iPlug 2: Desktop Plug-in Framework Meets Web Audio Modules

Oliver Larkin
Creative Coding Lab
The University of Huddersfield
oli@webaudiomodules.org

Alex Harker
Creative Coding Lab
The University of Huddersfield
a.harker@hud.ac.uk

Jari Kleimola
Web Audio Modules
Espoo, Finland
jari@webaudiomodules.org

ABSTRACT

This paper introduces iPlug 2: a desktop C++ audio plug-in framework that has been extended and reworked in order to support Web Audio Modules, a new format for browser-based audio effects and instruments, using WebAssembly. iPlug 2 provides a complete solution and workflow for the development of cross-platform audio plug-ins and apps. It allows the same concise C++ code to be used to create desktop and web-based versions of a software musical instrument or audio effect, including audio signal processing and user interface elements. This new version of the framework has been updated to increase its flexibility so that alternative drawing APIs, plug-in APIs and platform APIs can be supported easily. We have added support for the distributed models used in recent audio plug-in formats, as well as new graphics capabilities. The codebase has also been substantially modernised. In this paper, we introduce the problems that iPlug 2 aims to address and discuss trends in modern plug-in APIs and existing solutions. We then present iPlug 2 and the work required to refactor a desktop plug-in framework to support the web platform. Several approaches to implementing graphical user interfaces are discussed as well as creating remote editors using web technologies. A real-world example of a WAM compiled with iPlug 2 is hosted at https://virtualcz.io, a new web-based version of a commercially available synthesizer plug-in.

1. INTRODUCTION

Audio plug-in formats such as VST, Audio Units, LV2 and AAX are well-established as a means of providing specific audio synthesis, audio processing or MIDI capabilities to a host environment. Each format has an application programming interface (API) that is based on a specific model of interaction with the host. Nevertheless, these formats all share the fundamental requirements of processing audio samples, handling parameter changes, managing state and displaying a user interface. The Web Audio Module (WAM) format aims to offer similar functionality in the browser [7].

The iPlug 2 C++ framework presented in this paper allows the same codebase that is used to create a desktop plug-in to be used to generate a Web Audio Module. Figure 1 shows VirtualCZ, a software synthesizer programmed using iPlug 2, running in both a popular desktop Digital Audio Workstation (DAW), and in a web browser, demonstrating a highly consistent visual appearance between the two.

Figure 1. VirtualCZ VST in the Reaper DAW

WAM in Google Chrome

1.1 Audio Plug-in Frameworks

There are many challenges that audio plug-in developers face when making a product for a wide audience. Most plug-ins must work with a range of different hosts, which are generally not limited to a single operating system and will only support certain plug-in formats. A *cross-platform* user interface (UI) toolkit must be used for what is usually a single window interface, often featuring controls such as sliders, meters, knobs and buttons. The UI toolkit is responsible for drawing to the window, handling events and creating contextual controls, potentially supporting both mouse and touch interaction. Yet more platform differences occur when files must be read or written. Finally, plug-ins often need to be able to support multiple processor architectures.

For audio plug-in developers targeting multiple formats and platforms, it is impractical to code for each one individually, as this approach results in long development times, and a large codebase that is hard to maintain. Ideally, the same code is used to target multiple formats *and* platforms, whilst ensuring compliance with all the APIs involved - plug-in APIs, drawing APIs and platform APIs. This is the task of an *audio plug-in framework* such as iPlug. In our view, such a framework should allow a developer to iterate on ideas quickly and focus on the creative elements of their task, whilst at the same time producing reliable, performant binaries in multiple formats and with a high degree of parity in look and feel across all platforms. In a real-time audio context, there are many details of implementation, such as ensuring thread-safety, that can distract a developer from their creative focus. Ideally a plug-in framework should handle such details elegantly. Thus, we introduce iPlug 2 which aims to address these issues, improving upon the original iPlug, whilst maintaining a degree of backward compatibility with existing code. iPlug 2 adds support for additional target formats and platforms, including Web Audio

Web Audio Conference WAC-2018, September 19–21, 2018, Berlin, Germany.

Modules. For the first time, the same C++ DSP and UI code can be used to create desktop and web versions of an audio plug-in - opening new possibilities for distribution, documentation, and promotional activities.

