Transmisja danych dźwiękiem w JavaScript od podstaw

Część 2: Web Audio API

W pierwszej części artykułu (numer 8/2016) poznaliśmy od podstaw zagadnienie Dyskretnej Transformaty Fouriera z punktu widzenia programisty. Do realizacji prostej wymiany danych dźwiękiem w JavaScript potrzebujemy jednak czegoś więcej. Następnym klockiem w układance jest Web Audio API. Postaramy się wykorzystać z niego tylko te elementy, które będą przydatne do realizacji naszego celu. Pozwoli nam to zaimplementować własną klasę AudioMonoIO, która opakuje wspomniane API do postaci prostego interfejsu. Na końcu stworzymy aplikację, będącą wirtualnym pianinem, za pomocą której będziemy mogli przesyłać dźwięki pomiędzy dwoma urządzeniami.

Web Audio API jest bardzo rozbudowanym systemem udostępnianym przez większość nowoczesnych przeglądarek. Daje ono możliwość przetwarzania sygnałów dźwiękowych oraz kontrolę tego procesu bezpośrednio z poziomu JavaScript. Otwiera nam to drogę do aplikacji umożliwiających np. obróbkę dźwięku, filtrowanie czy tworzenie muzyki ze zbioru sampli. W przypadku gier będą to wszelkie efekty dźwiękowe. Najprostszym przykładem może być potrzeba odwzorowania akustyki sceny (echo, pogłos) czy też emitowanie dźwięku kroków z proporcjonalną głośnością w lewym lub prawym głośniku w zależności od strony, z której nadciąga wróg. Web Audio API dla wielu z tych przypadków posiada już gotowe „klocki".

Jak widzimy, oferowane możliwości są bardzo duże. Oczywiście nie będziemy korzystać ze wszystkich dostępnych elementów. Musimy się zatem zastanowić, co tak naprawdę będzie nam potrzebne do realizacji prostej wymiany danych. Zanim jednak tego dokonamy, przeprowadźmy małą retrospekcję tego, co udało nam się zrealizować do tej pory. Pozwoli nam to wstępnie określić wymagania odnośnie Web Audio API nawet bez jego znajomości.

W poprzedniej części artykułu poznaliśmy od podstaw prosty i intuicyjny algorytm Dyskretnej Transformaty Fouriera. Do jego przetestowania stworzyliśmy sztuczny sygnał składający się z kilku przebiegów sinusoidalnych. Jako że rzeczywisty kanał transmisyjny nigdy nie jest pozbawiony zakłóceń, zdecydowaliśmy się dodać łatwy do zasymulowania biały szum. Zbliżyliśmy się zatem w małym stopniu do realnego świata. Tak przygotowany sygnał przepuściliśmy przez wspomniany algorytm DTF. Ostatecznie udało nam się z powodzeniem wyciągnąć główne częstotliwości tworzące sygnał, nawet gdy był on dość zakłócony. Oprócz częstotliwości przebiegów sinusoidalnych udało nam się także wydobyć informację o ich przesunięciu fazowym. Warto zaznaczyć, że przy tych wszystkich działaniach nie została użyta żadna gotowa biblioteka DSP. Cel, jaki wyznaczyliśmy, to poznanie tematu od podstaw bez żadnego magicznego pudełka, którego nie bylibyśmy w stanie zaimplementować samodzielnie. Wyjątkiem jest tutaj oczywiście sam proces próbkowania i emisji dźwięku. Tę część zwyczajnie musimy pozostawić sprzętowi.

Niestety ceną prostoty użytego algorytmu DTF jest jego bardzo wolne działanie w porównaniu z alternatywą, jaką jest FFT (ang. *Fast Fourier Transform*). Również generowanie w czasie rzeczywistym tylko jednej sinusoidy przy standardowym próbkowaniu 44.1 kHz wymaga wywołania 44100 razy funkcji `Math.sin()`. Warto o tym pamiętać, ponieważ może być to szczególnie istotne, gdy jednym z urządzeń, jakie użyjemy do testów, będzie np. smartfon.

Już na pierwszy rzut oka widać, że najważniejsze, czego potrzebujemy od Web Audio API, to uzyskanie dostępu do prawdziwych próbek audio z naszego mikrofonu. Jako że rolą drugiego urządzenia będzie nadawanie sygnału, musimy mieć także możliwość odtwarzania strumienia próbek na głośnikach. W obydwu przypadkach z powodzeniem wystarczy tylko jeden kanał dźwięku (mono).

Niestety, jak przekonałem się podczas testów praktycznych, z uwagi na wolniejsze urządzenia nie jest możliwe skorzystanie tylko z naszych własnych rozwiązań DSP. Na szczęście w Web Audio API możemy znaleźć gotowe elementy do przetwarzania wejścia, jak i generowania wyjścia.

Osobiście nie lubię takiej formy „oszukiwania" po wcześniejszym ustaleniu założeń projektu. Niestety jest to jedyne wyjście w przypadku np. smartfonów. Algorytmy wbudowane w API przeglądarki są dużo szybsze niż ich JavaScriptowe odpowiedniki oraz często są dodatkowo akcelerowane sprzętowo. Finalnie zatem postaramy się zaimplementować dwie drogi prowadzące do tego samego celu.

Wygląda na to, że mniej więcej wiemy już, czego oczekujemy od Web Audio API. Nie pozostaje nam zatem nic innego, jak prześledzić szczegółowo jego elementy i połączyć wszystko w logiczną całość. Dobrze byłoby także opakować całość komunikacji z Web Audio API do postaci prostego interfejsu zawierającego tylko to, co jest nam potrzebne. Mając do dyspozycji taką pojedynczą klasę, uprościmy znacznie kod przykładów, które będą pojawiać się w dalszej części artykułu. Naszą klasę nazwiemy `AudioMonoIO`.

To tyle tytułem wstępu. Nasz plan jest gotowy, zatem do dzieła!

1. WEB AUDIO API – WPROWADZENIE

Pierwszym krokiem do wykorzystania potencjału, jaki oferuje nam Web Audio API, jest stworzenie tak zwanego kontekstu audio. Jak tego dokonać? Wystarczy powołać do życia obiekt klasy `AudioContext` za pomocą operatora `new`. Instancja tak stworzonego obiektu stanowić będzie kontener dla węzłów audio, w których tak naprawdę odbywać się będzie faktyczne przetwarzanie strumienia dźwiękowego. Węzły te możemy łączyć ze sobą w dowolny sposób za pomocą metody `connect`. Wyjście jednego z nich może posłużyć jako wejście innego. Na przykład `GainNode` umożliwia zmianę poziomu głośności sygnału podanego na wejściu. Na wyjściu otrzymamy zatem podgłośniony lub ściszony strumień, który może posłużyć za wejście innego węzła bądź węzłów.

Węzły, które posiadają jedynie wyjścia, nazywamy źródłowymi (`SourceNode`). W naszym przypadku najważniejszym będzie `MediaStreamSourceNode`, który umożliwia dostęp do strumienia audio z naszego mikrofonu. Jego inicjalizacja wymaga nieco więcej zabiegów niż w przypadku innych węzłów, ale to opiszemy bardziej szczegółowo w dalszej części artykułu. Przykładem innego węzła źródłowego jest `OscillatorNode`, który generuje sygnał okresowy o zadanej częstotliwości i kształcie.

W naszym kontekście audio może istnieć wiele węzłów źródłowych. Strumienie te oraz ich dalsze drogi wcale nie muszą tworzyć jednej wspólnej połączonej całości. Każdy z nich może tworzyć osobny graf efektów, przez który przechodzić będzie pewien strumień audio.

Skoro dostępne są m.in. węzły źródłowe, to czy istnieje także węzeł wyjścia? Oczywiście istnieje i nazywa się `AudioDestinationNode`. Dostęp do jego instancji zrealizowany jest przez właściwość `destination` stworzonego wcześniej obiektu kontekstu. Węzeł ten jest inicjalizowany automatycznie wraz z kontekstem audio i reprezentuje „finalny cel" strumienia audio. W większości przypadków będzie on równoznaczny z głośnikami naszego komputera. Jeżeli zatem chcemy usłyszeć dźwięk wyprodukowany przez graf naszych nodów, należy ostatni jego element podłączyć do wspomnianego węzła destynacji.

Skoro instancja węzła destynacji dostępna jest w obiekcie kontekstu z automatu, to czy podobnie jest z innymi węzłami? Otóż nie – obiekty innych węzłów musimy utworzyć samodzielnie. Jest to całkiem logiczne, ponieważ w naszym grafie możemy zażyczyć sobie kilka nodów tego samego typu. Mając po jednej instancji każdego węzła, byłoby to po prostu niemożliwe. Tutaj z pomocą przychodzi nam sam kontekst audio. Udostępnia on metody wytwórcze dla każdego typu węzła. Innymi słowy operator `new` staje się zbędny.

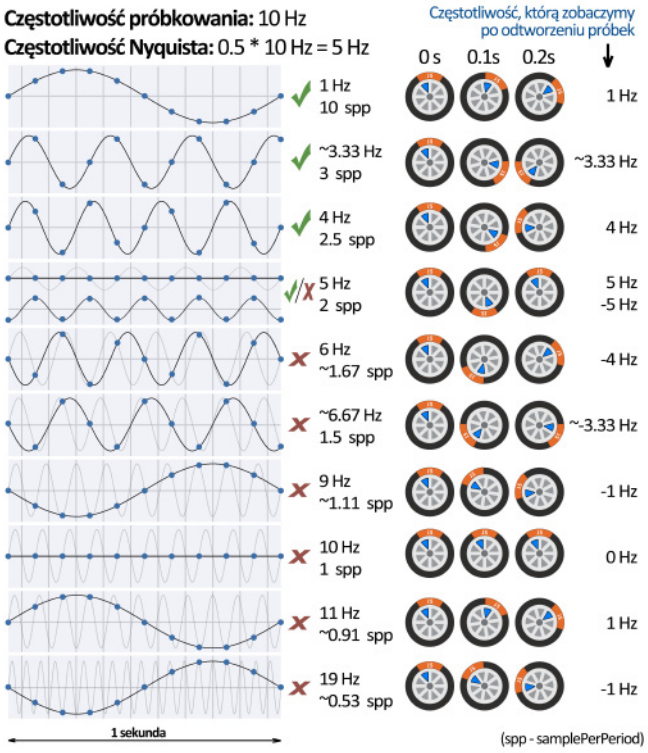
Chcąc utworzyć np. `GainNode`, musimy wywołać metodę `audioContext.createGain()`, oczywiście przy założeniu, że nasz kontekst audio znajduje się w zmiennej o nazwie `audioContext`. Uogólniając, konwencja tworzenia nodów jest następująca: chcąc powołać do życia obiekt typu `{Typ}Node`, musimy posłużyć się metodą `audioContext.create{Typ}()`.

Dlaczego zrezygnowano z podejścia wykorzystującego operator `new`? Zapewne między innymi dlatego, że każdy z węzłów działa tylko w obrębie swojego kontekstu audio. Próba wywołania metody `connect` na nodach należących do różnych kontekstów wyrzuci wyjątek. Użycie wzorca metody wytwórczej pozwala m.in. „pod maską" dokonać automatycznego powiązania stworzonego węzła z odpowiadającym mu kontekstem.

W Web Audio API dostępnych jest prawie 20 różnych typów nodów. W tym artykule skupimy się tylko na tych, które umożliwią nam realizację transmisji danych. Zanim jednak przejdziemy do ich szczegółowego omawiania, zaczniemy od pojęć podstawowych.

2. KILKA SŁÓW O PRÓBKOWANIU

Po poprawnej inicjalizacji obiektu kontekstu audio możemy z niego odczytać bardzo ważną wartość, jaką jest częstotliwość próbkowania: `audioContext.sampleRate`. W większości przypadków będzie ona równa 44.1 kHz lub 48 kHz. Dlaczego akurat tyle? Dlatego aby w pełni pokryć zakres częstotliwości, jakie są w stanie usłyszeć ludzie. Przypomnijmy: przeciętny człowiek słyszy częstotliwości z zakresu od 16 Hz do 20 kHz. Skoro zatem największa częstotliwość słyszana przez człowieka to 20 kHz, dlaczego więc próbkujemy z częstotliwością ponad dwukrotnie większą? Aby odpowiedzieć na to pytanie, prześledzimy przypadek z dużo mniejszą częstotliwością próbkowania równą 10 Hz (Rysunek 1).



Rysunek 1. Częstotliwość próbkowania oraz częstotliwość Nyquista

Jak widzimy, przy częstotliwości próbkowania 10 Hz zapisanie jednego pełnego okresu sinusoidy o częstotliwości 1 Hz wymaga użycia 10 sampli. W poprzedniej części artykułu do tego celu używaliśmy określenia `samplePerPeriod`. Zwiększanie częstotliwości sinusoidy sprawia, że coraz mniej sampli przypada na jej jeden okres. Okazuje się, że gdy dochodzimy do częstotliwości sinusoidy równej połowie częstotliwości próbkowania (5 Hz), na jeden jej okres przypadają dokładnie dwa sample. W zależności od fazy fali może być to już za mało, by móc ją poprawnie zapisać. Po przekroczeniu 5 Hz w naszym materiale zaczną pojawiać się częstotliwości, których nie ma w rzeczywistości. Zniekształcenia tego typu noszą nazwę aliasingu. Wrócimy jeszcze do tego zagadnienia w dalszej części artykułu.

Z tego przykładu możemy wyciągnąć prosty wniosek. Maksymalna częstotliwość zawarta w sygnale nie może przekraczać połowy częstotliwości próbkowania. Wartość połowy częstotliwości próbkowania nazywana jest częstotliwością Nyquista. Warto to zapamiętać, gdyż ta nazwa będzie się przewijać w dalszej części tego artykułu.

Wróćmy do kontekstu audio. Dla jednej z popularnych częstotliwości próbkowania 44100 Hz połową będzie 22050 Hz. Oznacza to, że najwyższa częstotliwość, jaką będziemy mogli poprawnie od-

tworzyć z zapisanych próbek, będzie mniejsza niż 22050 Hz. Skoro przeciętny człowiek słyszy dźwięki o maksymalnej częstotliwości 20 kHz, po co nam zatem zakres od 20 kHz do 22.05 kHz? Wynika to z faktu, że sygnał przed próbkowaniem musi zostać poddany filtracji (filtr dolnoprzepustowy), aby usunąć wszystkie częstotliwości, których nie bylibyśmy w stanie poprawnie zapisać. Problem w tym, że nie istnieje filtr, który przepuszcza w pełni częstotliwości poniżej 20 kHz, usuwając przy tym całkowicie częstotliwości ponad 20 kHz. W praktyce zawsze musi istnieć pewien zakres pośredni. W tym przypadku ten zakres ma szerokość 2050 Hz. Czy jednak to my musimy martwić się filtrowaniem sygnału z mikrofonu przed jego próbkowaniem? Odpowiedź brzmi: nie, gdyż robi to za nas sprzęt. Od Web Audio API dostajemy jedynie informację, jaka jest obowiązująca wartość częstotliwości próbkowania. Niestety nie istnieje żadna metoda do zmiany tej wartości z poziomu skryptu.

Dla dociekliwych: wartość 44.1 kHz wybrano jakiś czas temu w wyniku porozumienia twórców sprzętu. Stała się ona standardem np. w formacie płyt CD z muzyką. Dodatkowo ma ciekawą właściwość, gdyż jest ona równa iloczynowi kwadratów czterech pierwszych liczb pierwszych: $44100 = 2^2 * 3^2 * 5^2 * 7^2$. Sprawia to, że dla wielu wygodnych dzielników (np. 2, 3, 4, 5, 6, 7, 9) dzieli się bez reszty. Warto jednak dodać, że obecnie odchodzi się od tej wartości na rzecz 48 kHz ($48000 = 2^7 * 3 * 5^3$).

