INTERACTIVE SWARM ORCHESTRA
A GENERIC PROGRAMMING ENVIRONMENT FOR SWARM BASED COMPUTER MUSIC

Daniel Bisig *Martin Neukom* *John Flury*

Zurich University of the Arts
Institute for Computer Music and Sound Technology
Baslerstrasse 30
CH-8048 Zürich
Switzerland

ABSTRACT

The project Interactive Swarm Orchestra (ISO) employs swarm algorithms to control sound synthesis and sound positioning. As part of this endeavour, a set of Open Source software libraries has been developed that serves as flexible toolkit for musicians and scientists who intend to experiment with swarm based music. This paper introduces the libraries for sound synthesis (ISO Synth) and swarm simulation (ISO Flock). It puts particular emphasis on the associated programming concepts and data structures and provides source code examples.

1. INTRODUCTION

The ISO project tries to explore novel methods for generative composition and performance that derive from Artificial Life research into self-organised and autonomous systems. Such systems can give rise to adaptive, dynamic and complex patterns on different scales and offer interesting opportunities and challenges with regard to interactivity, feedback and control [1]. Swarm simulations explore principles of self-organisation and emergence in the appearance of group behaviour [2, 3]. These algorithms are particularly interesting for generative art and computer music for several reasons. They can exhibit a wide variety of different behaviours that range from simple reactive responses to highly complex and adaptive processes. They lend themselves to intuitive and natural forms of interaction. And they can be easily tailored to deal with any number and dimension of parameters. For a more profound discussion of the conceptual aspects of the ISO project please refer to [4, 5].

Several artists and musicians have employed swarm algorithms in their work (see [4] for an overview). All these works represent highly idiosyncratic examples that illustrate some of the aesthetic and interactive possibilities of swarm-based art. On the other hand, none of these examples explore the potentials of swarm algorithms for art on a more general and broad level. The authors believe that there is a need for a conceptual and technical foundation that supports and promotes research and artistic activities in swarm based music and art. The project ISO hopes to contribute to the establishment of such a foundation. The first tangible result of this project is a set of software tools for the creation of swarm based music.

2. ISO SOFTWARE ENVIRONMENT

Throughout the course of the ISO project, several C++ libraries (see table 1) have been developed in order to facilitate research and experimentation in swarm based computer music. These libraries are provided as Open Source software and can be downloaded from the project's web-site [6]. This web-site also hosts detailed documentation and tutorials. The libraries have been tested on MacOSX. A port to Linux is currently under way. Since the ISO libraries depend solely on cross platform third party libraries that are available for OSX, Linux and Windows, porting ISO to Windows should be straight forward.

ISO Base	data and exception types
ISO Math	vectors, matrices, quaternions
ISO Data	arrays, buffers, pools
ISO Event	event scheduling and execution
ISO Com	direct and network based communication
ISO XML	xml reader and writer
ISO Serialize	serialisation
ISO Geom	splines, meshes, grids
ISO Font	font to spline conversion
ISO Space	space representations, distance algorithms
ISO GL	OpenGL wrappers
ISO MIDI	MIDI handling
ISO Tracker	camera based tracking
ISO Synth	sound synthesis and processing
ISO Flock	swarm simulation

Table 1. ISO Libraries

At the onset of the ISO project in 2006, we specified a list of requirements that needed to be met by a prospective implementation of a swarm simulation and sound synthesis software environment. All software elements should be based on the same programming concepts and standardised design. Their functionality should be generic, flexible and easily expandable. They should provide a simple and well documented API and require a gradual learning curve when moving from high to low level implementations. The libraries should reflect a clear separation of their different functionality and data management requirements. Finally, the libraries should support the implementation of applications that are either monolithic (i.e. combine all functionality in a single application) or distributed (i.e. functionality is divided among several applications that may run on different computers). We decided that the development of entirely new software libraries constitutes the best approach to meet these requirements

This paper will mainly discuss the rationale and functionality of the two main ISO libraries: ISO Synth and

ISO Flock. The smaller ISO libraries will only be mentioned in the context of these two main libraries.

