Radium: A Music Editor Inspired by the Music Tracker

Kjetil Matheussen
Norwegian Center for Technology in Music and the Arts. (NOTAM)
Sandakerveien 24D, Bygg F3
N-0473 Oslo
Norway
k.s.matheussen@notam02.no

Abstract

Radium is a new type of music editor inspired by the *music tracker*. Radium's interface differs from the classical music tracker interface by using graphical elements instead of text and by allowing musical events anywhere within a tracker line.

Chapter 1: The classical music tracker interface and how Radium differs from it. Chapter 2: Radium Features: *a*) The Editor; *b*) The Modular Mixer; *c*) Instruments and Audio Effects; *d*) Instrument Configuration; *e*) Common Music Notation. Chapter 3: Implementation details: *a*) Painting the Editor; *b*) Smooth Scrolling; *c*) Embedding Pure Data; *d*) Collecting Memory Garbage in C and C++. Chapter 4: Related software.

Keywords

Radium, Music Tracker, GUI, Pure Data, Graphics Programming.

1 Introduction

The tracker interface appeared on the AmigaOS platform in late 80s and early 90s with programs such as Soundtracker, NoiseTracker and Protracker. The first tracker was called "The Ultimate Soundtracker",[1] and was released in 1987 by Karsten Obarski.[2]

In the classical tracker interface, time goes downwards. Notes placed higher on the screen are played before notes placed below.[3] Instead of moving the cursor up or down, the whole editor scrolls up or down, and the cursor is just locked in the middle of the screen.

The tracker editor shows a two-dimensional table in which musical events can be stored. We can think of it as a spreadsheet with *tracks* as columns and *lines* as rows.

Musical events are defined with pure text. The event `C#3 5-32-000` plays the note C sharp at octave 3 using instrument number 5 at volume 32. The last three zeroes can be used for various types of sound effects, or to set new tempo.

The tables are called *patterns*, and a song usually contains several patterns. To control the order patterns are playbed back, we use a *playlist*. For example, if we have three patterns, a typical song could have a playlist like this: 1, 2, 1, 2, 3, 1, 2.

1.1 How Radium Differs from the Classical Tracker Interface

Radium[4] differs from the music tracker interface by using graphical elements instead of text and by allowing any number of events to be placed anywhere.[5] The latter means that a line in Radium is essentially just a graphical hint. It should be possible to compose millions of years of music within just one tracker line.

These differences are so fundamental, that it's questionable whether Radium can be defined as a tracker.

1.2 History of Radium

The first version of Radium was released in year 2000 under the GPL license, and it only supported MIDI. After the initial release, Radium was developed actively for around a year, followed by a period between 2001 and 2012 with less development. Since 2012, Radium has been actively developed again.

The features presented in this paper have mostly been implemented in 2012 and 2013. Audio support was introduced in November 2012.

[1] According to Wikipedia: `http://en.wikipedia.org/wiki/Music_tracker`

[2] `http://en.wikipedia.org/wiki/Ultimate_Soundtracker`

[3] When I started making radium, I also considered letting time go in the horizontal direction. I don't remember why I chose the vertical direction.

[4] `http://users.notam02.no/~kjetism/radium/`

[5] This feature may not be useful, depending on how you compose music. But at least when using splitted lines, and for accurately importing midi files and other music formats, it's a necessary feature.

1.3 Portability

The first version of Radium was released for the Amiga Operating System (*AmigaOS*), version 3.0 or later. The code was written in a portable style, where non-portable code was clearly separated and easy to replace. An alpha version for Linux was available already in 2001.

Radium is at the time of writing available for Linux, Windows, and Mac OS X, where Linux is the main development platform and the platform with the most features. It should be straight forward to port Radium to a platform which has Jack, POSIX, and Qt.

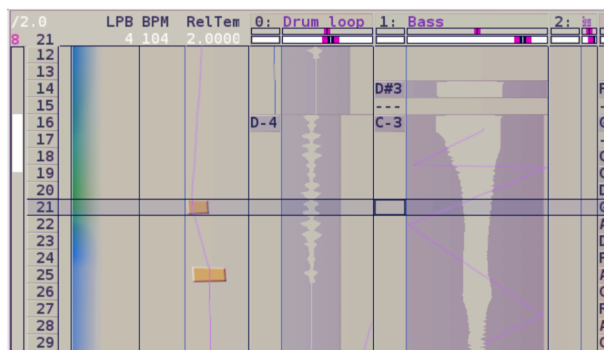
1.4 Term

In the rest of this paper, the word "line" means a tracker line, and not a vertical graphical line (i.e. a row of pixels) or an automation line. In cases where we refer to a graphical line, the expression "graphical line" will be used instead. In cases where we refer to an automation line, the expression "automation line" or "break point line" will be used.