3. FALA AKUSTYCZNA

W tej sekcji opowiemy nieco, jak fala akustyczna podróżuje w powietrzu oraz jak ją generować i odbierać. Otóż rozchodzi się ona w formie zaburzeń jego gęstości i ciśnienia z prędkością rzędu 320-350 m/s (nieco ponad 1000 km/h). Przekłada się to na długość fali rzędu np. 6.7 m dla tonu 50 Hz, 33.5 cm dla tonu 1 kHz czy 2.2 cm dla tonu 15 kHz. Do wytworzenia takich zaburzeń nasze głośniki muszą być wyposażone w pewien ruchomy element. Ten element nazywamy membraną. Jej wychylenie jest zależne od naszych próbek audio. Wartość binarna każdej z nich trafia kolejno w stałych odstępach czasu na przetwornik cyfrowo-analogowy. Otrzymane napięcie, po wygładzeniu „schodków" filtrem, używane jest do sterowania wychyleniem membrany.

Gdy zatem w naszych próbkach zapisany jest np. czysty sinus o częstotliwości 8 kHz, spowoduje on naprzemienne wypychanie i cofanie membrany 8 tysięcy razy w ciągu sekundy. Skutkuje to naprzemiennym kompresowaniem i rozrzedzaniem powietrza przy membranie. W efekcie otrzymamy falę akustyczną.

W mikrofonie w skrócie zachodzi proces odwrotny. Tam także znajduje się membrana, lecz to nie napięcie, ale fala akustyczna wprawią ją w drgania. Drgania te powodują proporcjonalne zmiany napięcia. Tak otrzymany sygnał przepuszczany jest przez filtr dolnoprzepustowy, którego celem w tym przypadku jest pozostawienie tylko dolnej części widma poniżej częstotliwości Nyquista. Ostatecznie sygnał przepuszczony jest przez przetwornik analogowo-cyfrowy, który umożliwia zapisanie wartości napięcia z membrany w postaci liczby binarnej. Jest ona wartością naszego pojedynczego sampla audio. Przy częstotliwości próbkowania 44.1 kHz otrzymujemy zatem nieco ponad 44 tysięcy liczb binarnych w każdej sekundzie. Najczęściej każda z nich jest 16-bitowym typem całkowitoliczbowym ze znakiem. Przekłada się to na wartości próbki z przedziału od -32768 do 32767.

4. PIERWSZA APLIKACJA W WEB AUDIO API

Zadaniem naszej pierwszej aplikacji będzie generowanie co jedną sekundę prostego tonu (czysta pojedyncza sinusoida) o losowej częstotliwości z zakresu od 1000 Hz do 3000 Hz oraz losowej głośności z zakresu od 0% do 1%. Ograniczenie maksymalnej głośności jest celowe. Zwyczajnie nie chcę mieć na sumieniu słuchu kogoś, kto otworzy ten przykład, mając słuchawki na uszach na pełnej głośności. Dodatkowo nasza aplikacja będzie inicjalizować wspomniany wcześniej węzeł mikrofonu. Wymaga on użycia strumienia audio zwracanego przez inne API przeglądarki.

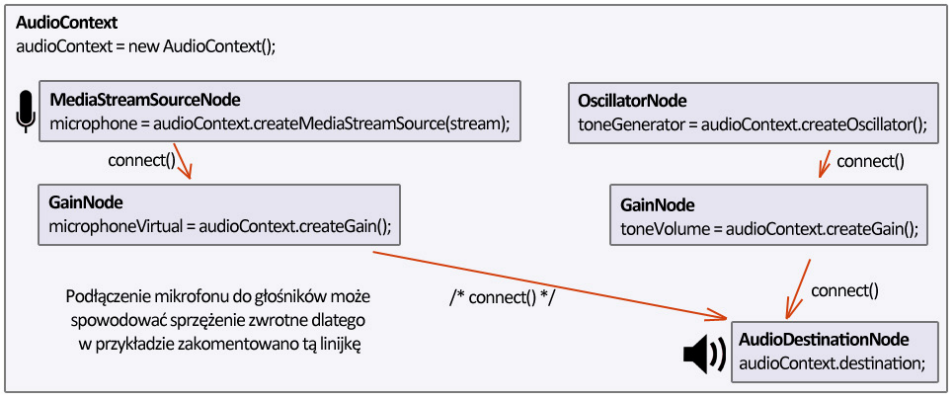
W pierwszym przykładzie nie będziemy się jeszcze zajmować przetwarzaniem napływających sampli. Jedyne, co możemy tutaj zrobić, to podłączyć węzeł mikrofonu do węzła destynacji, co da nam możliwość słuchania np. swojego głosu na głośnikach. Z uwagi jednak na możliwość wystąpienia sprzężenia zwrotnego linijka łącząca te dwa węzły została zakomentowana. Zapraszam do testowania we własnym zakresie. Na Rysunku 2 przedstawiono węzły użyte w przykładzie w postaci grafu.

W Listingu 1 przedstawiono kod pierwszej aplikacji. Dla oszczędzenia miejsca pominięto w nim obsługę błędów oraz zabiegi potrzebne do pracy na starszych przeglądarkach. Brakujące elementy umieścimy w klasie `AudioMonoIO`.

Listing 1. Generowanie prostego tonu oraz inicjalizacja węzła ze strumieniem z mikrofonu

```
function init() {
  audioContext = new AudioContext();

  // input
  microphoneVirtual = audioContext.createGain();
```



Rysunek 2. Graf węzłów pierwszej aplikacji

```
  connectMicrophoneTo(microphoneVirtual);

  // output
  toneGenerator = audioContext.createOscillator();
  toneVolume = audioContext.createGain();
  updateRandomTone();
  toneGenerator.start();
  toneGenerator.connect(toneVolume);
  setInterval(updateRandomTone, 1000);

  // przypięcie węzłów do głośników
  /*
  microphoneVirtual.connect( // uwaga na sprzężenie zwrotne
    audioContext.destination
  );
  */
  toneVolume.connect(audioContext.destination);
}

function connectMicrophoneTo(microphoneVirtual) {
  var constraints, audioConfig;

  audioConfig = {
    googEchoCancellation: false,
    googAutoGainControl: false,
    googNoiseSuppression: false,
    googHighpassFilter: false
  };
  constraints = {
    video: false,
    audio: {
      mandatory: audioConfig,
      optional: []
    }
  };
  navigator.mediaDevices.getUserMedia(constraints)
    .then(function (stream) {
      microphone = audioContext
        .createMediaStreamSource(stream);
      microphone.connect(microphoneVirtual);
    });
}

function updateRandomTone() {
  var frequency, gain, now;

  now = audioContext.currentTime;

  frequency = 1000 + Math.random() * 2000;
  toneGenerator.frequency.value = frequency;
  toneGenerator.frequency.setValueAtTime(frequency, now);

  gain = Math.random() * 0.01;
  toneVolume.gain.value = gain;
  toneVolume.gain.setValueAtTime(gain, now);
}
```

Pełny kod powyższego przykładu dostępny jest poniżej:

- https://audio-network.rypula.pl/audio-context-init
- https://audio-network.rypula.pl/audio-context-init-src

W dalszej części przyjrzyjmy się bardziej szczegółowo użytym węzłom. Wszystkie opisy zakładają, że instancja naszego kontekstu audio znajduje się w zmiennej o nazwie `audioContext`.

4.1 GainNode

Węzeł ten jest bardzo prosty. Można go stworzyć przy użyciu obiektu kontekstu audio poprzez wywołanie na nim metody `createGain`. Jego zadaniem jest zmiana poziomu głośności strumienia, który przez niego przechodzi. Sprowadza się to do pomnożenia wartości napływających sampli przez zadany współczynnik (>1 głośniej, <1 ciszej). Z dokumentacji możemy wyczytać, że poziomem tym sterujemy, przypisując odpowiednią wartość do zmiennej `value` we właściwości `gain`. Jest to prawda, lecz wiąże się z tym jeden problem. Otóż czas potrzebny na zmianę tej wartości jest zależny od implementacji Web Audio API. W niektórych przeglądarkach (np. Chrome) domyślnie włączone jest „wygładzenie zmian" tejże wartości. Oznacza to, że głośność nie zostanie natychmiast zmieniona z poziomu A do poziomu B. Zamiast tego wartość ta będzie płynnie zmieniać się w czasie, tak by finalnie zatrzymać się na wartości B. W przypadku Chrome domyślny czas wygładzania wynosi 50 ms. W przeglądarce Firefox czas ten jest domyślnie zerowy.

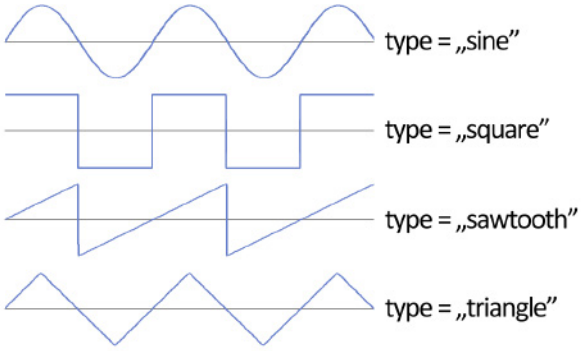
Podobne zachowanie występuje także w węźle `OscillatorNode`. Jak w przypadku zmian głośności nie jest to tak mocno zauważalne, tak przy zmianach częstotliwości stanowi to już większy problem. My jednak naprawimy oba przypadki. Rozwiązaniem jest metoda `setValueAtTime`. Przyjmuje ona dwa parametry. Pierwszy to wspomniana wcześniej wartość współczynnika, drugi to czas wykonania tej zmiany. W naszym przypadku zmianę chcemy wykonać natychmiast. W obiekcie naszego kontekstu audio dostępna jest specjalna właściwość zwracająca aktualny czas – `audioContext.currentTime`.

Podsumujmy zatem wszystko. By uniknąć problemów, należy najpierw ustawić nową wartość w „tradycyjny" sposób przy użyciu `gain.value`, a następnie w kolejnej linijce użyć metody `setValueAtTime`. Przykład z życia wzięty znajdziemy w Listingu 1 w metodzie `updateRandomTone`.

4.2 OscillatorNode

Węzeł ten umożliwia generowanie na swoim wyjściu sygnału okresowego o zadanej częstotliwości. Tworzymy go za pomocą metody `createOscillator`. Aby węzeł rozpoczął generowanie przebiegu, musimy go jeszcze „uruchomić" poprzez wywołanie metody `start`. Z dokumentacji wyczytamy, że emitowaną częstotliwość możemy zmienić, przypisując nową wartość do zmiennej `value` we właściwości `frequency`. Tutaj także ustawienie nowej wartości skutkuje nieco odmiennym zachowaniem w zależności od implementacji modułu Web Audio API w przeglądarce. Do uzyskania natychmiastowej zmiany musimy zatem także posłużyć się dodatkowo metodą `setValueAtTime`. To rozwiązanie wyeliminuje niepożądane, płynne „przepływanie" dźwięku do zadanej częstotliwości. Efekt ten mógłby wpływać negatywnie na naszą transmisję danych.

Domyślnie wybranym typem kształtu generowanej fali jest „sinus" (typ `sine`). Do dyspozycji mamy także trzy inne kształty: „prostokąt", „piłę" oraz „trójkąt" (Rysunek 3). Aby zmienić domyślny typ, wystarczy do właściwości `type` przypisać string o wartości odpowiednio „square", „sawtooth" lub „triangle".



Rysunek 3. Kształty wbudowane w OscillatorNode

Istnieje także możliwość skonfigurowania własnego kształtu za pomocą metody `setPeriodicWave` oraz klasy `PeriodicWave`. Do celów transmisji danych w zasadzie moglibyśmy wybrać typ wbudowany „sine". Wiąże się z tym jednak pewne ograniczenie. Otóż

domyślny sinus nie pozwala na zmianę fazy emitowanego przebiegu okresowego. By zatem mieć pełną kontrolę nad sygnałem, musimy skorzystać z możliwości konfiguracji własnego kształtu przebiegu. Oczywiście także wymodelujemy sinusa, jednak dodatkowo z możliwością ustawienia przesunięcia fazowego.

4.2.1 Sygnały okresowe – teoria

Konfiguracja obiektu `PeriodicWave` wymaga jednak trochę wstępu teoretycznego o budowie sygnałów okresowych. Wykroczymy nieco poza zakres minimalnej wiedzy niezbędnej do transmisji danych. Jest to jednak o tyle istotne, że nie sposób o tym nie powiedzieć. Naszym celem jest stworzenie okresowego przebiegu o zadanej częstotliwości podstawowej. Innymi słowy, gdy narysujemy wynikowe próbki na wykresie w funkcji czasu, ich kształt powinien się cyklicznie powtarzać. Takim zachowaniem cechuje się oczywiście „czysta" sinusoida.

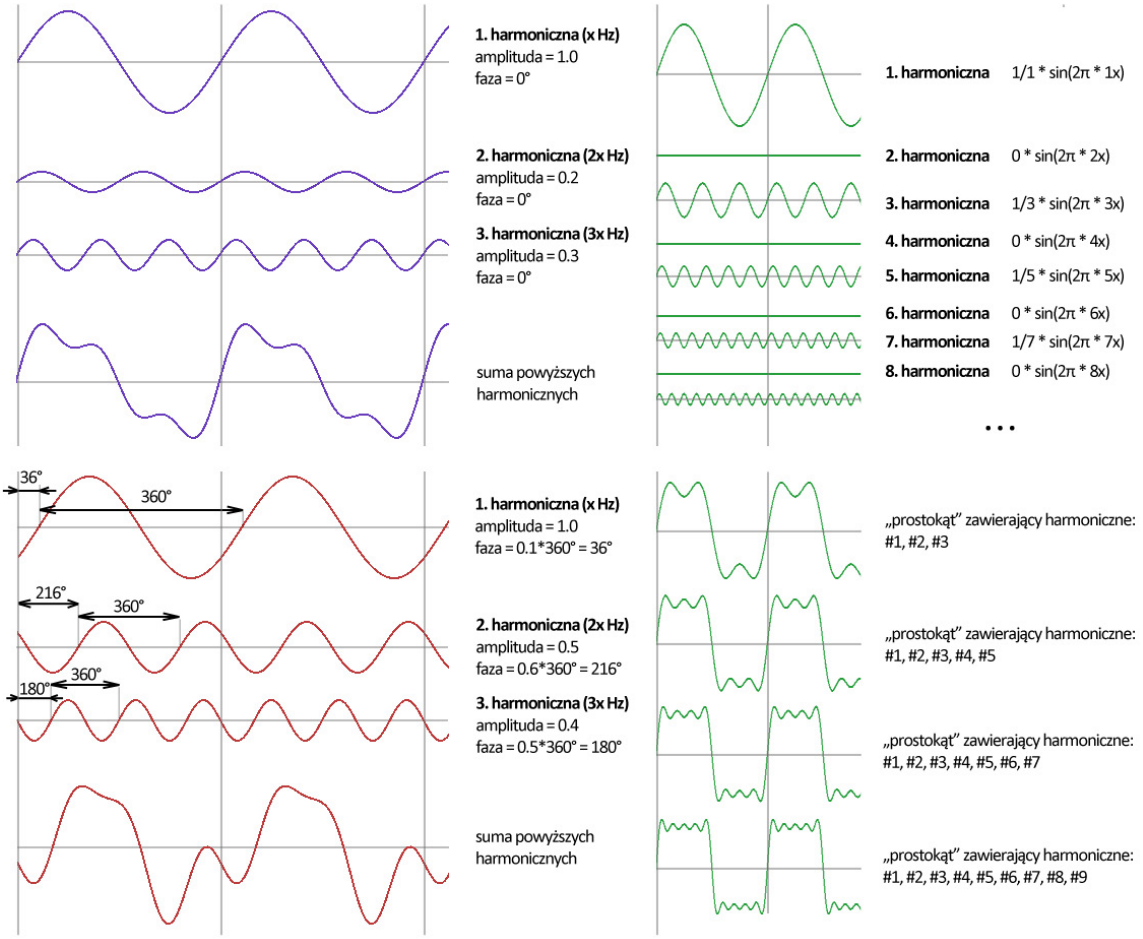
Zadajmy sobie teraz pytanie – czy możliwe jest manipulowanie wyglądem sygnału tak, by wciąż zachować okresowe powtarzanie się tego samego kształtu z częstotliwością podstawową? Otóż jest to możliwe. Wystarczy do naszej „bazowej" sinusoidy dodać drugą sinusoidę, której częstotliwość będzie dwukrotnie większa niż częstotliwość podstawowa. Dwukrotne zwiększenie częstotliwości sprawi, że w czasie trwania jednego cyklu naszej pierwszej sinusoidy „zmieszczą" się dokładnie dwa pełne cykle naszej drugiej sinusoidy. Idąc dalej, do tak zmienionego sygnału możemy dodać trzecią sinusoidę, której częstotliwość będzie 3x większa od częstotliwości podstawowej. Oznacza to, że w jednym okresie naszej pierwszej sinusoidy zmieszczą się dokładnie 3 pełne okresy trzeciej sinusoidy. W analogiczny sposób możemy dokładać kolejne sinusoidy. Oczywiście limitem częstotliwości będzie częstotliwość Nyquista.