2. CONTEXT

The VST audio plug-in standard introduced by Steinberg in 1997 is a landmark in the history of music technology, which significantly changed the world of music production. It created a platform for software virtual instruments and effects which offered portability, affordability and scalability for consumers, compared with hardware alternatives.

Technology and software development practices have changed significantly since the time of the first plug-in instruments and effects. Modern plug-in APIs reflect technological developments and address problems and limitations in earlier APIs. VST2, which is still widely used, has a simplistic C interface which gives plug-in and host developers significant freedom but it exemplifies some core problems in the domain. A key example is the ambiguity regarding the timing and threading of parameter change events, with respect to the audio processing. On today's multi-core processors, thread-safety is a far greater concern than it may have been during the late nineties, perhaps explaining the lack of a clear threading specification in the original API.

Here we present an overview of the defining features of some modern plug-in formats that have been considerations for the development of iPlug 2[1].

2.1 VST3

Steinberg's VST3[2] was released in 2008 and has a C++ API which is much more verbose than VST2, explicitly specifying many details that were previously ambiguous. VST3 introduces a distinction between a *controller* and a *processor* part of an audio plug-in, and if the developer follows the recommendations of the API documentation, he/she must assume that these two elements may run in different address spaces, or even on different computers. Whilst this separation promotes good practice of isolating user interface code from real-time audio processing code, it also presents challenges in terms of keeping the state of the two elements synchronised, as well as for transferring any data between the two. Normalized parameters are used for single value communication, which may include additional "read only" parameters in order to allow functionality such as metering visualisations. For more complex synchronisation a messaging system exists which can send arbitrary data, but, depending on the complexity of a plug-in's requirements, the developer may need to implement multiple queueing mechanisms for sending/receiving messages safely on the high priority audio thread.

2.2 Audio Unit Extensions (AUv3)

Apple's AUv3 API provides the first unified API for Audio Unit development across the macOS and iOS platforms. An AUv3 plug-in must be implemented using the Objective-C or Swift programming languages, with the audio processing callback using C/C++. Possibly the biggest change in AUv3 is the introduction of a model that runs the Audio Unit in a separate, sandboxed process, protecting the host from misbehaving plug-ins. This adds a small amount of overhead, and from a plug-in developer's point of view - especially when targeting multiple formats - flexibility and functionality are limited. Since the extension is sandboxed it no longer has access to the computer's file system (which is an issue when resources are shared with other formats) and it may not be possible to do things such as communicate with other instances or external hardware. If AUv3 plugins are to be used on both desktop and mobile platforms different HCI modalities must be considered in UIs, as well as the different screen space afforded by different devices.

2.3 LV2

The LV2 plug-in API is written in C with RDF semantic web descriptors to provide information about the functionality the plug-in supports [11]. It is highly extensible and addresses many issues discussed here. For example, it promotes the separation of DSP and user interface into different binaries, which can be run out of process. LV2 offers a great deal of flexibility at the expense of ease of use, although this complexity may be reduced by targeting via an intermediary audio plug-in framework.

2.4 Web Audio Modules

The Web Audio Module (WAM) API [7] is the most recent of all APIs discussed here. It is still under development, as it uses technology that has only recently become available in mainstream web browsers and only a small number of web-based audio plug-in host environments exist [2][5]. The WAM format has been conceived and developed outside of industry, and aims to support a wide range of end users and use cases, with a community website[3] to get feedback from other interested parties whilst the API is developed. It has been designed taking into consideration some of the issues discussed with existing plug-in APIs and aims to be lightweight and flexible, so that it may be used with existing plug-in frameworks and codebases, and readily support experimental and creative uses of audio plug-ins. Like VST3, the WAM API splits the concept of a plug-in into two components, although the relationship between the two parts is different [7]. The *controller* is implemented in JavaScript and extends the Web Audio API's AudioWorkletNode, running on the browser's main thread. The *processor* is typically implemented in C/C++ and is compiled to WebAssembly using the emscripten toolchain[4] [12]. The processor extends AudioWorkletProcessor and is thus sandboxed for security reasons, since it runs on browser's high priority audio thread. User interfaces are implemented on the controller side with any technologies available in the browser. Communication between the two parts is asynchronous, and the parts cannot directly access each other's properties.

