# Supercollider Environment for Tangible Objects

Till Bovermann, Bielefeld University

In this subsection we want to give an insight into interface techniques as they are needed for tangible computing setups. Beginning with, we motivate the usage of Tangible Computing and show a possible setup. Then we will look into a small example on how to implement a basic Tangible Computing Environment in SuperCollider. Finally, the SuperCollider Environment for Tangible Objects (SETO) is presented and its application shown at hand of two example setups.

Tangible Computing means to use physical objects for manipulating digital information, or–from a different point of view–to enhance physical objects with digital functionality. This on first sight very simple idea turns out to be a powerful approach for the design of complex, yet still naturally operate-able interfaces. Due to the fact that physical objects are the main part for user interaction, all tangible interfaces are multi-user, multi-handed without the need to for the programmer to explicitly implement it. While it is still possible to design uncomfortable systems, the user's and therefore the programmer's everyday knowledge on physical objects and their handling encourages an intuitive interface design. This makes it a strong tool for HCI. Often, alongside the design process, a natural display of the digital system's state via the physical object's relation appears. The input device turns into a bidirectional component, featuring also a display of the system. All these features support that the user develops a feeling of flow as described by Csikszentmihalyi [csi], just as known by playing a traditional musical instrument. Last not least using sound for the display of data manipulation via tangible computing is promising, because the user is naturally surrounded by sound and it is easy to perceivably connect object manipulations with the change of soundscape.

Unfortunately, Tangible Computing needs more attention for its technical realization then controlling concepts. First of all, an object tracking system is needed providing real-time information about physical object manipulation to the system. The type of tracking software depends on the objects, the frame-rate for pseudo-continuous data and the desired degrees of freedom of which should be tracked, however also the amount of money willing to spend has to be considered. A variety of systems may be taken into account, lasting from cheap, one camera, vision-based systems to many-camera or electronic-based marker tracking systems by far out of the financial scope for private use. A relatively low-cost variant is the open-source software reacTIVision [react], for which a (fast) camera, appropriate lighting, objects to be tracked and a glass surface are required.

Apart from the ability to observe more or less DOFs, the tracking system does not influence the resulting sounds, but the quality of the interaction and therefore the flow experience may drastically differ. This is mostly due to different tracking rates (10Hz vs. 120Hz) and latencies (2ms vs 50ms). Since the tracking process is done in a separate process (probably even not on the same computer), an interface between the tracking system and SuperCollider is needed.

TUIO is a protocol designed just for this purpose, i.e. to transmit tracked physical object states between an object recognition system and the application to be controlled, in our case SuperCollider. Strictly speaking, TUIO is a specification of a fixed set of URL-style commands and method names for OSC focusing particularly on reliability and performance. Table 3.1 shows the complete set of commands available in TUIO.

| Parameter | Type | Description | Range |
|---|---|---|---|
| S | int32 | Object ID | |
| I | int32 | Class ID | |
| x, y, z | float32 | Position | (0…1) |
| a, b, c | float32 | Angle in Euler-not. | (0…2pi) |
| u, v, w | float32 | Angle in axis-not. | |
| X, Y, Z | float32 | Velocity (pos.) | |
| A, B, C | float32 | Velocity (rot.) | |
| M | float32 | Acceleration (pos.) | |
| R | float32 | Acceleration (rot.) | |
| P | Defined by OSC | Free Parameter | |

| Identifier | Key | Parameters |
|---|---|---|
| /tuio/[profileName] | “set” | <ID> [Parameters] |
| /tuio/[profileName] | “alive” | [ID … ID] |
| /tuio/[profileName] | “fseq” | <int32> |