Zauważmy, że każdy nowy cykl sinusoidy o częstotliwości podstawowej będzie także początkiem cyklów wszystkich innych sinusoid składających się na nasz finalny sygnał. Kształt następnego cyklu będzie zatem taki sam. Sinusoidy spełniające te warunki nazywamy „składowymi harmonicznymi", a pierwszą z nich „składową podstawową" lub „tonem podstawowym". Jego częstotliwość jest równa naszej częstotliwości podstawowej. Sterując amplitudami harmonicznych oraz ich przesunięciem fazowym, możemy modelować kształt naszego przebiegu. Na Rysunku 4 przedstawiono kilka przykładów pokazujących, jak zmienia się przebieg w zależności od parametrów harmonicznych. Brak harmonicznej oznacza, że jej amplituda jest równa zeru.

Jak widzimy, za pomocą odpowiedniej liczby harmonicznych, ich amplitud oraz przesunięć fazowych możemy wygenerować w zasadzie dowolny kształt, włączając w to typy wbudowane w `OscillatorNode`, takie jak „prostokąt", „piła" czy „trójkąt".

4.2.2 Sygnały okresowe w praktyce, czyli klasa PeriodicWave

Do konfiguracji fali okresowej należy posłużyć się obiektem klasy `PeriodicWave`. Tworzymy go, wywołując metodę `createPeriodicWave` na obiekcie naszego kontekstu audio. Przyjmuje ona dwa parametry będące tablicami liczb zmiennoprzecinkowych. Tablice te powinny mieć takie same długości. Każda para liczb o takich samych indeksach w tablicach opisuje jedną składową harmoniczną naszego przebiegu. Do opisu harmonicznych



Rysunek 4. Modelowanie kształtu sygnału okresowego za pomocą harmonicznych

klasa `PeriodicWave` używa liczb zespolonych. Pierwszą tablicę traktujemy jako części rzeczywiste, natomiast drugą jako części urojone liczb zespolonych. Poprzez odpowiednie sterowanie wartościami w tablicach możemy wygenerować w sygnale wynikowym sinusoidy składowe o takich amplitudach i przesunięciach fazowych, jakie sobie życzymy.

Na tym etapie możemy zauważyć podobieństwo do tego, o czym mówiliśmy przy okazji omawiania Dyskretnej Transformaty Fouriera w poprzedniej części tego artykułu. Wynikiem jego pracy były liczby zespolone mówiące o amplitudzie oraz przesunięciu fazowym badanych fal. Czy zatem `OscillatorNode` realizuje w pewnym sensie proces odwrotny? Oczywiście tak! Klasa `PeriodicWave` jest zdolna do transformacji opisu harmonicznych z dziedziny częstotliwości na dziedzinę czasu.

Bardzo ważny jest fakt, że para liczb znajdujących się w tablicach pod indeksem 0 nie opisuje pierwszej harmonicznej, lecz częstotliwość 0 Hz. Cokolwiek byśmy tam jednak nie wpisali, przeglądarka i tak wyzeruje tę wartość. Dla dociekliwych – moduł tej liczby zespolonej to tak zwany „DC offset", czyli nic innego jak wartość średnia wszystkich próbek. Zmieniając tę wartość, moglibyśmy przesuwać nasz przebieg w górę lub w dół na wykresie.

Dlaczego zatem w implementacjach Web Audio API ta wartość jest zerowana? Takie przesunięcie przebiegu na wykresie nie ma po prostu większego sensu, gdyż nie dałoby żadnego zauważalnego efektu akustycznego. Membrana głośnika drgałaby bowiem tak samo z tą różnicą, że średnio byłaby nieco głębiej lub nieco płycej w głośniku. To jednak zredukowałoby zakres, w jakim znajdowałby się nasz sygnał. Zabawa zaczyna się zatem od indeksu 1, który reprezentuje naszą częstotliwość podstawową (pierwsza harmoniczna). Każdy następny indeks będzie reprezentował kolejną harmoniczną.

Zauważmy, że klasa `PeriodicWave` nie posiada żadnej informacji o wartości częstotliwości podstawowej. Podajemy jedynie współczynniki szeregu harmonicznego. To `OscillatorNode` podczas swojej pracy generuje z podanych współczynników ostateczny strumień próbek. Częstotliwość jego pierwszej harmonicznej odpowiada ustawionej wcześniej wartości właściwości `frequency`.

W Listingu 2 pokazano, w jaki sposób wygenerować prosty kształt za pomocą klas `OscillatorNode` oraz `PeriodicWave`. Dla prostoty wybrano kształt „prostokąt", gdyż do opisu jego harmonicznych nie potrzebujemy żadnych przesunięć fazowych. Wystarczą w zupełności odpowiednio dobrane wartości amplitud.

Listing 2. Generowanie przebiegu okresowego o kształcie „prostokąt"

```
oscillatorNode = audioContext.createOscillator();
baseFrequency = 1000;
amplitudeList = [
  0,   // [0] 0 Hz (DC offset) ta wartość i tak jest zerowana
  1/1, // [1] harmoniczna #1 - {baseFreq} x 1 ton podstawowy
  0,   // [2] harmoniczna #2 - {baseFreq} x 2
  1/3, // [3] harmoniczna #3 - {baseFreq} x 3
  0,   // [4] harmoniczna #4 - {baseFreq} x 4
  1/5  // [5] harmoniczna #5 - {baseFreq} x 5
  // dla parzystych numerów amplituda wynosi zero,
  // dla nieparzystych amplituda jest równa 1/{nrHarm.}
];
// kształt prostokąt nie wymaga przesunięć
// harmonicznych w fazie - część rzeczywista
// to same zera
real = new Float32Array(amplitudeList.length);
// część urojona to dane z tablicy amplitud
imag = new Float32Array(amplitudeList);

periodicWave = audioContext.createPeriodicWave(real, imag);
oscillatorNode.setPeriodicWave(periodicWave);
oscillatorNode.frequency.value = baseFrequency;
oscillatorNode.frequency.setValueAtTime(
  baseFrequency, audioContext.currentTime
);
oscillatorNode.start();
```

Pełny kod oraz źródła znajdziemy poniżej:

» https://audio-network.rypula.pl/square-wave
» https://audio-network.rypula.pl/square-wave-src

Oczywiście nasz przebieg będzie tylko przybliżeniem idealnego przebiegu o kształcie „prostokąt". Do jego budowy użyliśmy tylko 5 harmonicznych, dlatego nasz prostokątny kształt będzie w rzeczywistości pofalowany (Rysunek 4). Jego wygląd możemy poprawić poprzez dodanie kolejnych harmonicznych o odpowiednich amplitudach.

Łatwo zauważyć, że nasze wartości amplitud podaliśmy tylko w tablicy reprezentującej części urojone. O tym, dlaczego tak zrobiliśmy, powiemy nieco dalej.

Warte wspomnienia jest to, że wyjście z oscylatora jest normalizowane. Oznacza to, że wartość żadnej z próbek nie powinna przekroczyć zakresu od -1 do 1. Definiując tablice części rzeczywistej i urojonej, zwróćmy zatem uwagę nie na faktyczne wartości długości wektorów harmonicznych (wartości bezwzględne liczb zespolonych), tylko na proporcje między tymi długościami.

Skoro zatem finalny strumień audio jest normalizowany, tak by wypełnić całość zakresu od -1 do 1, to jak go „przyciszyć"? Odpowiedzią jest `GainNode`. Wystarczy wyjście z oscylatora podpiąć bezpośrednio na jego wejście i tam sterować finalną głośnością. Takie rozwiązanie zastosowano w przykładzie z Listingu 1 oraz Listingu 3.

4.2.3 Klasa PeriodicWave – dodajemy przesunięcie fazowe

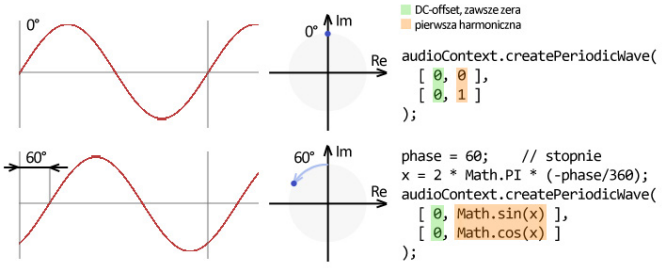
Na koniec sprawdźmy, w jaki sposób możemy sterować przesunięciem fazowym. W skrypcie z Listingu 2 wartości współczynników harmonicznych wpisaliśmy do tablicy reprezentującej części urojone oraz były one zawsze dodatnie. Stosując analogię do wektorów 2D, możemy powiedzieć, że wszystkie z nich były zawsze skierowane w górę na godzinę dwunastą. Dla obiektu klasy `PeriodicWave` jest to równoznaczne z brakiem przesunięcia fazy. W przypadku kształtu „prostokąt" było to wystarczające.

Zanim przejdziemy dalej, warto zaznaczyć, że będziemy tu rozważać dwa przypadki przesunięcia fazowego. Pierwszy to „lokalne" przesunięcie fazy każdej ze składowych z osobna. W połączeniu z odpowiednią wartością amplitudy daje to możliwość modelowania kształtu przebiegu. Drugi to „globalne" przesunięcie w fazie całego naszego kształtu. Będziemy zatem szukać sposobu na taką zmianę przesunięć fazowych wszystkich harmonicznych, by w efekcie nasz gotowy kształt nie „rozjechał" się.

Warto teraz przypomnieć sobie diagram konstelacji z poprzedniej części artykułu. Punkt na diagramie reprezentował udział konkretnej częstotliwości w badanym sygnale. Odległość tego punktu od środka układu współrzędnych mówiła o amplitudzie powiązanej z nim fali, natomiast kąt pomiędzy osią Y a punktem o przesunięciu fazowym. Warto na tym etapie zaznaczyć jedną ważną różnicę. W przypadku diagramu konstelacji przyjęliśmy obrót punktu zgodnie z ruchem wskazówek zegara, aby zwizualizować przesuwanie wykresu w prawą stronę. Jest to odwrotna konwencja do tej, jaka obowiązuje w klasie `PeriodicWave`. Warto o tym pamiętać, by nie pogubić się w obrotach.

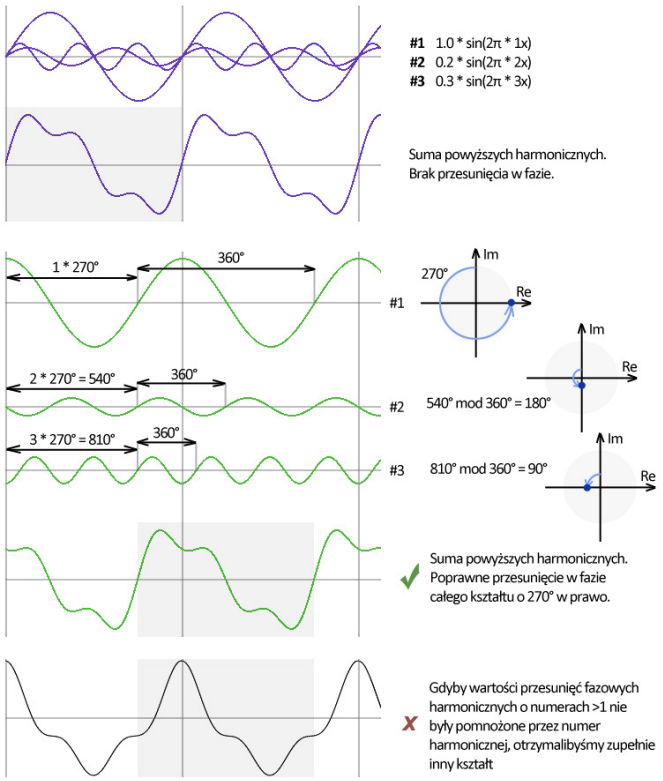
Co zatem zrobić, aby przesunąć harmoniczną w fazie? Musimy zwyczajnie obrócić powiązany z nią wektor o pewien kąt, nie zmieniając jego długości. W którą stronę? Jak już ustaliliśmy – przeciwnie do ruchu wskazówek zegara. W rezultacie nasza sinusoida zostanie

przesunięta w prawo. Na Rysunku 5 przedstawiono najprostszy przypadek, w którym nasz przebieg posiada tylko pierwszą harmoniczną. Oczywiście postępując analogicznie, możemy przesuwać w fazie harmoniczną o dowolnym numerze.



Rysunek 5. Przesuwanie składowej harmonicznej w fazie

Na tym etapie powinniśmy już wiedzieć, jak przy użyciu klasy `PeriodicWave` wymodelować w zasadzie dowolny kształt. Sprawdźmy więc teraz, w jaki sposób nasz kształt przesuwać w fazie jako całość. W przypadku pierwszej harmonicznej sprawa jest prosta – wartość kąta obrotu jest równa wymaganemu przesunięciu fazowemu. Na przykład obrót wektora pierwszej harmonicznej o 90 stopni da nam w rezultacie przesunięcie o 1/4 długości fali o częstotliwości podstawowej w prawo. Obrót o 180 stopni to przesunięcie o 1/2 długości fali. W przypadku kolejnych harmonicznych wydawać by się mogło, że należy także obracać o ten sam kąt. Okazuje się jednak, że to spowodowałoby „rozjechanie" się kształtu naszego przebiegu (Rysunek 6).



Rysunek 6. Przesunięcie w fazie całego kształtu zbudowanego z wielu harmonicznych

Jak widzimy, aby zachować kształt wykresu podczas przesunięcia, wektor każdej następnej harmonicznej o numerze większym od 1 musi „dogonić" obrót harmonicznej o numerze 1. Dzieje się tak dlatego, że długość fali każdej kolejnej harmonicznej stopniowo zmniejsza się. Aby to skompensować, należy pomnożyć kąt obrotu przez numer harmonicznej. W efekcie każda następna harmoniczna dogoni przesunięcie pierwszej z nich. To sprawi, że wszystkie harmoniczne zaczną swój nowy cykl w tym samym miejscu, a nasz kształt w każdym następnym okresie częstotliwości podstawowej pozostanie taki sam. Oczywiście, gdy harmoniczne naszego kształtu miały już pewne „lokalne" przesunięcia w fazie decydujące o jego wyglądzie, należy je także dodać.

W Listingu 3 przedstawiono kod generujący również przebieg o kształcie „prostokąt", jednak tym razem z możliwością zmiany zarówno „globalnego" przesunięcia fazowego, jak i poszczególnych harmonicznych z osobna. Dodatkowo użyty został węzeł `GainNode` umożliwiający zmianę głośności strumienia wytworzonego przez `OscillatorNode`.

