Csound 6, Old Code Renewed

LAC2013

John ffitch, Victor Lazzarini and Steven Yi

National University of Ireland, Maynooth

LAC Graz, May 2013

Introduction

In March 2012, a decision was taken to move the development of Csound from version 5 to a new major version, 6.

Moving to a new version allowed developers to rethink key aspects of the system, without the requirement of keeping ABI or API compatibility with earlier iterations.

The only restriction, which is a fundamental one for Csound, is to provide backwards language compatibility, ensuring that music composed with the software will continue to be preserved.

Introduction

In March 2012, a decision was taken to move the development of Csound from version 5 to a new major version, 6.

Moving to a new version allowed developers to rethink key aspects of the system, without the requirement of keeping ABI or API compatibility with earlier iterations.

The only restriction, which is a fundamental one for Csound, is to provide backwards language compatibility, ensuring that music composed with the software will continue to be preserved.

Introduction

In March 2012, a decision was taken to move the development of Csound from version 5 to a new major version, 6.

Moving to a new version allowed developers to rethink key aspects of the system, without the requirement of keeping ABI or API compatibility with earlier iterations.

The only restriction, which is a fundamental one for Csound, is to provide backwards language compatibility, ensuring that music composed with the software will continue to be preserved.

A Short History of Csound

Csound can be traced back to Barry Vercoe's MUSIC 360 package for computer music (1968), which was itself a variant of Max Mathews' and Joan Miller's MUSIC IV (1964).

Following the introduction of the PDP-11 minicomputer, a modified version of the software appeared as MUSIC 11.

Later, with the availability of C (and UNIX), this program was re-written in that language as Csound, allowing a simpler cycle of development and portability, in comparison to its predecessor. In the early 2000s, the final releases of version 4 attempted to retrofit an application programming interface (API), so that the system could be used as a library.

Background

A Short History of Csound

Csound can be traced back to Barry Vercoe's MUSIC 360 package for computer music (1968), which was itself a variant of Max Mathews' and Joan Miller's MUSIC IV (1964).

Following the introduction of the PDP-11 minicomputer, a modified version of the software appeared as MUSIC 11.

Later, with the availability of C (and UNIX), this program was re-written in that language as Csound, allowing a simpler cycle of development and portability, in comparison to its predecessor. In the early 2000s, the final releases of version 4 attempted to retrofit an application programming interface (API), so that the system could be used as a library.

ffitch, Lazzarini and Yi (NUIM) Csound 6 LAC2013 3 / 23

A Short History of Csound

Csound can be traced back to Barry Vercoe's MUSIC 360 package for computer music (1968), which was itself a variant of Max Mathews' and Joan Miller's MUSIC IV (1964).

Following the introduction of the PDP-11 minicomputer, a modified version of the software appeared as MUSIC 11.

Later, with the availability of C (and UNIX), this program was re-written in that language as Csound, allowing a simpler cycle of development and portability, in comparison to its predecessor. In the early 2000s, the final releases of version 4 attempted to retrofit an application programming interface (API), so that the system could be used as a library.

Csound 5

The need for the further development of the Csound API, as well as other innovations, prompted a code freeze and a complete overhaul of the system into version 5, released as a library with a few basic frontends, in 2006.

Much of this development included updating 1970s programming practices by applying more modern standards. One of the major aims was to make the code reentrant, so that its use as a library could be made more robust.

Csound 5

The need for the further development of the Csound API, as well as other innovations, prompted a code freeze and a complete overhaul of the system into version 5, released as a library with a few basic frontends, in 2006.

Much of this development included updating 1970s programming practices by applying more modern standards. One of the major aims was to make the code reentrant, so that its use as a library could be made more robust.

Csound operation in a nutshell

As a MUSIC-N language, Csound incorporates a compiler for instruments. During performance, these can be activated (instantiated) by various means, the traditional one being the standard numeric score.

