TINT: Towards a Pure Python Augmented Reality Framework

Ulrich Eck*
University of South Australia
Magic Vision Lab

Christian Sandor†
University of South Australia
Magic Vision Lab

ABSTRACT

This paper describes our software framework, called TINT, which is targeted towards rapid development of augmented reality applications. It is entirely written in the Python programming language and optimized with compiled modules to achieve realtime performance without sacrificing simplicity and maintainability. The design goal for TINT is, to make it possible to develop applications and framework components in pure Python code. This increases the productivity of the developer and is less error-prone.
TINT implements a set of components: dataflow network with hardware sensors, video-capturing, record / playback functionality, presentation library with compositing, networking, and application classes.
In this paper we give a high level overview of TINT and compare it with existing frameworks and libraries. We also elaborate in detail on our techniques to support realtime augmented reality applications written in pure Python and give some examples its usage.

Index Terms: H.5.1 [Multimedia Information Systems]: Artificial, augmented, and virtual realities—; D.3.2 [Language Classifications]: Python, C, C++—; D.3.3 [Language Constructs and Features]: Frameworks—; D.2.10 [Design]: Rapid Prototyping—

1 INTRODUCTION

Many mature frameworks and libraries for research and development of augmented reality (AR) and mixed reality (MR) applications are available today. However, it is difficult to get most of them running on a specific platform, because of their dependent libraries which need to be installed in a supported version. Furthermore, most frameworks are quite complex to get started with for novice developers and still won't allow even advanced programmers to reach their goals quickly.
The problem arises mostly from using compiled and statically typed programming languages which require linking against their dependencies and do not support modifications at runtime unless a plugable component architecture is used or created. On the other hand, interpreted, dynamically typed programming languages are known to be slow in execution speed and therefore are not suitable for realtime graphics applications.
To overcome these drawbacks, we decided to implement a new framework using a dynamically typed scripting language. As Python has a very clear syntax, is object-oriented, platform independent, and simple to extend with custom binary modules, we decided to implement the core framework of TINT entirely in this programming language. Furthermore we followed the design principles for Python as stated in "The Zen of Python"[1] to create a framework that feels pythonic[2] when used. Bruce Eckel said in his keynote [5], that development done in Python is 5-10 times more productive and less error-prone.
TINT is still capable of running AR and MR applications in realtime on most laptop computers, with decent graphics card, because we are offloading computational intensive tasks to either the graphics card, one of the few dependent libraries like numpy[3], or custom platform-independent python modules that are written in Cython[4], C, or C++. Our framework reduces the number of required, binary dependencies to a minimum of four standard libraries and does not force the developer or user of an application to install libraries which are not used.
The rest of the paper is structured as follows: Section 1.1 describes our contribution. Section 2 compares our work with existing frameworks and libraries. Section 3 shows options for Python - C/C++ interoperability and gives a detailed overview of our components and how they work together. Section 4 provides some examples and Section 5 finally states our conclusions and what we plan to improve in future versions of TINT.

1.1 Contribution

Using the Python programming language in AR and MR applications is not new. Other frameworks that use Python, either use it only to implement single components (DWARF [3]), or use a wrapped C++ framework to make the components available in Python (Panda3D [8], AvangoNG [12]). We designed TINT in the spirit of frameworks like Panda3D, but we've taken the ideas one step further: First, we decided, that we want to develop as many components as possible in Python. Second, we wanted to increase the development efficiency (see Section 3.1). Third, we wanted unrestricted interactive shell access (see Section 4.2).
Our contribution is to demonstrate that almost pure Python AR and MR frameworks are possible and desirable. The framework was written from scratch in approximately two man-months development time. It was used to efficiently create a mobile MR navigation system [17]. New team members were able to get started and contribute to the development within less than a week. This is a remarkable improvement over other existing frameworks.
In cases, where the Python implementation is not fast enough, critical code blocks are optimized using Cython. Numeric array operations written in Cython for example, execute up to 500 times faster than a pure python implementation [20].
Our approach might also be interesting for other research projects, which often work with students, making flexibility and simplicity the key design goals of a software framework.