Listing 3. Generowanie przebiegu okresowego o kształcie „prostokąt" – tym razem z możliwością przesunięcia fazowego i zmiany głośności

```
volume = 0.5;  // 50% głośności
phase = 0.75;  // przesunięcie całości w prawo o 270 stopni
hAmplitude = [ // DC offset został pominięty
  1, 0, 1/3, 0, 1/5
];
hPhase = [0, 0, 0, 0, 0]; // kształt prostokąt nie wymaga
                          // przesunięcia faz harmonicznych
real = new Float32Array(1 + hAmplitude.length);
imag = new Float32Array(1 + hAmplitude.length);
globalP = TWO_PI * (-phase); // 'globalne' przesunięcie fazy
real[0] = 0;   // DC-offset
imag[0] = 0;   // DC-offset
for (i = 0; i < hAmplitude.length; i++) {
  harmonicNr = 1 + i;
  localP = TWO_PI * (-hPhase[i]); // 'lokalne' przesunięcie
  real[harmonicNr] =
    hAmplitude[i] * Math.sin(globalP * harmonicNr + localP);
  imag[harmonicNr] =
    hAmplitude[i] * Math.cos(globalP * harmonicNr + localP);
}
// ...
oscillatorNode.connect(gainNode);
gainNode.connect(audioContext.destination);
// ...
periodicWave = audioContext.createPeriodicWave(real, imag);
oscillatorNode.setPeriodicWave(periodicWave);
// ...
gainNode.gain.value = volume;
gainNode.gain.setValueAtTime(
  volume,
  audioContext.currentTime
);
```

Pełny kod oraz źródła znajdziemy poniżej:

» https://audio-network.rypula.pl/square-wave-phase
» https://audio-network.rypula.pl/square-wave-phase-src

Zastanówmy się teraz, czy jesteśmy w stanie sami zaimplementować funkcjonalność klas `OscillatorNode` oraz `PeriodicWave`. Dokładniej chodzi tu o możliwość generowania przebiegów okresowych składających się z wielu harmonicznych. Przypomnijmy: w poprzedniej części artykułu do generowania przebiegów okresowych używaliśmy funkcji `generateSine`. Zwracała ona kolejne wartości próbek sinusoidy o zadanej częstotliwości (podanej w `samplePerPeriod`), fazie oraz amplitudzie. Bardziej złożone przebiegi generowaliśmy poprzez sumowanie wartości próbek otrzymanych z fal składowych. Gdy zatem odpowiednio dobierzemy częstotliwości fal składowych, po zsumowaniu jesteśmy w stanie osiągnąć dokładnie taki sam efekt. Mamy zatem dwie drogi do osiągnięcia tego samego celu.

4.3 MediaStreamSourceNode

Po przeczytaniu opisu klasy `OscillatorNode` i `PeriodicWave` wiemy już, jak emitować dźwięk. Teraz skupimy się na tym, jak ten dźwięk wychwycić. Z Rysunku 2 możemy odczytać, że część grafu odpowiadająca za przechwytywanie danych z mikrofonu składa się z dwóch węzłów. Pierwszy to `MediaStreamSourceNode`, a drugi to `GainNode`. O tym, jaka jest rola drugiego, powiemy za chwilę. Pierwszy natomiast możemy stworzyć, wywołując metodę `createMediaStreamSource` z obiektu naszego kontekstu audio. To jednak nie koniec, gdyż metoda ta wymaga parametru typu `MediaStream`. By stworzyć obiekt o takim typie, musimy posłużyć się innym API przeglądarki, a dokładniej metodą `getUserMedia` z obiektu `navigator.mediaDevices`. Jak widać, faktyczny strumień danych z mikrofonu nie pochodzi z Web Audio API. `MediaStreamSourceNode` jedynie go opakowuje, tak by można go było podłączać do innych węzłów.

Metoda `getUserMedia` przyjmuje parametr będący „listą wymogów" co do strumienia, jaki chcemy otrzymać. Lista ta to obiekt z dwoma polami – `audio` i `video`. W naszym przypadku interesuje nas tylko strumień audio. Najprostsza konfiguracja to po prostu przypisanie do pola `audio` wartości `true`, a do `video` wartości `false`. Pominięcie pola `video` jest równoznaczne z podaniem wartości `false`.

Okazuje się, że sama wartość `true` nie wystarczy. Przeglądarki domyślnie starają się „poprawić" dane napływające z mikrofonu poprzez zastosowanie różnego rodzaju filtrów. Chodzi tutaj o automatyczne dostosowywanie głośności czy też usunięcie zakłóceń, takich jak biały szum, echo itp.

Aby zmienić domyślną konfigurację, należy w polu `audio` umieścić obiekt z dwoma polami – `optional` i `mandatory`. W każdym z nim możemy umieścić kolejny obiekt z listą ustawień oraz flag `true` lub `false`. Tutaj niestety ciężko dokładnie powiedzieć, jaka jest lista wszystkich opcji i czy powinniśmy umieścić ją w polu `optional` czy `mandatory`. Na stan obecny można odnieść wrażenie, że to API ciągle się rozwija i finalny kształt nie został jeszcze ustalony. Przeszukując Internet, najczęściej jednak można natrafić na strukturę z Listingu 1. Oprócz obiektu konfiguracyjnego warto sprawdzić ustawienia mikrofonu w systemie. Czasem możemy tam znaleźć kilka opcji, które warto wyłączyć (np. *Automatic Gain Controll* – AGC). Generalnie im bardziej „nietknięte" będą nasze sample, tym lepiej.

O tym, jak ważne jest wyłączenie wszystkich „poprawiaczy" dźwięku, przekonałem się osobiście, próbując przechwycić ton o stałej częstotliwości na jednym z laptopów. Pozostawienie domyślnej konfiguracji spowodowało, że czysta sinusoida nadawana przez inne urządzenie zostanie całkowicie wytłumiona na urządzeniu odbierającym po kilku milisekundach. Może jest to niezły bajer w przypadku nagrywania głosu, jednak w przypadku transmisji danych, w której ta sinusoida byłaby nośną dla danych, jest to, delikatnie mówiąc, niewskazane.

Ok, wiemy, jak wygląda konfiguracja, ale ciągle nie odpowiedzieliśmy na pytanie, dlaczego zdecydowaliśmy się na dwa węzły. Otóż metoda `getUserMedia` zwraca nam obiekt typu `Promise`. Oznacza to, że wynik działania tej metody wcale nie musi być natychmiastowy. Z punktu widzenia naszej aplikacji możemy jednak zwyczajnie chcieć kontynuować konfigurację dalszych węzłów i połączeń między nimi. Dlatego właśnie dodany został drugi węzeł `GainNode`, który w zasadzie pełni tu rolę pośrednika. Możemy go zatem podłączyć do dalszej części grafu i traktować tak, jakby był on czymś w rodzaju wirtualnego mikrofonu.

Dlaczego w Web Audio API zastosowano rozwiązanie z promisem? Otóż dlatego, że dostęp do mikrofonu czy kamery wymaga jawnej zgody użytkownika. Po wywołaniu metody `getUserMedia` przeglądarka wyświetla odpowiedni komunikat z pytaniem. Jest to oczywiście podyktowane względami bezpieczeństwa, gdyż użytkownik może zwyczajnie nie chcieć udostępniać tego, co się dzieje w okolicy jego komputera. Fakt oczekiwania na akceptację lub rezygnację wspomnianego już komunikatu nie powinien jednak blokować wykonywania skryptu. Jak widać, użycie przez twórców Web Audio API obiektu typu `promise` jest tutaj jak najbardziej uzasadnione.

Jeżeli już mówimy o względach bezpieczeństwa, warto wspomnieć, że dostęp do strumieni zwracanych przez metodę `getUserMedia` nie jest już teraz tak łatwy jak to miało miejsce przed końcem 2015 roku. Wtedy właśnie w przeglądarce Chrome wprowadzono zmianę, która do poprawnego przechwytywania strumienia dźwięku lub/i obrazu wymaga dodatkowo użycia protokołu `https`. Nieszyfrowany protokół `http` zadziała tylko w przypadku korzystania z lokalnego serwera (127.0.0.1, localhost). Jest bardzo prawdopodobne, że niebawem wszystkie przeglądarki będą zachowywać się tak samo jak Chrome. W przeszłości nie było takich restrykcji. Nie zdziwmy się więc, gdy natrafimy w Internecie na „martwe" przykłady użycia Web Audio API. Nie będą one działać w większości przypadków tylko dlatego, że użyto protokołu `http` zamiast `https`.

Wróćmy do naszego obiektu `promise` zwracanego przez metodę `getUserMedia`. Po jego pozytywnym rozwiązaniu otrzymujemy strumień audio (`WebRTC MediaStream`). To właśnie jego musimy podać jako parametr metody `createMediaStreamSource`. W skrypcie z Listingu 1 wszystkie opisane wyżej kroki umieszczono w metodzie `connectMicrophoneTo`. Przyjmuje ona jeden parametr będący węzłem audio. Właśnie do tego węzła dołączany jest asynchronicznie strumień audio z mikrofonu. W naszym przykładzie tym parametrem jest obiekt typu `GainNode` przypisany do zmiennej `microphoneVirtual`.

Na koniec informacja, która może oszczędzić czas i nerwy. Nigdy nie deklarujemy zmiennej przetrzymującej węzeł stworzony przez `createMediaStreamSource` jako lokalnej w funkcji będącej handlerem `then`. W zależności od implementacji silnika JS w przeglądarce może ona zostać usunięta przez *Garbage Collector* po kilku sekundach od inicjalizacji. Gdy tak się stanie, nasz obiekt typu `MediaStreamSourceNode` po prostu zniknie i pozostajemy z głuchą ciszą. Nie pomaga tutaj nawet fakt, że podłączamy go do zmiennej `microphoneVirtual` poprzez metodę `connect`. W teorii GC nie powinien go wtedy usuwać, jednak w praktyce okazuje się, że nie zawsze tak jest. Rozwiązaniem jest po prostu zadeklarowanie tej zmiennej w takim zasięgu, który nie zostanie usunięty po opuszczeniu funkcji umieszczonej w `then`.

5. WĘZŁY UMOŻLIWIAJĄCE PRZETWARZANIE DANYCH Z MIKROFONU

W tej sekcji dowiemy się, w jaki sposób możemy wykorzystać strumień audio z naszego mikrofonu. W przykładzie z Listingu 1 przygotowaliśmy już nieco potrzebne klocki do pracy. Użyliśmy do tego celu węzła `GainNode` (zmienna `microphoneVirtual`) oraz asynchronicznie przypinanego do niego węzła `MediaStreamSourceNode` (zmienna `microphone`). Takie połączenie umożliwiło już na etapie inicjalizacji aplikacji podpięcie zmiennej `microphoneVirtual` do dalszej części naszego grafu. Nie było więc konieczne czekanie na prawdziwe dane z mikrofonu.

Co może pełnić rolę dalszej części grafu? Może być to np. węzeł `destination`. Do zastosowań transmisji danych podłączanie mikrofonu do głośników nie ma jednak większego sensu. Jedyne, co potrzebujemy, to przetwarzenie napływających próbek audio w taki sposób, by wyciągnąć z nich strumień symboli nadawanych przez inne urządzenie. Aby tego dokonać, mamy do wyboru dwie drogi.

Pierwszą jest użycie węzła `AnalyserNode`. Jest to rozwiązanie, które wcześniej nazwaliśmy pewną formą oszukiwania, gdyż wprowadza „magiczne pudełko", które wykonuje za nas operacje DSP. Jak już jednak wspomnieliśmy, jest to jedyne rozwiązanie, jeśli chodzi o wolniejsze urządzenia mobilne, takie jak np. smartfony. Węzeł ten używa niezwykle wydajnego algorytmu FFT (ang. *Fast Fourier Transform*), więc jest w tym przypadku idealny.

Druga droga to praca na surowych samplach audio, które będziemy musieli sami przetworzyć. Oznacza to, że będziemy musieli zaprzęgnąć do pracy algorytm DTF opisany w pierwszej części tego artykułu. Jego zasadniczą wadą jest bardzo powolne działanie, jednak jest on relatywnie prosty do zrozumienia. Do pracy wymaga na wejściu tablicy z wartościami kolejnych próbek. Nasz strumień z mikrofonu musi zatem zostać w jakiś sposób zamieniony na tablice JavaScriptowe. Z pomocą przychodzi `ScriptProcessorNode`. Jest to węzeł, który paczkuje napływający strumień próbek do postaci tablic o równych rozmiarach. Aby uzyskać dostęp do tych danych, wystarczy zarejestrować w węźle handler do eventu `onaudioprocess`. Zdarzenie to będzie cyklicznie wywoływane w stałych odstępach czasu.

W naszej klasie `AudioMonoIO` zaimplementujemy możliwość skorzystania z obydwu dróg. Prześledźmy zatem szczegółowo obydwa węzły.

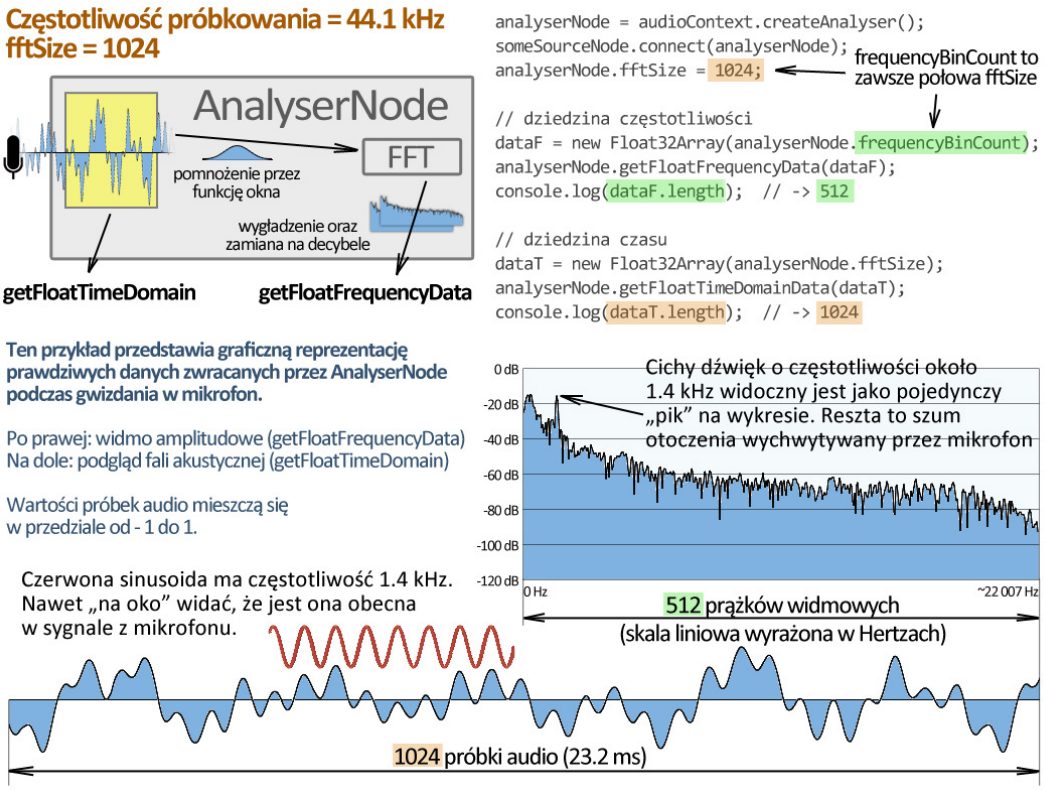
5.1 AnalyserNode

Węzeł ten tworzymy jak każdy inny węzeł za pomocą odpowiedniej metody kontekstu audio. W tym przypadku ta metoda to `createAnalyser`. W skrócie `AnalyserNode` umożliwia podgląd w czasie rzeczywistym danych zarówno z dziedziny czasu, jak i z dziedziny częstotliwości. Dostęp do tych informacji możliwy jest za pomocą dwóch metod: `getFloatTimeDomainData` oraz `getFloatFrequencyData`. Obie z nich wymagają jednego parametru, który jest tablicą typu `Float32Array` o odpowiednim rozmiarze. Gdy wywołamy obydwie metody zaraz po sobie, otrzymane wyniki możemy traktować jako parę. W środku naszego węzła tablica próbek dziedziny czasu jest po prostu użyta jako wejście algorytmu obliczającego tablicę dziedziny częstotliwości.