The steps involved in the compiler can be divided into two: parsing, and compilation proper.

At instantiation, an init-pass loop is performed, executing all the once-off operations for that instance. This is then inserted in a list of active instruments, and its performance code is executed sequentially, processing vectors (audio signals), scalars (control signals) or frames of spectral data.

Csound operation in a nutshell

As a MUSIC-N language, Csound incorporates a compiler for instruments. During performance, these can be activated (instantiated) by various means, the traditional one being the standard numeric score.

The steps involved in the compiler can be divided into two: parsing, and compilation proper.

At instantiation, an init-pass loop is performed, executing all the once-off operations for that instance. This is then inserted in a list of active instruments, and its performance code is executed sequentially, processing vectors (audio signals), scalars (control signals) or frames of spectral data.

Background

Csound operation in a nutshell

As a MUSIC-N language, Csound incorporates a compiler for instruments. During performance, these can be activated (instantiated) by various means, the traditional one being the standard numeric score.

The steps involved in the compiler can be divided into two: parsing, and compilation proper.

At instantiation, an init-pass loop is performed, executing all the once-off operations for that instance. This is then inserted in a list of active instruments, and its performance code is executed sequentially, processing vectors (audio signals), scalars (control signals) or frames of spectral data.

ffitch, Lazzarini and Yi (NUIM) Csound 6 LAC2013 5 / 23

Motivation

Following the 2011 Csound Conference in Hannover, there were a number of user requests that would be more easily achievable with a version bump:

- the capacity of new orchestra code, ie. instruments and user-defined opcodes (UDOs), to be added to a running instance of the engine
- additions to the orchestra language, for instance, generic arrays
- rationalisation of the API to allow further features in frontends
- loadable binary formats, API construction of instruments
- further development of parallelism
- improved facilities for live coding

Csound6

Motivation

Following the 2011 Csound Conference in Hannover, there were a number of user requests that would be more easily achievable with a version bump:

- the capacity of new orchestra code, ie. instruments and user-defined opcodes (UDOs), to be added to a running instance of the engine
- additions to the orchestra language, for instance, generic arrays
- rationalisation of the API to allow further features in frontends
- loadable binary formats, API construction of instruments
- further development of parallelism
- improved facilities for live coding

ffitch, Lazzarini and Yi (NUIM) Csound 6 LAC2013 6 / 23

Motivation

Following the 2011 Csound Conference in Hannover, there were a number of user requests that would be more easily achievable with a version bump:

- the capacity of new orchestra code, ie. instruments and user-defined opcodes (UDOs), to be added to a running instance of the engine
- additions to the orchestra language, for instance, generic arrays
- rationalisation of the API to allow further features in frontends
- loadable binary formats, API construction of instruments
- further development of parallelism
- improved facilities for live coding

Csound6

Motivation

Following the 2011 Csound Conference in Hannover, there were a number of user requests that would be more easily achievable with a version bump:

- the capacity of new orchestra code, ie. instruments and user-defined opcodes (UDOs), to be added to a running instance of the engine
- additions to the orchestra language, for instance, generic arrays
- rationalisation of the API to allow further features in frontends
- loadable binary formats, API construction of instruments
- further development of parallelism
- improved facilities for live coding

ffitch, Lazzarini and Yi (NUIM) Csound 6 LAC2013 6 / 23

Motivation

Following the 2011 Csound Conference in Hannover, there were a number of user requests that would be more easily achievable with a version bump:

- the capacity of new orchestra code, ie. instruments and user-defined opcodes (UDOs), to be added to a running instance of the engine
- additions to the orchestra language, for instance, generic arrays
- rationalisation of the API to allow further features in frontends
- loadable binary formats, API construction of instruments
- further development of parallelism
- improved facilities for live coding

Csound6

Motivation

Following the 2011 Csound Conference in Hannover, there were a number of user requests that would be more easily achievable with a version bump:

