UnitInstrument: Easy Configurable Musical Instruments

Yutaro MARUYAMA
Kobe University, Japan
maruyama@stu.kobe-u.ac.jp

Yoshinari TAKEGAWA
Kobe University, Japan
take@eedept.kobe-u.ac.jp

Tsutomu TERADA
Kobe University, Japan
tsutomu@eedept.kobe-u.ac.jp

Masahiko TSUKAMOTO
Kobe University, Japan
tuka@kobe-u.ac.jp

ABSTRACT

Musical instruments have a long history, and many types of musical instruments have been created to attain ideal sound production. At the same time, various types of electronic musical instruments have been developed. Since the main purpose of conventional electronic instruments is to duplicate the shape of acoustic instruments with no change in their hardware configuration, the diapason and the performance style of each instrument is inflexible. Therefore, the goal of our study is to construct the *UnitInstrument* that consists of various types of musical units. A unit is constructed by simulating functional elements of conventional musical instruments, such as output timing of sound and pitch decision. Each unit has connectors for connecting other units to create various types of musical instruments. Additionally, we propose a language for easily and flexibly describing the settings of units. We evaluated the effectiveness of our proposed system by using it in actual performances.

Keywords

Musical instruments, Script language

1. INTRODUCTION

In the long history of musical instruments, many types of musical instruments such as wind, string, percussion, and keyboard have been developed. At the same time, various types of electronic musical instruments were developed such as digital pianos and electronic guitars. These erectronic instruments have many functions such as diapason change and tone change.

However, the main aim of most conventional electronic instruments is to duplicate the shape of acoustic instruments, their hardware configuration cannot be easily changed. This means that the performance style of each instrument is inflexible. For example, pianists cannot play music composed for the organ with dual manuals using a digital piano with a single manual, and guitarists cannot play music for a long scale guitar with 24 frets with a short scale guitar with 12 frets.

NIME2010, 15-18th June, 2010, Sydney, Australia

Therefore, the goal of our study is to construct the *UnitInstrument*, which consists of various types of musical units. We assume that most musical instruments can be categorized by functional properties. A unit is a functional element of a conventional musical instrument, such as output timing of sound and pitch decision. Figure 1 shows the concept of our *UnitInstrument* and examples of units and their combinations. The figure shows a *KeyboardUnit* with only 12 keys (seven white and five black) and a *FingerboardUnit* with only four frets. We can build various instrument configurations by combining multiple units, as shown at the bottom of this figure. The diapason and configuration of musical instruments are extended to combine the same types of units, and new musical instruments can be created by combining different types of units. For example, by connecting the *PickupUnit* of a guitar with a *KeyboardUnit*, we construct a new keyboard that can easily produce vibrato. Additionally, we propose a language for easily and flexibly define the settings of units.

The remainder of this paper is organized as follows. Section 2 explains related work. Section 3 describes the design of *UnitInstrument* and Section 4 presents the implementation of a prototype system. Finally, Section 5 describes conclusions and our planned future work.

2. RELATED WORK

There is a large amount of research whose main goal is the improvement of system functions by combining simple functional units. For example, users can control an object in a video game by combining LEGO block[1], and browse web sites by combining triangle board[2], and control programs with combined block[3]. However, the goal of these studies is not to construct flexible musical instruments.

There are systems for producing music composition by combining blocks that have marker[4][5]. The goal of these systems is to compose music, while the goal of our study is to construct musical instruments.

The concept of the *UnitKeyboard*[6], in our previous work, is similar to *UnitInstrument*. We can construct various types of configurations on a musical keyboard using the *UnitKeyboard*. On the other hand, the *Unitkeyboard* realizes configurable keyboards, while *UnitInstrument* creates new musical instruments because it combines various types of musical instruments. This means that *UnitInstrument* requires flexible configuration control; therefore we propose a new language to flexibly define configurations of the *UnitInstrument*.