2 RELATED WORK

The scope of TINT is similar to MRPlatform [18], which provides a complete framework for developers of AR and MR applications with sophisticated abstraction and separation of concerns. It is also comparable to frameworks like Panda3D [8], and AvangoNG [12] as it utilizes the Python programming language to ease the development of AR applications and provides a rich set of predefined components to build on. However TINT lacks features

*e-mail: ueck@net-labs.de
†e-mail: Christian.Sandor@unisa.edu.au

[1] http://www.python.org/dev/peps/pep-0020/
[2] http://cafepy.com/article/59/
[3] http://numpy.scipy.org/
[4] http://www.cython.org/

like calibration/registration tools, support for high end hardware like head mounted displays, and does currently not provide a full scenegraph implementation.
The design of our dataflow network as acyclic directed graph of nodes, was inspired by OpenTracker [16] and UNIT [15] in respect to flexibility and configurability; but, it is much easier to use and extend and does not try to adapt to distributed environments transparently. For example a simple source object in Opentracker needs 3 files with about 100 lines of code (LOC) (taken from "How to use Opentracker"[5]), while a similar sensor in TINT requires one file with 15 LOC (see Section 4.1).
Finally, our presentation layer was inspired by the work of [10]. In their work on interactive focus and context visualizations for AR, they used shaders to combine rendered layers using frame buffer objects (FBOs). TINT provides a set of components, that simplify the usage of these technologies and allow developers to create advanced graphics output without knowing all the details of OpenGL FBOs and the necessary infrastructure to do compositing on the graphics card using shaders.
Based on the ubiquitous computing survey [6], we provide a record for TINT to ease the comparison of our framework with other existing frameworks and libraries in Appendix A.

3 System Design

Section 3.1 gives an overview of available options to provide Python - C/C++ interoperability and how to increase the speed of execution for computational intensive tasks. A detailed decomposition of our framework components is given in Section 3.2.

3.1 Python - C/C++ Interoperability

To make a Python application suitable for realtime applications with rich 3D graphics and video streams, the processing of large arrays of data must not be done directly in Python, but offloaded to optimized, external modules. Running the application logic and simple data processing on the other hand is not a problem with refresh rates commonly used in AR applications.
A developer usually starts implementing an algorithm in Python. If the resulting code is not fast enough, it can eventually be optimized by using special data types, like numpy ndarrays. Finally, if the code contains loops over large lists or array operations, small code blocks need to be optimized using a compiled language. We needed a simple way of creating optimized, binary modules that extend our framework. Only time intensive calculations are replaced with compiled code, so that the flexibility which is provided through Python is still a benefit.
To be able to implement algorithms for array manipulations like computer vision or image manipulation we decided to use the Cython language. The Cython language is very close to the Python language, but Cython additionally supports calling C functions and declaring C types on variables and class attributes. This allows the compiler to generate very efficient C code from Cython code.
There is also active development towards a fast just in time (JIT) compiler for Python. The PyPy project[6] adds a JIT for restricted Python code. Unladen-swallow[7] a Python implementation using an LLVM based JIT, which enables further speed up in future without the need of binary modules at all.
Access to existing libraries like libdc1394[8] for video capturing or other drivers for external hardware like trackers or input devices is also required. We evaluated numerous tools that enable interoperability between Python and libraries written in C/C++. There are different ways to access external libraries:
Wrappers take some information from a header- and/or specification-file and create C/C++ code which is then compiled into a Python module. They usually produce fast code with transparent object conversion, but it can be tedious to create the specification files; examples are Swig [4] and SIP[9].
Foreign Function Interfaces (FFIs) can directly use external binary libraries without the need of compiling a module. This is very convenient when optional libraries are accessed. Compared to wrappers the calling overhead is higher, as object conversion needs to be done in Python. There is a default FFI implementation available in Python called ctypes[10].
Libraries require the framework to be implemented in a specific way, so that the library tools can automatically generate Python interoperability. They are usually as fast as wrappers and provide transparent object conversion. The need to create the framework so that is compatible with the library on the other hand, reduces the flexibility, increases complexity or the resulting wrapper feels less pythonic when used. Some libraries need to be recompiled even for minor modifications, which can slow down the development cycle. Examples are Boost.python[11] and Interrogate from Panda3D [8] for C++ interoperability.
Native modules are implemented in C using the Python extension module API [19]. This tends to be error-prone as the developer needs to take care of topics like threading and reference counting when implementing the module. They are fast, but much more work to implement.
We decided to use the FFI ctypes because it does not require compilation. Therefore, optional dependencies can be ignored when installing TINT onto a new computer, when certain functionality is not required.