Tak naprawdę istnieją jeszcze dwa odpowiedniki wspomnianych metod. Są nimi `getByteTimeDomainData` oraz `getByteFrequencyData`. Pod względem wydajnościowym nie ma jednak znaczenia, czy użyjemy wersji `Float` czy `Byte`. Skoro nie widać różnicy, to… wybieramy wersję `Float`, gdyż oferuje większą precyzję.

Twórcy Web Audio API skorzystali z faktu, iż tablice podane jako parametr funkcji przekazywane są przez referencję. Dane wpisywane są więc bezpośrednio do przekazanych w parametrze tablic. By nie zgubić żadnych danych, tablice te muszą mieć ustalony z góry rozmiar. Zależy on ściśle od wartości parametru `fftSize`, o którym opowiemy szczegółowo za chwilę. W przypadku metody `getFloatTimeDomainData` rozmiar ten powinien wynosić dokładnie `fftSize`, natomiast dla `getFloatFrequencyData` musi być to połowa `fftSize`. Dla uproszczenia wartość połowy `fftSize` możemy pobrać bezpośrednio z instancji węzła za pomocą właściwości `frequencyBinCount`.

Czym jest więc `fftSize`? Jest to liczba sampli audio, które zostaną użyte do wykonania na nich algorytmu Szybkiej Transformaty Fouriera (FFT). Innymi słowy jest to szerokość okna, przez które podglądany jest napływający strumień próbek audio w dziedzinie czasu. Domyślnie szerokość ta jest równa 2048. Można ją jednak zmienić, przypisując nową wartość do właściwości `fftSize` naszego obiektu węzła. Rozmiar ten zawsze musi być liczbą będącą



Rysunek 7. Interpretacja wyników pracy AnalyserNode

potęgą dwójki z zakresu od 32 do 32768. Mamy zatem do dyspozycji 11 różnych wartości: 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768. Na Rysunku 7 pokazano interpretację wyników pracy `AnalyserNode` dla `fftSize` równego 1024.

Używając elementu `<canvas>` oraz danych zwracanych przez `AnalyserNode`, możemy w łatwy sposób przedstawić graficznie falę akustyczną oraz jej widmo amplitudowe. Otrzymany wykres widma będzie jednak nieco inny niż ten, który analizowaliśmy w poprzedniej części artykułu. Różnica wynika z użytej na osi poziomej jednostki częstotliwości. W `AnalyserNode` jednostką jest Hertz, podczas gdy w naszej implementacji DTF użyliśmy dość nietypowo `samplePerPeriod`. Warto jednak zaznaczyć, że w obydwu przypadkach skala na osi poziomej jest liniowa. Użycie jednostki `samplePerPeriod` spowodowało zniekształcenie „górek" reprezentujących nasze dominujące częstotliwości. Wykres był zwyczajnie rozciągnięty z lewej strony, a ściśnięty z prawej. W przypadku `AnalyserNode` taki efekt nie występuje. Każda pojedyncza sinusoida o danej częstotliwości będzie zatem na wykresie „pikiem" o podobnej szerokości.

Kod oraz źródła przykładu używającego elementu `<canvas>` znajdziemy poniżej:

» https://audio-network.rypula.pl/analyser-node
» https://audio-network.rypula.pl/analyser-node-src

5.1.1 AnalyserNode pod lupą – algorytm FFT

W tej sekcji przyjrzymy się z grubsza działaniu FFT. Jego najbardziej znaną formą jest algorytm Cooley–Tukey pracujący na tablicach o rozmiarach będących potęgą dwójki. To właśnie dzięki niemu `AnalyserNode` jest w stanie zwracać dane z dziedziny częstotliwości niezwykle szybko. Nie będziemy jednak zagłębiać się w szczegóły jego implementacji, gdyż wykracza to poza zakres tego artykułu. FFT potraktujemy zatem jak czarne pudełko z wejściem i wyjściem, do którego wchodzą i wychodzą dane określonego typu. Warto tutaj jednak zaznaczyć ważną rzecz – `AnalyserNode` nie udostępnia żadnych metod do bezpośredniego korzystania z FFT. Wszystko dzieje się automatycznie podczas pracy węzła. O FFT mówimy tylko dlatego, by przybliżyć zasadę działania klasy `AnalyserNode`.

Zanim aktualny blok próbek audio z dziedziny czasu trafi na wejście FFT `AnalyserNode` stosuje na nim funkcję okna. Ten zabieg ma na celu wyeliminowanie efektu wycieku widma. Omawialiśmy to zagadnienie w poprzedniej części artykułu. Wewnątrz algorytmu FFT tak przygotowany blok wejściowy poddawany jest m.in. podziałowi na mniejsze bloki w myśl zasady „dziel i zwyciężaj". Finalnie na wyjściu FFT otrzymujemy blok liczb zespolonych. W każdej z nich zawarta jest informacja o określonej częstotliwości znajdującej się w badanym sygnale. Powiązane z otrzymanymi liczbami zespolonymi częstotliwości zależą jednak ściśle od wybranego na początku rozmiaru `fftSize`. Jest to nieco odmienne działanie niż w przypadku algorytmu z pierwszej części tego artykułu. W jego przypadku możliwe było wybranie dowolnej częstotliwości, na którą chcemy się „nastroić". Ważne jest jednak, że w przypadku obydwu algorytmów interpretacja otrzymanych liczb zespolonych jest taka sama.

Przypomnijmy: z tak otrzymanych liczb zespolonych możemy wyciągnąć informacje o przesunięciach fazowych oraz amplitudach fal z nimi związanych. Niestety w przypadku `AnalyserNode` w momencie wywołania metody `getFloatFrequencyData` lub `getByteFrequencyData` ta pełna informacja jest spłaszczona do postaci jedynie danych o amplitudzie. Innymi słowy to, co otrzymujemy, to jedynie wartości bezwzględne liczb zespolonych znajdujących się w „najświeższym" buforze wyników FFT. Niestety twórcy Web Audio API nie przewidzieli żadnej innej metody do otrzymania informacji o przesunięciu fazowym. Dlaczego? Niestety nie znalazłem nigdzie informacji na ten temat…

Wróćmy do rozmiarów tablic. Faktem jest, że w wyniku działania FFT otrzymamy na wyjściu tyle samo liczb zespolonych co wartości podanych na wejściu (`fftSize`). Dlaczego zatem `AnalyserNode` zwraca nam tylko połowę? Aby rozwiązać tę zagadkę, wystarczy spojrzeć, jak rozlokowane są częstotliwości powiązane z każdą liczbą zespoloną w tablicy otrzymanej na wyjściu FFT. Pod indeksem 0 znajdziemy częstotliwość 0 Hz. Część rzeczywista tej liczby zespolonej jest to tak zwany DC-offset, który jest niczym innym jak wartością średnią próbek z okna. Wspomnieliśmy o nim krótko przy okazji omawiania `OscillatorNode` i `PeriodicWave`. Jak zapewne podpowiada nam intuicja, idąc dalej po indeksach tablicy, każda następna liczba zespolona będzie powiązana z coraz to wyższą częstotliwością (indeksy 1, 2, 3, …). Jest to prawda. Okazuje się jednak, że najwyższa użyteczna wartość częstotliwości przypadnie nie na samym końcu tablicy (indeks `fftSize - 1`), lecz w jej środku (indeks `0.5 * fftSize`). Rozmiar tablicy to liczba parzysta, więc dla ścisłości powinniśmy bardziej powiedzieć: w pierwszym możliwym indeksie drugiej połowy, patrząc od lewej. Nasz „środkowy" indeks będzie powiązany z częstotliwością Nyquista.

Faktem jest, że przekroczenie częstotliwości Nyquista powoduje efekt nazwany aliasingiem. Wtedy w zapisanym sygnale pojawią się częstotliwości, których nie ma w rzeczywistości. Najprostszą analogią jest przypadek filmowania kół samochodu, który rusza z miejsca, a następnie porusza się ruchem jednostajnie przyspieszonym. Po odtworzeniu filmu zobaczymy, że w miarę upływu czasu koła kręcą się coraz szybciej. Gdy jednak częstotliwość obrotu kół przekroczy połowę częstotliwości, z jaką nasza kamera zapisuje klatki obrazu, koła samochodu zaczną obracać się w przeciwnym kierunku. Dodatkowo ich prędkość obrotu będzie stopniowo spadać (Rysunek 1). Gdy częstotliwość obrotu kół będzie równa częstotliwości zapisywania klatek obrazu, wtedy na filmie ujrzymy jadący samochód, którego koła nie obracają się wcale. Dalsze zwiększanie prędkości samochodu spowoduje cyklicznie pojawianie się na filmie tego samego efektu (ang. *wagon-wheel effect*). Okazuje się, że Dyskretna Transformata Fouriera ma analogiczne właściwości związane z częstotliwością obrotu wektora jednostkowego, który omawialiśmy w poprzedniej części artykułu.

Wróćmy do FFT. Ustaliliśmy, że każdy kolejny prążek powiązany jest z coraz to wyższą częstotliwością. Efekt aliasingu spowoduje jednak, że w drugiej połowie tak naprawdę zobaczymy widmo częstotliwości o wartościach ujemnych. Dla uproszczenia załóżmy, że rozpatrujemy wartości bezwzględne częstotliwości ujemnych powiązanych z indeksami. Przy takim założeniu każda następna liczba zespolona o indeksie większym od `0.5 * fftSize` będzie już powiązana z coraz to niższą częstotliwością. Wartości częstotliwości za połową będą zatem spadać tak samo szybko jak rosły przed połową. W pierwszej połowie częstotliwości rosną liniowo, natomiast w drugiej połowie liniowo maleją. Liczba zespolona pod indeksem `0.5 * fftSize - x` będzie zatem w pewnym sensie powiązana z taką samą częstotliwością co liczba zespolona pod indeksem `0.5 * fftSize + x`. Widzimy zatem, że mamy tu do czynienia z symetrią częstotliwości względem indeksu `0.5 * fftSize`. Przesuwając się cały czas w prawo, dotrzemy finalnie do końca tablicy. Ostatni indeks tablicy (`fftSize - 1`) będzie powiązany z taką samą częstotliwością jak indeks 1. Jak widzimy, tablica „skończyła się", zanim dotarliśmy do elementu DC-Offset, który jest pod indeksem 0.

Pamiętajmy jednak, że do tej pory omówiliśmy symetrię częstotliwości powiązanych z elementami tablicy. O samych wartościach nic jeszcze nie powiedzieliśmy. O tym, czy one także wykazują pewną formę symetrii, opowiemy nieco dalej.

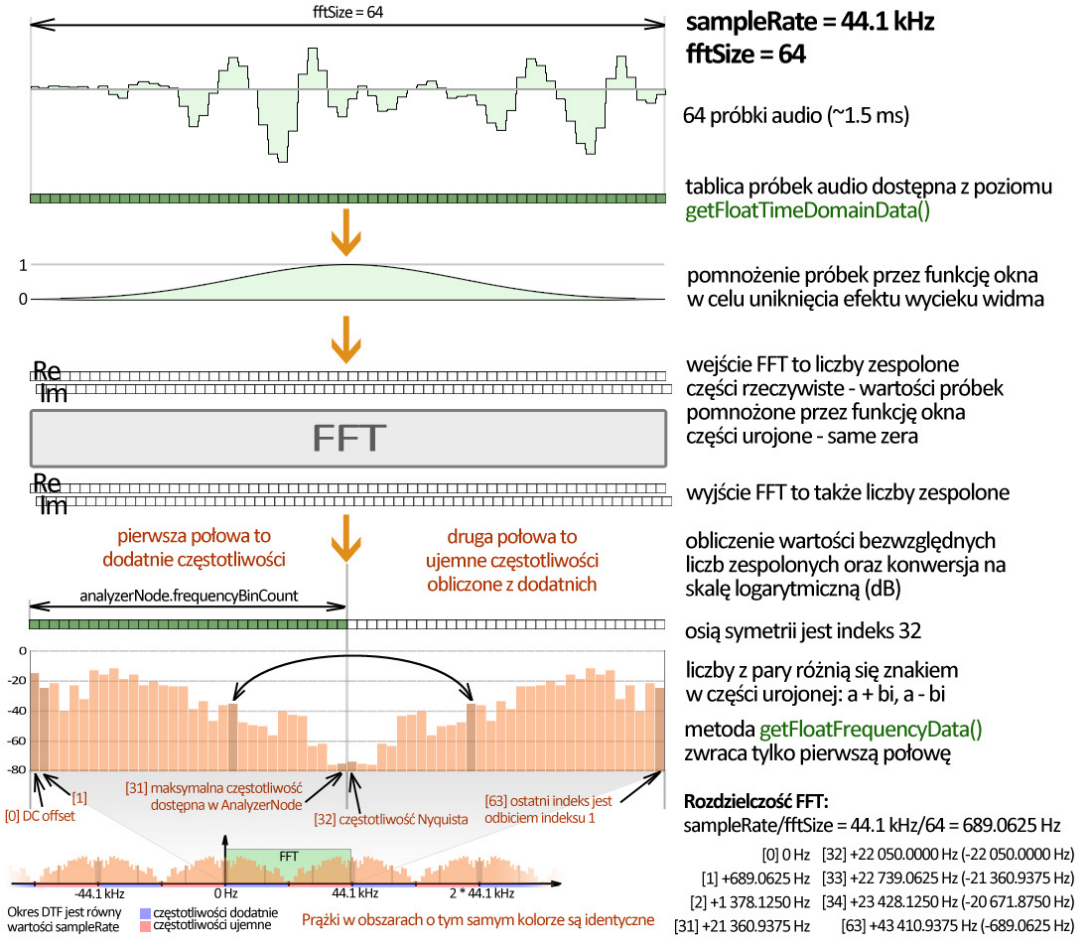
Wróćmy na chwilę do typu danych, jakie możemy podać na wejściu FFT. Ustaliliśmy, że `AnalyserNode` na wejściu otrzymuje wartości sampli audio z naszego okna. Jak na próbki audio przystało, są one liczbami rzeczywistymi. Jak się jednak okazuje, algorytm FFT na wejściu z powodzeniem przyjmuje także liczby zespolone.

Chwileczkę… liczby zespolone na wejściu w dziedzinie czasu? Przecież wartość próbki audio to zapis binarny napięcia wytwarzanego przez wychylenie membrany mikrofonu! Okazuje się jednak, że w przetwarzaniu sygnałów liczby zespolone w dziedzinie czasu nie są niczym dziwnym. Sam byłem tym dość zdziwiony, dlatego że jest to trochę sprzeczne z intuicją. Jak bowiem wyobrazić sobie urojone wychylenie membrany mikrofonu? Nie będziemy tutaj jednak zastanawiać się, jaki jest tego fizyczny sens. Zastanowimy się bardziej, jak ten „problem" obejść. Otóż rozwiązaniem jest stworzenie takiej liczby zespolonej, której część rzeczywista będzie równa wartości naszej próbki, a część urojona będzie równa zero. Problem rozwiązany! Dodatkowo właśnie tutaj zaczyna się finalna odpowiedź na pytanie, dlatego otrzymujemy tylko połowę `fftSize`. Okazuje się, że gdy części urojone wszystkich elementów wejściowych są równe zero, to po transformacie moduły otrzymanych liczb zespolonych będą symetrycznie odbite względem indeksu `0.5 * fftSize`. Dla dociekliwych – liczby zespolone z „pary" będą miały zwyczajnie przeciwny znak w części urojonej, np. `a + bi` oraz `a - bi`. Zmiana znaku nie zmienia długości wektora, więc moduły są takie same. Zaznaczmy jednak raz jeszcze: jest tak tylko wtedy, gdy liczby zespolone z wejścia FFT mają w częściach urojonych same zera.

Skoro zatem moduły liczb zespolonych drugiej połowy są jedynie odbiciem modułów pierwszej połowy, oznacza to, że z powodzeniem można którąś z nich pominąć. Tak właśnie zrobiono w `AnalyserNode`. Dane zwracane przez metodę `getFloatFrequencyData` są tak naprawdę pierwszą połową bloku zwróconego przez FFT. Do naszej dyspozycji jest zatem tablica o rozmiarze `0.5 * fftSize` od indeksu 0 (DC-Offset) do indeksu `0.5 * fftSize - 1` włącznie. Jak widzimy, dane o częstotliwości Nyquista (`indeks 0.5 * fftSize`) nie zawierają się w tym zbiorze, więc są dla nas niedostępne (Rysunek 8).