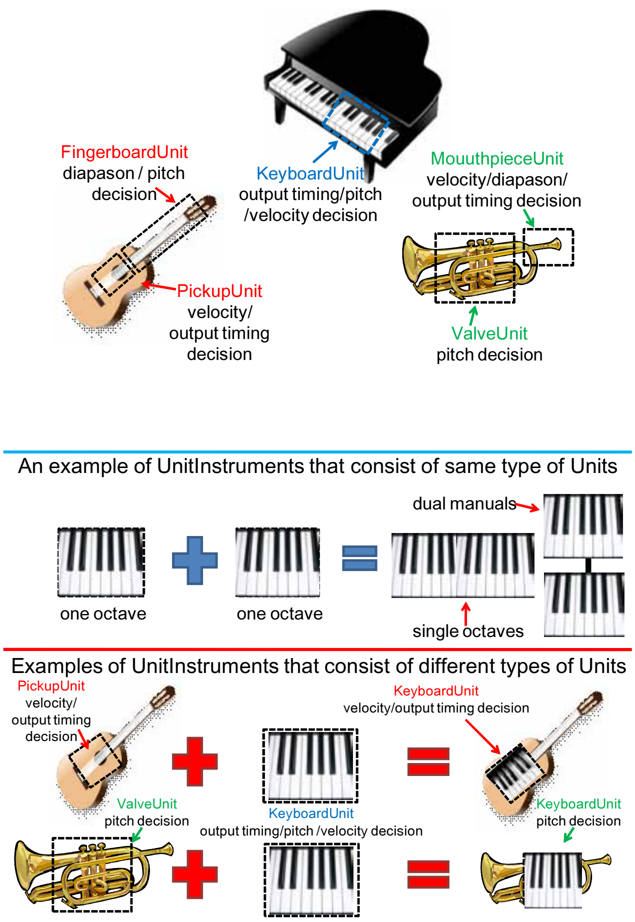

Figure 1: Concept of UnitInstruments

3. DESIGN

UnitInstrument is designed according to the following policies.

Extracting components from existing instruments

The *UnitInstrument* enhances the configuration of conventional instruments by connecting the same types of units as shown in Figure 1. For example, we can construct a two-octave keyboard by horizontally connecting two *KeyboardUnits*, or construct an organ with dual manuals by vertically connecting two *KeyboardUnits*.

In addition, the *UnitInstrument* can create new musical instruments by connecting different types of units. For example, by connecting the *MouthpieceUnit* of a wind instrument with the *KeyboardUnit*, we can construct a new keyboard that easily modulates the velocity and the diapason of keys being pressed. Additionally, by connecting a *FingerboardUnit* of a guitar to the *KeyboardUnit*, we can construct a new keyboard that can easily produce vibrato. In this way, we can construct new musical instruments that create new musical expressions, and this concept enhances the possibility of musical instruments. Note that the use of each unit is the same as conventional instruments. Therefore, users can apply the playing techniques and experience they acquired when learning the original instrument to using the *UnitInstrument*.

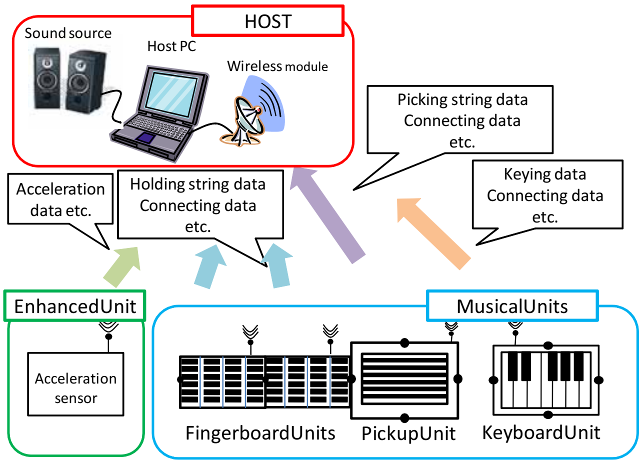

Figure 2: System structure

Flexible and dynamic reconfiguration

We can construct various types of musical instruments by connecting multiple units. This means that users need to configure various types of settings for each unit. To reduce the setting time, each unit should recognize its connection status, and the new settings suitable for the current situation should be automatically assigned to the unit. Additionally, there should be a script language to enable flexible configuration of units. The details of this language is explained in Section 3.2.

3.1 System structure

Figure 2 shows the system structure of the *UnitInstrument*. The system consists of units and a Host. Units consist of the following *MuscialUnits* and *EnhancedUnits*.

3.1.1 MusicalUnit

MusicalUnit is a generic term of instrument units, such as *KeyboardUnit*, *FingerboardUnit*, *PickupUnit*, and *MouthpieceUnit*. A *MusicalUnit* has at least one functional instrumental element: output timing of sound, pitch decision, or velocity decision.