- the capacity of new orchestra code, ie. instruments and user-defined opcodes (UDOs), to be added to a running instance of the engine
- additions to the orchestra language, for instance, generic arrays
- rationalisation of the API to allow further features in frontends
- loadable binary formats, API construction of instruments
- further development of parallelism
- improved facilities for live coding

ffitch, Lazzarini and Yi (NUIM) Csound 6 LAC2013 6 / 23

Motivation

Following the 2011 Csound Conference in Hannover, there were a number of user requests that would be more easily achievable with a version bump:

- the capacity of new orchestra code, ie. instruments and user-defined opcodes (UDOs), to be added to a running instance of the engine
- additions to the orchestra language, for instance, generic arrays
- rationalisation of the API to allow further features in frontends
- loadable binary formats, API construction of instruments
- further development of parallelism
- improved facilities for live coding

Developments to date: Build system and Tests

In Csound 6, the official build system is now the CMake-based build. Moving to CMake introduced some hurdles and changes in workflow, but it also brought with it generation of build system files, such as Makefiles, XCode projects, and Eclipse projects.

Unit and functional tests have been added to Csound 6's codebase. CTest is the test running utility used to execute the individual C-code tests. In addition, CUnit is employed to create the individual tests and test-suites within the test code files. Beyond C-code testing, the suite of CSD's used for application/integration testing continues to grow, and a new set of Python tests has also been added for testing API usage from a host language.

Developments to date: Build system and Tests

In Csound 6, the official build system is now the CMake-based build. Moving to CMake introduced some hurdles and changes in workflow, but it also brought with it generation of build system files, such as Makefiles, XCode projects, and Eclipse projects.

Unit and functional tests have been added to Csound 6's codebase. CTest is the test running utility used to execute the individual C-code tests. In addition, CUnit is employed to create the individual tests and test-suites within the test code files. Beyond C-code testing, the suite of CSD's used for application/integration testing continues to grow, and a new set of Python tests has also been added for testing API usage from a host language.

Developments to date: Code Reorganisation

The Csound code base is passing through a significant reorganisation:

- obsolete code, such as the old parser, has been removed.
- some opcodes have been completely rewritten
- The CSOUND struct has been rationalised and reorganised
- the public API has been redesigned

Developments to date: Code Reorganisation

The Csound code base is passing through a significant reorganisation:

- obsolete code, such as the old parser, has been removed.
- some opcodes have been completely rewritten
- The CSOUND struct has been rationalised and reorganised
- the public API has been redesigned

Csound6

Developments to date: Code Reorganisation

The Csound code base is passing through a significant reorganisation:

- obsolete code, such as the old parser, has been removed.
- some opcodes have been completely rewritten
- The CSOUND struct has been rationalised and reorganised
- the public API has been redesigned

Developments to date: Code Reorganisation

The Csound code base is passing through a significant reorganisation:

- obsolete code, such as the old parser, has been removed.
- some opcodes have been completely rewritten
- The CSOUND struct has been rationalised and reorganised
- the public API has been redesigned

Developments to date: Code Reorganisation

The Csound code base is passing through a significant reorganisation:

- obsolete code, such as the old parser, has been removed.
- some opcodes have been completely rewritten
- The CSOUND struct has been rationalised and reorganised
- the public API has been redesigned

Developments to date: Type System

The Csound language uses strongly typed variables and enforces these at compile-time. The system of types was hard-coded into the parser and compiler.

In Csound 6, a generic type system was implemented as well as tracking of variable names to types. The new system provides a mechanism to create and handle types, such that new types can be easily added to the language.

Variable definitions and types are kept after compilation. This allows the possibility of inspecting variables found in instruments or in the global memory space.

Csound6

Developments to date: Type System

The Csound language uses strongly typed variables and enforces these at compile-time. The system of types was hard-coded into the parser and compiler.

