KineTcl

Andreas Kupries ActiveState Software Inc. 409 Granville Vancouver, BC CA
andreask@ActiveState.com

ABSTRACT

This paper describes a package enabling Tcl scripts to talk to Microsoft's `Kinect` and related devices.

Technically `KineTcl` is a binding to the `OpenNI` framework and thus provides access to all depth sensor devices for which a sensor plugin exists. The best known device so far in that category is the `Kinect`.

The paper will describe the internal structure of the package (i.e. how it matches to the `OpenNI` API, and weaves both C and `Tcl` [12] together to make use of each others strengths) and point to supporting packages and tools used in the implementation.

1. OVERVIEW

`KineTcl` [2] is a new Tcl package providing a binding to Microsoft's `Kinect` [9], and related devices.

The project was started at the behest of the National Museum of Health and Medicine, Chicago[1] (short: `NMHMC`) for use in its exhibition space as one of the pieces of software linking real world activities and actions to interactive virtual displays.

Research into existing open source software for `Kinect` located two existing projects, `OpenKinect` [5] (aka `libfreenect`), and Open Natural Interaction (`OpenNI` [6]).

`OpenKinect` was created by the OSS and OSH communities through reverse engineering the Kinect's USB protocol. It is a low-level library providing access to the device without having to care about this USB protocol and the like. While not quite as low-level as a driver, it is not much higher. The developers have planned an analysis library for higher level operations (e.g. user detection and gesture recognition) but this was not yet implemented at the time of the research.

`OpenNI`, is a framework abstracting away from hardware devices and image processing for particular tasks (like user-, hand-, and skeleton-tracking). It was created and is maintained by `PrimeSense` [8], the developer and manufacturer of the depth sensor used in the Kinect. `OpenNI` is also "an industry-led, not-for-profit organization formed to certify and promote the compatibility and interoperability of Natural Interaction (NI) devices, applications and middleware."[6]. Both the framework itself and a generic sensor driver "node" for the `PrimeSense` sensor are available in source, under the LGPL. A derivative of the latter, specialized to the `Kinect` is available on github[10].

At this point of the research both possibilities were seen as roughly equivalent.

`OpenNI` was chosen because of the existence of the `NITE` [11] extensions, encapsulating all of the necessary higher level algorithms (i.e user detection, skeleton/joint tracking, gesture recognition, etc).

Given the time frame of the project (started in January 2012, a working system needed by May) it was considered difficult or impossible to invent and write such algorithms from scratch, as would be needed when using `libfreenect`. Having access to these through `NITE` outweighed the consideration that this part of the system is only available in binary, and not in source.

The next chapter gives a general overview of `KineTcl`'s design, implementation, and features. Following that, chapters 3 and 4 discuss limitations, possible applications and future directions for the package.

2. DESIGN & IMPLEMENTATION

2.1 OpenNI

`OpenNI`'s API is written in C, with an underlying class hierarchy [1] where the leaves represent the various data streams coming from a depth sensor, and the higher classes provide the general functionality and APIs. This is shown in figure 1. Note that the classes not only represent data streams from physical sensors, but also data coming out of higher level algorithms like user detection and tracking (i.e. virtual sensors).

These APIs contain the mandatory minimum supported functions for each class. For sensors going beyond these, `OpenNI` defines a series of standard "capabilities" they may provide. From a different point of view these could be called aspects, or mixins. As an example, figure 2 shows the capabilities which are defined for user detection and tracking.

For full details, see `OpenNI`'s reference documentation[7].

2.2 Basic Design

Generally all `OpenNI` classes and instances are represented as classes and instances to the Tcl script as well. Whenever

[1]Underneath the C API is actually C++

Tcl '2012 Chicago, IL, USA
.