2 Radium Features

2.1 The Editor

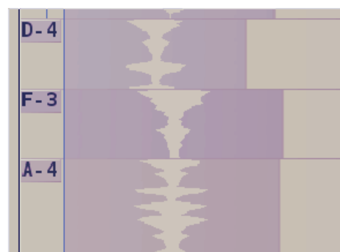
The image below shows a *Block* (the name of patterns in Radium).[6] From left to right, we see a vertical slider, line numbers (12-29), a green and blue area indicating tempo, an *LPB* track (Lines Per Beat), a *BPM* track (Beats Per Minute), a *RelTempo* track (for doing time-varying tempo changes), plus two sound tracks; a drum loop track and a bass track:



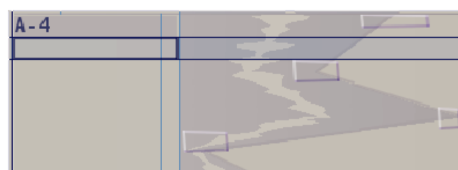
We also see that text is used to denote pitch ("D-4", "D#3", and "C-3"), while graphical break point lines are used to define tempo changes and effect automation. Pitch can be denoted with graphics too, using vertical lines, but text is clearer and more accurate.

2.1.1 Editor Elements

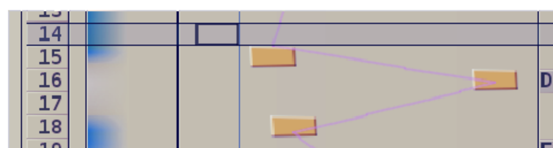
- Audio waveforms are shown in the tracks:



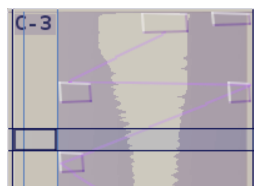
- Time-varying volume changes (crescendo/diminuendo) are defined using break point *curves*. The audio waveforms are updated in realtime:



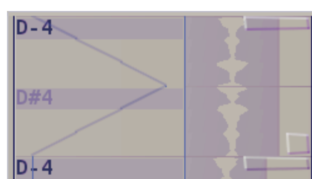
- Time-varying tempo changes (accelerando/ritardando) are defined with break point *lines*. The audio waveforms are updated in real time:



- Effects, e.g. reverb or chorus, are also defined with break point lines:



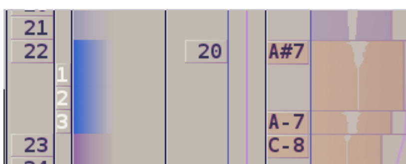
- ...And so are time-varying pitch changes (glissando's):



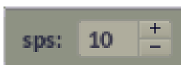
[6] The word "Block" comes from Octamed (`http://en.wikipedia.org/wiki/Octamed`). I think Block is a better name than Pattern, at least in Radium where events can be placed freely and doesn't have to follow a pattern.

- Pitch can be defined with unlimited precision. The pitch below is placed 82 cents above C sharp at octave 4:

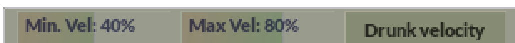
- Lines can be split. Splitting is essentially just a way to zoom in on one line so that you have more space to edit, but it can also be used to define measures. Furthermore, splitted lines can themselves be splitted, those lines again can also be splitted, and so forth:



- Updating the graphics too often can be tiring for the eyes. The SPS option (Scrolls Per Second) sets a limitation on the number of updates per second. SPS is an effective way to make the viewing more pleasurable when not using smooth scrolling.



- The velocity of new notes can be set using a random walk algorithm (drunk velocity). The algorithm tries to simulate how a musician varies volume while playing:



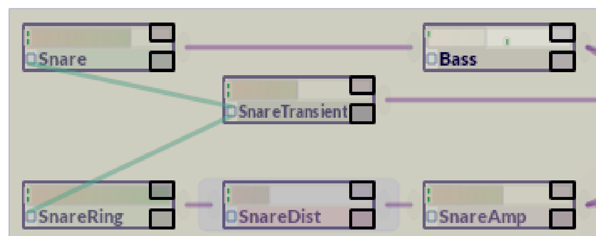
- All editing operations are undoable and redoable. The number of undoes is limited by system memory.