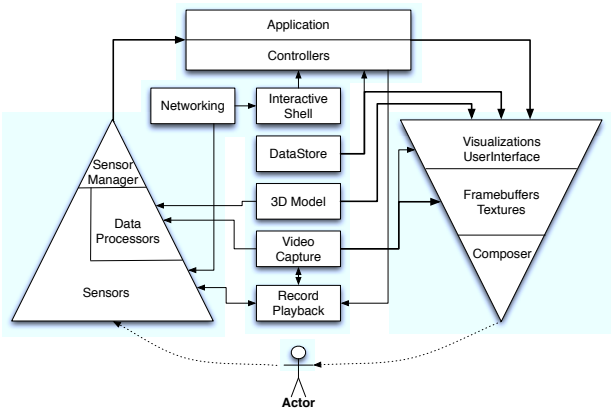

Figure 1: Functional Decomposition of TINT Components. Arrows denote dataflow

3.2 System Decomposition

The big picture of our framework components is shown in Figure 1. They can be grouped into three categories: dataflow components (left), core components (center), and presentation components (right). In the following sections we describe the components of each category and how they work together.

[5] http://studierstube.icg.tu-graz.ac.at/opentracker/html/howto_opentracker.pdf
[6] http://codespeak.net/pypy/dist/pypy/doc/
[7] http://code.google.com/p/unladen-swallow/
[8] http://damien.douxchamps.net/ieee1394/libdc1394/
[9] http://www.riverbankcomputing.com/software/sip
[10] http://docs.python.org/library/ctypes.html
[11] http://www.boost.org/

3.2.1 Core Components

A TINT application is developed using the model view controller (MVC) pattern. The main application provides a context to access the model data, for example textures, 3D geometries, and other entities. Controllers process information received from the dataflow network and define the behavior of a TINT application. They are usually implemented using finite state machines and extend predefined controller classes. The views are implemented using the presentation components (see Section 3.2.3).
The configuration library ZConfig [12] is used to setup the components from configuration files. All applications automatically come with a command line interface, so that there is a uniform way to run TINT applications. To store and retrieve tabular application data, the object relational mapper SQLAlchemy[13] with support for various relational database systems has been integrated.
Most graphical user interface (GUI) libraries allow modifications to their state only from within their own event loop, which causes problems when the application needs to react immediately on events like incoming packets of a network connection. An application that uses threading extensively is difficult to implement because of synchronization issues with the GUI event loop. We decided to implement TINT with two event loops, each one running in a separate thread, as shown in Figure 2: The GUI's main event loop that runs all operations as blocking function calls to handle GUI / OpenGL related tasks and an asynchronous event loop for things like network communication and hardware access. These two event loops are synchronized from within the GUI event loop to ensure all modifications are made in the right context. This approach is similar to the hybrid programming model described in [13], but we don't try to unify both concepts transparently into a programming model. The asynchronous event loop is implemented using a Twisted Reactor [11]. Twisted implements various TCP and UDP network protocols out of the box and it is fairly simple to write your own protocols. We implemented the open sound control[14] (OSC) protocol to enable remote control of application parameters (see Section 4.3). To enable ad hoc peer to peer (P2P) communication, a multicast DNS service discovery (M-DNS SD) client is integrated using pybonjour[15].
External components can be adapted to the system in several ways: A custom protocol can be implemented using Twisted, and integrated into the asynchronous eventloop, so that the external component is directly accessed over the network. Alternatively, an existing driver module can be integrated into the framework; if non-blocking read and write operations are supported, it can be integrated into the asynchronous eventloop, in other cases, blocking calls from the gui eventloop, or synchronized threads are used. Finally a custom wrapper can be created using one of the python interoperability options mentioned in Section 3.1
Video images are captured from the attached camera and provided to the data processors and application controllers in realtime. We currently support Firewire cameras on Linux/OSX and Quicktime Capturing on OSX. Our libdc1394 driver is implemented in pure Python using ctypes to access the external library. To effectively process the captured video image buffers, we use the ctypes property of numpy ndarrays to transfer the buffer into a shaped array on a C-level for further processing. This way, the large arrays of data are not directly accessed from Python and capturing runs at the speed of a binary driver module. The configuration and control of the library is done through regular Python calls as they do not involve larger amounts of data to be transferred.
Furthermore, there are tools to support the developer: an interactive application shell (see Example 4.2)) and a realtime data stream recorder / player (see Section 3.2.2). These tools are available in all TINT applications as they are provided by the core components.