2.5 Existing Solutions

Companies producing plug-ins in multiple formats will often make use of an intermediary plug-in framework for the reasons stated above, but many have proprietary solutions that are not publicly available. In the public domain, the JUCE library[5] is an extremely popular choice for audio plug-in developers, and provides a wide

[1] Although they both offer unique approaches and solutions to issues discussed here, AVID's AAX and Propellerhead's Rack Extensions are not included in this list, since the SDKs are not in the public domain.

[2] https://www.steinberg.net/en/company/technologies/vst3.html

[3] https://webaudiomodules.org

[4] http://emscripten.org

[5] https://juce.com

range of functionality. It might be considered alternatively as an application framework, plug-in framework, plug-in hosting framework, UI toolkit and even a DSP library. Other publicly available solutions include the DISTRHO plug-in framework[6], RackAFX[7] [10] and Dplug[8]. The VST3 SDK contains VSTGUI (which is suitable for use with a range of plug-in formats) as well as wrappers for several other plug-in APIs. Developers may create their plug-in as a VST3 primarily and then wrap it to other formats. Symbiosis[9] is another wrapper tool that some developers use to support the AUv2 format by wrapping VST2 plug-ins. Whilst wrappers allow for perfect compliance with the original API, this approach can cause problems where the wrapped plug-in's API is significantly different to the wrapper - an example of this might be the case of VST3 where MIDI controller messages are handled in a way that is very different to other plug-in APIs.

Support for individual plug-in formats varies across the solutions presented above. Although it promotes separation of audio processing and graphics code, JUCE does not currently support distributed plug-ins. It has a restrictive license due to its commercial nature. DISTRHO lacks AU, VST3 and AAX, support and has limited graphics functionality, although it does support distributed LV2, and a range of Linux plug-in formats. Dplug uses the D programming language, which has not been widely adopted. RackAFX is a stand-alone windows application that can be used to create code for VST2, VST3, AAX and AU plug-ins. None of the solutions above offer web browser support. One project that does support the web, but not desktop plug-ins is JSAP by Jillings et al. [3].

3. IPLUG 2

The iPlug framework was originally developed by John Schwartz and provided support for the VST2 and AUv2 formats. It was open-sourced in 2008 as part of Cockos' WDL[10]. Since then several modified versions have been available, including the first author's "WDL-OL" which added VST3, RTAS, AAX, and standalone app targets. At its core the original iPlug has two abstract C++ interfaces: *IPlug* - for plug-in API communication and *IGraphics* - for the UI, including layout, drawing, platform-specific event handling, contextual controls and filesystem access. The base class for a UI element is an *IControl*. In IDE projects and makefiles, pre-processor macros are used to switch between different graphics and plug-in API base classes, as required.

The development of iPlug 2 was motivated by the following aims:

- Creative Coding: iPlug 2's abstract interface cleanly presents the core elements of an audio plug-in. We believe this promotes creativity, allowing developers to quickly iterate on the most important parts: the DSP, UI and UX. For example, iPlug 2 simplifies the repetitive task of defining the characteristics of parameters, which can now be achieved with a single line of code [8].
- Simplicity of plug-in developer experience: iPlug 2 comes with a well-defined project structure and ready-made IDE example projects which should compile easily. Projects can quickly be created and duplicated. By abstracting the complicated details of plug-in APIs, the framework becomes well suited for teaching purposes - it is easy for learners to get results quickly
- Portability and extensibility: iPlug 2 can target different architectures, operating systems, plug-in APIs and UI drawing backends. More targets and functionality can be added easily.
- Minimal and lightweight API classes: iPlug 2 itself uses lightweight data structures - it is limited in scope, acting solely as a plug-in framework and aims to excel at that one function.
- Permissive license: iPlug 2 offers the freedom to have control over the elements of code that interact with the plug-in host.

3.1 Refactoring iPlug

The development of iPlug 2 was prompted by the desire to support Web Audio Modules, but also by a need to address limitations of the original framework in order to make it future proof and support modern plug-in APIs, whilst maintaining a good degree of backwards compatibility with existing product codebases. Thus, we did not have the option to completely rewrite the framework from the ground up.