2.2 The Modular Mixer

The modular mixer provides a graphical interface to route note events and audio signals between sound objects.

The role of a sound object is to produce audio, receive audio, produce note events, receive note events, or any combination of those.

Inside each sound object in the Mixer GUI, there is a volume slider, a mute button, a bypass button, VU meters (one for each channel), and a "diode" that lights up when receiving note events:



The green lines show connections for note events, such as *Note On*, *Note Off*, Note volume changes, and Note pitch changes.

The other connections (those painted in a color that resembles mortar[7]) show audio connections.

2.2.1 Separate channel routing

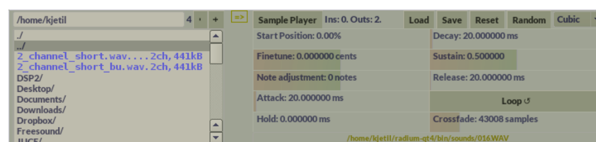
In the Mixer GUI, an audio connection sends all channels from one sound object to another. In order to (for instance) send only left channel, or only receive right channel, the audio connections must be routed through special channel routing objects.

The idea is that it's faster to use a little bit more time to route channels separately when necessary, than to always connect every channel manually.

2.3 Instruments and Audio Effects

2.3.1 Sampler Instrument

This instrument can play: 1) Normal soundfiles,[8] 2) Fasttracker instruments,[9] or 3) Soundfonts [Rossum and Joint, 1995]:



[7] "Name that color": http://chir.ag/projects/name-that-color/#594C5B

[8] All formats supported by libsndfile: http://www.mega-nerd.com/libsndfile/

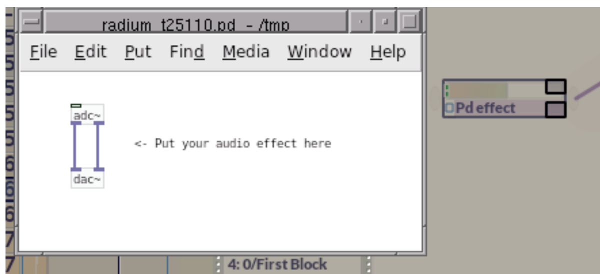
[9] Text files describing the Fasttracker "XI" instrument format: 1) "XI format description" by "KB / The Obsessed Maniacs / Reflex", 2) "The XM module format description for XM files version $0104" by "Mr.H of Triton" in 1994.

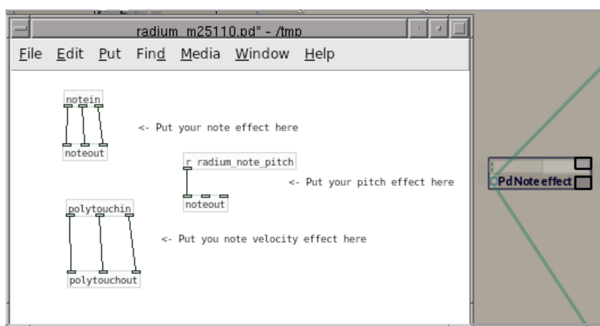
2.3.2 VST Plugins and instruments

Native VST plugins and instruments are supported on Linux, OSX and Windows:

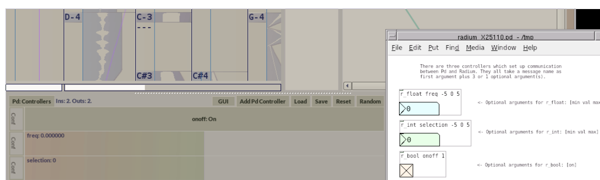
2.3.3 Pure Data (*Pd*)

Pd processes can be inserted anywhere in the sound graph. The Pd GUI is opened by double clicking the sound object. There is no limitation on the number of simultaneous instances:





Custom Pd controllers make it possible to control Pd from Radium, and to control Radium from Pd. The Pd controllers appear as Radium effects, similarly to "Max for Live":[10]



2.3.4 STK Instruments

Radium includes 20 STK instruments doing physical modeling [Cook and Scavone, 1999]. These are written by Romain Michon in the Faust language [Michon and Smith, 2011]. Michon's instruments have been slightly modified to be used as instruments in Radium.