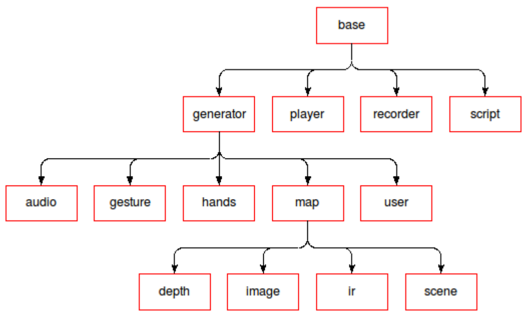

Figure 1: OpenNI class hierarchy

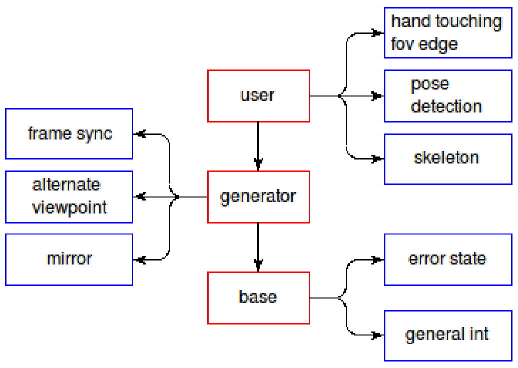

Figure 2: OpenNI User Tracking Capabilities

we mention a class in the future, we will also specify which of the three layers (`OpenNI`, C, or Tcl) we are talking about if it is not clear from the context.

Following the spirit of Poli-C [13] the binding is written using layers, with a low-level C layer implementing only the bare necessities which are then glued by the Tcl layer into the final user-visible API.

As mentioned, the C layer wraps each `OpenNI` "class" (which includes capabilities) into a Tcl class command whose methods map pretty much directly to `OpenNI` API functions. This is very much like Tk widgets. However, these classes do not know about the class hierarchy and superclasses. Each C class implements a binding to just the methods of their `OpenNI` class without regard for inherited methods.

This layering and the connections between the parts in the different layers is shown in figure 3, using the stack of classes for "depth image generator" nodes as example. We see not only the classes, but also the inheritance relationships (in blue), including the fact that `KineTcl`'s C layer does not use inheritance, and the use of instances (in red). The Tcl level depth image instances contain the C level instances of their class and all the required superclasses, which share the `OpenNI` handle for the node. This last point will be explained further in section 2.3.

This, and the mixin of the supported capabilities, is all handled in the Tcl layer. Here all the underlying classes are wrapped by `TclOO` [14] classes which instantiate all the required C classes so that the user may have access to the full set of methods, direct and inherited. The connection from the externally visible methods to the C methods is done through TclOO forwards, which also allows us to hide

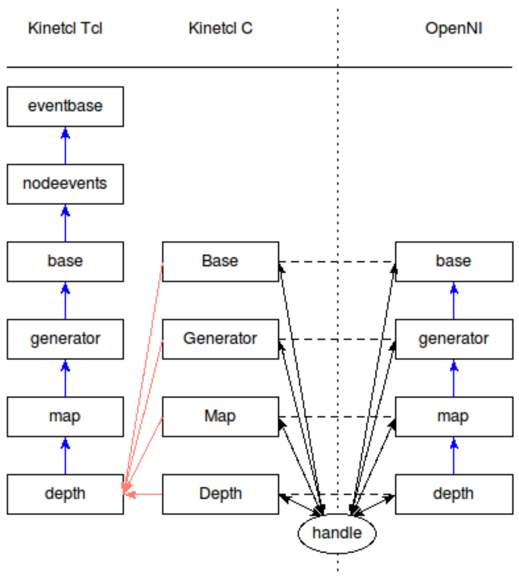

Figure 3: Package Layering

all special C methods needed by the Tcl layer which are irrelevant from the user's perspective. This includes, for example, the various introspection methods used to manage callbacks/events and capability mixing.

2.3 Object Construction

One tricky point in all this is that the various C instances constructed for the Tcl instance all have to operate on a single `OpenNI` handle for the object in question (see figure 3). How do we disseminate this information?