Firstly, we decided that the IPlug and IGraphics components should be able to operate independently, i.e. it should be possible to make a plug-in using the IPlug base classes but with a different UI toolkit. Likewise, it should be possible to use IGraphics to implement simple user interfaces without an iPlug plug-in underneath.

The next set of changes relate to drawing. iPlug's original drawing backend using Cockos' LICE was severely limited in the area of vector graphics. Modern plug-ins must work on a variety of screen resolutions and at different DPIs with high frame rates for smooth animation. A trend in modern plug-in UIs is to make use of vector graphics rather than bitmap-based approaches, since they are more suitable for rescaling and have a small memory footprint. This is particularly important on the web where the size of a website payload must be minimised. For these reasons, we implemented multiple new IGraphics drawing backends to evaluate different approaches, each of which has distinct advantages and disadvantages, in terms of portability, features and performance. We also added an abstract interface IGraphicsPathBase, which extends IGraphics with a generic way of supporting path-based drawing. Only a small number of methods need to be implemented to support a given path-based drawing API, and this approach has been used for implementations using Cairo[11], NanoVG[12], Anti-Grain Geometry[13], and the HMTL5 canvas. The IGraphics interface was also modified to support multi-resolution bitmaps and at the same time we now support drawing SVGs into a graphics context. A new library of controls has been added, including themeable and animated vector-based controls.

Another major change was a requirement for distributed UI and DSP components in order to support Web Audio Modules and

[6] https://github.com/DISTRHO/DPF

[7] http://www.willpirkle.com/rackafx/

[8] https://github.com/AuburnSounds/Dplug

[9] http://nuedge.net/article/5-symbiosis

[10] https://www.cockos.com/wdl/

[11] https://cairographics.org

[12] https://github.com/memononen/nanovg

[13] http://www.antigrain.com

provide better VST3 compliance. This change needed to be seamless and transparent whilst maintaining a simple API and so necessitated a more flexible class hierarchy.

Finally, we modernised the codebase. The original version of iPlug used a small subset of C++ 98 functionality, with numerous issues due to multiple developers of different levels of experience having modified the code over many years. Making use of C++11 allows for simpler and safer code constructions, such as the inclusion of 'actions' attached to UI controls, which use C++11 lamda functions. This means that developers don't have to write custom controls for custom functionality. Lamdas are also used to implement custom animations, parameter display strings and several other useful features.

4. IPLUG 2 WAM SUPPORT

In [7] the authors discussed a preliminary version of iPlug with support for Web Audio Modules, but no graphics capability. Since that time, the WAM API has changed slightly due to the arrival of the AudioWorklet and WebAssembly[3].

4.1 Workflow

iPlug 2 aims to provide a quick workflow for desktop plug-ins and the same should apply when producing a WAM - ideally building a WAM should be as easy as any other plug-in target. A build script handles compilation via the emscripten toolchain followed by the packaging of all the required files. This may be executed from an IDE such as Xcode, allowing the build process to be handled alongside other formats. To compile a WAM, the plugin developer just needs to install the emscripten toolchain and place the WAM SDK in the correct folder.

There are two options provided when generating a WAM - the first is a "stand-alone" web page designed to allow the WAM to be used in isolation, with boilerplate code for connecting with the Web Audio graph and Web Midi [6]. The second option produces a *Web Component* ready for embedding into WAM hosting applications [2] – where each Web Component has a shadow DOM that avoids namespace conflicts when multiple plug-ins are hosted on a single page.

4.2 Distributed Elements

In the build process, there are two WebAssembly (WASM) compilation stages - one for the IPlugWAM WASM module that extends the AudioWorkletProcessor, and one for the IPlugWeb WASM module that communicates with the AudioWorkletNode and which owns the plugin UI (see Figure 2). Integration with the Web Audio API is via the WAM processor and controller interfaces. In order to support iPlug's single class approach, these compilation stages involve inheriting two different base classes and the switching of base classes is handled by the build script, so the plug-in developer does not need to deal with the separation.