In Csound 6, a generic type system was implemented as well as tracking of variable names to types. The new system provides a mechanism to create and handle types, such that new types can be easily added to the language.

Variable definitions and types are kept after compilation. This allows the possibility of inspecting variables found in instruments or in the global memory space.

ffitch, Lazzarini and Yi (NUIM) Csound 6 LAC2013 9 / 23

Developments to date: Type System

The Csound language uses strongly typed variables and enforces these at compile-time. The system of types was hard-coded into the parser and compiler.

In Csound 6, a generic type system was implemented as well as tracking of variable names to types. The new system provides a mechanism to create and handle types, such that new types can be easily added to the language.

Variable definitions and types are kept after compilation. This allows the possibility of inspecting variables found in instruments or in the global memory space.

Developments to date: Generic Arrays

In Csound 5, a ‘t’ type was added that provided a user-definable length, single-dimension array of floating-point numbers.

In Csound 6, with the introduction of the generic type system, the code for t-types was extended to allow creation of homogenous, multi-dimensional arrays of any type. Additionally, the argument list specification for opcodes was extended to allow denoting arrays as arguments.

Developments to date: Generic Arrays

In Csound 5, a ‘t’ type was added that provided a user-definable length, single-dimension array of floating-point numbers.

In Csound 6, with the introduction of the generic type system, the code for t-types was extended to allow creation of homogenous, multi-dimensional arrays of any type. Additionally, the argument list specification for opcodes was extended to allow denoting arrays as arguments.

Developments to date: On-the-fly Compilation

The first steps necessary for on-the-fly compilation, were in the latter versions of Csound 5, with the introduction of the new parser. Also, compilation from text files was replaced by a new core (memory) file subsystem, so now strings containing Csound code could be presented directly to the parser.

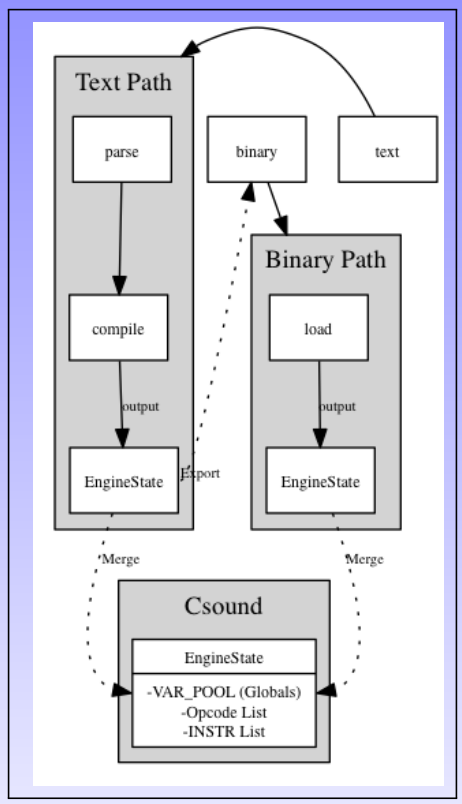
In Csound 6, the monolithic API call to compile/start Csound is broken down into `csoundParseOrc()` + `csoundCompileTree()` (with these two combined in `csoundCompileOrc()`) and `csoundStart()`. The parsing function creates an abstract syntax tree (AST) from a string containing Csound code. The compilation function then creates the internal data structures that the AST represents, ready for engine instantiation. These operations can be performed repeatedly.

Developments to date: On-the-fly Compilation

The first steps necessary for on-the-fly compilation, were in the latter versions of Csound 5, with the introduction of the new parser. Also, compilation from text files was replaced by a new core (memory) file subsystem, so now strings containing Csound code could be presented directly to the parser.

In Csound 6, the monolithic API call to compile/start Csound is broken down into `csoundParseOrc()` + `csoundCompileTree()` (with these two combined in `csoundCompileOrc()`) and `csoundStart()`. The parsing function creates an abstract syntax tree (AST) from a string containing Csound code. The compilation function then creates the internal data structures that the AST represents, ready for engine instantiation. These operations can be performed repeatedly.