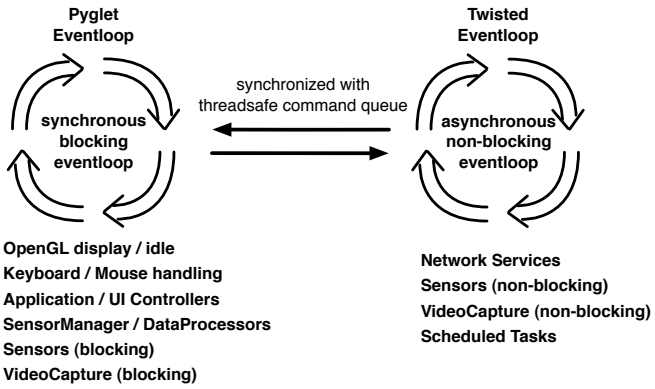

Figure 2: Synchronized Event Loops

3.2.2 Dataflow Network

Similar to Opentracker's approach of using XML files to specify the acyclic graph of source and filter objects, TINT loads its data from the application configuration to set up the dataflow. Sensors (source objects) represent local or remote input devices like trackers, Intersense Cube3 Gyro, GPS devices or Phidgets [9]. Dataprocessors (filter objects) can be used to combine and process data streams from sensors and other dataprocessors. All node types can be implemented with non-blocking, blocking, or threaded behavior, so that external devices can be easily integrated into the framework.
A Sensormanager controls the life cycle of dataflow network nodes and collects information from them when requested by the application controller. It also integrates transparently with the record / playback functionality of the core application classes to support configurable record and playback of data streams. This allows the developer to replay recorded sessions and optimize algorithms on computers without having connected the hardware that is required to run the application.
To record a set of sensor-data- and video-streams, we dynamically create a HDF5 [7] database schema for the dataflow network using pytables[16][1] and subsequently append data to the tables with time stamps. When a session is played back, the original sensor nodes are replaced by playback nodes, which are configured from the metadata in the recorded session. The sensor-data and video-streams are then played back in realtime by subsequently reading the tables contents and setting it as sensor output.

3.2.3 Presentation Components

The view components use the cross platform multimedia library pyglet[17] to provide the graphic context. Views itself are very simple and do not make any assumptions on how the developer will output graphics. Advanced rendering techniques like GLSL shaders and render buffers are supported. The output of a single view is usually rendered into an FBO; the resulting FBOs are then combined in the compositing component before being presented to the user. The default composting component is a GLSL shader that combines the FBOs using alpha blending as illustrated in Figure 3.