3. ISO SYNTH

ISO Synth is a library for real time sound synthesis, signal processing, and audio spatialisation. It follows the famous Music N paradigm [7]. Accordingly, audio handling is distributed among discrete units that exchange audio and control data via their interconnected ports. There are no limitations with regard to the topology of these interconnections (e.g. cyclic audio processing graphs via recurrent connections are valid). ISO Synth doesn't distinguish between audio-rate and control-rate but rather allows the specification of different sampling rates for individual units and ports. Scores and MIDI-based control are realised via the ISO event system. The state of an ISO Synth patch can be stored and recreated at any time via XML based serialisation functionality. Finally, ISO Synth relies on the ISO Com system to exchange data directly or via UDP with other ISO applications.

ISO Synth currently supports a variety of standard signal processing and sound synthesis techniques [4] as well as audio spatialisation via two and three-dimensional ambisonic projection [8].

3.1. Implementation

Audio data flow in ISO Synth proceeds according to the pull model. It interfaces with audio hardware and other audio applications via the virtual ports of the Jack Audio Connection Kit [9]. Audio and control data are encapsulated in buffer objects and can consist of an arbitrary number of channels and a power of two number of frames. Buffer objects are exchanged among audio units via interconnected ports. The connections are handled by instances of the link class. These links take care of changes in sampling rate, channel count and frame count that might be necessary between connected ports. Units can possess five different types of ports that all derive from a basic port class. Input, output, control and internal ports deal with continuous audio data. Control and switch ports can be manually set to particular values or receive event based signals. A unit can possess an arbitrary number of control and switch ports but only zero or one single input, output and internal port (see Figure 1). Input and output ports form an unit's external entry and exit points for audio data. Internal ports represent a special form of output port that allows units to encapsulate internal units (see Figure 1 right side). Control ports modify parameters of a unit (such as frequency). Control ports can be continuously updated by connecting them to other units' output ports or they can be set either manually or via events. Switch ports serve to reconfigure a unit in a more fundamental way than control ports do (for example to enable or disable looping in a sample playback unit). Switch ports cannot be connected via links but are set either manually or via events.

Custom units are implemented by deriving from one of the unit's base classes and overwriting its process function. ISO Synth assures that all unit ports have been updated prior to the execution of the user's function code and therefore grant access to properly refreshed audio and control data.

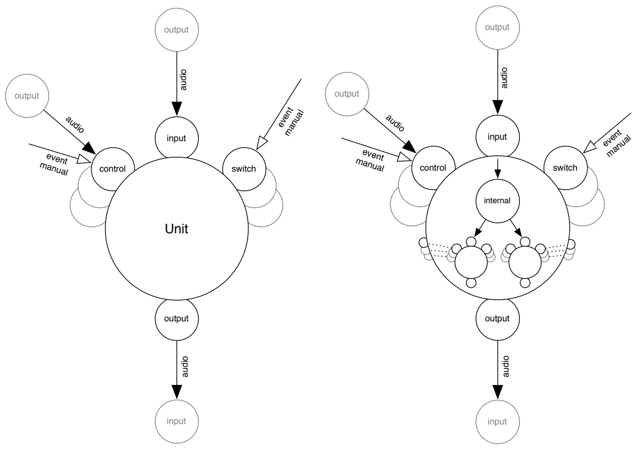

Figure 1. Ports in ISO Synth. Large central circles represent units. Small peripheral circles represent ports. Dimmed peripheral circles indicate an arbitrary number of ports. Filled arrows indicate ports connected via links. Outlined arrows indicate manual or event based port modifications. The left image depicts a standard unit. The right image shows a complex unit that encapsulates internal units. The dashed lines indicate that control and switch ports of internal units may mimic ports of the encompassing unit.

ISO Synth provides a patch class that can be subclassed to encapsulate the creation and configuration of commonly used combinations of units. Also, this class simplifies communication with other ISO programs by providing the virtual function "consumeMessage" that is called whenever a message is received.