Table1

The command string (e.g. "_ixya") refers to the set of parameters each object provides, and is in conjunction with the object's ID the unique identifier for the objects. For a valid transfer the object tracker has to send a set-message for each object change (including appearance/vanishing) and an alive-message containing the IDs of all available objects. All other messages are optional. This makes it relatively easy to implement a basic interface for TUIO, especially in SuperCollider, since it has native OSC support. Assuming that the object tracking system sends TUIO messages of '_ixya' a patch modulating the frequency of a corresponding sine wave by the rotation angle (a) of an object may be written as shown in the following listing:

```
SynthDef(\testTUIO, {|freq = 400, out = 0, amp = 0, vol = 0.25|
    Out.ar(out, SinOsc.ar(freq, 0, amp*vol)!2)
}).send(s);

(
var synths, numObj = 4, resp;

// Create a Synth for each object. make sure it will not play.
synths = Array.fill(numObj, {
    Synth(\testTUIO, [\vol, numObj.reciprocal, \amp, 0])
});

// set up OSC responder
resp = OSCresponder(nil, "/tuio/_ixya", {|time, resp, msg|
    var id, classID, x, y, a, pos, amps;

    // if object state is updated, change frequency of corresponding synth
    (msg[1] == \set).if{
        # id, classID, x, y, a = msg[2..6];
        synths[id].set(
            \freq, a.wrap(0, 2pi).linexp(0, 2pi, 400, 800)
        )
        };

        // only play synths for alive (i.e. visible) objects
        (msg[1] == \alive).if{
             amps = Array.fill(numObj, 0);
             msg[2..].do{|i|
                  amps[i] = 1;
             };
             synths.do{|synth, i|
                  synth.set(\amp, amps[i])
             }
        };
    }).add;
    )
```

The described approach does what is expected, however, it does not make use of the whole potential the TUIO interface definition provides, for example to manage object representations and interactions. For this, a more elaborated system like *SETO* can be used, which decouples functionality from implementation details by providing a higher level of abstraction, such that the user can concentrate more on the intended functionality then on the low-level features. SETO therefore takes care of object changes including their visibility and provides interfaces to actions and interactions of objects.