[12] http://www.zope.org/Members/fdrake/zconfig/
[13] http://www.sqlalchemy.org/
[14] http://www.opensoundcontrol.org/
[15] http://code.google.com/p/pybonjour/
[16] http://www.pytables.org/
[17] http://www.pyglet.org/

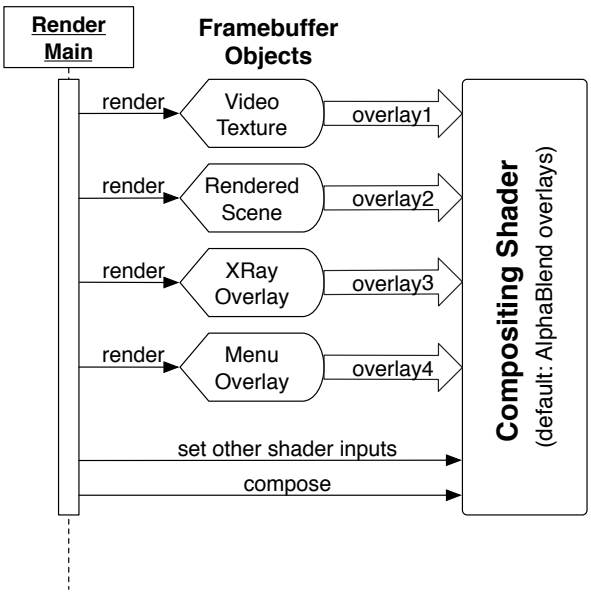

Figure 3: Compositing Component

Advanced compositing components can support techniques like X-ray vision [2] by supplying shader logic that combines the output of the camera texture with the occluded object based on metadata that is provided by the view. Our framework also includes a simple scenegraph implementation that enables loading, querying, and displaying of geometries with textures, and a simple animation engine.

4 EXAMPLES

In this section we show how to create a minimal sensor node and how to change the implementation of such a node at runtime using the interactive Python shell. Finally we describe an iPhone controller example that can be used to interact with a TINT application over a wireless network.

4.1 Minimal Sensor Node

This example illustrates a simple sensor node for the dataflow network, that can be loaded from an application configuration file. This sensor outputs a constant tuple of float values which were specified in the configuration. The implementation of such a sensor needs only 15 LOC:

```
from TINT.sensor import AbstractSensor, NotStartedException

class ConstantFloatSensor(AbstractSensor):
    """ Float Sensor with constant output. """

    def __init__(self, name, value=()):
        """ class constructor. """
        super(ConstantFloatSensor, self).__init__(name)
        self._current_value = tuple([float32(v) for v in value])

    @classmethod
    def fromZConfig(cls, cfg):
        """ config helper """
        return cls(cfg.name, value=cfg.values)

    def describe_output(self):
        """ describe output for constant float sensor. """
        return tuple([('val%d' % i, 'float32') \
                      for i in range(len(self._current_value))])

    def update(self):
        """ just output a constant value. """
        if self.running == False:
            raise NotStartedException(self.name)
        return self._current_value
```

The `AbstractSensor` base class implements all necessary logic to integrate with the life cycle management of the sensor manager. Only two methods need to be implemented: `describe_output` provides information on what type of data the receiving component has to expect and `update` returns the current value as a tuple to the caller.

To register the new sensor with the ZConfig configuration library, an entry in the schema is required:

```
<sectiontype
    name="ConstantFloatSensor"
    datatype=".sensor.ConstantFloatSensor.fromZConfig"
    implements="TINT.Sensor"
    extends="AbstractSensor">
    <description>constant output sensor.</description>

    <multikey
        name="Value"
        datatype="float"
        attribute="values"
        required="no">
        <description>A List of Values.</description>
    </multikey>

</sectiontype>
```

The schema is used to validate given configurations, and to instantiate the specified components into a tree of nodes. An example configuration to set up such a sensor would look like:

```
<SensorManager>
    ...

    <ConstantFloatSensor>
        name demo_sensor
        value 1.0
        value 2.0
        value 3.0
    </ConstantFloatSensor>

    ...
</SensorManager>
```

4.2 Interactive Python Shell

While experimenting with new algorithms, the developer often needs to change the behavior of the newly created component. Some frameworks, e.g. DWARF [3], allow to parameterize components and modify these parameters during runtime. If the implemented algorithm needs to be revised, the developer needs to recompile, and restart the application. This can be tedious when trying to enhance parts like tracking quality or other improvements with visual feedback.