ISO Synth provides hash maps for associating units, ports and links with unique names. These names can be user specified (units) or are automatically generated (ports and links). The synth class provides functions for retrieving instances of these classes via their names. This approach improves readability of code at the cost of postponing feedback about coding errors from compile to run time. Most ISO libraries make heavy use of exception handling and try their best to generate meaningful messages whenever a runtime error occurs.

3.2. Examples

```
// create units for simple FM synthesis
WaveTableOscil* carrier =
     new WaveTableOscil ("sinewave");
WaveTableOscil* mod =
     new WaveTableOscil("sawtoothwave");
OutputUnit* dac =
     new JackOutputUnit(2, "Built-in Audio");
// manually set port values
mod->set("frequency", 600.0);
mod->set("offset", 900.0);
mod->set("amplitude", 200.0);
// connect output port to control port
mod->connect(carrier, "frequency");
// connect carrier's output port to DAC's input port
carrier->connect(dac);
// set control port via events
Synth::get().eventManager().createEvent(2000.0,
     mod->controlPort("amplitude"), 50.0, 1000.0);
Synth::get().eventManager().createEvent(4000.0,
     mod->controlPort("amplitude"), 400.0, 1000.0);
```

Example 1. Unit Creation and Manual, Event Based and Link Based Control Port Changes.

```
class LimiterUnit : public ProcessUnit {

     ControlPort* mThreshold1;
     ControlPort* mThreshold2;

     Buffer* mThresh1Buffer;
     Buffer* mThresh2Buffer;

     LimiterUnit():ProcessUnit() {
          mThreshold1 = createControlPort("thresh1");
          mThreshold2 = createControlPort("thresh2");
          mThresh1Buffer = mThreshold1->buffer();
          mThresh2Buffer = mThreshold2->buffer();
          //set thresholds to default values
          mThreshold1->set(1.0f);
          mThreshold2->set(-1.0f);
      }

      void process(Buffer* pBuffer) {
          Unit::process(pBuffer);
          if (!mActive) return;

          pBuffer->truncate(
                 *mThresh1Buffer,*mThresh2Buffer);
     }
}
```

Example 2. Custom Unit Class Derived from ProcessUnit.

```
class AdditiveSynthesisPatch : public Patch {
   unsigned int mNumOscs;  //num of partials
   sample mRootFreq;  //fundamental frequency
   WaveTableOscil** mOscils; //bank of oscilators
   ControlPort** mFreqPorts; //frequency control ports
   ControlPort** mAmpPorts; //amplitude control ports

   AdditiveSynthesisPatch():Patch()
      , mNumOscs(10), mRootFreq(220)
   {}

   void construct() {
      mOscils = new WaveTableOscil*[mNumOscs];
      mFreqPorts = new ControlPort*[mNumOscs];
      mAmpPorts = new ControlPort*[mNumOscs];

      for (unsigned int i = 0; i < mNumOscs; i++) {
         //init oscilators
         mOscils[i] = new WaveTableOscil("sinewave");
         //connect each oscilator to audio output
         mOscils[i]->connect(output);
         //obtain control ports
         mFreqPorts[i] = mOscils[i]->
            controlPort("frequency");
         mAmpPorts[i] = mOscils[i]->
            controlPort("amplitude");
      }
   }

   void consumeMessage(const com::Message& pMessage) {
      unsigned int partialIndex = 0;
      sample harmonicFreq = mRootFreq;
      sample* values;
      unsigned int valueCount;

      while (partitialIndex < mNumOscs) {
         pMessage.nextValues(valueCount, &values);
         harmonicFreq += mRootFreq;
         mFreqPorts[partialIndex]->set(
            values[1]*harmonicFreq);
         mAmpPorts[partialIndex]->set(values[0]);
         partialIndex++;
      }
   }
```

Example 3. Implementation of "construct" and "consumeMessage" Function in Custom Patch Class.