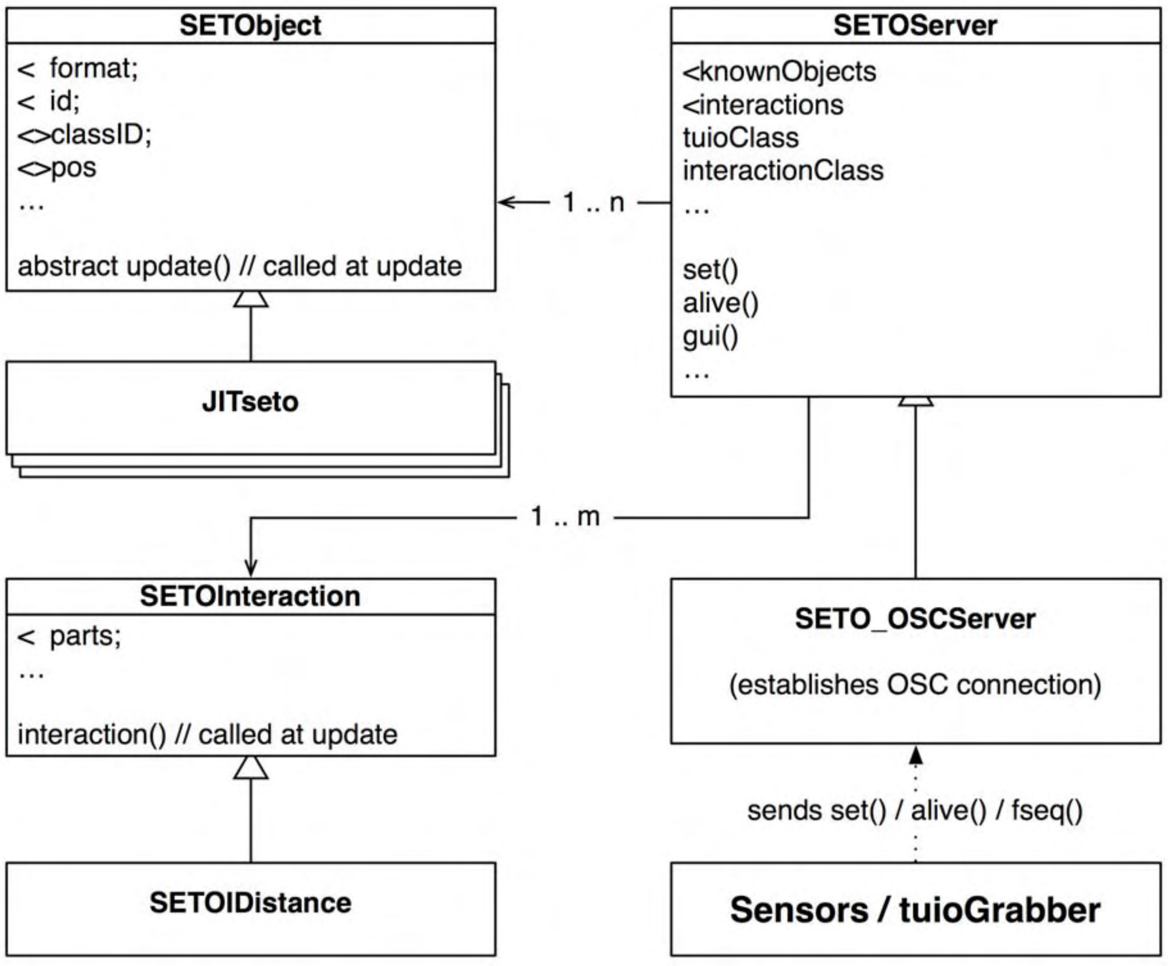

Figure 1: Dependency Diagram of SETO classes

As shown in Figure 1, SETO mainly consists of three types of classes. Each tracked object is represented by an instance of a class derived by **SETObject**. Digital added behavior, e.g. the modulation of a corresponding sound depending on the physical object's motion can be implemented either as a fixed method of a custom class derived from **SETObject**, or by implementing the **action** of **JITseto**, a class providing just-in-time functionality for tangible objects. Interactions between physical objects may be defined alike, but by deriving from **SETOInteraction**.

With SETO, the above shown example can be written as

```
SynthDef(\testTUIO, {|freq = 400, out = 0, amp = 0, vol = 0.25, famp=1|
	Out.ar([out, out+1], SinOsc.ar(freq, 0, (amp.lag(0.01)*vol*famp)))
}).send(s);

q = ();
(
q.synths = IdentityDictionary.new; // a storage for synths

JITseto.action = {|me|
	s.bind{
		// make sure there is a synth
		q.synths[me].isNil.if{
			q.synths[me] = Synth(\testTUIO, [\vol, 0.2, \amp, 0])
		};
		s.sync;
		me.visible.if({
			q.synths[me].set(
				\freq, me.rotEuler[0].wrap(0, 2pi).linexp(0, 2pi, 400, 800),
				\amp, 1
			)
		}, {
			q.synths[me].set(
				\amp, 0
			)
		})
	}
}
)

// instantiate SETOServer
t = SETO_OSCServer('_ixya', setoClass: JITseto);
t.gui;
t.start;
t.stop;
```

In the example, a **SETO_OSCServer** is instantiated that holds all representations of tangible objects for the programmer. All objects tracked by the tracking system are represented as **JITseto** objects, a child of **SETObject** that evaluates a global class function whenever they are updated. That is the place where the actual audio controlling takes place. At first sight this example might look as complex as the non-SETO one, however it has several benefits compared to it: (a) While the simple approach is limited to an explicit number of objects of which the identification numbers have to be successors of each other, here an arbitrary number of objects might be used. (b) While the mapping of object-behavior to sound parameters is fixed once the system is set up, in the SETO example it is interchangeable at runtime.