First, only the leaf C classes can create a new handle, a property the binding inherits directly from `OpenNI`. Knowing that the Tcl glue will construct the leaf first then walk up the Tcl class hierarchy to construct the required C level superclass instances, the code for a leaf class saves the obtained handle into a per-interpreter structure of the package. The superclasses' code then retrieves the handle from there. Doing things in this manner avoids having to expose and pass a C level pointer through the Tcl layer.

It should be further noted that the C base class provides a special method (`@unmark`) to explicitly clear this handle store. This is not done automatically by the C base class during its construction, because of the capabilities. The handle storage has to be kept around until the Tcl glue has mixed them in, thus the responsibility to signal its release falls to the Tcl layer.

2.4 Object to Handle Conversion

Another issue which has to be solved in the cooperation of C and Tcl layers is that various `OpenNI` (and thus C) methods take a second handle as input, requiring us to convert from a Tcl object command to the underlying `OpenNI` handle.

At the C level, this is managed by calling out to the Tcl procedure `::kinetcl::Valid` which performs both validation of a Tcl_Obj* as a proper Tcl object (command) and its conversion, leaving the resulting `OpenNI` handle in the same storage area as used during object construction. The caller can retrieve it from there after the procedure returns.

At the Tcl level, `::kinetcl::Valid` uses a dictionary of the active instances managed by the base class to validate the argument as a Kinetcl object. For the arguments passing this test `::kinetcl::Valid` then uses its knowledge of the Tcl object internals, namely the existence and name of the C base class instance in the object to directly access it and invoke the special C method (`@mark`) which will store the desired handle in the storage area for the C level to pick it up from.

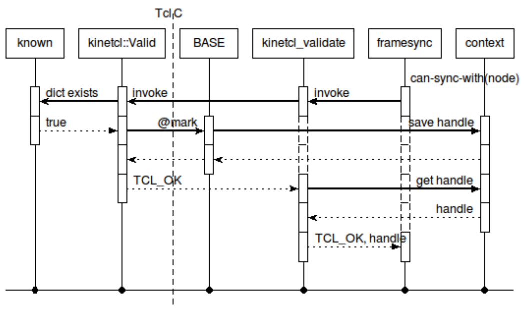

Figure 4: Object to Handle validation and conversion

Figure 4 shows all of the above in a UML sequence diagram.

2.5 Events and Callbacks

The last area of cooperation to talk about are the 34 `OpenNI` callbacks. Unfortunately, they are invoked from `OpenNI`'s internal threads, making it impossible to use them "as is" (i.e. let them directly call up into Tcl).

This issue was mainly solved by converting the callbacks into events, for which we have Tcl API functions to safely enqueue them regardless of which thread they come from and are going to. However, even with that we had two problems left.

First, one of the callbacks is very high-rate, generated several times per second. I am talking about the 'new frame' event for all the map generators, signaling the presence of a new image frame (image, depth, IR, ...). Because a single such signal is good enough this event is throttled by allowing only one per object into the event queue and discarding the remainder until the event in the queue has been processed.

The other remaining issue arises again from the fact that events are generated by threads outside of Tcl's control. It means that new events not only can, but will arrive while Tcl is processing the queued events. Without safeguards Tcl's event queue will never be empty, and the processing loop will never end, starving out idle-events processing.

While a solution was found for this, it doesn't look very nice. Readers of the example applications will see code like that shown in listing 1. This is essentially an emulation of Tcl's event loop using `while` and `update`, and inserting the necessary calls to (a) drive `OpenNI`'s processing (`waitUpdate`) and (b) safeguard (estart, estop) Tcl's event loop while processing events. `estop` causes the system to defer incoming events into a spill-over queue, whereas `estart` restores regular processing and moves all defered events into the main Tcl event queue.

With the pressure for getting a working system now gone,

Listing 1: Event loop

```
while {1} {
   kinetcl waitUpdate
   kinetcl estop
   update
   kinetcl estart
}
```

better solutions for the event integration should be investigated (e.g. Tcl's API for "Event Sources").