A *MusicalUnit* has a wireless module to communicate with the Host and connectors to connect it to other units. It sends various operation data to the Host, such as keying, velocity, pitch, and connection. We explain the design of the three *MusicalUnits* below.

KeyboardUnit

Each key on a conventional keyboard has all functional elements. Additionally, the 12 keyboard keys are intuitively arranged according to a musical scale, and many pianists are familiar with normal-sized keys. Therefore, a *KeyboardUnit* has 12 keys (seven white and five black) that are normal size. Additionally, it is equipped with four connectors on the left, right, top, and bottom, for connecting to other units.

FingerboardUnit

A *FingerboardUnit* is a unit extracted from the fingerboard of a conventional guitar. There are four frets on a *FingerboardUnit*, because we generally play single sounds and basic chords with four consecutive frets. In addition, the positions of the frets in a *FingerboardUnit* are the same as a conventional guitar because players of *UnitInstruments*

Table 1: Member variables of Unit object

Name (Data type)	Function
basepitch (*pitch* C4)	Reference tone of Unit
mode (*mode* KEY_DEC)	mode
child (**Switch* sw)	Reference to the collection of switches
right (**unit* ri)	Reference to the unit connected of the right side
up (**unit* up)	Reference to the unit connected at the top
left (**unit* le)	Reference to the unit connected of the left side
down (**unit* dow)	Reference to the unit connected at bottom
type (*type* PICKUP)	Reference to unit type
id (*int* ID)	Reference to unit ID

Table 2: Member functions of Unit object

Name	Function
setTone (*tone* CLEAN_GT)	Setting of unit tone
setTuning (*tuning* NORMAL)	Setting of unit tuning

should be able to use the techniques of playing a conventional guitar.

PickupUnit

A *PickupUnit* is a unit extracted from the pickup of a conventional guitar. There are six strings on a *PickupUnit* because we generally play melodies and basic chords with six consecutive strings. In addition, the size of a *PickupUnit* is the same as a conventional guitar because of the same reason of *FingerboardUnit*.

3.1.2 *EnhancedUnit*

An *EnhancedUnit* is equipped with connectors, sensors or actuators, and a wireless module, to enhance the functionality of a *MusicalUnit*. For example, users control the tone of *MusicalUnit*s with their posture, which is calculated and detected from data of the acceleration sensor on the *EnhancedUnit*. The acceleration data is then sent to the Host. At the same time, users can control diapasons of a *MusicalUnit* neighboring an *EnhancedUnit* equipped with distance sensors. For example, the longer the distance between the *KeyboardUnit* and the *EnhancedUnit*, the higher the diapason of the *KeyboardUnit*.

3.1.3 Host

The Host manages the connection status, and controls the settings of all units. Additionally, it generates a MIDI messages based on the status of units and data sent from the units. The settings are managed by mapping between the physical inputs on the unit and the actual output of sound. This mapping is described in the script language we explain in the next section.

Table 3: Member variables of Switch object

Name (Data type)	Function
priority (*int* pri)	Setting of control priority
pitch (*pitch* pi)	Setting of pitch
parent (**unit* pa)	Reference to belonging Unit
tone (*tone* JAZZ_GT)	Setting of tone
targetUnit (**unit* tau)	Reference to the trigger unit
targetSwitch (**switch* tas)	Reference to the trigger switch
id (*int* ID)	Reference to the switch ID

Table 4: Data types

Data type	Function
int	integer
tone	tone
pitch	pitch
mode	mode
type	type
tuning	tuning
*unit	collection of the units
*switch	collection of the switches

Table 5: Operators

Type	Function
=	Assign
==	Comparison (equal)
!=	Comparison (not equal)
+	Addition
−	Subtraction
/	Division
+=	Add and Assign
−=	Substract and Assign
<	Less than

Table 6: Control statements

Name	Function
for(*Reset*;*Condition*;*Statements*) *Statements* end for	Allowing code to be repeatedly executed
foreach(*variable* in *Object*) *Statements* end foreach	Traversing items in a collection
do *Statements* end do	Allowing code to be repeatedly
if(*Condition*) *Statements* {else *Statements*} end if	Showing if-then sentence of condition

3.2 Script language

We assume that players will frequently reconfigure *UnitInstruments*, even while they are playing them. Therefore, each unit should be able to recognize its connection status, and the system should automatically and immediately assign the tone and the diapason to the unit. Additionally, we are able to flexibly assign functional elements to each unit with this *UnitInstrument* framework. As a result we can make various types of new instruments. However, users need to configure various types of settings for each unit. To reduce the setting time, we propose an object-oriented script language for programming *UnitInstruments*. This script language enables users to reduce the setting time