Developments to date: On-the-fly Compilation



Developments to date: Sample Accuracy

Csound has always allowed sample-level accuracy, a feature present since its MUSIC 11 incarnation, by setting the processing block, (`ksmps`), to 1 sample.

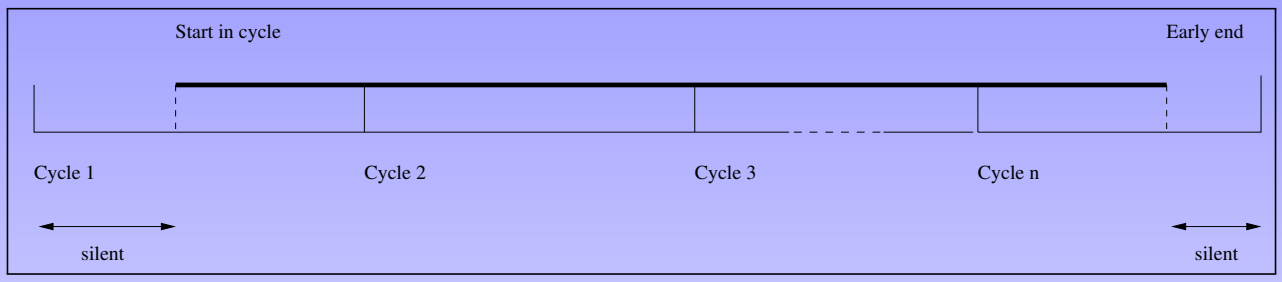
In Csound 6, an alternative sample accuracy method has been introduced, using an offset into the processing block, which will round the start time of an event to a single sample. Similarly, event durations are also made to be sample accurate, as the last iteration of each processing loop is limited to the correct number of samples.

Developments to date: Sample Accuracy

Csound has always allowed sample-level accuracy, a feature present since its MUSIC 11 incarnation, by setting the processing block, (`ksmps`), to 1 sample.

In Csound 6, an alternative sample accuracy method has been introduced, using an offset into the processing block, which will round the start time of an event to a single sample. Similarly, event durations are also made to be sample accurate, as the last iteration of each processing loop is limited to the correct number of samples.

Developments to date: Sample Accuracy



This option is provided with the non-default `--sample-accurate` flag, to preserve backward compatibility.

Developments to date: Realtime Priority Mode

Csound has been a realtime audio synthesis engine since 1990, although it was never provided with strict realtime-safe behaviour.

Given the multiple applications of Csound, it makes sense to provide separate operation modes for its engine. In Csound 6, we introduce the realtime priority mode, set by the `--realtime` option, which aims to provide better support for realtime safety, with complete asynchronous file access and a separate thread for unit generator initialisation.

Developments to date: Realtime Priority Mode

Csound has been a realtime audio synthesis engine since 1990, although it was never provided with strict realtime-safe behaviour.

Given the multiple applications of Csound, it makes sense to provide separate operation modes for its engine. In Csound 6, we introduce the realtime priority mode, set by the `--realtime` option, which aims to provide better support for realtime safety, with complete asynchronous file access and a separate thread for unit generator initialisation.

Developments to date: Multicore Operation

In 2009 an experimental system for using multiple cores for parallel rendering of instruments was written, and later incorporated in Csound 5. While the design was generally semantically correct it only delivered moderate performance gains.

For Csound 6 we are developing a different approach, where semantic analysis and the creation/consumption of the DAG of dependencies is only done on instantiation/destruction of instruments. This uses watch-lists as found in SAT-solvers.

Developments to date: Multicore Operation

In 2009 an experimental system for using multiple cores for parallel rendering of instruments was written, and later incorporated in Csound 5. While the design was generally semantically correct it only delivered moderate performance gains.