Our framework implements an interactive Python shell that is available locally or through a network connection using telnet or ssh. Once the user opens such an interactive Python shell, they are presented a command prompt similar to starting the interpreter from the command line. The interactive shell binds a context that allows the developer to access the instantiated components of the running application instantly. Using this feature, the developer can modify parameters by setting the relevant attributes directly to new values. Furthermore it is possible to change the implementation of instances or classes due to Python's dynamic nature. A new implementation could be activated by using the `reload` command in Python to reimport a modules implementation from a file or it can be done interactively as shown below:

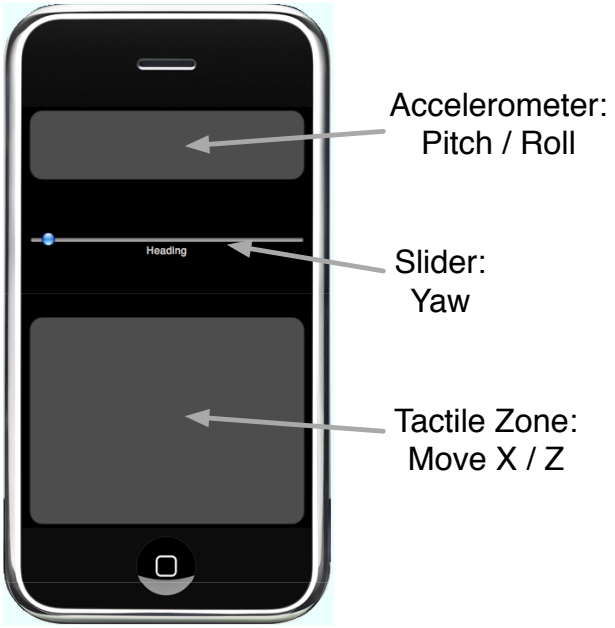

Figure 4: Our iPhone controller can be used to conveniently simulate components. This example shows a 5DOF sensor input as tracker replacement.

```
>>> s1 = sensormanager.get_sensor('demo_sensor')
>>> s1.update()
(1.0, 2.0, 3.0)
>>> old_func = s1.update
>>> def new_func(self):
...     return tuple([x*2.0 for x in old_func(self)])
...
>>> s1.update = new_func
>>> s1.update()
(2.0, 4.0, 6.0)
```

In the first line, a reference to the `demo_sensor` is assigned to the variable `s1` and the second command fetches the current value from the sensor. The next line assigns a reference of the function object `update` to the variable `old_func`. Then a new implementation `new_func` of `update` is defined, which doubles the values of the tuple returned from the old implementation `old_func`. It is activated by overwriting the `update` attribute for this specific instance `s1` of the `demo_sensor`. The last line finally fetches the current value from the modified sensor.
When the modifications are sufficient, the code snippet can be copied into the source file of the sensor. This option applies to all framework classes that are implemented in Python.

4.3 iPhone Controller

Our iPhone controller can be used to remote control a running TINT application. It is similar to a personal unified controller (PUC), as described in [14]. The controller automatically connects to the application and creates an application specific user interface.
The client application is realized with the Mrmr Client[18] on an iPhone. It automatically connects to the TINT application using a M-DNS SD request and uses the Mrmr protocol to talk to the application. The Mrmr protocol is an extension to the OSC protocol, which defines a user interface description language and a naming scheme for the transferred data. A Mrmr sensor component in a TINT application defines a schema for automatically creating the UI and mapping back the incoming data to the sensors output.
The implementation of this feature took only two days. During the development of our mobile MR navigation system [17], it was extremely useful. When developers wanted to experiment with outdoor user interface ideas at their desktop PCs, they needed to simulate outdoor trackers. Our iPhone controller (see Figure 4) enabled them to do this conveniently: Roll, pitch, and yaw simulate input from a Gyroscope; X and Z movement simulate GPS tracking. GPS measurements in the up-direction (Y) were too unreliable for our purposes; therefore, we have hardcoded them to the typical height of a human (1.80 meters).