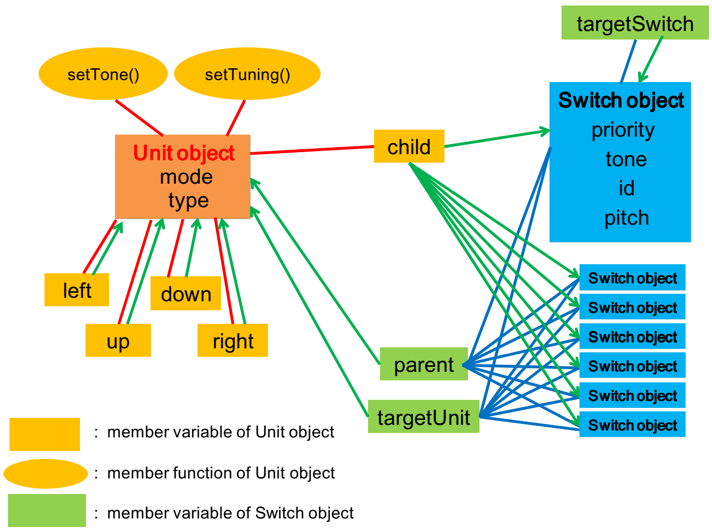

Figure 3: Relationship between Unit and Switch objects

```
unit f,  p;
switch s;
int i;
foreach(p in unit)
 if(p.type == PICK_UP)
  p.mode = SOUND_DEC;
  p.setTuning(NORMAL);
  p.setTone(STEEL_GT);
 end if
end foreach

foreach(f in unit)
 if(f.type == FINGER_PLATE)
  f.mode = KEY_DEC
  for(i=0; i<23; i++)
   f.child(i).priority = i;
   f.child(i).pitch = i+1;
   if(i>3)
    f.child(i).pitch += 1;
    f.child(i).priority -= 4;
   end if
 if(i>7)
    f.child(i).pitch += 1;
    f.child(i).priority -= 4;
   end if
   if(i>11)
    f.child(i).pitch += 1;
    f.child(i).priority -= 4;
   end if
   if(i>15)
    f.child(i).priority -= 4;
   end if
   if(i>19)
    f.child(i).pitch += 1;
    f.child(i).priority -= 4;
   end if
  end for
 end if
end foreach

f = unit(1).right;
f.basepitch = E2;
do
 f.mode = KEY_DEC;
 foreach(s in f.child)
s.targetUnit = 1;
s.tone = OVERDRIVE_GT;
  s.targetSwitch = s.id / 4;
 end foreach
 if(f.left != NULL)
  f.basepitch += 4;
  f = f.left;
  f.basepitch = f.right.basepitch - 4;
 else
  break;
 end if
end do

f = unit(1).up;
f.basepitch = E2;
do
 f.mode = KEY_DEC;
 foreach(s in f.child)
  s.targetUnit = 1;
  s.tone = JAZZ_GT;
  s.targetSwitch = s.id / 4;
 end foreach
if(f.left != NULL)
  f.basepitch += 4;
  f = f.left;
  f.basepitch = f.right.basepitch - 4;
 else
  break;
 end if
end do
```

