Wasm Audio Worklets API

The AudioWorklet extension to the Web Audio API specification enables web sites to implement custom AudioWorkletProcessor Web Audio graph node types.

These custom processor nodes process audio data in real-time as part of the audio graph processing flow, and enable developers to write low latency sensitive audio processing code in JavaScript.

The Emscripten Wasm Audio Worklets API is an Emscripten-specific integration of these AudioWorklet nodes to WebAssembly. Wasm Audio Worklets enables developers to implement AudioWorklet processing nodes in C/C++ code that compile down to WebAssembly, rather than using JavaScript for the task.

Developing AudioWorkletProcessors in WebAssembly provides the benefit of improved performance compared to JavaScript, and the Emscripten Wasm Audio Worklets system runtime has been carefully developed to guarantee that no temporary JavaScript level VM garbage will be generated, eliminating the possibility of GC pauses from impacting audio synthesis performance.

Audio Worklets API is based on the Wasm Workers feature. It is possible to also enable the -*pthread* option while targeting Audio Worklets, but the audio worklets will always run in a Wasm Worker, and not in a Pthread.

Development Overview

Authoring Wasm Audio Worklets is similar to developing Audio Worklets API based applications in JS (see MDN: Using AudioWorklets), with the exception that users will not manually implement the JS code for the ScriptProcessorNode files in the AudioWorkletGlobalScope. This is managed automatically by the Emscripten Wasm AudioWorklets runtime.

Instead, application developers will need to implement a small amount of JS <-> Wasm (C/C++) interop to interact with the AudioContext and AudioNodes from Wasm.

Audio Worklets operate on a two layer “class type & its instance” design: first one defines one or more node types (or classes) called AudioWorkletProcessors, and then, these processors are instantiated one or more times in the audio processing graph as AudioWorkletNodes.

Once a class type is instantiated on the Web Audio graph and the graph is running, a C/C++ function pointer callback will be invoked for each 128 samples of the processed audio stream that flows through the node. Newer Web Audio API specs allow this to be changed, so for future compatibility use the `AudioSampleFrame`'s `samplesPerChannel` to get the value.

This callback will be executed on a dedicated separate audio processing thread with real-time processing priority. Each Web Audio context will utilize only a single audio processing thread. That is, even if there are multiple audio node instances (maybe from multiple different audio processors), these will all share the same dedicated audio thread on the AudioContext, and will not run in a separate thread of their own each.

Note: the audio worklet node processing is pull-mode callback based. Audio Worklets do not allow the creation of general purpose real-time prioritized threads. The audio callback code should execute as quickly as possible and be non-blocking. In other words, spinning a custom *for(;;)* loop is not possible.

Programming Example

To get hands-on experience with programming Wasm Audio Worklets, let's create a simple audio node that outputs random noise through its output channels.

1. First, we will create a Web Audio context in C/C++ code. This is achieved via the `emscripten_create_audio_context()` function. In a larger application that integrates existing Web Audio libraries, you may already have an `AudioContext` created via some other library, in which case you would instead register that context to be visible to WebAssembly by calling the function `emscriptenRegisterAudioObject()`.

Then, we will instruct the Emscripten runtime to initialize a Wasm Audio Worklet thread scope on this context. The code to achieve these tasks looks like:

```
#include <emscripten/webaudio.h>

uint8_t audioThreadStack[4096];

int main()
{
  EMSCRIPTEN_WEBAUDIO_T context = emscripten_create_audio_context(0);

  emscripten_start_wasm_audio_worklet_thread_async(context, audioThreadStack,
sizeof(audioThreadStack),
                                                   &AudioThreadInitialized, 0);
}
```

2. When the worklet thread context has been initialized, we are ready to define our own noise generator AudioWorkletProcessor node type:

```
void AudioThreadInitialized(EMSCRIPTEN_WEBAUDIO_T audioContext, bool success, void
*userData)
{
  if (!success) return; // Check browser console in a debug build for detailed errors
  WebAudioWorkletProcessorCreateOptions opts = {
    .name = "noise-generator",
  };
  emscripten_create_wasm_audio_worklet_processor_async(audioContext, &opts,
&AudioWorkletProcessorCreated, 0);
}
```