Skoro już wiemy, jaki jest sens danych zwracanych przez metodę `getFloatFrequencyData`, zastanówmy się teraz, jaki jest skok częstotliwości między dwoma sąsiadującymi indeksami. Skoro indeks `0.5 * fftSize` to częstotliwość Nyquista (czyli połowa częstotliwości próbkowania) oraz potrzeba `0.5 * fftSize` elementów, by dotrzeć tam liniowo od 0 Hz, to możemy zapisać to wzorem: `(sampleRate / 2) / (fftSize / 2)`. Po uproszczeniu otrzymujemy `sampleRate / fftSize`. Podstawiając wartości, czyli np. próbkowanie 44.1 kHz oraz rozmiar `fftSize` równy np. 2048, otrzymamy wartość około 21.53 Hz. Ten wynik mówi nam o rozdzielczości częstotliwościowej FFT.

Czy to dużo? Oczywiście to zależy. Rozważmy więc najprostszy przykład z życia wzięty, czyli gwizdanie. W przybliżeniu możemy założyć, że częstotliwość gwizdania mieści się w przedziale od 1 kHz do 2 kHz. W tym zakresie odległości między kolejnymi często-



Rysunek 8. Symetria wyjścia FFT dla danych wejściowych będących liczbami rzeczywistymi

tliwościami nut wynoszą mniej więcej od 50 do 100 Hz. Widzimy zatem, że otrzymana rozdzielczość jest wystarczająco duża, by rozróżnić pojedyncze dźwięki osoby gwiżdżącej, np. popularne „do re mi fa sol la si do". Są to kolejno zagrane białe klawisze w oktawie. Na Rysunku 9 do zapisu nazw dźwięków użyto metody znanej jako „Scientific Pitch Notation". Różni się ona od zapisu używanego w Polsce, lecz moim zdaniem ten zapis jest bardziej logiczny i krótszy. Częstotliwości dźwięków wygenerowano, bazując na powszechnie stosowanym strojeniu równomiernie temperowanym o 12 dźwiękach na oktawę. Częstotliwość dźwięku A4 to 440 Hz.

5.1.2 Podgląd w czasie rzeczywistym

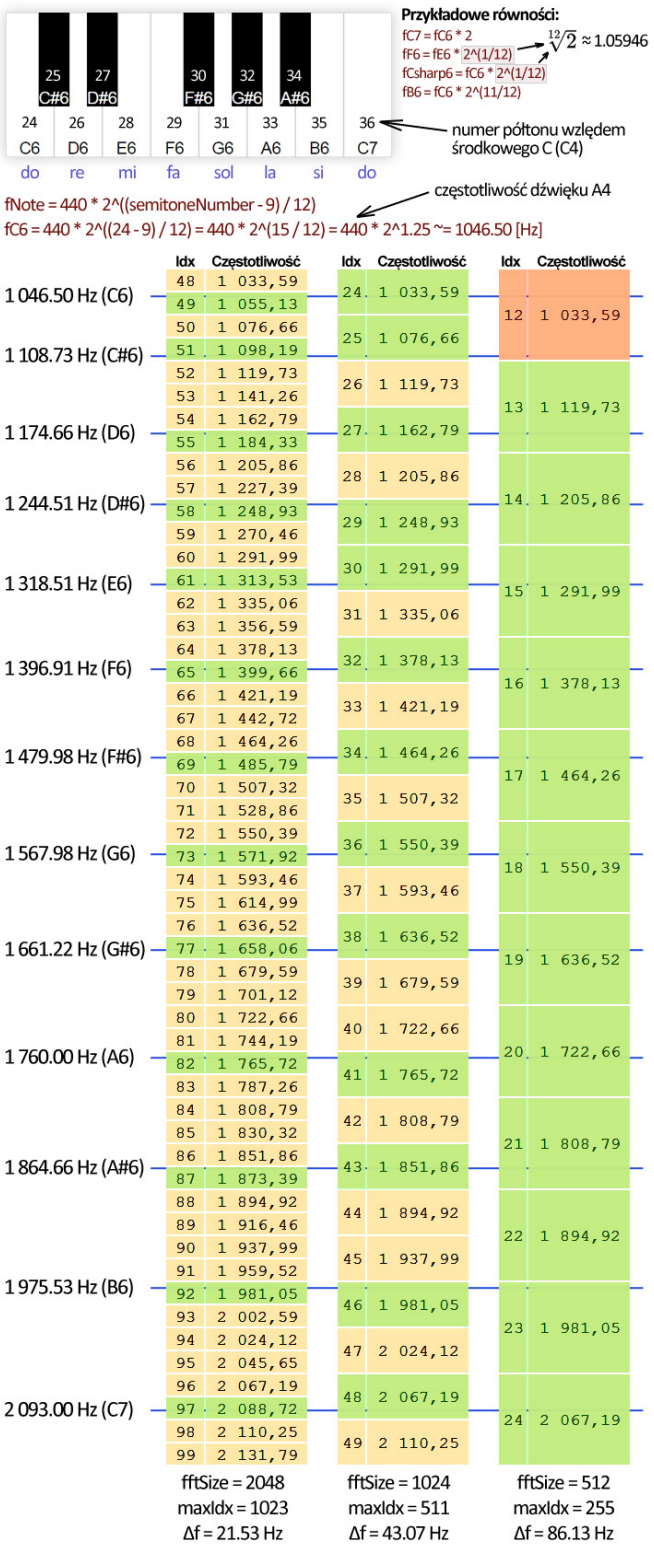
Na koniec zastanówmy, co tak naprawdę oznacza, że `AnalyserNode` umożliwia podgląd naszego sygnału w czasie rzeczywistym. Wiemy już, że algorytm FFT w nim użyty działa na bloku danych wejściowych o określonym rozmiarze. Zastanówmy się zatem, ile czasu taki blok trwa. Najmniejsza możliwa wartość `fftSize` to 32. Przy próbkowaniu 44.1 kHz taki pojedynczy blok trwa około 0.7 ms. Dla kontrastu największy możliwy blok ma długość 32768 sampli. Jego czas trwania jest już znacznie dłuższy, gdyż wynosi około 743 ms.

Jak to się przekłada na wychwytywanie sygnałów? W praktyce w dłuższych blokach krótko trwające sygnały będą wydawały się słabsze. Tym słabsze, im mniejszy procent okna będą wypełniać. Z kolei „czas reakcji" wykresu widma na pojawienie się sygnału będzie dłuższy.

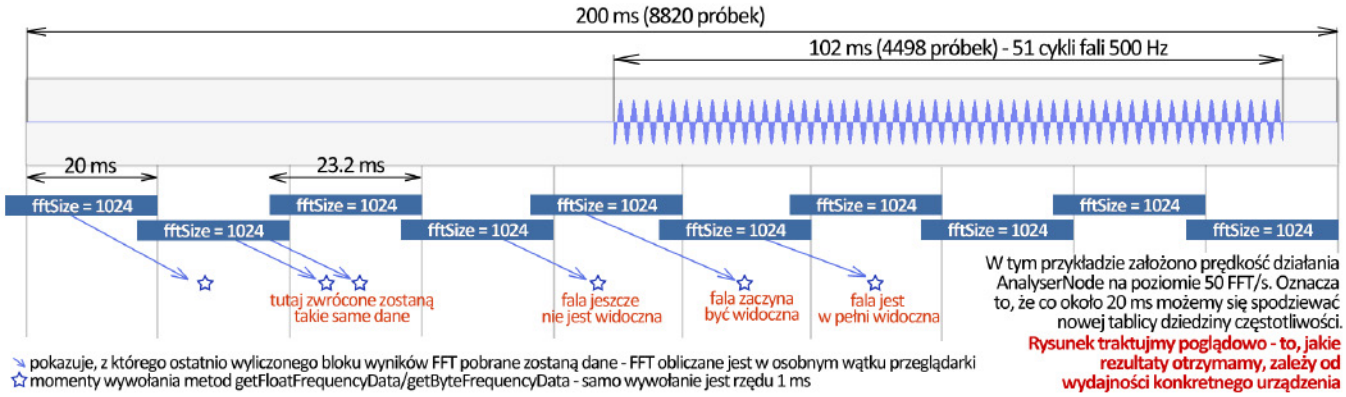
Mając na myśli rozmiar bloku, chodzi nam tu o wielkość okna, przez które patrzymy na nasz sygnał w dziedzinie czasu. Do uzyskania wrażenia płynnego podglądu w czasie rzeczywistym okno to musi się w jakiś sposób przesuwać w miarę napływu nowych sampli audio. Powstanie zatem coś w rodzaju filmu, w którym każda klatka generowana jest na podstawie aktualnej ramki z wynikiem FFT. Aby zapewnić płynne przejścia pomiędzy klatkami, w `AnalyserNode` dostępna jest opcja wygładzania dziedziny częstotliwości. Jest ona domyślnie włączona. W skrócie działa to tak, że elementy bloku danych zwracanego przez metodę `getFloatFrequencyData` wyliczane są jako średnia ważona elementów z bloku poprzedniego i aktualnego. Wygładzaniem tym możemy sterować, ustawiając wartość z przedziału od 0 do 1 do właściwości `smoothingTimeConstant` w instancji klasy `AnalyserNode` (0 oznacza brak wygładzania, 1 największe wygładzanie). Wartością domyślną jest 0.8.

Wygładzanie niestety niesie za sobą pewne konsekwencje, jeśli chodzi o transmisję danych. Przypuśćmy, że nadajemy wiadomość Morsem na jednej stałej częstotliwości. Okresy ciszy rozdzielają kolejne symbole dłuższych i krótszych impulsów. W szczególnym przypadku stosując wygładzanie, spowodujemy zlanie się wszystkich symboli w jeden. Nasze przerwy nie byłyby zwyczajnie widoczne jako wyraźne spadki amplitudy prążka widmowego powiązanego z naszą częstotliwością nośną. Na potrzeby transmisji danych generalnie lepiej jest całkowicie wyłączyć wygładzanie. Możemy tego dokonać, przypisując 0 do właściwości `smoothingTimeConstant`.

Na koniec warto by zastanowić się, jaki jest limit odświeżania podglądu w „czasie rzeczywistym". Podsumujmy – by nie „zgubić" żadnej części sygnału i uzyskać zadowalającą szybkość odświeżania, następujące po sobie okna powinny zachodzić „na zakładkę", a przesunięcia powinny być możliwie minimalne. Czy zatem możliwe jest tak częste odświeżanie danych zwracanych przez `AnalyserNode`, by kolejno występujące po sobie okna oddalone były od siebie o wartość jednej próbki? Otóż nie. W Web Audio API zdefiniowano pojęcie o nazwie „render quantum". Jest to z góry określona liczba sampli, którą jednorazowo, w sposób atomowy, przetwarza kontekst audio. Dokumentacja Web Audio API definiuje tę wartość jako równą 128. To ograniczenie skutkuje tym, że nie jest możliwe odświeżenie np. danych zwracanych przez `getFloatFrequencyData` częściej niż co 128 sampli (około 2.9 ms). Po prostu wywołania metod zawierające się w tym samym „kwancie renderowania" będą zawsze zwracały te same dane. Stanie się tak również wtedy, gdy będziemy starać się odczytywać dane szybciej niż nasz sprzęt



Rysunek 9. Rozdzielność FFT a zdolność wykrywania dźwięków podczas gwizdania



Rysunek 10. W jaki sposób AnalyserNode dostarcza podglądu w czasie rzeczywistym

jest w stanie je przetwarzać. Tutaj warto jednak zaznaczyć, że algorytm FFT działa w osobnym wątku przeglądarki i nie powoduje „przycinania" strony (Rysunek 10).

5.1.3 Wydajność FFT w porównaniu z metodą intuicyjną DTF

Na końcu przeanalizujmy wydajność użytego w `AnalyserNode` algorytmu FFT. Jego złożoność wynosi `O(N * log2(N))`, podczas gdy złożoność metody opisanej w poprzedniej części artykułu to `O(N^2)`. Taki zapis może na początku zbyt wiele nie mówić, lecz gdy podstawimy do wzorów liczby szybko, zobaczymy, jak duża jest to różnica. Dla `fftSize` o wartości 4096 będzie to odpowiednio 49152 (4096 * log2(4096)) oraz 16777216 (4096 * 4096). Algorytm FFT będzie zatem szybszy około 341x. Idąc dalej, dla `fftSize` o wartości 32768 będzie to już odpowiednio 491520 (32768 * log2(32768)) oraz 1073741824 (32768 * 32768). Dla tego przypadku FFT będzie już szybszy około 2185x. Im większa wartość N, tym większe stają się różnice wydajności na korzyść FFT. Załóżmy, że mamy maszynę zdolną wykonać milion rozważanych operacji w ciągu sekundy. W pierwszym przykładzie (N = 4096) obliczenia będą trwać około 50 milisekund dla FFT oraz prawie 17 sekund dla metody standardowej. W drugim przykładzie (N = 32768) będzie to już około 500 ms dla FFT oraz prawie 18 minut dla metody standardowej. Widać wyraźnie, że jedyną opcją dla generowania widma sygnału w czasie rzeczywistym jest FFT. Nie bez powodu ten algorytm często nazywa się jednym z najważniejszych algorytmów w historii. To właśnie jego szybkość sprawiła, że dzisiaj możemy się cieszyć technologiami takimi jak Wi-Fi, LTE, DVB-T. Nawet leciwy już format MP3 używa FFT. Oczywiście praktycznych zastosowań tej metody jest o wiele więcej.

5.1.4 AnalyserNode – podsumowanie

W tej sekcji przeanalizowaliśmy dość szczegółowo `AnalyserNode`. Po krótkiej analizie wydajnościowej można stwierdzić, że bardzo trudno jest zrezygnować z rozwiązań szybkich na rzecz tylko rozwiązań prostych i intuicyjnych zaimplementowanych przez nas samodzielnie. Nie chodzi już tutaj tylko o wolniejsze urządzenia mobilne. Przy tworzeniu aplikacji transmitującej dane dość często potrzebne będzie użycie widma sygnału zwyczajnie po to, by zweryfikować okolice częstotliwości, na którą staramy się nastroić. Dlatego też w naszej klasie `AudioMonoIO` udostępnimy także potencjał, jaki oferuje FFT. Pamiętajmy jednak, że gdy tylko interesuje nasz pojedynczy prążek, wciąż możemy skorzystać z „naszego" intuicyjnego algorytmu.

Ważna uwaga co do samego algorytmu FFT. Otóż nie zwraca on wyników będących pewnego rodzaju przybliżeniem Dyskretnej Transformaty Fouriera. Jest to algorytm dający dokładnie te same wyniki, tylko w nieporównywalnie krótszym czasie. Niestety `AnalyserNode` „spłaszcza" je do postaci jedynie wartości bezwzględnych liczb zespolonych. Oznacza to, że informacja o przesunięciu fazowym jest dla nas niedostępna. Ciągle jednak jesteśmy w stanie narysować np. wykres widma amplitudowego, co w większości przypadków jest wystarczające.

5.2 ScriptProcessorNode

Węzeł ten umożliwia dostęp do pojedynczych próbek przechodzących przez nasz graf stworzony w kontekście audio. Jego działanie i konfiguracja jest bardzo prosta. Wystarczy stworzyć nowy obiekt za pomocą metody `createScriptProcessor` znajdującej się w kontekście audio. Przyjmuje ona 3 parametry. Pierwszy to rozmiar paczki z samplami, jaką będziemy cyklicznie przetwarzać i/lub generować podczas pracy naszej aplikacji. Wartość tego parametru musi być liczbą będącą potęgą dwójki z przedziału od 256 do 16384. Mamy zatem do dyspozycji 7 możliwości – 256, 512, 1024, 2048, 4096, 8192 oraz 16384. Mały rozmiar bufora umożliwia uzyskanie mniejszych opóźnień, jednak może powodować powstawanie niepożądanych trzasków. Duży rozmiar bufora powinien wyeliminować te problemy jednak wtedy musimy liczyć się z większymi opóźnieniami. Istnieje także ósma możliwość, czyli podanie zera. Wtedy Web Audio API podczas tworzenia węzła sam dobierze najbardziej odpowiedni rozmiar bufora, biorąc pod uwagę wydajność środowiska, na jakim pracuje.