[10] https://www.ableton.com/en/live/max-for-live/

2.3.5 Zita Reverb

Fons Adriaensen's "Zita Rev1" reverb (*Zita Reverb*),[11] implemented by Julius O. Smith III in Faust [Smith, 2012].[12] Zita Reverb is also used as the default reverb when creating a new song.[13]

2.3.6 Multiband Compressor

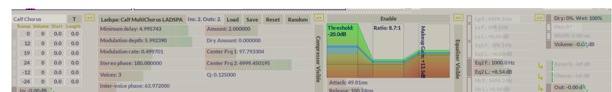
A multiband compressor. The DSP is implemented in Faust by using components written by Julius O. Smith III [Smith, 2012].: Compressor, lookahead limiter, bandsplit, and smoothing.

2.3.7 Other Instruments and Audio Effects

a) LADSPA plugins. Richard Furse's Linux Audio Developer's Simple Plugin API. *b*) Maarten de Boer's multitap delay "Tapiir" [De Boer, 2001], implemented by Yann Orlarey in Faust. *c*) A Fluidsynth instrument, using libfluidsynth.[14] *d*) Sound objects to send or receive audio to and from jack clients. *e*) A pipe object. *f*) Channel routing objects (section 2.2.1). *g*) MIDI output.

2.4 Instrument and Plug-in Configuration Widget

Sound goes through five parts in the instrument and plugin-in configuration widget. From left to right in the picture below, we see: 1) A *Note Duplicator*, 2) An automatically created Plugin/Instrument GUI, 3) A Compressor, 4) An Equalizer, 5) Settings for dry/wet, panning, stereo width, reverb, chorus, and output volume:



1) Instruments can play several note events when a note is played, using the *note duplicator*. This is convenient to, for instance, double the bass, or add a simple echo. Up to six notes can be played for each incoming note event, and for each of those six notes, the user can specify values for transposition, volume change, delay, and duration. If

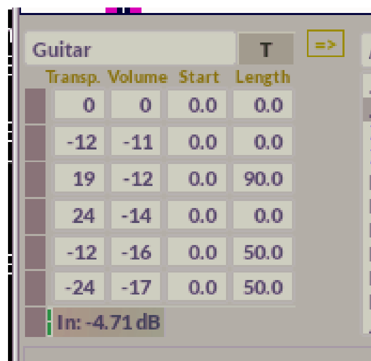
[11] http://kokkinizita.linuxaudio.org/linuxaudio/zita-rev1-doc/quickguide.html

[12] https://ccrma.stanford.edu/~jos/Reverb/Zita_Rev1_Reverberator.html

[13] The "Calf Multichorus" LADSPA plugin (written by Krzysztof Foltman) is used as the default Chorus effect.

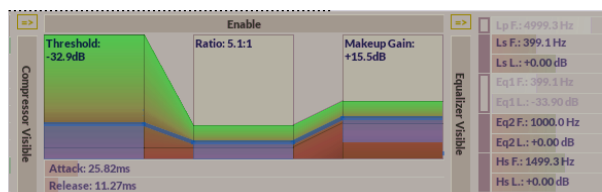
[14] http://www.fluidsynth.org

you need more than six notes, you can connect the sound object to yet another sound object to duplicate the notes further:



2) Sliders and buttons are automatically created for all instruments and plug-ins, based on the controllers they provide:

3) The compressor has a novel interface which tries to show more intuitively how the sound is squashed together. The DSP code is written in Faust by Julius O. Smith III [Smith, 2012].



2.5 Common Music Notation

Scores can be generated from Radium files automatically with Bill Schottstaedt's notation software *Common Music Notation* (CMN) [Schottstaedt, 1997].[15] The generated scores can be further tweaked in CMN, either by editing the generated CMN code, or by writing code that further modifies the CMN code. The latter technique is used to generate this score:

3 Implementation Details

Radium is mainly written in C and C++. Some code is also written in Python, Faust [Orlarey et al., 2009] and Scheme.[16]

3.1 Painting the Editor

The visible part of the editor is painted line by line to a backbuffer. When the editor is scrolling, we just copy corresponding tracker lines from the backbuffer into the screen. When a tracker line is not visible anymore, it is marked as free, and available for painting a new line. This way, we don't have to repaint everything for every update or scroll the screen or the backbuffer.