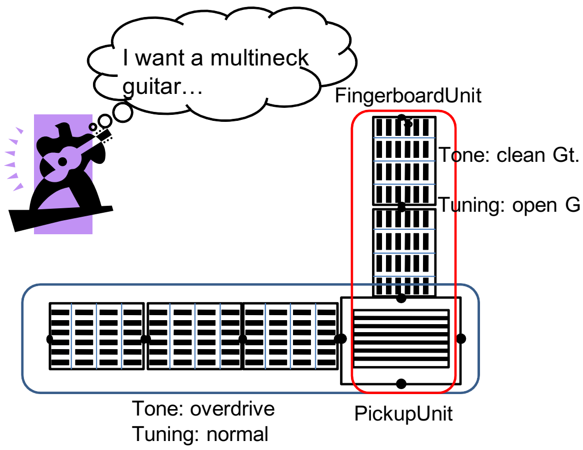

Figure 4: Example script of multineck guitar

```
unit b, f;
switch s, k;
int i;
foreach(b in unit)
 if(b.type == KEYBOARD)
  b.tone = PIANO;
  b.mode = SOUND_DEC;
  b.basepitch = C4;
  for(i=0; i<11; i++)
   b.child(i).pitch = i;
   b.child(i).priority = 0;
  end for
 end if
end foreach

foreach(f in unit)
 if(f.type == FINGER_PLATE)
  f.mode = KEY_DEC
  for(i=0; i<23; i++)
   f.child(i).priority = i;
   f.child(i).pitch = i+1;
 if(i>3)
    f.child(i).pitch += 1;
    f.child(i).priority -= 4;
   end if
 if(i>7)
    f.child(i).pitch += 1;
    f.child(i).priority -= 4;
   end if
   if(i>11)
    f.child(i).pitch += 1;
    f.child(i).priority -= 4;
   end if
 if(i>15)
    f.child(i).priority -= 4;
   end if
 if(i>19)
    f.child(i).pitch += 1;
    f.child(i).priority -= 4;
   end if
  end for
 end if
end foreach
f = unit(5).left;
f.basepitch = E2;

do
 f.mode = KEY_DEC;
 foreach(s in f.child)
  s.targetUnit = 5;
  s.tone = OVERDRIVE_GT;
  if(s.id < 4)
   s.targetSwitch = 0;
  else if(s.id < 8)
   s.targetSwitch = 2;
  else if(s.id < 12)
   s.targetSwitch = 4;
  else if(s.id < 16)
   s.targetSwitch = 5;
  else if(s.id < 20)
   s.targetSwitch = 7;
 else if(s.id < 24)
   s.targetSwitch = 9;
  end if
 end foreach

if(f.left != NULL)
  f.basepitch += 4;
  f = f.left;
  f.basepitch = f.right.basepitch - 4;
 else
  break;
 end if
end do
```

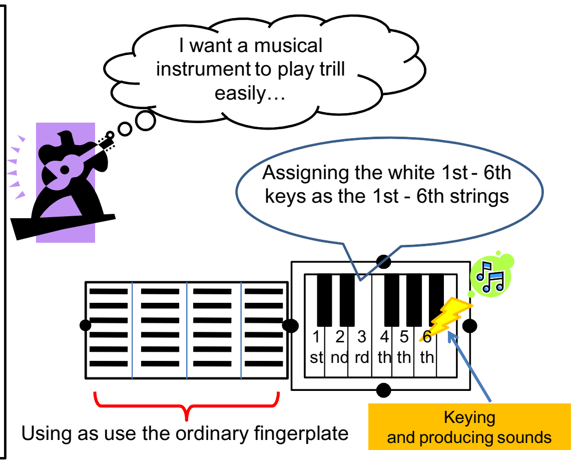

Figure 5: Example script of instrument that replaces pickup with keyboard

and use *UnitInstruments*.

3.2.1 *Script specification*

The script language consists of *Unit* objects, *Switch* objects, data types, operators, and control statements. *Unit* and *Switch* objects also have member variables and functions. The relationship between *Unit* and *Switch* objects is shown in Figure 3. The member variables and functions of *Unit* objects are listed in Tables 1 and 2, respectively, the member variables of *Switch* objects are listed in Table 3, data types in the script language are listed in Table 4, operators are listed in Table 5, and the control statements are listed in Table 6.

To define the characteristics of units, a user first chooses *Unit* objects or *Switch* objects and defines the type of unit. Additionally, the tone and diapason of each unit can be set using control sentences and the member variables *tone* and *pitch*. In addition, units that are connected via each connector are referred to *right* and *left*, and the user can define the characteristics of connected units recursively.

3.2.2 Examples of script

A multineck guitar

Figure 4 shows an example script that produces a multineck guitar structure. The first characteristics of the *PickupUnit* is defined from lines 4 to 10 and that of the *FingerboardUnit* is defined from lines 11 to 38. Lines 39 to 55 means that when a right neck (a group of *FingerboardUnits* connected to the left side of *PickupUnit*) is performed, the system outputs sound with the tone of an *OverDrive guitar*. On the other hand, lines 56 to 72 define when the other neck is performed. The system outputs sound by *opening G tuning*, and the tone of the soud is of a *clean guitar*. Additionally, *do* statement recursively sets multiple *FingerboardUnits*. Therefore, the diapason of the necks can be easily increased.