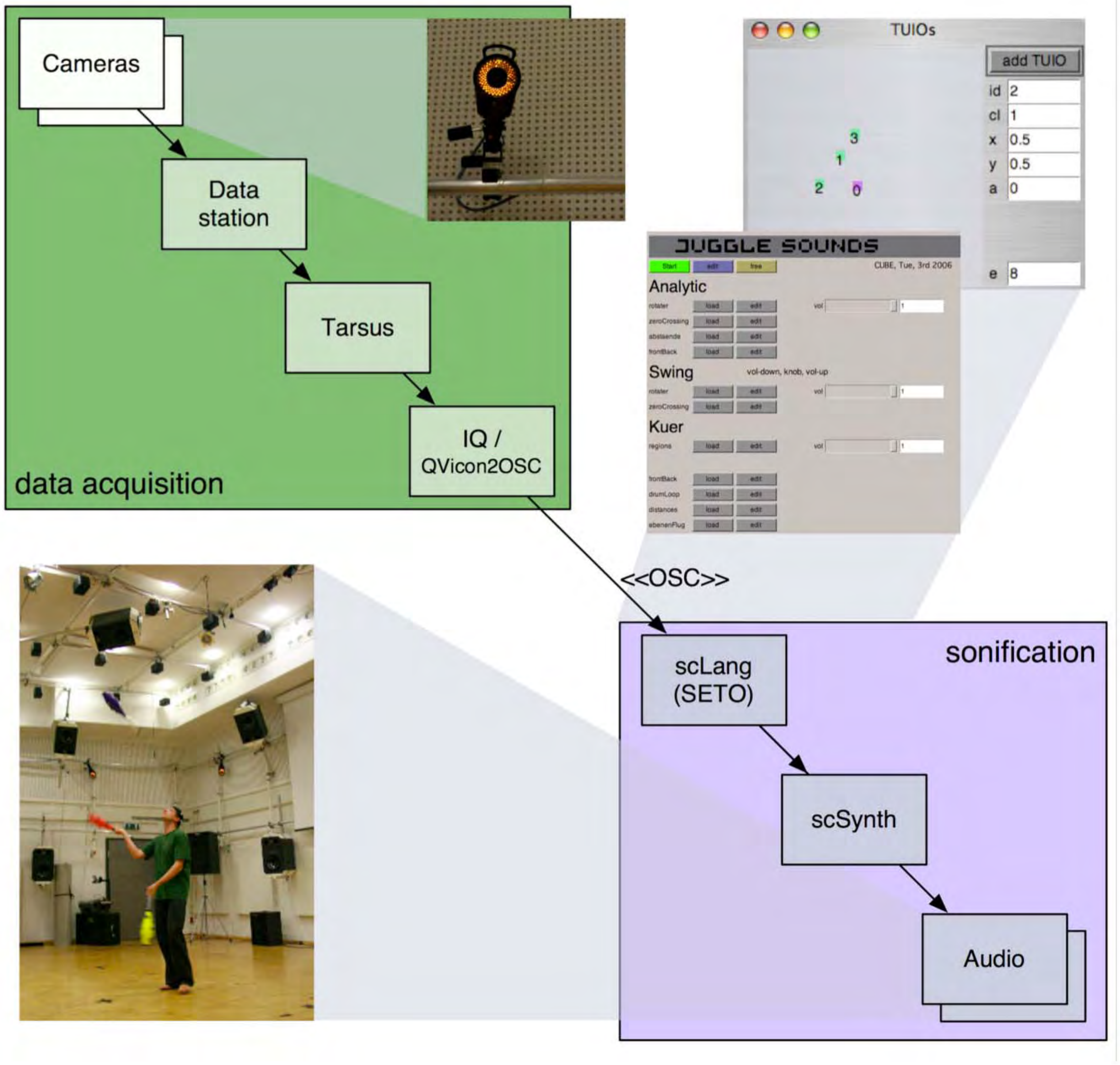

Figure 2: The JugglingSounds system.