Unfortunately, this strategy causes the order of the lines in the backbuffer not to be chronological (newer lines often appear below older lines). Non-chronological order makes it impossible to paint graphical elements that span several lines in one operation. This limitation causes breakpoint boxes to be squashed up against the ceiling and floor of a tracker line,[17] and automation lines to be slightly not quite connected (or too much connected) because of anti-aliasing artifacts.[18] Scrolling the backbuffer[19] would not solve the problem either since graphical elements can start before the visible area, or end after the visible area. Therefore, at least some graphical elements has to be painted in several operations anyway.

An alternative solution that would solve the graphical problems, is to make the backbuffer big enough to contain the complete block. But this solution could occupy too much memory.[20]

[15] https://ccrma.stanford.edu/software/cmn/

[16] Using the Guile interpreter

[17] This effect probably looks more like a feature,

[18] while this effect probably looks more like a bug.

[19] or modifying Qt so that the underlying coordinate system would match the order of the lines in the backbuffer

[20] There are of course other solutions as well that would solve the graphical problems while still keeping the backbuffer, but I think they would complicate the code too much to be worth the effort.

However, since today's desktop computers (2014) seems fast enough to just repaint the screen when necessary, the strategy of painting tracker lines one by one in a backbuffer will, albeit so efficient that it made the program usable on hardware from 1992,[21] probably be removed in the near future. The advantage of not using a custom backbuffer is simpler code and less graphical artifacts, plus that it is simpler to add new graphical features when graphical operations are not bounded to be performed within tracker lines.

3.2 Smooth scrolling

Music trackers have traditionally updated the screen only when the current tracker line changes (i.e. scrolling line by line). By updating the screen at each vertical blank instead, we get smooth scrolling.

Smooth scrolling looks amazing compared to scrolling line by line, but perhaps more importantly is that smooth scrolling seems significantly less tiring for the eyes.

3.2.1 Render using the CPU

The first attempt to achieve smooth scrolling was to make Radium render the screen by copying line by line from the backbuffer at each vertical blank. To achieve sub-pixel accuracy, all painting operations on the backbuffer were performed n times, painting to n different back buffers, where each backbuffer was slightly skewed to the next one, all within the span of one pixel in the vertical direction. A good value for n would be at least 4.

One problem with this attempt was that the amount of time to render a frame varied a bit, and it was easy to lose the vertical blank deadline and get a frame glitch. The usual vertical blank period for an LCD screen is $16+\frac{2}{3}$ ms, so we don't have much time, and we can't trust the OS to wake us up soon enough if the current process for some reason has yielded in the middle of rendering.

A graphical glitch is very apparent when the whole screen moves in one direction at a constant speed, so to avoid frame glitches, Radium rendered frames in a separate thread and put them on a ringbuffer which the main thread would read from.[22] If a single frame took more time to render than $16+\frac{2}{3}$ ms, we still avoided a glitch if the average rendering time was less than $16+\frac{2}{3}$ ms.

However, this strategy didn't play very well with the current painting system (i.e. the code became very complicated), plus that it had a quite high CPU usage (which also made it more prone to frame glitches), so it was abandoned.

3.2.2 Render using the GPU

A more successful attempt at achieving smooth scrolling has been to use OpenGL in 2D mode. By letting the GPU repaint everything at each vertical blank, we achieve both smooth scrolling and a very low CPU usage. Another advantage is significantly smaller and simpler code since we don't use the type of backbuffer described in section 3.1 plus that scrolling is only a matter of sending updated y coordinates to OpenGL for the graphical objects.

This code is currently under development and should replace the current system soon.

3.3 Embedding Pd

Radium uses Peter Brinkmann's *libpd*[23] as basis to embed Pd.

Libpd is a thin layer of code that makes Pd into a library [Brinkmann et al., 2011]. Libpd doesn't include the Pd GUI, and it has some other limitations as well, so a "Radium fork" of libpd has been made for including features needed by Radium.[24]

The first modification was to re-add the GUI and create an API to control it. Several other enhancements and required modifications followed, such as loading and saving patches and adding a *void∗* argument to the midi functions.

3.3.1 Libpds (libpd with an extra 's')

However, the biggest challenge for using libpd is that only one Pd instance can run in a process simultaneously. With only one instance, you can't send sound from one patch to another in the Radium mixer (at least not if there is a non-pd sound object in the middle of those two). Or, for that matter, you can't make a LADSPA or VST plugin out of a Pd patch.