Instruments that replace pickup with keyboard

Figure 5 shows an example script that produces a keyboard structure with fingerboard, lines 4 to 14 in this script denote that the *KeyboardUnit* is configured like a conventional keyboard. However, lines 45 to 72 denote that a new musical instrument can be constructed that allocates one string to one white key in the *KeyboardUnit*, the pitch from the connected *FingerboardUnits*, and the production of sounds from the pressing of the keys. This can facilitate sweep-picking, which produces the sounds for sweeping all strings and the trill picking that alternately outputs the sounds.

4. IMPLEMENTATION

Figure 6 shows a prototype system including a *KeyboardUnit*, *FingerboardUnit*, *PickupUnit*, *EnhancedUnit* equipped with an acceleration sensor, *EnhancedUnit* equipped with a power supply module, and a Host.

4.1 Host

The Host has a sound module, a wireless module for communicating with units. For the prototype, we used a Panasonic CF-Y7 with Windows XP as the Host PC, Microsoft Visual C++ .NET 2005 for implementing the application to manage unit settings, Allow7 UM-100 as a wireless module, and Roland SC-8820 as a MIDI sound generator. In the current version of the system, the user configures the settings by adding scripts to the source code of the prototype.

4.2 MusicalUnit and EnhancedUnit

We used the UnitKeyboard [6], which we developed , as the *KeyboardUnit*. We constructed the *FingerboardUnit* and *PickupUnit* using a YAMAHA EZ-AG electric guitar. Since it is an electric guitar and has 12 frets in a fingerboard, we

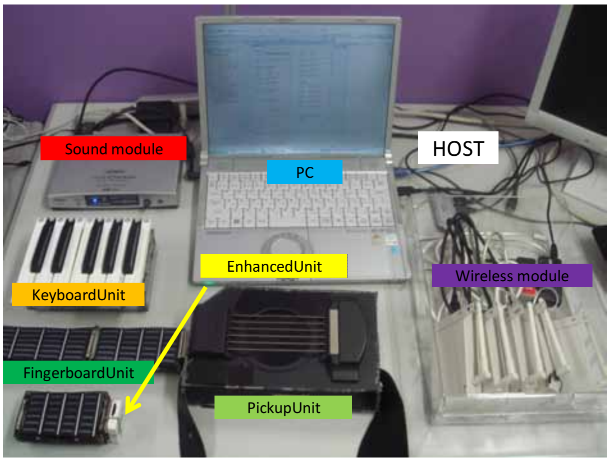

Figure 6: Snapshot of prototype system

Figure 7: Actual performance at Kobe Luminarie in 2008

cut it to make the four-fret *FingerboardUnits*. We used Microchip Technology PIC16F877A to control the *MusicalUnits*. The software on the *MusicalUnits* is implemented in C language on Microchip Technology MPLAB.

A *FingerboardUnit* has a connector on the right and left side, a wireless module, a microcontroller, and 24 switches. Additionally, the connectors have magnets for easily connecting/unconnecting another unit. The electrical power for the *FingerboardUnit* is supplied by the *EnhancedUnit* equipped with a power supply module. The *PickupUnit* has a connector on the right, left, top, and bottom, a wireless module, strings, six vibration sensors for detecting the vibration of the strings, and a microcontroller. We also installed an *EnhancedUnit* equipped with an acceleration sensor for detecting the player's motion.

4.3 Actual use

We used the prototype in several actual performances. We discuss how effective our proposed device was in these performances.

Kobe Luminarie Citizens Stage in 2008

We performed using the prototype at the Kobe Luminarie event on December 13 and 14, 2008. Kobe Luminarie is a large-scale event held in Kobe, Japan every December. It began in 1995 and commemorates the Great Hanshin earth-

Figure 8: Actual performance at Tsukamoto laboratory's 5th year anniversary party