Drugi i trzeci parametr mówi o liczbie kanałów kolejno dla wejścia i wyjścia. Do naszych zastosowań wystarczy dźwięk „mono", więc ta wartość powinna wynosić jeden lub zero. Dlaczego zero? Ponieważ gdy będziemy potrzebować węzła pełniącego rolę tylko generatora próbek, interesować nas będzie jedynie wyjście. Wtedy liczba kanałów wejścia może być równa zero. Czy zatem w sytuacji odwrotnej, gdy interesują nas tylko napływające próbki, możemy postąpić analogicznie (1 kanał wejścia, 0 kanałów wyjścia)? W zasadzie tak, ale nie do końca. Otóż w przeglądarce Chrome od dobrych kilkunastu miesięcy występuje pewien błąd. Gdy do węzła `ScriptProcessorNode` podłączymy wejście, ale nie podłączymy wyjścia, wtedy nasz węzeł nie będzie działać prawidłowo. Nie pomaga nawet fakt podania zera jako liczby kanałów wyjścia. Po prostu zdarzenie z paczką do przetworzenia nie będzie uruchamiane. Rozwiązaniem jest podłączenie wyjścia naszego script procesora np. do węzła `destination`. Tablica próbek wyjścia jest domyślnie wypełniona zerami, więc naszego węzła nie będzie „słychać". Chcąc zastosować sztuczkę z węzłem `destination`, liczbę kanałów wyjścia musimy ustawić na 1 (mono).

W zasadzie jednak ten problem nas nie dotyczy, gdyż wcale nie musimy tworzyć osobnych węzłów `ScriptProcessorNode` dla sampli napływających i sampli generowanych. Możemy zwyczajnie utworzyć jeden wspólny obiekt `ScriptProcessorNode`. W zdarzeniu, które jest przez niego odpalane, mamy dostęp zarówno do tablicy wejścia, jak i wyjścia.

Jak podpiąć się do tego zdarzenia? Wystarczy do właściwości `onaudioprocess` przypisać zwykłą funkcję. Będzie do niej przekazywany jeden parametr `audioProcessingEvent`, z którego możemy wydobyć wszystko, czego potrzebujemy (Listing 4).

Listing 4. ScriptProcessorNode – tablice z samplami

```
spNode = audioContext.createScriptProcessor(4096, 1, 1);

spNode.onaudioprocess = function (audioProcessingEvent) {
  var monoIn, monoOut;

  // potrzebujemy tylko kanału o indeksie zero (mono)
  monoIn = audioProcessingEvent.inputBuffer
    .getChannelData(0),
  monoOut = audioProcessingEvent.outputBuffer
    .getChannelData(0);

  sampleInHandler(monoIn);
  sampleOutHandler(monoOut, monoIn);
};

function sampleInHandler(monoIn) {
  console.log(monoIn.length);      // -> 4096
}

function sampleOutHandler(monoOut, monoIn) {
  console.log(monoOut.length, monoIn.length);
  // -> 4096 4096
}
```

W przykładzie z Listingu 4 przetwarzanie i generowanie sampli oddelegowaliśmy do osobnych funkcji, używając przy tym jednego węzła. Jego rozmiar bufora ustawiliśmy na 4096. Oznacza to, że każda z przekazywanych tablic będzie także tego rozmiaru. Funkcja `sampleInHandler` otrzyma zatem jeden parametr będący tablicą 4096 próbek wejścia np. z mikrofonu. Jest to w tym przypadku fragment o długości około 93 ms, zakładając standardowe próbkowanie (1000 * 4096 / 44100). Z tej tablicy będziemy jedynie czytać. W przypadku funkcji `sampleOutHandler` będą to dwa parametry – `monoOut` oraz `monoIn`. Zmienna `monoOut` to także fragment o długości około 93 ms, jednak tym razem strumienia audio, który zostanie odegrany np. na naszych głośnikach. Do tej tablicy będziemy więc tylko pisać. Gdy jednak zależy nam, by nasz węzeł „generował" ciszę, wystarczy nie pisać wcale. Jak już wspomnieliśmy, domyślnie nasza tablica `monoOut` jest wypełniona zerami. Drugi parametr `monoIn` możemy traktować jako opcjonalny. Możemy dzięki niemu generować dane wyjściowe na podstawie danych wejściowych.

Możemy teraz zapytać – po co w ogóle używać `ScriptProcessorNode` do przetwarzania napływających sampli? Przecież `AnalyserNode` ma metodę `getFloatTimeDomainData`. Ona także zwraca sample z wejścia węzła. Generalnie tak, jednak w przypadku `AnalyserNode` nie bardzo mamy kontrolę nad tym, jaki konkretnie blok dostaniemy. Oczywiście możemy próbować „celować" w odpowiednie miejsca w czasie np. timerem. Nigdy jednak nie będziemy mieć pewności, czy występujące po sobie bloki nie będą zachodzić na siebie lub czy nie będzie między nimi małej przerwy. Ze `ScriptProcessorNode` jest inaczej. O ile nasz procesor będzie nadążał z przetwarzaniem, będą to zawsze bloki występujące bezpośrednio po sobie co do sampla. Zachowana jest zatem ciągłość napływających próbek i każda z nich wystąpi tylko raz.

`ScriptProcessorNode` ma jednak jedną duża wadę. Otóż działa on w głównym wątku JavaScriptowym. Wątek ten odpowiedzialny jest także za interakcję z użytkownikiem. Możemy się o tym łatwo przekonać, uruchamiając `while (true)` w skrypcie bezpośrednio dołączonym do HTMLa. Nasza zakładka po prostu przestanie odpowiadać. Gdy zatem przetwarzanie naszych sampli będzie trwało zbyt długo, możemy spowodować nie tylko trzaski w audio, ale też „przywieszenie" naszej strony. Rozwiązaniem byłoby tu przeniesienie tych operacji do osobnego wątku. Okazuje się, że Web Audio API dokładnie do tego zmierza. W dokumentacji `ScriptProcessorNode` od dawna oznaczony jest jako *deprecated*. Zostanie on zastąpiony czymś w rodzaju audio Web Workerów. Na razie jednak nowe rozwiązania nie są dostępne. Warto jednak o tym pamiętać, by w pewnym momencie dokonać migracji.

Kod oraz źródła przykładu wykorzystującego `ScriptProcessorNode` znajdziemy poniżej:

- https://audio-network.rypula.pl/script-processor-node
- https://audio-network.rypula.pl/script-processor-node-src

6. WEB AUDIO API – PODSUMOWANIE

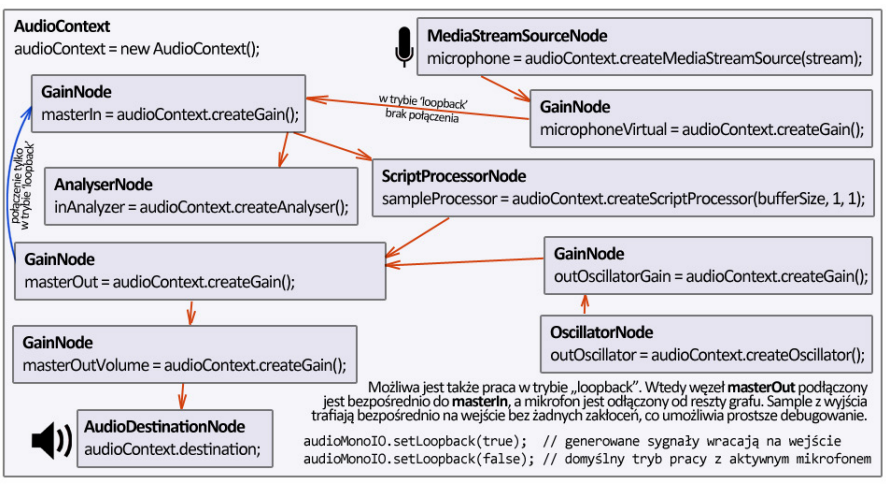
Węzły, które omówiliśmy, powinny w zupełności wystarczyć do zrealizowania prostej wymiany danych. Warto tutaj zaznaczyć, że Web Audio API jest ciągle rozwijane. Dobrym tego przykładem jest `ScriptProcessorNode`, który niebawem zostanie zastąpiony rozwiązaniem pracującym na osobnym wątku.

Opisane węzły to jedynie część potencjału drzemiącego w Web Audio API. Gdy chcemy dowiedzieć się czegoś więcej, warto zajrzeć na poniższe strony:

- https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API
- https://webaudio.github.io/web-audio-api/

7. IMPLEMENTUJEMY WŁASNĄ KLASĘ AUDIOMONOIO

Na tym etapie powinniśmy na tyle dużo wiedzieć o Web Audio API, by móc wyciągnąć z niego tylko to, co nam potrzeba. Do transmisji



Rysunek 11. Graf węzłów audio w klasie AudioMonoIO

danych będziemy potrzebowali tylko jednego kanału (mono). Jak wcześniej ustaliliśmy, oprócz ręcznego generowania próbek chcemy także wykorzystać potencjał rozwiązań wbudowanych, takich jak `AnalyserNode` czy `OscillatorNode`. Musimy zatem użyć w zasadzie wszystkich opisanych wcześniej węzłów. Na Rysunku 11 przedstawiono graf węzłów, jaki chcemy zaimplementować w klasie `AudioMonoIO`.

Po dodaniu obsługi błędów i zabiegów potrzebnych do pracy na starszych przeglądarkach w wyniku otrzymamy kompletną klasę `AudioMonoIO`. Jej kod dostępny jest na stronie projektu Audio-Network. Jest to jeden plik JavaScript bez żadnych dodatkowych zależności. Możemy go użyć w swoim projekcie, gdy nie chcemy wgryzać się w szczegóły komunikacji z Web Audio API.

» https://audio-network.rypula.pl/audio-mono-io-class

8. AUDIOMONOIO W PRAKTYCE

Na pierwszy ogień przetestujemy bardzo podstawowe użycie naszej klasy. Zaczniemy więc od utworzenia jej obiektu: `var audioMonoIO = new AudioMonoIO(fftSize, bufferSize, smoothingTimeConstant)`. Pierwszy parametr konstruktora mówi sam za siebie. Jest to rozmiar FFT. Kolejny parametr o nazwie `bufferSize` jest to rozmiar bufora `ScriptProcessorNode`. Jak pamiętamy, im niższa wartość, tym mniejsze opóźnienia. Ostatni parametr mówi o wygładzaniu wyniku FFT. Na potrzeby transmisji danych najlepiej ustawiać go na zero. Warto zaznaczyć, że parametry konstruktora nie są wymagane. Gdy ich nie podamy, zostaną użyte wartości domyślne.

Co zatem będzie robić nasza testowa aplikacja? Otóż na głośnikach postaramy się wygenerować dźwięk „syreny", składający się z dwóch tonów podstawowych zmieniających się co jedną sekundę. Nasza syrena będzie dodatkowo zmiksowana z białym szumem, by zaprezentować obie metody generowania dźwięku, jakie mamy do dyspozycji w `AudioMonoIO`. Dźwięk syreny wygenerujemy za pomocą metody `setPeriodicWave`. W naszym przykładzie użyjemy tylko dwóch pierwszych jej parametrów: częstotliwości oraz głośności. Kolejne trzy to „globalne" przesunięcie fazowe, tablica amplitud harmonicznych oraz tablica faz harmonicznych. Skorzystamy z nich w jednym z późniejszych przykładów. Pod spodem ta metoda używa `OscillatorNode`, więc nie powinna obciążać naszego procesora tak jak generowanie fali samodzielnie próbka po próbce.

Wracając do białego szumu – wytworzymy go za pomocą metody `setSampleInHandler`. Przyjmuje ona funkcję będącą handlerem do napływających paczek próbek. Paczka ta będzie dostępna jako parametr `monoIn` dostępny w funkcji naszego handlera. Parametr ten to oczywiście tablica sampli danego wycinka strumienia wyjściowego. By uzyskać pożądany efekt, wystarczy więc do każdego z tych sampli przypisać losową wartość z przedziału od -1 do 1. W praktyce nie chcemy, by nasz szum przyćmił syrenę, więc ograniczymy zakres do wartości z przedziału od -0.05 do 0.05.

Strumień sampli z mikrofonu przepuścimy natomiast przez „szukacz" największej wartości bezwzględnej amplitudy próbek z dziedziny czasu. Da to nam w rezultacie informację o ogólnej głośności sygnału z wejścia. W aplikacji tę informację wyświetlimy w formie paska, który będzie wypełniał się proporcjonalnie do głośności. Jako że sample dziedziny czasu możemy odczytać na dwa sposoby, umieścimy dwa paski. Będziemy mogli dzięki temu sprawdzić, czy zachowują się tak samo. Do tego celu użyjemy więc metody `getTimeDomainData` (tak naprawdę metoda `getFloatTimeDomainData` z `AnalyserNode`) oraz handlera ustawionego przez `setSampleInHandler` (pod spodem `ScriptProcessorNode`). W teorii powinniśmy uzyskać zbliżone wyniki, ponieważ ich okna powinny być relatywnie blisko siebie.

Nie byłoby zabawy, gdybyśmy nie wykorzystali także potencjału FFT. Odczyt wyników pracy transformaty możliwy jest poprzez metodę `getFrequencyData`. Przypomnijmy: zwracane dane to amplitudy częstotliwości składowych sygnału. Gdyby więc przeszukać całą tablicę w poszukiwaniu indeksu tablicy o najwyższej amplitudzie, moglibyśmy odgadnąć dominującą częstotliwość, jaką „słyszy" nasz mikrofon. Brzmi nieźle, pytanie tylko, jak zamienić numer indeksu na częstotliwość? Otóż rozdzielczość FFT jest równa częstotliwości próbkowania podzielonej przez rozmiar FFT. Wystarczy więc pomnożyć otrzymaną rozdzielczość przez znaleziony indeks, a otrzymana wartość będzie tym, czego szukamy.

Wszystkie operacje inicjalizacji tablicy `Float32Array` o odpowiednim rozmiarze ukryto w `AudioMonoIO`, by maksymalnie uprościć komunikację. Zarówno metoda `getFrequencyData`, jak i `getTimeDomainData` zwrócą nam zatem tablicę w tradycyjny sposób jako wynik jej działania. Zobaczmy więc, jak wygląda fragment kodu naszego przykładu (Listing 5).