To give an insight into the possibilities of SETO, we now describe the implementation of *JugglingSounds*, a system for real-time auditory monitoring of juggling patterns designed and implemented in 2006 at IEM, Graz, Austria [juggling]. For detailed information on its use and especially on the sonification process, please cf. to the Sonification chapter of this book.

JugglingSounds consists of two main parts, data acquisition via motion tracking and sonification via SETO as shown in Figure 2. For proper sonification results, the used tracking system has to support capturing of Objects in 6DOF at an update rate of at least 40Hz. In the particular setup, we used for this an optical motion capturing system developed by the Vicon Company. It consists of at least 6 cameras, a data station and a PC running the *ViconiQ* software as well as the server application (*Tarsus*), which is connected to the data station via Ethernet. This setup is able to compute the 6DOF position of rigid bodies (here three juggling clubs and the juggler's head) about 120 times per second. This pseudo-continuous multidimensional stream of tracking data is translated into OSC messages by *QVicon2OSC*, and sends to the sonification part, running on a separate computer. The non-TUIO-conforming OSC interface of QVicon2OSC required special **SETOServer** class. Its definition (**SETOTarsusServer**) may be used as a guideline for implementing new tracking interfaces for SETO.

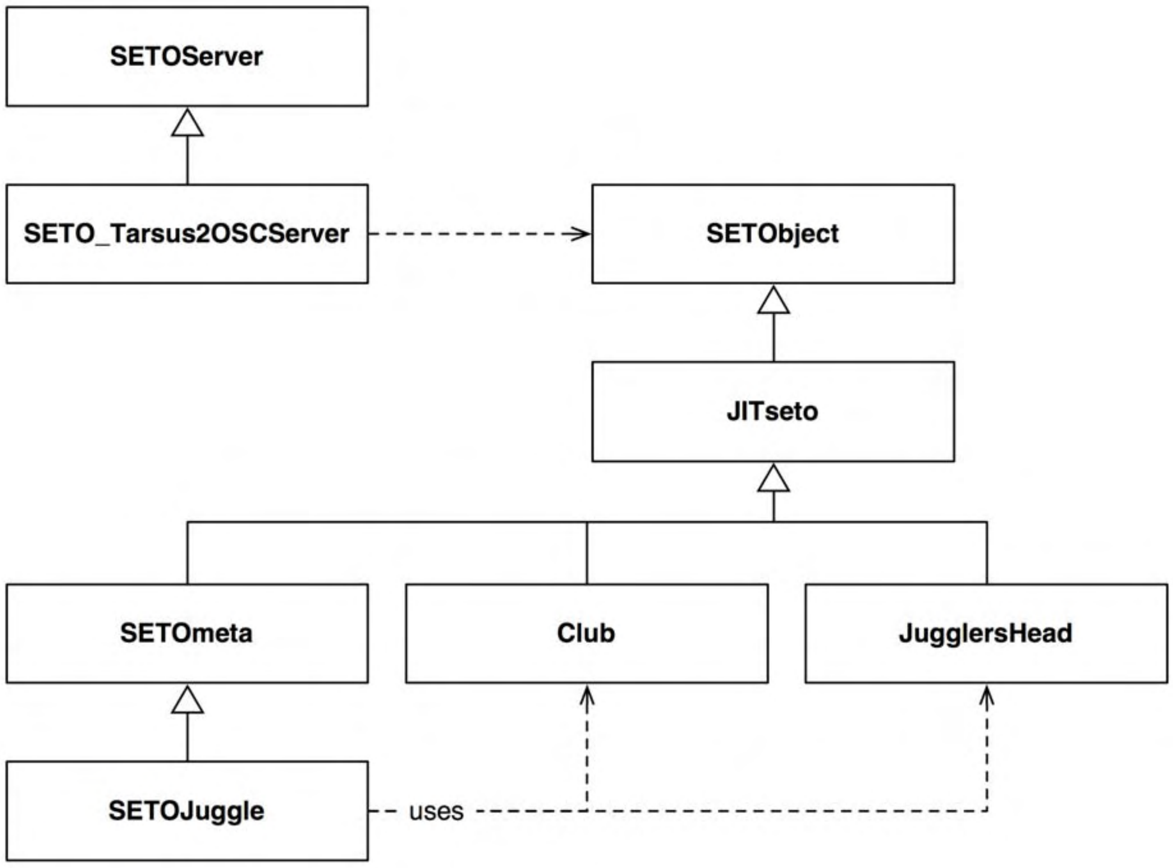

Figure 3: Dependency diagram of JugglingSounds classes

Internally, SETO uses homogeneous transformations, consisting of a translational and a rotational part put into a matrix. However, the objects 6DOF position may also be provided in a different representation form, e.g. in axis notation as it is done by our tracking system. As soon as a homogeneous representation is needed, SETO computes the transformation and provides it to the user.[i]

The complexity of JugglingSounds forced us to subclass from SETObject to represent juggling clubs and the juggler's head. Their class dependencies from standard SETO classes are shown in Figure 3. Here, **Club** represents one juggling club, **JugglersHead** is the juggler's head. Each club or head motion triggers the computation of sonification-relevant features, like the location of the club's symmetric axis's (**symAxis**), an indicator if this axis has crossed the z-plane in the last update (**zeroCrossing**) etc.