5 Future Work and Conclusions

We have presented our framework TINT for rapid prototyping of AR and MR applications. Due to its clean and simple implementation it is easy to get started with and simple to extend. Our framework has been successfully used to create several smaller tools and a mobile MR navigation system [17]. The navigation system runs at 20 fps on an Intel Pentium-M 2.0 GHz computer with Nvidia GeForce 6600 graphics; Videos and detailed information can be found on our website[19].
While this has been a great success for the framework, we realized that there are several things that should be improved: we favored a simple inheritance paradigm to build the framework classes. A better approach is to use the interface and adapter patterns to decouple the implementation while improving the interoperability of the framework components. An event dispatcher should be used to further decouple the components. Furthermore, a sophisticated scenegraph implementation is needed for rapid prototyping. Existing components will be extracted from our applications and added to the framework, to increase the amount of reusable components and device drivers. As most core components are completely platform independent, a working version for Windows will be implemented soon, together with some native device drivers.
In the future we plan to integrate a CUDA or OpenCL adapter to enable offloading of computational intensive image manipulation and tracking tasks directly to the GPU and therefore minimize the need of custom extension modules. As the Python language evolves and has reached the version 3.1 recently, new features are available; e.g. new buffer interface that is integrated directly into the language which will simplify the transfer of large array data between external libraries and numpy arrays. These features can be used once our main dependencies reach a stable version which is compatible with Python 3.1. Other research projects like JIT compilation of Python code using PyPy to speed up pure python modules as described in [13] need to be investigated, as they tend to be even more pythonic than our current prototype.

References

[1] F. Alted. Keynote: On the data access issue (or why modern cpus are starving). EuroSciPy, 2009.
[2] B. Avery, C. Sandor, and B. H. Thomas. Improving spatial perception for augmented reality x-ray vision. In *Proceedings of IEEE VR*, pages 79–82, March 2009.
[3] M. Bauer, B. Brügge, G. Klinker, A. MacWilliams, T. Reicher, C. Sandor, and M. Wagner. An architecture concept for ubiquitous computing aware wearable computers. In *Proceedings of ICDCSW*, pages 785–790, Washington, DC, USA, 2002. IEEE Computer Society.
[4] D. M. Beazley. Swig: an easy to use tool for integrating scripting languages with c and c++. In *Proceedings of USENIX TCLTK*, pages 15–15, Berkeley, CA, USA, 1996. USENIX Association.
[5] B. Eckel. Keynote at the 9th international python conference. 2001.