Listing 5. Fragment przykładu wykorzystującego AudioMonoIO

```
/* ... */
function nextAnimationFrame() {
  timeDomain = audioMonoIO.getTimeDomainData();
  freqDomain = audioMonoIO.getFrequencyData();
  frequencyPeak = getIndexOfMax(freqDomain) * fftResolution;
  timeDomainMaxAbsValueAnalyser = normalizeToUnit(
    timeDomain[getIndexOfMaxAbs(timeDomain)]
  );

  domGaugeAnalyser.style.width =
    (timeDomainMaxAbsValueAnalyser * 100) + '%';
  domPeakFrequency.innerHTML = frequencyPeak.toFixed(2) + ' Hz';

  requestAnimationFrame(nextAnimationFrame);
}

function init() {
  txFrequency = 2000;
  /* ... */
  audioMonoIO = new AudioMonoIO(8192, 8192); // fftSize, bufferSize
  fftResolution =
    audioMonoIO.getSampleRate() / audioMonoIO.getFFTSize();

  audioMonoIO.setSampleInHandler(function (monoIn) {
    timeDomainMaxAbsValueRaw = normalizeToUnit(
      monoIn[getIndexOfMaxAbs(monoIn)]
    );
    domGaugeRaw.style.width =
      (timeDomainMaxAbsValueRaw * 100) + '%';
  });

  audioMonoIO.setSampleOutHandler(function (monoOut, monoIn) {
    for (var i = 0; i < monoOut.length; i++) {
      monoOut[i] = 0.05 * (Math.random() * 2 - 1);  // biały szum
      // możliwe jest także użycie sampli z mikrofonu:
      // monoOut[i] += 0.1 * monoIn[i];
    }
  });

  audioMonoIO.setPeriodicWave(txFrequency, 0.5);  // głośność 50%
  setInterval(function () {
    txFrequency = (txFrequency === 2000 ? 2500 : 2000);
    audioMonoIO.setPeriodicWave(txFrequency, 0.5);  // głośność 50%
  }, 1000);
  /* ... */
}
```

Pełna wersja jest niewiele dłuższa. Poniżej możemy znaleźć linki do wersji live oraz źródła:

» https://audio-network.rypula.pl/audio-mono-io-basic
» https://audio-network.rypula.pl/audio-mono-io-basic-src

Po uruchomieniu przykładu na pierwszy rzut oka widać, że czas reakcji pierwszego paska ogólnej głośności oraz wartości najgłośniejszej częstotliwości jest w zasadzie natychmiastowy. Możemy to przetestować, zwyczajnie pukając w mikrofon. Obydwa bazują na `AnalyserNode`. Tego jednak nie można powiedzieć o drugim pasku głośności używającym metody `setSampleInHandler` bazującej na `ScriptProcessorNode`.

Różnica odświeżania wynika to z faktu, iż `AnalyserNode` z założenia powinien dostarczać dane jak najszybciej, dając wrażenie pracy w czasie rzeczywistym. Można to osiągnąć, umieszczając wywołania metod `getTimeDomainData` oraz `getFrequencyData` w konstrukcji używającej `requestAnimationFrame`. Jest to funkcja informująca przeglądarkę, że chcemy narysować kolejną klatkę ekranu z poziomu JavaScriptu. Przeglądarka kolejkuje nasze żądanie i wywołuje podany handler tuż przed jej wewnętrznym cyklem przerysowania ekranu. Dla małych wartości `fftSize` węzeł `AnalyserNode` działa bardzo szybko. W praktyce oznacza to, że nasze dane będą się odświeżać w zasadzie z prędkością odświeżania ekranu (60 fps, czyli co około 16 ms).

Z paczką sampli dostępną w handlerze metody `setSampleInHandler` jest inaczej. Tam prędkość odświeżania zależy od rozmiaru bufora. W naszym przypadku przy inicjalizacji obiektu `AudioMonoIO` ustawiliśmy ten rozmiar na 8192. Oznacza to, że w jednej sekundzie mamy tylko około 5 „klatek" (44100 / 8192 = 5.4 fps). Przekłada się to na odświeżanie co około 185.8 ms.

Taki czas odświeżania to prawie 1/5 sekundy, więc można to także usłyszeć podczas startu naszej aplikacji. Syrena uruchamia się od razu, podczas gdy biały szum pojawia się z lekkim opóźnieniem. Web Audio API na starcie zwyczajnie nie ma danych i daje nam trochę czasu na ich wypełnienie (do 185.8 ms w tym przypadku). Jeżeli nie zdążymy zmieścić się w tym czasie, nasz dźwięk będzie poszarpany. Dlatego też mały rozmiar bufora daje mniejsze opóźnienie, lecz może powodować problemy objawiające się np. trzaskami.

Na zakończenie przyjrzyjmy się bliżej informacji o najgłośniejszej częstotliwości. Nasza syrena składa się z dwóch naprzemiennie emitowanych tonów o częstotliwości 2000 Hz oraz 2500 Hz. Gdy zatem nasz mikrofon wyłapie dźwięk głośników, będziemy w stanie zweryfikować, czy te wartości się pokrywają. Przy założeniu, że nasza maszyna pracuje na standardowym próbkowaniu 44100 Hz, a `fftSize` jest równe 8192, daje nam to rozdzielczość widma około 5.3833 Hz (44100/8192). Dla takiej rozdzielczości najbliższy prążek dla 2000 Hz będzie miał indeks 372. Oznacza to, że gdy transmitowany jest ton 2000 Hz, powinniśmy go odebrać za pomocą FFT jako 2002.59 Hz (372 * 5.3833 Hz). Analogicznie dla tonu 2500 Hz będzie to indeks 464, który związany jest z częstotliwością 2497.85 Hz (464 * 5.3833 Hz). Testy pokazują, że jesteśmy w stanie otrzymać dokładnie takie rezultaty. Pamiętajmy, że rozdzielczość możemy zwiększyć, używając większych rozmiarów `fftSize`.

Jak widać, ten przykład stanowi podwaliny pod transmisję danych. Gdy naszą syrenę uruchomimy na innej maszynie, podczas gdy druga maszyna będzie „słuchać", będziemy w stanie łatwo odtworzyć, który ton jest aktualnie transmitowany.

8.1 Zabawa harmonicznymi i przesunięciem fazowym

Metoda `setPeriodicWave` zdolna jest także do generowania dźwięku na podstawie amplitud oraz przesunięć fazowych składowych harmonicznych. Ich zmiana przy zachowaniu tej samej częstotliwości będzie wpływać na barwę dźwięku. To właśnie dzięki nim jesteśmy w stanie odróżnić różne instrumenty muzyczne czy też różne osoby emitujące ten sam dźwięk. Wiemy z sekcji o `OscillatorNode`, że harmoniczne o numerach większych niż 1 są to wielokrotności tonu podstawowego. Na wykresie widma amplitudowego pojawią się zatem w formie serii „pików".

Możemy się pokusić o uproszczenie działania naszego ucha i ośrodka słuchu w mózgu do postaci zagadnień, które już poznaliśmy. Nasz aparat słuchowy dokonuje zatem w pewnym sensie Transformaty Fouriera. Potem nasz mózg „wybiera" pierwszy pik ze spektrum (patrząc od lewej) i odczytuje jego częstotliwość. To powoduje wrażenie dźwięku o danej wysokości. Każdy następny pik zmienia już tylko jego barwę. Dzięki temu jesteśmy w stanie powiedzieć, czy ktoś nam np. wspomniany dźwięk zagrał na skrzypcach czy zwyczajnie zagwizdał. Oczywiście to bardzo duże uproszczenie, ale daje ono pewien pogląd na to, jak działa nasz słuch.

Metoda `setPeriodicWave` umożliwia także sterowanie „globalnym" przesunięciem fazowym całego kształtu. W dziedzinie czasu będzie to oznaczać przesuwanie naszego kształtu w prawo lub w lewo bez zmiany jego wyglądu. Czy to także da się „usłyszeć"? Niestety, nie. Odgrywane po sobie takie same przebiegi okresowe będą brzmiały dokładnie tak samo, nawet gdy będą przesunięte w fazie.

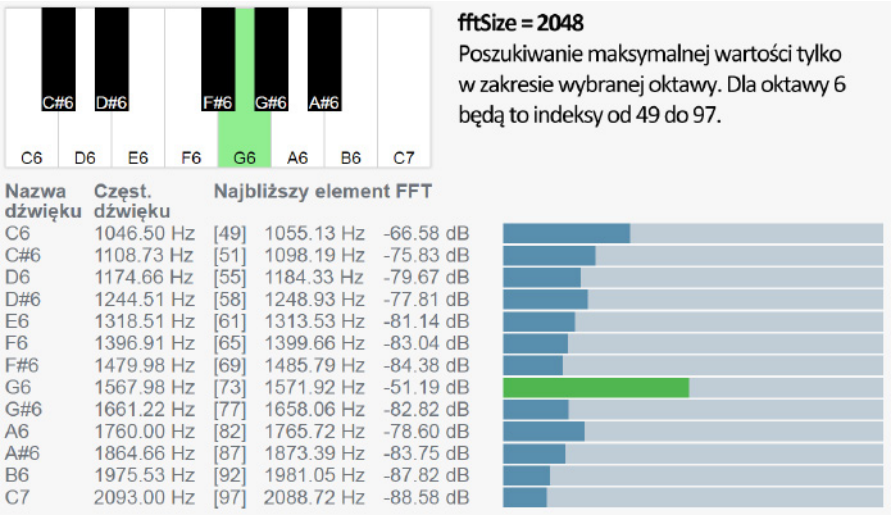
Na koniec wróćmy do przykładu. Działającą aplikację oraz jej kod możemy znaleźć pod linkami podanymi poniżej. Umożliwia ona oprócz generowania dźwięków także ich wizualizację. O samym procesie rysowania powiemy nieco więcej w następnej części tego artykułu. W przykładzie oprócz zwykłego sinusa dostępne są także przybliżenia przebiegów takich jak „piła", „prostokąt" czy „trójkąt". Wygenerowano je w całości za pomocą tablic amplitud harmonicznych. W Internecie można też znaleźć amplitudy harmonicznych dla instrumentów takich jak pianino czy wiolonczela. Brzmią one ciągle „komputerowo", lecz są już nieco bardziej przyjemne niż czysta sinusoida.

» https://audio-network.rypula.pl/harmonics-and-phase
» https://audio-network.rypula.pl/harmonics-and-phase-src

8.2 Wykrywanie dźwięków pianina

Mając możliwość generowania dźwięku oraz wykrywania dominującej częstotliwości, możemy pokusić się o stworzenie nieco bardziej użytkowej aplikacji. Co powiemy na proste pianino z możliwością detekcji emitowanego dźwięku na drugim urządzeniu? Podwaliny pod tego typu aplikację przedstawiliśmy przy okazji przykładu z „syreną". Do realizacji tego pomysłu musimy także dobrać odpowiednią częstotliwość do każdego klawisza. Potrzebne wzory znajdziemy na Rysunku 9. Mogą być one jednak nieco enigmatyczne, dlatego warto rozwinąć je o dodatkowy komentarz.

W skrócie: dźwięki na pianinie pogrupowane są w oktawy. Można je łatwo zlokalizować, gdyż tworzą charakterystyczny powtarzający się wzór złożony z klawiszy białych i czarnych. Wybierzmy zatem dla przykładu dowolny klawisz i znajdźmy jego odpowiednik w oktawie wyżej. Okazuje się, że jeśli częstotliwość pierwszego dźwięku oznaczymy jako x Hz, to częstotliwość klawisza z oktawy wyżej będzie równa 2x Hz. Analogicznie dźwięk z oktawy niższej będzie miał częstotliwość 0.5x Hz. Jak zatem dobrać częstotliwość pozostałych klawiszy? Otóż w oktawie mieści się 12 półtonów (klawisze białe i czarne). Przyjętym standardem jest strojenie równomiernie temperowane. W praktyce oznacza to, że częstotliwość klawisza $n + 1$ możemy wygenerować, mnożąc częstotliwość klawisza n przez stałą. Ta stała to pierwiastek dwunastego stopnia z dwóch,



Rysunek 12. Nadawanie i odbieranie dźwięków muzycznych

czyli w przybliżeniu 1.059463. Skąd się wzięła taka dziwna liczba? Otóż stąd, że po wspomnianych 12 półtonach nasza częstotliwość musi być 2x większa niż częstotliwość, z której wyszliśmy. Gdy pomnożymy x dwanaście razy przez naszą stałą, otrzymamy finalnie 2x. Trafimy więc poprawnie na dźwięk o oktawę wyższy. Takie strojenie sprawia, że możemy zagrać melodię, zaczynając z dowolnego klawisza. Dopóki przesuniemy wszystkie dalsze dźwięki melodii o tyle samo półtonów co pierwszy, będzie ona ciągle „tą samą melodią". Po prostu dla naszego mózgu znaczenie mają stosunki częstotliwości kolejnych dźwięków, a nie ich faktyczne wartości. Więcej informacji na ten temat możemy znaleźć na Wikipedii:

» https://en.wikipedia.org/wiki/Scientific_pitch_notation
» https://en.wikipedia.org/wiki/Equal_temperament

Przypomnijmy: do znalezienia w sygnale częstotliwości dominującej wystarczy w tablicy wyników FFT znaleźć element o maksymalnej wartości. Z numeru indeksu możemy łatwo obliczyć szukaną częstotliwość. Przeszukiwanie całej tablicy wyników FFT ma jednak wadę. Możemy zwyczajnie wychwycić częstotliwości, które co prawda będą „najgłośniejsze", lecz niekoniecznie będą związane z nadawanymi przez drugie urządzenie dźwiękami. Ten problem możemy jednak łatwo rozwiązać, ograniczając zakres częstotliwości do tylko jednej oktawy. Wtedy przeszukiwanie tablicy wyników FFT ograniczy się tylko do wąskiego zakresu indeksów (Rysunek 12). W oktawie znajduje się 12 półtonów. Da to nam w rezultacie możliwość transmitowania 12 unikalnych symboli pomiędzy dwoma urządzeniami.

Działający przykład i kod możemy znaleźć pod poniższymi linkami. Pamiętajmy, by zawsze wybrać ten sam numer oktawy na obydwu urządzeniach.

» https://audio-network.rypula.pl/piano
» https://audio-network.rypula.pl/piano-src

Zaprezentowana aplikacja oferuje oczywiście bardzo podstawową formę komunikacji. O tym, jak przesłać prawdziwe bajty danych, opowiemy w następnej części tego artykułu.

8.3 Dokumentacja

Niektóre z przedstawionych w artykule przykładów używają klas, które z łatwością możemy użyć w innych projektach. Te klasy to np. `AudioMonoIO`, `FFTResult`, `MusicCalculator`. Dwie ostatnie zostały użyte w przykładzie z pianinem. Link do ich opisów znajdziemy poniżej:

» https://audio-network.rypula.pl/example-docs

9 PODSUMOWANIE

Mam nadzieję, że chociaż jedna osoba czytająca ten artykuł zgodzi się ze mną, że Web Audio API to bardzo ciekawy moduł z ogromnym potencjałem. Jego możliwości oczywiście nie kończą się tylko na opisanych węzłach. Omawiając każdy z nich, często wykraczaliśmy poza wiedzę niezbędną do realizacji prostej transmisji danych. Myślę jednak, że zawsze warto poznać temat od podszewki. Wtedy jest nam dużo łatwiej wyciągnąć tylko to, co jest nam potrzebne. Upraszcza nam to życie później. Efektem takiego uproszczenia jest nasza własna klasa `AudioMonoIO`. Ukrywa ona wszystkie szczegóły komunikacji z Web Audio API, wystawiając jedynie prosty interfejs.

Tematyka przetwarzania sygnałów jest o tyle specyficzna, że często szukając informacji, możemy natrafić na opisy bazujące na technice analogowej. Przez dziesięciolecia była to jedyna droga, by przesłać coś np. za pomocą fal radiowych. Dopiero rozwój elektroniki umożliwił osiągnięcie podobnych efektów w sposób cyfrowy. Przekłada się to także na przetwarzanie dźwięku. Jednym z celów tej serii jest ujednolicenie tych wszystkich informacji i przekazanie ich w zrozumiałej dla programistów formie. Programiści zawsze widzą tablice próbek, a nie zmieniające się w czasie napięcie. Filtry to dla nas algorytmy, a nie układy zbudowane z kondensatorów, oporników czy cewek.

Naszym celem jest wykorzystanie Web Audio API do realizacji wymiany danych binarnych. W następnej części artykułu zabierzemy się zatem za brakujące elementy układanki. Omówimy m.in. techniki modulacji cyfrowej, opakujemy intuicyjny algorytm Dyskretnej Transformaty Fouriera z części pierwszej artykułu do postaci klasy oraz stworzymy kilka prostych aplikacji przesyłających dane za pośrednictwem dźwięku.

ROBERT RYPUŁA
robert.rypula@gmail.com

Pasjonat komputerów i programowania. Od wielu lat związany z aplikacjami webowymi. Obecnie zatrudniony w PGS Software na stanowisku Frontend Developer. Wcześniej doświadczenie zdobywał w firmie Okinet. Poza technologiami webowymi twórca m.in. aplikacji HgtReader umożliwiającej rendering topografii całej Ziemi (OpenGL/Qt). Fan pisania własnych rozwiązań od zera wszędzie tam, gdzie jest to możliwe.