Subsequently, the action methods of **Club** and **JugglersHead** transfer all collected information to specific **NodeProxies**:

```
Club.action = {|me|
    me.flipAngleVel.isNaN.not.if({
        ~rotVel.set(me.id, me.flipAngleVel);
    }, {
        ~rotVel.set(me.id, 0);
    });
```

```
	~absX.set(me.id, me.pos[0]);
	~absY.set(me.id, me.pos[1]);
	~height.set(me.id, me.pos.last);

	~relX.set(me.id, me.posRelHead[0]);
	~relY.set(me.id, me.posRelHead[1]);

	~zeroCrossing.set(me.id, me.zeroCrossing.binaryValue);
	~catched.set(me.id, me.catched.binaryValue);
	~posRelGPointX.set(me.id, me.posRelGroundPoint[0]);
	~posRelGPointY.set(me.id, me.posRelGroundPoint[1]);
	~posRelGPointZ.set(me.id, me.posRelGroundPoint[2]);
	~posRelHeadX.set(me.id, me.posRelHead[0]);
	~posRelHeadY.set(me.id, me.posRelHead[1]);
	~posRelHeadZ.set(me.id, me.posRelHead[2]);
};

JugglersHead.action = {|me|
	~regionChanged.set(0, me.regionChanged.binaryValue);
	~region.set(0, me.region);
	me.regionChanged.if{
		q.regionChange(me.region);
	};
	~headAbsX.set(0, me.pos[0]);
	~headAbsY.set(0, me.pos[1]);
	~headHeight.set(0, me.pos[2]);

};
JugglingInteraction.headClubAction = {|distance, isValid, head, club|
	~dist.set(club.id, distance);
};
```

A basic setup of JugglingSounds can be found on this DVD.

To conclude, SETO can be used to set up complex environments for tangible computing. It is especially suitable for environments where interaction with the code is needed; here it provides easy access to higher-level functionality.

## REFERENCES

[react] Bencina, R., Kaltenbrunner, M., and Jordà, S. (2005). Improved topological fiducial tracking in the reactivision system. In *Proceedings of the IEEE International Workshop on Projector-Camera Systems (Procams 2005)*, San Diego, USA.

[juggling] Bovermann, T., Groten, J., de Campo, A., and Eckel, G. (2007). Juggling sounds. In *Proceedings of the 2nd International Workshop on Interactive Sonification*, York.

[i] Footnote: For more information on the mathematical background on rotational representations, please cf. e.g. to http://en.wikipedia.org/wiki/Rotation_representation_(mathematics) )