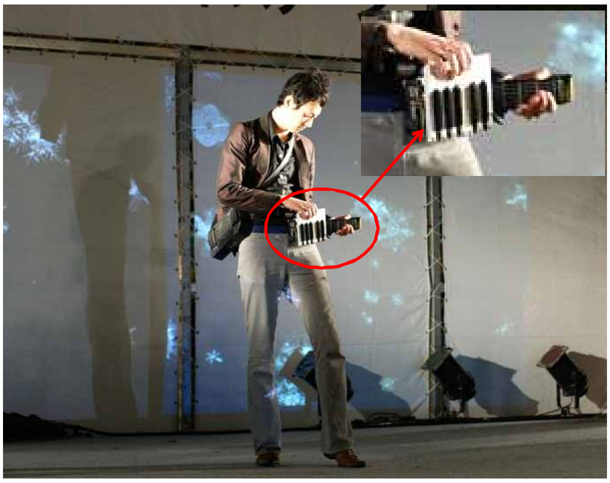

Figure 9: Actual performance at Kobe Luminarie in 2009

quake of that year. In this performance, we performed with two *KeyboardUnits*, a *FingerboardUnit*, a *PickupUnit*, and an *EnhancedUnit* equipped with high-intensity light emitting diodes (LEDs). There were two performers; one performer played the *KeyboardUnit*, the other played the *FingerboardUnit* and the *PickupUnit* while singing. The *FingerboardUnit* and the *EnhancedUnit* were connected. The performer playing the guitar played the *FingerboardUnit* with his left hand, and performed while waving his left hand as shown in Figure 7. We showed that the *UnitInstruments* were able to be taken apart during the performance.

Tsukamoto Laboratory's 5th year anniversary party

We played the prototype at the 5th year anniversary party of Tsukamoto Laboratory on October 30, 2009. We used two *KeyboardUnits*, a *FingerboardUnit* and a *PickupUnit*. There were three performers, and we connected/disconnected two *KeyboardUnits*, as shown in Figure 8. We programmed tone and diapason assigned to each *KeyboardUnit* to change based on their connection status. Two performers played the *KeyboardUnits*. The other played the *FingerboardUnit* and the *PickupUnit*, and he performed and connected/ disconnected these units to change the tone and diapason.

Kobe Luminarie citizens stage in 2009

We again performed with the prototype at the Kobe Luminarie event on December 12, 2009. We used a *KeyboardUnit*, a *FingerboardUnit*, and a *PickupUnit*, and connected/disconnected different types of *MusicalUnits*. The tone and diapason of each unit was changed according to its configuration. There were two performers. First, one performer played the *KeyboardUnit* and the other played the *FingerboardUnit* and the *PickupUnit*. In the middle of performance, the performer playing the *FingerboardUnit* and the *PickupUnit* connected the *FingerboardUnit* and the *KeyboardUnit* after taking the *KeyboardUnit* from the other performer, and played the new instrument, as shown in Figure 9. He easily played sweep-picking to use this instrument. We showed that we could connect different types of *MusicalUnits* to the *UnitInstrument*.

5. CONCLUSIONS

We proposed the *UnitInstrument*, which consists of various types of units extracted from conventional instruments from the viewpoint of functional elements. We can build various kinds of musical instruments by connecting multiple units.

We intend to design and implement other musical units such as percussion instruments and wind instruments. Additionaly, We will evaluate the hardware characteristics and usability of our system.

6. ACKNOWLEDGMENTS

A part of our study was supported by the Hayao Nakayama Science and Technology Cultural Foundation research promotion.

7. REFERENCES

[1] D. Anderson, J. Frankel, J. Marks, A. Agarwala, P. Beardsley, J. Hodgins, D. Leigh, K. Ryall, E. Sullivan, and J. Yedida: Tangible Interaction Graphical Interpretation: A New Approach to 3D Modeling, *Proceedings of Special Interest Group on Computer GRAPHics (SIGGRAPH2000)*, pp. 393-402, 2000.

[2] M. Gorbet, M. Orth, and H. Ishii: Triangles: Tangible Interface for Manipulation and Exploration of Digital Information Topography, *Proceedings of Computer-Human Interaction (CHI1998)*, pp. 49–56, 1998.

[3] H. Suzuki, and H. Kato: Interaction-level support for collaborative learning: AlgoBlock an open programming language, *Proceedings of Computer Support for Collaborative Learning (CSCL2002)* pp.349–355, 2002.

[4] N. D. Henry, H. Nakano, and J. Gibson: Block Jam,*Proceedings of Special Interest Group on Computer GRAPHics (SIGGRAPH2002)*, p. 67, 2002.

[5] M. Kaltenbrunner, S. Jorda, G. Geiger, and M. Alonso: The reacTable: A Collaborative Musical Instrument, *Proceedings of Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise (WETICE2006)*, pp. 406–411, 2006.

[6] Y. Takegawa, T. Terada, and T. Tsukamoto: UnitKeyboard: An Easily Configurable Compact Clavier, *Proceeding of International Conference on New Interfaces for Musical Expression (NIME2008)*, pp. 289–292, 2008.