To circumvent this limitation, an additional library called *libpds* has been added to the Radium fork of libpd. Libpds makes it possible to load several Pd instances and communicate with them separately. Libpds has almost the same API as libpd, except that most functions take an additional "pd instance" parameter.

[21] Amiga 1200

[22] This strategy is similar to how we reliably get sound in real time from a non-deterministic source, for instance a hard drive.

[23] `http://libpd.cc`

[24] `http://github.com/kmatheussen/libpd`

Libpds works by dynamically loading a new libpd library file for each new Pd instance. To avoid symbol clash for the global variables between the various Pd instances, *dlopen* is called with the *RTLD_LOCAL* flag when opening "libpd.so". The RTLD_LOCAL flag prevents symbols from being shared globally.

Unfortunately, this behavior causes problems when loading Pd externals (i.e. plugins which are loaded during runtime). Pd externals require access to functions and global variables provided by Pd, but since Pd doesn't share its symbols globally, the externals fail to load.

The selected solution for the problem is to statically link the most common Pd externals into libpd. 921 externals are currently included, and among them are most of the externals distributed with the Pd distribution *Pd-Extended*.[25] In order to compile that many externals without manually writing a large Makefile, a script recursively scans a list of directories and compiles all externals it can find.

A slightly simpler way to load externals would be to link the Pd externals directly (i.e. instead of recompiling), but using a ".so" file as a static library does not work.

It is likely that there are better ways to support externals, such as implementing a new dynamic linking system, but the current solution seems to work well for now

3.4 Garbage collection

Radium has from the start used Hans Boehm's garbage collector for C and C++ as memory manager (BDW-GC) [Boehm and Weiser, 1988]. It is not necessary to free memory manually when using a garbage collector, so Radium has fewer lines of code, and most likely fewer bugs, because of this choice.

There has been no trouble with BDW-GC, and Radium has not had memory leaks. It is strange that BDW-GC is not used in most large programs written for C or C++.

4 Related software and how their features compare to Radium

4.1 Jeskola Buzz

Jeskola Buzz[26] appeared in 1997-1998.[27] Jeskola Buzz was probably the first tracker with a modular mixer. The modular mixer in Radium is inspired by the one in Jeskola Buzz, but the modular mixer in Jeskola Buzz doesn't support sending note events or sound objects with more than two channels.

4.2 Aodix

Aodix[28] was released before 2002, but I don't know when. Aodix may have been the first *tickless* tracker, depending on how old Aodix is. Tickless means that events are not bounded by tracker lines, a feature which is shared with Radium. Another feature shared with Radium is that you can apparently zoom in and out of the patterns.

4.3 Renoise

Renoise[29] was released in 2002. Renoise is a more traditional tracker than Jeskola Buzz and Aodix, but has more features.

Renoise uses one instrument per track, which is similar to Radium, but Renoise lets you organize tracks further by optionally grouping tracks and instruments. For instance by grouping all drum tracks or all vocal tracks. Grouping makes patterns visually clearer and simpler to navigate and it simplifies adding effects to a group of instruments (since they are already grouped). Grouping is a feature that is currently missing in Radium.

Renoise also supports effect automation and tempo automation, but unlike Radium, the graphics is placed horizontally in a separate area below the tracks, and not in the tracks themselves.

5 Conclusion

Radium presented a radical change to the classical tracker interface when it was released fourteen years ago.

The following is a list of larger tracker features that first appeared in Radium (at least to my knowledge). An appending * means that Radium is still the only tracker, or tracker-like, program that provides this feature, at least to my knowledge:

a) Smooth scrolling*; *b*) Limitation on the number of scrolls per second*; *c*) Tickless timing (may have been introduced in Aodix before Radium); *d*) Zoom in/out (may have been introduced in Aodix before Radium); *e*) Waveform data visible in tracks; *f*) The "Radium Compressor" compressor interface*; *g*) Pitch values shown graphically; *h*) Tempo automation*; *i*) Effect automation*; *j*) Volume automation*;

[25] http://puredata.info/downloads/pd-extended

[26] http://www.jeskola.net/buzz/

[27] http://en.wikipedia.org/wiki/Jeskola_Buzz

[28] http://www.kvraudio.com/product/aodix-by-arguru-software/details

[29] http://www.renoise.com

k) Pitch automation*; *l*) Adjustable track widths*; *m*) Pd or Max/MSP integration*; *n*) Track headers with volume control and instrument name*; *o*) Automatic MIDI preset change when playing note for instrument with different preset*; *p*) Line splitting (including line split splitting, line split split splitting, etc.)*; *q*) Unlimited number of simultaneously playing notes per track, and no limitation when they are allowed to start and stop playing*;[30] *r*) Unlimited number of blocks, tracks and lines; *s*) Generate scores with CMN*; *t*) Unlimited undo/redo; *u*) Send pitch change events between instruments*;[31] *v*) Configurable menus*.