[18] http://poly.share.dj/projects/#mrmr

[19] http://www.magicvisionlab.com/projects/mars/

[6] C. Endres, A. Butz, and A. MacWilliams. A survey of software infrastructures and frameworks for ubiquitous computing. *Mob. Inf. Syst.*, 1(1):41–80, 2005.
[7] M. Folk, R. McGrath, and N. Yeager. Hdf: an update and future directions. In *Proceedings of IEEE IGARSS*, volume 1, pages 273–275 vol.1, 1999.
[8] M. Goslin and M. R. Mine. The panda3d graphics engine. *Computer*, 37(10):112–114, 2004.
[9] S. Greenberg and C. Fitchett. Phidgets: easy development of physical interfaces through physical widgets. In *Proceedings of ACM UIST*, pages 209–218, New York, NY, USA, 2001. ACM.
[10] D. Kalkofen, E. Mendez, and D. Schmalstieg. Interactive focus and context visualization for augmented reality. In *Proceedings of ISMAR*, pages 1–10, Washington, DC, USA, 2007. IEEE Computer Society.
[11] K. Kinder. Event-driven programming with twisted and python. *Linux J.*, 2005(131):6, 2005.
[12] R. Kuck, J. Wind, K. Riege, and M. Bogen. Improving the avango vr/ar framework: Lessons learned. In *Virtuelle und Erweiterte Realität : 5. Workshop der GI-Fachgruppe VR/AR*, Magdeburg, DE, 2008.
[13] P. Li and S. Zdancewic. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. In *Proceedings of ACM PLDI*, pages 189–199, New York, NY, USA, 2007. ACM.
[14] J. Nichols, B. A. Myers, M. Higgins, J. Hughes, T. K. Harris, R. Rosenfeld, and M. Pignol. Generating remote control interfaces for complex appliances. In *Proceedings of ACM UIST*, pages 161–170, New York, NY, USA, 2002. ACM.
[15] A. Olwal and S. Feiner. Unit: modular development of distributed interaction techniques for highly interactive user interfaces. In *Proceedings of GRAPHITE*, pages 131–138, New York, NY, USA, 2004. ACM.
[16] G. Reitmayr and D. Schmalstieg. Opentracker-an open software architecture for reconfigurable tracking based on xml. In *Proceedings of IEEE VR*, page 285, Washington, DC, USA, 2001. IEEE Computer Society.
[17] C. Sandor, A. Cunningham, U. Eck, D. Urquhart, G. Jarvis, A. Dey, S. Barbier, M. Marner, and S. Rhee. Egocentric space-distorting visualizations for rapid environment exploration in mobile mixed reality. In *To appear in VR 2010: Proceedings of the IEEE Conference on Virtual Reality*, Waltham, MA, USA, March 2010.
[18] S. Uchiyama, K. Takemoto, K. Satoh, H. Yamamoto, and H. Tamura. Mr platform: A basic body on which mixed reality applications are built. In *Proceedings of ISMAR*, page 246, Washington, DC, USA, 2002. IEEE Computer Society.
[19] G. van Rossum. Extending and embedding the python interpreter release 2.5. Technical report, Python Software Foundation, 2006.
[20] I. M. Wilbers, H. P. Langtangen, and A. Odegard. Using cython to speed up numerical python programs. 2008.

A CLASSIFICATION BASED ON THE UBIQUITOUS COMPUTING SURVEY:

A.1 Type and background

Group / company: Wearable Computer Lab, University of South Australia, Adelaide, Australia
Time / manpower: Since Feb. 2009, 2-3 active developers
Development focus: Research
Research goals: Develop prototypes for new MR and AR techniques.
Contact: Christian Sandor
Target environment: Mobile mixed and augmented reality.
System description: TINT is a lightweight framework for augmented reality application prototyping. TINT consists of a number of components that help developers to create new AR applications in Python and C/C++ easily. All compo- nents have a Python implementation - some may also have a C/C++ implementation if higher performance is needed. There are several options to access different hardware de- vices via sensors and process their output within data pro- cessors. An advanced configuration system sets up the com- ponents which allows easy adoption of different platforms and environments. There is extensive record- and playback- functionality built into the main application components, so that all sensors and captured video images can be recorded into sessions and played back later. Also TINT uses best-of- breed 3rd party components/libraries that fit into the frame- work nicely to avoid duplicate efforts and increase the sta- bility of the overall framework.

A.2 Underlying technology

Language: Python, Cython, C/C++
Network protocol: Various TCP and UDP protocols
Supported platforms: Linux, Mac OSX
Scalability: Single host, single process
Underlying paradigm: Create applications and framework extensions in pure python and replace performance critical parts with Cython/C implementations.

A.3 Components

Types: Python objects provided by the framework.
Granularity: Medium (sensors, dataprocessors, video capture, menus, views and controllers)
Description: Developer creates app by using Framework components.
Instantiation: By application provided from developer.
Configuration: Platform specific and global settings are loaded from schema validated config files.
Communication / lookup: P2P communication, multicast-dns discovery

A.4 General information

Accessibility: Not specified
Level of abstraction: The programmer works with reusable framework classes and libraries.
Modules and services: Application support (menus, controllers, config), dataflow network (sensors, dataprocessors), 2D/3D rendering (framebuffer composer, simple scene graph)
Suitable for: TINT is designed to create prototypes for future mobile devices and runs on standard pc/mac hardware.
Key publications: This paper.