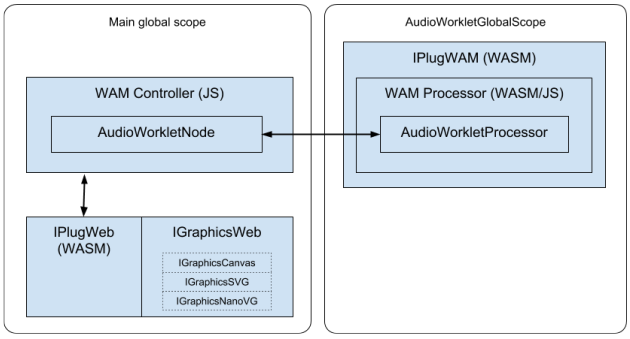

Figure 2. Separation of IPlugWeb and IPlugWAM base classes with respect to AudioWorklet and WAM interfaces

The distribution of editor and processor parts of a plug-in is facilitated by the doubling of the plug-in state (i.e. the current values of all parameters plus any arbitrary state data). This allows either element of the plug-in to access state data rapidly. Although this wasn't a requirement for WAM support, when dealing with other distributed plug-in formats, such as VST3, this means that the editor can feature custom preset handling functionality, even though the main state serialisation code must be handled on the processor side.

4.2.1 Plug-in Processor

The audio processing part of iPlug 2's WAM support differs only slightly from the implementation discussed in [7]. This is because the WAM processor is now in the AudioWorkletGlobalScope whereas previously, when the ScriptProcessorNode was used, it was in global scope. When the IPlugWAM class is compiled, audio processing, midi processing and parameter handling methods are valid but UI-related code is excluded, along with preset handling. The plug-in developer must exclude any UI only code via the C pre-processor.

4.2.2 Plug-in Editor

In [7] we provided options to create generic user interfaces, from the WAM JSON descriptor, but new user interfaces had to be implemented for the web. This can be seen with WebCZ101 - a prototype WAM using the same DSP as the first author's plug-in VirtualCZ, but with a new UI written using polymer[14]. In the process of updating iPlug we investigated whether using web technologies (HTML/JS/CSS) for UI across desktop and web targets was an option, to leverage the ever-increasing layout and rendering capabilities of modern browser engines. At the time of writing, platform web views are inconsistent across different operating systems, as are the methods available for communicating with them, so this was not a viable solution. We also investigated embeddable web rendering engines, but everything currently available was either a significant dependency (a large dynamic library) or had a complicated build system, conflicting with the overall aims of iPlug 2.

To achieve a consistent UI across all platforms we decided to continue to use IGraphics (via C++) and to create a second WASM module - which inherits the IPlugWeb base class. This *editor* module runs in the global scope, and is thus able to communicate with a WAM's controller, as well as the DOM.

[14] https://www.polymer-project.org/

4.3 IGraphics on the Web

IGraphics classes for the web are implemented in C++, but the code largely consists of JavaScript calls transliterated using emscripten's embind library. Events are handled by the IGraphicsWeb platform base class using emscripten's HTML5 library.

There are several options for adding drawing support, each extending IGraphicsWeb into a concrete class:

4.3.1 IGraphicsCanvas

IGraphicsCanvas draws to a single HTML5 canvas object. Vector drawing is simplified by inheriting the IGraphicsPathBase interface, with most methods mapped to a single canvas method call, except for *PathStroke* and *PathFill*, which require a larger number of calls to set multiple properties before drawing.

Bitmap support poses a specific challenge in the web environment where resource loading is typically asynchronous, unlike in desktop applications which have a synchronous model. For parity with the desktop we cannot make use of the browser's file loading mechanisms to load and decode image resources, and must instead preload files via the emscripten virtual file system.

4.3.2 WebGL via IGraphicsNanoVG

NanoVG is an antialiased 2D vector graphics library for OpenGL which has been adapted to work with recent GPU based technologies such as Apple's Metal. On the desktop, IGraphicsNanoVG provides iPlug 2 with an option for hardware accelerated vector graphics, and the integration of 3D displays and shaders with IControls, which may be useful for some plug-in UIs. NanoVG can also be compiled with emscripten using it's GLFW emulation library to render with WebGL. In this case the same text rasterizer can be used, making this a good option for parity of look and feel across all platforms, at the expense of increasing the payload of the web page and duplication of functionality already available in the browser. Whilst it is likely that most browser canvas drawing is hardware accelerated in some way, WebGL may still offer better performance with large full screen high-resolution UIs, as well as the assurance of hardware acceleration where available.

