Computer Organization & Design: The Hardware/-Software Interface.* Morgan Kaufmann, 1994.

[7] J. K. Ousterhout, *Tcl and the Tk Toolkit.* Addison Wesley, 1994.

[8] Wikipedia, "Scripting language — wikipedia, the free encyclopedia," 2005. [Online; accessed 30-November-2005].

[9] L. Prechelt, "An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program."

[10] D. Schmalstieg, "Studierstube augmented reality project," 1996.

[11] Dr. Roman Geus, "Python Wrapper Tools: A Performance Study." EuroPython2004 conference, Götheborg, Sweden, 7 June 2004.

[12] J. Wernecke, *The Inventor Mentor.* Reading, Massachusetts: Addison Wesley, 1994.

[13] Henrik Tramberend, *Avocado: A Distributed Virtual Environment Framework.* PhD thesis, University of Bielefeld, 2003.

[14] Henrik Tramberend and Bernd Fröhlich, "On Scripting in Distributed Virtual Environments," in *4th International Immersive Projection Technology Workshop*, 2000.

[15] Calle Lejdfors and Lennart Ohlsson, "PyFX: A framework for real-time graphics effects," tech. rep., Lund University, 2005.

[16] Cary Sandvig and Jesse Schell, "Panda3D - Introduction Scenegraph Design," tech. rep., Disney, 2000.

[17] S. Shodhan, “Panda3D - The Panda3D Engine - Python Scripting for Game and Simulation Development,” in *Pycon Python conference*, 2004.

[18] G. Ward, *Distributing Python modules.* Python Software Foundation.

[19] S. Knight, *SCons User Guide.* Python Software Foundation.

[20] D. M. Beazley, *SWIG Users Manual.* The SWIG project.

[21] J. Wernecke, *The Inventor Toolmaker.* Reading, Massachusetts: Addison Wesley, 5 ed., 1995.

[22] J. Newman, M. Wagner, M. Bauer, A. MacWilliams, T. Pintaric, D. Beyer, D. Pustka, F. Strasser, D. Schmalstieg, and G. Klinker, “Ubiquitous tracking for augmented reality.,” in *ISMAR*, pp. 192–201, 2004.

[23] F. Melchior, T. Laubach, and D. de Vries, “Authoring and user interaction for the production of wave field synthesis content in an augmented reality system.,” in *ISMAR*, pp. 48–51, 2005.

[24] G. van Rossum, *Extending and Embedding the Python Interpreter.* Python Software Foundation.

[25] D. M. Beazley, “Automated scientific software scripting with SWIG,” *Future Gener. Comput. Syst.*, vol. 19, no. 5, pp. 599–609, 2003.

[26] I. Pulleyn, “Embedding Python in Multi-Threaded C/C++ Applications,” *Linux J.*, vol. 2000, no. 73es, p. 3, 2000.

[27] G. Ward, *Installing Python modules.* Python Software Foundation.

[28] G. van Rossum and F. L. Drake, Jr., *Python/C API Reference manual.* Python Software Foundation.

[29] T. Fahmy and D. Schmalstieg, “High level 3D graphics programming in Python,” in *Pycon Python conference*, 2004.

[30] T. L. Cottom, “Using SWIG to Bind C++ to Python,” *Computing in Science and Engg.*, vol. 5, no. 2, pp. 88–96, c3, 2003.

[31] G. van Rossum, *Python Tutorial.* Python Software Foundation.

[32] M. Pilgrim, *Dive into Python.* APress, 2004.

[33] D. Reed, “Rapid application development with Python and glade,” *Linux J.*, vol. 2004, no. 123, p. 1, 2004.

[34] D. M. Beazley, “Interfacing C/C++ and Python with SWIG,” in *7th International Python Conference, SWIG Tutorial*, 1998.

[35] M. Lutz, *Programming Python.* O’Reilly & Associates, Inc., 1996.

[36] D. M. Beazley and G. V. Rossum, *Python Essential Reference.* New Riders Publishing, 1999.

[37] J. Petrone, “3D programming with Python,” *Linux J.*, vol. 2002, no. 94, p. 4, 2002.

[38] G. van Rossum, *Python Library Reference.* Python Software Foundation.

[39] M. Lutz, *Programming Python (2nd edn).* O’Reilly & Associates, Inc., March 2001.

[40] A. Koenig and B. E. Moo, *Accelerated C++: practical programming by example.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[41] K. McDonald, *The Quick Python Book.* Greenwich, CT, USA: Manning Publications Co., 1999.

[42] T. Moller and E. Haines, *Real-Time Rendering.* A. K. Peters Limited, 2002. In press.

[43] S. B. Lippman and J. Lajoie, *C++ Primer: 3rd edition.* Reading, Mass.: Addison-Wesley Publishing Co., 1991.

[44] M. Woo *et al.*, *OpenGL programming guide: the official guide to learning OpenGL, version 1.2.* Reading, MA, USA: Addison-Wesley, third ed., 1999.

[45] T. Fahmy, “3D graphics in Python,” *PyZine*, vol. 2004, no. 6, p. 1, 2004.

[46] E. S. Raymond, *The Cathedral and the Bazaar.* Sebastapol, CA: O’Reilly, 2 ed., 2001. Forward by Bob Young. Full text available online.

[47] Calle Lejdfors, “Techniques for implementing embedded domainspecific languages in dynamic languages,” Master’s thesis, Lund, Feb. 2006.