This list of (more or less useful) new features shows that Radium has tried to be an innovator for tracker software. Radium will try to be an innovator in the future as well.

6 Acknowledgements

Radium is an open source program which includes code from several other programs and uses several open source libraries. Today's Radium would not exists without the open source community. Some of the people who have written code that's used in Radium are (apologies to those I've forgotten):

Fons Adriaensen: Zita REV1; Conrad Berhörster / Josh Green / Peter Hanappe / David Henningsson / Pedro López-Cabanillas / Antoine Schmitt: Fluidsynth; Michele Bosi: Visualisation Library; Hans Boehm / Ivan Maidanski: BDW-GC; Peter Brinkmann: libpd; Rui Nuno Capela: code from QTractor to auto-create Plugin GUI's and show VST GUI's; Paul Davis / Stephane Letz: Jack; Ray Donnelly / Alexey Pavlov / Roumen Petrov: MinGW Python; Dominique Fober / Albert Gräf / Stephane Letz / Yann Orlarey / Julius O. Smith III: Faust; Krzysztof Foltman: The CALF multichorus LADSPA plugin; Grigor Iliev: The Soundfont parser in libgig; Giles Hall: The python-midi library; Bob Ham: Code from Jack-Rack to organize LADSPA plugins using liblrdf; Steve Harris: liblrdf; Erik de Castro Lopo: libsamplerate and libsndfile; Romain Michon: The Faust STK instruments; Paul Mineiro: Fast functions to calculate exponential and logarithmic values; Javier Serrano Polo: Vestige; Miller Puckette: Pd; Yann Orlarey: The Tapiir effect implementation and smooth delay code; Bjorn Roche: Memory barrier code; Gary P. Scavone: RtMidi; Bill Schottstaedt: CMN; Julius O. Smith III: Compressors / lookahead limiter / filters / equalizer; Hans-Christoph Steiner et al.: Pd-Extended; www.magnetophon.nl: The included Blowfish demo song; TumaGonx Zakkum: LADSPA plugins for Windows.

I also want to especially thank Yann Orlarey for creating the Faust programming language and Julius O. Smith III for all the DSP code he has written for Faust. Their work has saved me a lot of time and ensured professional sound quality.

[31]I.e polyhponic aftertouch for pitch instead of volume

References

Hans-Juergen Boehm and Mark Weiser. 1988. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807–820.

Peter Brinkmann, Peter Kirn, Richard Lawler, Chris McCormick, Martin Roth, and Hans-Christoph Steiner. 2011. Embedding Pure Data with libpd. In *Proceedings of the Pure Data Convention*.

Perry R Cook and Gary Scavone. 1999. The Synthesis Toolkit (STK). In *Proceedings of the International Computer Music Conference*, pages 164–166.

Maarten De Boer. 2001. Tapiir, a Software Multitap Delay. In *Conference on Digital Audio Effects, Limerick, Ireland*.

Romain Michon and Julius O Smith. 2011. Faust-STK: a Set of Linear and Nonlinear Physical Models for the Faust Programming Language. In *Proceedings of the 11th International Conference on Digital Audio Effects (DAFx-11)*, page 199.

Yann Orlarey, Dominique Fober, and Stephane Letz. 2009. FAUST: an efficient functional approach to DSP programming. *New Computational Paradigms for Computer Music, Editions Delatour France*, pages 65–96.

Dave Rossum and E Joint. 1995. The SoundFont® 2.0 File Format.

Bill Schottstaedt. 1997. Beyond midi. chapter Common Music Notation, pages 217–221. MIT Press, Cambridge, MA, USA.

Julius O Smith. 2012. Signal Processing Libraries for Faust. In *Proceedings of the Linux Audio Conference 2012*, pages 153–161.