4.3.3 IGraphicsSVG

The last option we explored was an experiment using the browser's vector graphics backend. An IGraphicsSVG implementation provided an SVG surface holding any number of interactive or static controls. Each control was encapsulated inside an SVG group element for individual attachment, detachment and transformation actions. Although all SVG elements are inherently mouse sensitive, IGraphicsSVG had to accept mouse events at the root element level to make it consistent with the other IGraphics implementations. Instead of conforming to IGraphicsPathBase's procedural path drawing primitives, it followed SVG's object-based paradigm and offered corresponding primitives as classes. The primitives were instantiated during the control initialization phase and attached as children of the SVG group element. The runtime control state was reflected through the primitive's SVG attributes and CSS properties.

Since IGraphicsSVG was only suitable for the web, it was removed from the codebase, however it demonstrates the extensibility of the iPlug 2 framework: from the plug-in developer's perspective, all IControls look alike despite the different graphics backends.

4.4 Cross-platform issues

Targeting multiple platforms with different user interface modalities and different restrictions such as sandboxing, means it is good to have flexibility in choice of the approaches used for UI, depending on the project. An example would be in the case of pop-up menus and text entry controls. Users may expect to see platform native-style controls - but on some platforms there are limited options for customising such controls. Furthermore, in the web browser contextual menus and text entry controls are regularly implemented with differing look and feel to the native platform. For this reason, we have added the option of using bespoke controls for menus and text entries which are then drawn using IGraphics.

5. REMOTE EDITORS

Remote editor functionality arises as a side-effect of enabling separation between plug-in processor and editor in iPlug 2, along with IGraphics web support. Given these features of the framework it becomes possible to provide near identical-looking remote editors for desktop plug-ins via WebSockets, by changing the communication layer between editor and processor. In this use case, tablets or other devices run the editor which is used to interact with a desktop plug-in running on a remote machine via a local network. The IPlugWeb class – the plugin editor - is compiled as a WASM module, but instead of being connected to the IPlugWAM WASM module in the browser, messages to and from the user interface are sent over a WebSocket connection. This has been made possible by defining a messaging protocol between editor and processor parts, including variations for sending parameter changes, MIDI and arbitrary data. The WebSocket server in the desktop plug-in seamlessly keeps all the clients for a particular plug-in instance in sync.

Figure 3. Remote editor and visualizer for a suite of spatialisation plug-ins using WebVR, via the aframe.io JavaScript library.

The technologies available in the browser may also be used to provide supplementary interfaces for desktop audio plug-ins that would not be feasible inside a traditional single window audio plug-in UI, and iPlug 2 comes with a generic WebSocket server interface in order to facilitate these use cases. An example of this has been implemented using WebVR to visualise panning locations (azimuths and elevations) in a suite of spatial audio tools developed at the University of Huddersfield (see Figure 3). Since the plug-ins

are running a server, devices can connect as WebSocket clients over a local network. This allows us to support a wide range of VR headsets, including mobile phone-based systems such as Google cardboard.

6. FUTURE WORK

At the time of writing iPlug 2 is still under development. Work needs to be done on documentation, portability, optimisation and further simplifying the codebase. There are issues that will need to be addressed as more desktop plug-ins are ported to the Web Audio Module format, including the translation of platform SIMD and multi-threaded code to WebAssembly-friendly equivalents.

A more formal evaluation of the performance of different graphics libraries and drawing approaches is necessary both for web and desktop deployment. Initial findings are positive, suggesting that despite calls between code written in different programming languages user interfaces remain responsive on current hardware, even with complex graphical elements.

In this paper, we have presented approaches to creating *imperative* UIs using C++ code originally developed for desktop plug-ins. Another approach for future exploration in the context of iPlug is to use embedded web views for desktop plug-in UIs. This is dependent on the availability of lightweight platform web views or embeddable rendering engines capable of consistent results across platforms.