For Csound 6 we are developing a different approach, where semantic analysis and the creation/consumption of the DAG of dependencies is only done on instantiation/destruction of instruments. This uses watch-lists as found in SAT-solvers.

Developments to date: Multicore Operation, preliminary results

-j	CloudStrata	Xanadu		Trapped...		
	ksmps=500 (sr=96000)	ksmps=10	ksmps=100	ksmps=10	ksmps=100	ksmps=1000
1	1	1	1	1	1	1
2	0.54	0.57	0.55	0.75	0.79	0.78
3	0.39	0.40	0.40	0.66	0.76	0.73
4	0.32	0.39	0.33	0.61	0.72	0.70

Relative performance with multiple threads in three existing Csound code examples, -j indicates the number of threads used

Other developments

Since the time of the writing of this paper, other developments have taken place:

- Reordering of compilation stages and overhaul of the semantic system.
- support for functions with more than one arguments and opcodes as functions (that can be inlined in expressions) was added.

****

```
out linen(moogladder(vco2(p4,p5),
                     1000+linen(2000,0.5,p3,0.5),0.7),
                     1,p3,.1)
```


Csound6

Other developments

Since the time of the writing of this paper, other developments have taken place:

- Reordering of compilation stages and overhaul of the semantic system.
- support for functions with more than one arguments and opcodes as functions (that can be inlined in expressions) was added.

****

```
out linen(moogladder(vco2(p4,p5),
                     1000+linen(2000,0.5,p3,0.5),0.7),
                     1,p3,.1)
```

Other developments

Since the time of the writing of this paper, other developments have taken place:

- Reordering of compilation stages and overhaul of the semantic system.
- support for functions with more than one arguments and opcodes as functions (that can be inlined in expressions) was added.

****

```
out linen(moogladder(vco2(p4,p5),
                     1000+linen(2000,0.5,p3,0.5),0.7),
                     1,p3,.1)
```


Other developments

- multiple strings in event parameters are now allowed
- support for array operations has been added
- the 6.00 API has been finalised
- further internal changes (local ksmps in instrs, hash tables for strings, more code reorganisation, etc)
- Csound 6 Release Candidate 1 has been finalised, RC2 on its way.

Other developments

- multiple strings in event parameters are now allowed
- support for array operations has been added
- the 6.00 API has been finalised
- further internal changes (local ksmps in instrs, hash tables for strings, more code reorganisation, etc)
- Csound 6 Release Candidate 1 has been finalised, RC2 on its way.

Csound6

Other developments

- multiple strings in event parameters are now allowed
- support for array operations has been added
- the 6.00 API has been finalised
- further internal changes (local ksmps in instrs, hash tables for strings, more code reorganisation, etc)
- Csound 6 Release Candidate 1 has been finalised, RC2 on its way.

Other developments

- multiple strings in event parameters are now allowed
- support for array operations has been added
- the 6.00 API has been finalised
- further internal changes (local ksmps in instrs, hash tables for strings, more code reorganisation, etc)
- Csound 6 Release Candidate 1 has been finalised, RC2 on its way.

Csound6

Other developments

- multiple strings in event parameters are now allowed
- support for array operations has been added
- the 6.00 API has been finalised
- further internal changes (local ksmps in instrs, hash tables for strings, more code reorganisation, etc)
- Csound 6 Release Candidate 1 has been finalised, RC2 on its way.

Other developments

- multiple strings in event parameters are now allowed
- support for array operations has been added
- the 6.00 API has been finalised
- further internal changes (local ksmps in instrs, hash tables for strings, more code reorganisation, etc)
- Csound 6 Release Candidate 1 has been finalised, RC2 on its way.