4. ISO FLOCK

ISO Flock is a library that supports the creation of multi-agent simulations. It emphasises simulations of large numbers of point-like agents that possess simple behaviours. Apart from this restriction, ISO Flock constitutes a highly generic toolkit that supports a wide diversity of swarm simulations. In particular, ISO Flock doesn't impose any restrictions on the number, type and dimensionality of parameters that agents can possess. ISO Flock provides functions to calculate spatial relationships among parameters as well as between parameters and other spatial structures such as splines or vector-fields. Agent behaviours establish functional relationships among these parameters. ISO Flock closely resembles ISO Synth in its integration with ISO Event and ISO Com. Agent parameters can be modified in a score like fashion via events. ISO Flock can exchange data with other ISO applications. Two and three dimensional agent parameters and spaces can be visualised with ISO GL. This library provides a set of OpenGL wrapper classes and provides simple functionality for mouse and keyboard based navigation through the visualisation space.

ISO Flock currently provides a basic collection of behaviours and spatial structures. This includes behaviours for dealing with space boundaries, for synchronising neighbouring parameters and for responding to the presence of spatial objects. Spatial structures exist that handle distance calculations between point-objects (Quadtree, Octree or higher dimensional N-Tree), bounding boxes (R-Tree) and vector-fields (region averaging or highest value search).

4.1. Implementation

The implementation of ISO Flock defines a set of classes from which simulations can be built either by configuring these classes or by creating derived classes. Similar to ISO Synth, ISO Flock relies heavily on hash maps to uniquely identify class instances via names for retrieval and manipulation. This time, this feature does not only lead to improved code readability but provides the basic means for class introspection and is an important prerequisite for the generic implementation of the agent system. The simulation class manages all agents and updates their behaviours, parameters and associated spatial structures at regular intervals. The swarm class is a convenience class that simplifies the creation, deletion and management of groups of agents that possess the same properties. This class provides a similar interface as the agent class but its functions affect all agents contained within a swarm. The agent class itself provides very little functionality and mainly serves as a labelled container for parameters and behaviours. An agent's state is represented by its parameter values. Parameters are composed of two vectors of arbitrary dimensions that handle the parameter's current and buffered value. Furthermore, parameters store spatial relationships with neighbouring parameters (Euclidian distance and direction) within so called neighbour groups. Each neighbour group is associated with a particular parameter space. During every simulation step, the parameter spaces themselves calculate these relationships and update the neighbour groups accordingly. Agent behaviours define functional relationships among parameters. Behaviours distinguish between input parameters, internal parameters and output parameters. Internal parameters are specific for a particular behaviour and are automatically created when the behaviour is instantiated for the first time. Whenever a behaviour is executed, it reads from its input and internal parameters as well as associated neighbour groups and writes into its output parameters. Custom behaviours can be developed by subclassing the abstract base behaviour class and implementing its "act" function. This function is called once during a simulation step for each instance of the behaviour and is supposed to provide the necessary functionality to read and write an agent's parameters.

4.2. Examples

```
Swarm s("demo swarm");

// create and set parameters
s.addParameter("pos", 3); //position
s.addParameter("vel", 3); //velocity
s.addParameter("force", 3); //force
s.addParameter("acc", 3); //acceleration
s.addParameter("mass", 1); //mass
s.set("mass",1.0); //set parameter value

// create behaviours and set their internal parameters
s.addBehavior("reset",ResetBehavior("", "force"));
s.addBehavior("rand",RandomizeBehavior("", "force"));
s.set("rand_range", 0.1);
s.addBehavior( "damp",DampingBehavior("vel",
   "force"));
s.set("damp_prefVelocity", 0.20);
s.set("damp_amount", 0.5);
s.addBehavior("acc", AccelerationBehavior(
   "mass vel force", "acc" ));
s.set("acc_maxAngularAcceleration", 0.1);
s.addBehavior("integration", EulerIntegration(
   "pos vel acc", "pos vel"));
s.set("integration_timestep", 0.1);

// create swarm agents
s.addAgents(200);
```