While `OpenNI`'s C API for callbacks allows the registration of an arbitrary number of actual callbacks for a specific event the C classes were kept simpler, handling only one actual callback per specific event, managed by associated set and unset methods.

The distribution of events to many observers is then again handled by the Tcl glue code, in two TclOO classes which are superclasses to the nomimal Tcl base class for `OpenNI` instances (see figure 3). These two classes, `kinetcl::eventbase` and `kinetcl::nodeevents`, provide a more event-like API, where users can `bind` to and `unbind` from events. The various Tcl sub-classes register the events they support with them, after using the C classes' method introspection facilities to determine this set. A small detail of the implementation is that a C level callback is set if and only if observers have been bound to the event it will be invoked for. This part of the functionality relies on a feature of the internally used `uevent` [15] package. That is, its ability to watch for and invoke commands when event bindings are set and removed (available since version 0.3.1).

2.6 Implementation

Now, how do we implement 39 C classes (14 core, 25 capabilities) quickly yet safely, especially in light of the large amount of virtually identical boilerplate needed to manage the class and instance commands and associated data structures?

By automating as much as possible.

Thus, a significant part of the time was not spent on writing the binding directly, but on writing the `critcl::class` generator package to encapsulate all the boilerplate and its templating. Having this generator in place, writing the binding became almost trivial, at least in most places. An only slighly abbreviated example is shown in listing 2.

Please note that the code in this listing represents the state of the Kinetcl head and of the `critcl::class` head officially released with `critcl 3.1 [4]`, which also makes use of the additional features for custom argument and result type processing.

The code currently in use by the NMHMC, found at the tag "nmhmc" in the `KineTcl` and `critcl repositories` is less streamlined, containing various argument- and result-processing C code fragments multiple times. For the class shown, the difference is only about half a kilobyte (4 versus 4.5 KB). This class gets converted into roughly 25 KB of C code. From this we can estimate that about 84% of the result is boilerplate code, generated, instead of manually written.

This was further simplified by agressively using Tcl's meta coding abilities to factor out the common parts of the various classes (leaf vs inner classes, the integer capability classes),

Listing 2: kinetcl::map implementation excerpt

```
critcl::class def ::kinetcl::Map {
    ::kt_abstract_class

    method bytes-per-pixel proc {} int {
        return xnGetBytesPerPixel (instance->handle);
    }

    method modes proc {} ok {
        XnStatus s;
        int lc;
        Tcl_Obj** lv = NULL;
        XnMapOutputMode* modes;

        lc = xnGetSupportedMapOutputModesCount (instance->handle);
        if (lc) {
            int i;

            modes = (XnMapOutputMode*) ckalloc (lc * sizeof (XnMapOutputMode));
            s = xnGetSupportedMapOutputModes (instance->handle, modes, &lc);
            CHECK_STATUS_GOTO;

            lv = (Tcl_Obj**) ckalloc (lc * sizeof (Tcl_Obj*));
            for (i = 0; i < lc; i++) {
              ...
            }

            ckfree ((char*) modes);
        }

        Tcl_SetObjResult (interp, Tcl_NewListObj (lc, lv));

        if (lc) {
            ckfree ((char*) lv);
        }

        return TCL_OK;
    error:
        ckfree ((char*) modes);
        return TCL_ERROR;
    }

    method @mode? proc {} ok {
        XnStatus          s;
        XnMapOutputMode mode;
        Tcl_Obj* mv [3];

        s = xnGetMapOutputMode (instance->handle, &mode);
        CHECK_STATUS_RETURN;

        ...

        Tcl_SetObjResult (interp, Tcl_NewListObj (3, mv));
        return TCL_OK;
    }

    method @mode: proc {int xres int yres int fps} XnStatus {
        XnMapOutputMode mode;

        mode.nXRes = xres;
        ...

        return xnSetMapOutputMode (instance->handle, &mode);
    }

    ::kt_callback mode \
        xnRegisterToMapOutputModeChange \
        xnUnregisterFromMapOutputModeChange \
        {} {}

    support {
        #define kinetcl_NUM_PIXELFORMATS (5)
          ...
    }
}
```

and generating the whole of the callback support from short descriptions as seen in listing 3.