We added support for bespoke controls for popup menus and text entries (rather than using platform controls) but this has implications for the accessibility of plug-in UIs. Screen readers have no knowledge of such custom controls, or of any IControls, for that matter. iPlug has previously been used for accessible interfaces for audio software using techniques such as sonification [9]. Further work is necessary to develop solutions that can natively offer good accessibility and this is something that should be handled at framework-level to proliferate accessible interfaces. In this regard, *declarative* HTML-based UIs are an appealing option since they can retain a high degree of semantic information for screen readers.

7. CONCLUSION

iPlug 2 offers a solution for rapidly deploying audio plug-ins to a range of targets, including the web browser. The framework allows the creation of desktop and web plug-ins from a single codebase, easing the process of porting plug-ins from existing C++ code, including their graphical user interfaces.

In this paper, the key features of iPlug 2 related to web audio have been discussed. These include seamless support for distributed editor and processor parts (required for using IGraphics with the WAM format) as well as a more modular approach allowing different drawing APIs to be used. We have also presented several options for web-based drawing backends and we discussed using these with WebSockets to enable remote editors for desktop plug-ins. iPlug 2 brings together web technologies and the world of audio plug-ins. We hope that this work will help create a richer ecosystem for Web Audio.

iPlug 2 is liberally licensed and is available for free under the WDL license at https://github.com/iPlug2. The WAM version of VirtualCZ can be tried online at https://virtualcz.io

8. ACKNOWLEDGEMENTS

We would like to acknowledge the support of The Creative Coding Lab at The University of Huddersfield. Thanks to Justin Frankel and John Schwartz from Cockos, as well as all contributors to the iPlug framework and the open-source libraries it uses.

9. REFERENCES

[1] Adenot, P., Toy, R., Wilson, C., Web Audio API, W3C Working Draft, 08 December 2015 and W3C Editor's Draft, 19 April 2018. Available online at http://www.w3.org/TR/webaudio/ and http://webaudio.github.io/web-audio-api/

[2] Buffa, M., Lebrun, J., Kleimola, J., Larkin, O., Letz, S. 2018. Towards an open Web Audio plug-in standard. In *Proceedings of the International World Wide Web Conference (WWW '18)*. Lyon, France.

[3] Choi, H. 2018. AudioWorklet: The future of web audio. In *Proceedings of the International Computer Music Conference (ICMC2018)*, Daegu, South Korea.

[4] Jillings, N., Wang Y., Reiss, J.D. and Stables, R. 2016. JSAP: A plugin standard for the Web Audio API with intelligent functionality. In *Proceedings of the Audio Engineering Society Convention 141,* Los Angeles, USA.

[5] Jillings, N. and Stables, R. 2017. An Intelligent audio workstation in the browser. In *Proceedings of 3rd Web Audio Conference*, London, UK.

[6] Kalliokoski, J., and Wilson, C. Web Midi API, W3C Working Draft, 17 March 2015 and W3C Editor's Draft, Draft 07 November 2017. Available online at http://www.w3.org/TR/webmidi/ and http://webaudio.github.com/web-midi-api/

[7] Kleimola, J., and Larkin, O. 2015. Web Audio Modules. In *Proceedings of the 12th Sound and Music Computing Conference (SMC-2015)*. Maynooth, Ireland.

[8] Larkin, O. 2018. Faust in iPlug 2: Creative coding audio plug-ins. In *Proceedings of the 1st International Faust Conference (IFC-18)*. Mainz, Germany.

[9] Martin, F., Metatla, O., Bryan-Kinns, N., Stockman T. 2016. Accessible Spectrum Analyser. In *Proceedings of the 22nd International Conference on Auditory Display (ICAD-2016)*. Atlanta, USA.

[10] Pirkle, W. 2012. Designing Audio Effect Plug-Ins in C++: With Digital Audio Signal Processing Theory. Focal Press (Taylor & Francis Group).

[11] Robillard, D. 2014. LV2 Atoms: A Data Model for Real-Time Audio Plugins. In *Proceedings of the Linux Audio Conference (LAC-2014)*. Karlsruhe, Germany.

[12] Zakai, A. 2011. Emscripten: an LLVM-to-JavaScript compiler. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*. New York, USA.