Example 1. Creation of a Swarm

```
// create parameter space
space::SpaceManager::get().addSpace(
   new space::PointSpace("pos_space", 3));

// create neighbour group for parameter
s.assignNeighbors( "pos", "pos_space", true,
   new space::NeighborGroupAlg( 1.7, 4, true ) );

// create BOIDS behaviors
s.addBehavior( "cohesion", CohesionBehavior(
   "pos:pos_space", "force"));
s.set("cohesion_minDist", 0.0);
s.set("cohesion_maxDist", 1.7);
s.set("cohesion_amount", 0.1);

s.addBehavior("alignment",AlignmentBehavior(
   "pos:pos_space vel", "force") );
s.set("alignment_minDist", 0.0);
s.set("alignment_maxDist", 1.7);
s.set("alignment_amount", 0.5);

s.addBehavior("evasion",EvasionBehavior(
   "pos:pos_space","force"));
s.set("evasion_maxDist", 1.0);
s.set("evasion_amount", 0.5);
```

Example 2. Extension of Example 1 That Deals With Neighborhood Calculations.

```
class AverageBehavior : public Behavior {
   Parameter* mParIn1; //input parameter 1
   Parameter* mParIn2; //input parameter 2
   Parameter* mParOut; //output parameter

   //...

   void act(){
      math::Vector<real>& valIn1 = mParIn1->values();
      math::Vector<real>& valIn2 = mParIn2->values();
      math::Vector<real>& valOut =
         mParOut->backupValues();

      valOut = (valIn1 + valIn2) / 2.0;
   }
}
```

Example 3. Creation of a Custom Behaviour

5. CONCLUSION AND OUTLOOK

The ISO software libraries have reached a state of functionality and stability that allows the project to transition from its prior stage of pure software development into a phase that balances engineering and musical experimentation more evenly. On the musical application side, we are currently collaborating with the Tanz Akademie Zürich (http://www.tanzakademie.ch) to realise performances that employ ISO software for linking dance movements to acoustical feedback. We are also currently constructing a dodecahedral scaffold for 3D sound-projection, which will serve as a flexible and mobile space for ISO-based sound installations. Concerning further software development, we plan to quickly reach the following goals: porting of ISO libraries to Linux and Windows, implementing an OSC layer within the ISO COM library, transfer of camera tracking functionality into a dedicated library (instead of a fixed application as is currently the case). In the long term, we would like to develop a visualisation library that is highly customisable and can be easily integrated with ISO Flock and ISO Synth. And last but not least, we are looking forward to connect the ISO project to research in gestural musical interfaces in order to explore and possibly develop hardware devices that support more adequate and intuitive forms of interaction with swarm based environments than camera tracking solutions are able to.

6. REFERENCES

[1] Sommerer, C. and Mignonneau, L. "Modeling Complex Systems for Interactive Art", *Applied Complexity - From Neural Nets to Managed Landscapes*, Institute for Crop & Food Research, Christchurch, New Zealand, 2000.

[2] Martinoli, A. "Swarm intelligence: emergence and self-organization in natural and artificial systems." *Course notes*, EPFL, 2005.

[3] Eberhart, R., Shi, Y. and Kennedy, J. *Swarm Intelligence*, Morgan Kaufmann, 2001.

[4] Bisig, D., Neukom, M. and Flury, J. "Interactive Swarm Orchestra", *Proceedings of the Generative Art Conference*, Milano, Italy, 2007.

[5] Bisig, D., Neukom, M. and Flury, J. "Interactive Swarm Orchestra, an Artificial Life Approach to Computer Music", *International Computer Music Conference*, Belfast, Ireland, 2008.

[6] ISO project website: http://www.i-s-o.ch

[7] Dodge, C. and Jerse, T. A. *Computer Music*, Schirmer Books, New York, USA, 1985.

[8] Malham, D. G. and Anthony, M., "3-D Sound Spatialization using Ambisonic Techniques", *Computer Music Journal* 19(4), 1995.

[9] Jack, http://jackaudio.org/