Third-party frontend example

(example)

Cabbage Live Coder frontend by Rory Walsh

Future developments

A number of ideas have also been put forward, which will be tackled in due course. These include for instance:

- support for alternative orchestra languages (through access to the parse tree format or some sort of intermediary representation) and programmatic building/editing of instruments.
- further language features (e.g. namespaces, tuples, block scoping, dynamic typing)
- decoupling of widget opcodes from FLTK dependency (and exposure through API)
- input / output buffer reorganisation (output buffers added to instruments)

Conclusions

Future developments

A number of ideas have also been put forward, which will be tackled in due course. These include for instance:

- support for alternative orchestra languages (through access to the parse tree format or some sort of intermediary representation) and programmatic building/editing of instruments.
- further language features (e.g. namespaces, tuples, block scoping, dynamic typing)
- decoupling of widget opcodes from FLTK dependency (and exposure through API)
- input / output buffer reorganisation (output buffers added to instruments)

Future developments

A number of ideas have also been put forward, which will be tackled in due course. These include for instance:

- support for alternative orchestra languages (through access to the parse tree format or some sort of intermediary representation) and programmatic building/editing of instruments.
- further language features (e.g. namespaces, tuples, block scoping, dynamic typing)
- decoupling of widget opcodes from FLTK dependency (and exposure through API)
- input / output buffer reorganisation (output buffers added to instruments)

Conclusions

Future developments

A number of ideas have also been put forward, which will be tackled in due course. These include for instance:

- support for alternative orchestra languages (through access to the parse tree format or some sort of intermediary representation) and programmatic building/editing of instruments.
- further language features (e.g. namespaces, tuples, block scoping, dynamic typing)
- decoupling of widget opcodes from FLTK dependency (and exposure through API)
- input / output buffer reorganisation (output buffers added to instruments)

ffitch, Lazzarini and Yi (NUIM) Csound 6 LAC2013 21 / 23

Future developments

A number of ideas have also been put forward, which will be tackled in due course. These include for instance:

- support for alternative orchestra languages (through access to the parse tree format or some sort of intermediary representation) and programmatic building/editing of instruments.
- further language features (e.g. namespaces, tuples, block scoping, dynamic typing)
- decoupling of widget opcodes from FLTK dependency (and exposure through API)
- input / output buffer reorganisation (output buffers added to instruments)

Conclusions

We hope to have demonstrated how the technology embodied in this software package has been renovated continuously in response to developments in Computer Science and Music. Our aim is to continue to support a variety of styles of computer music composition and performance, as well as the various ways in which Csound can be used for application development.

It is also important to note, for readers, that the re-engineering of Csound is taking place quite publicly in the Csound 6 git repository on Sourceforge (`git://git.code.sf.net/p/csound/csound6-git`). Anyone is welcome to check out and examine our struggles with computer technology and the solutions we are putting forward here.

Conclusions

We hope to have demonstrated how the technology embodied in this software package has been renovated continuously in response to developments in Computer Science and Music. Our aim is to continue to support a variety of styles of computer music composition and performance, as well as the various ways in which Csound can be used for application development.

It is also important to note, for readers, that the re-engineering of Csound is taking place quite publicly in the Csound 6 git repository on Sourceforge (`git://git.code.sf.net/p/csound/csound6-git`). Anyone is welcome to check out and examine our struggles with computer technology and the solutions we are putting forward here.

Acknowledgements

Our thanks go to the Csound community for their indulgence, suggestions and support. In addition, we would like to thank Martin Brain who introduced the idea of watch-lists and co-developed the detailed performance algorithm. We also acknowledge the implicit support from Sourceforge hosting

This work has been partly funded by the Irish HEA PRTLI-5 Digital Arts and Humanities programme.

Acknowledgements

Our thanks go to the Csound community for their indulgence, suggestions and support. In addition, we would like to thank Martin Brain who introduced the idea of watch-lists and co-developed the detailed performance algorithm. We also acknowledge the implicit support from Sourceforge hosting

This work has been partly funded by the Irish HEA PRTLI-5 Digital Arts and Humanities programme.