3. After the processor has initialized, we can now instantiate and connect it as a node on the graph. Since on web pages audio playback can only be initiated as a response to user input, we will also register an event handler which resumes the audio context when the user clicks on the DOM Canvas element that exists on the page.

```
void AudioWorkletProcessorCreated(EMSCRIPTEN_WEBAUDIO_T audioContext, bool success,
void *userData)
{
  if (!success) return; // Check browser console in a debug build for detailed errors

  int outputChannelCounts[1] = { 1 };
  EmscriptenAudioWorkletNodeCreateOptions options = {
    .numberOfInputs = 0,
    .numberOfOutputs = 1,
    .outputChannelCounts = outputChannelCounts
  };

  // Create node
  EMSCRIPTEN_AUDIO_WORKLET_NODE_T wasmAudioWorklet =
emscripten_create_wasm_audio_worklet_node(audioContext,
                                                            "noise-generator",
&options, &GenerateNoise, 0);

  // Connect it to audio context destination
  emscripten_audio_node_connect(wasmAudioWorklet, audioContext, 0, 0);

  // Resume context on mouse click
  emscripten_set_click_callback("canvas", (void*)audioContext, 0, OnCanvasClick);
}
```

4. The code to resume the audio context on click looks like this:

```
bool OnCanvasClick(int eventType, const EmscriptenMouseEvent *mouseEvent, void
*userData)
{
  EMSCRIPTEN_WEBAUDIO_T audioContext = (EMSCRIPTEN_WEBAUDIO_T)userData;
  if (emscripten_audio_context_state(audioContext) != AUDIO_CONTEXT_STATE_RUNNING) {
    emscripten_resume_audio_context_sync(audioContext);
  }
  return false;
}
```

5. Finally we can implement the audio callback that is to generate the noise:

```
#include <emscripten/em_math.h>

bool GenerateNoise(int numInputs, const AudioSampleFrame *inputs,
                      int numOutputs, AudioSampleFrame *outputs,
                      int numParams, const AudioParamFrame *params,
                      void *userData)
{
  for(int i = 0; i < numOutputs; ++i)
    for(int j = 0; j < outputs[i].samplesPerChannel*outputs[i].numberOfChannels; ++j)
      outputs[i].data[j] = emscripten_random() * 0.2 - 0.1; // Warning: scale down
audio volume by factor of 0.2, raw noise can be really loud otherwise

  return true; // Keep the graph output going
}
```

And that's it! Compile the code with the linker flags `-sAUDIO_WORKLET=1 -sWASM_WORKERS=1` to enable targeting AudioWorklets.

Synchronizing audio thread with the main thread

Wasm Audio Worklets API builds on top of the Emscripten Wasm Workers feature. This means that the Wasm Audio Worklet thread is modeled as if it was a Wasm Worker thread.

To synchronize information between an Audio Worklet Node and other threads in the application, there are three options:

1. Leverage the Web Audio "AudioParams" model. Each Audio Worklet Processor type is instantiated with a custom defined set of audio parameters that can affect the audio computation at sample precise accuracy. These parameters are passed in the `params` array into the audio processing function.

   The main browser thread that created the Web Audio context can adjust the values of these parameters whenever desired. See MDN function: setValueAtTime.

2. Data can be shared with the Audio Worklet thread using GCC/Clang lock-free atomics operations, Emscripten atomics operations and the Wasm Worker API thread synchronization primitives. See WASM_WORKERS for more information.
3. Utilize the `emscripten_audio_worklet_post_function_*()` family of event passing functions. These functions operate similar to the `emscripten_wasm_worker_post_function_*()` functions. They enable a `postMessage()` style of communication, where the audio worklet thread and the main browser thread can send messages (function call dispatches) to each other.

More Examples

See the directory tests/webaudio/ for more code examples on Web Audio API and Wasm AudioWorklets.