Listing 3: Callback definition

```
::kt_callback user-enter \
    xnRegisterToUserReEnter \
    xnUnregisterFromUserReEnter \
    {{XnUserID u}} {
        CB_DETAIL ("user", Tcl_NewIntObj (u));
    }
```

This last was made relatively simple by the very regular nature of `OpenNI`'s API for the (de)registration of callbacks, including the callback signatures. Even the places where two or even three callbacks were managed by a single pair of (de)registration functions could be fitted in.

3. LIMITATIONS

A number of `OpenNI`'s features were not given full attention, or not implemented at all, because `KineTcl`'s intended use in the NMHMC did not require them. These are:

1. The audio, player, recorder, and script classes are mainly shells without full implementation. They are certainly not tested.

2. Instances are constructed using only default arguments. `OpenNI` actually has an API allowing the user to configure a query object/structure to limit the search for the type of instance to specific vendors, versions, and the like. None of this is used.

   Create a "user generator", for example, and the system will simply provide a handle it believes is the best.

3. Similarly `OpenNI` has functionality to query it for the set of installed modules, their vendors, versions, provided node types, etc. This also includes the ability to query what node stacks exist (i.e. coherent collections of nodes able to perform a task). For example, a "hands tracker" may need a "user generator" and if multiple modules provide implementations of either, `OpenNI` can construct different processing networks (node stacks) by mixing and matching them.

   None of this functionality is exposed by `KineTcl`.

4. FUTURE DIRECTIONS

Some of the things we can/may do in the future of `KineTcl` are obvious. Just look at the limitations listed in the previous chapter.

Another relatively obvious direction is to write additional processing classes directly in Tcl (e.g. implement various types of gesture recognition). Some work on this has actually been done, but is not complete (and buggy). See the files `stance.tcl` and `examples/dance` for the experiment with a `FAAST` [16] inspired system.

Finally, there is the currently used hack for the final integration of events. Better solutions for this, such as Tcl's API for "Event Sources", should be investigated.

APPENDIX

A. REFERENCES

[1] National Museum of Health and Medicine, Chicago `http://www.nmhmchicago.org/`

[2] Andreas Kupries, KineTcl. `https://chiselapp.com/user/andreas_kupries/repository/KineTcl`

[3] Andreas Kupries, CRIMP. `http://wiki.tcl.tk/crimp`

[4] Andreas Kupries, Steve Landers, Jean-Claude Wippler, CriTcl. `http://jcw.github.com/critcl/`

[5] Various. OpenKinect, libfreenect. `http://openkinect.org/wiki/Main_Page`

[6] PrimeSense. OpenNI organization and framework. `http://www.openni.org`

[7] PrimeSense. OpenNI API Reference. `http://openni.org/Documentation/Reference/index.html`

[8] PrimeSense. `http://www.primesense.com`

[9] Microsoft. Kinect. `http://www.xbox.com/en-US/kinect/`

[10] Avin. SensorKinect. `https://github.com/avin2/SensorKinect`

[11] PrimeSense. NITE. `http://www.primesense.com/technology/nite3`

[12] Various, Tcl. `https://tcl.sourceforge.net`

[13] Jean-Claude Wippler, Poli-C. `http://wiki.tcl.tk/polic`

[14] Donal Fellows, TclOO `http://core.tcl.tk/tcloo`

[15] Various, Tcllib. `https://tcllib.sourceforge.net`

[16] ICT, Flexible Action & Articulated Skeleton Toolkit `http://projects.ict.usc.edu/mxr/faast/`