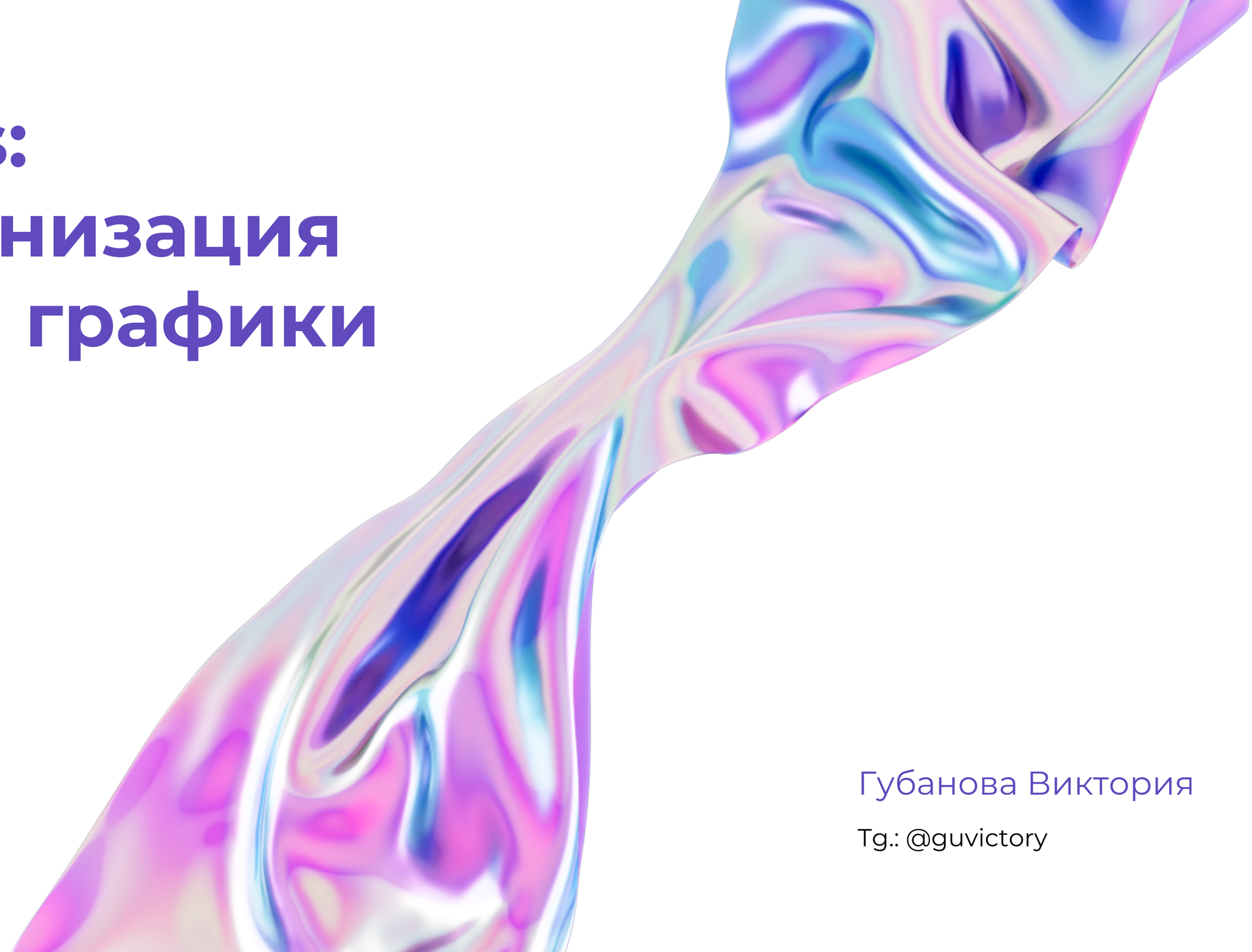
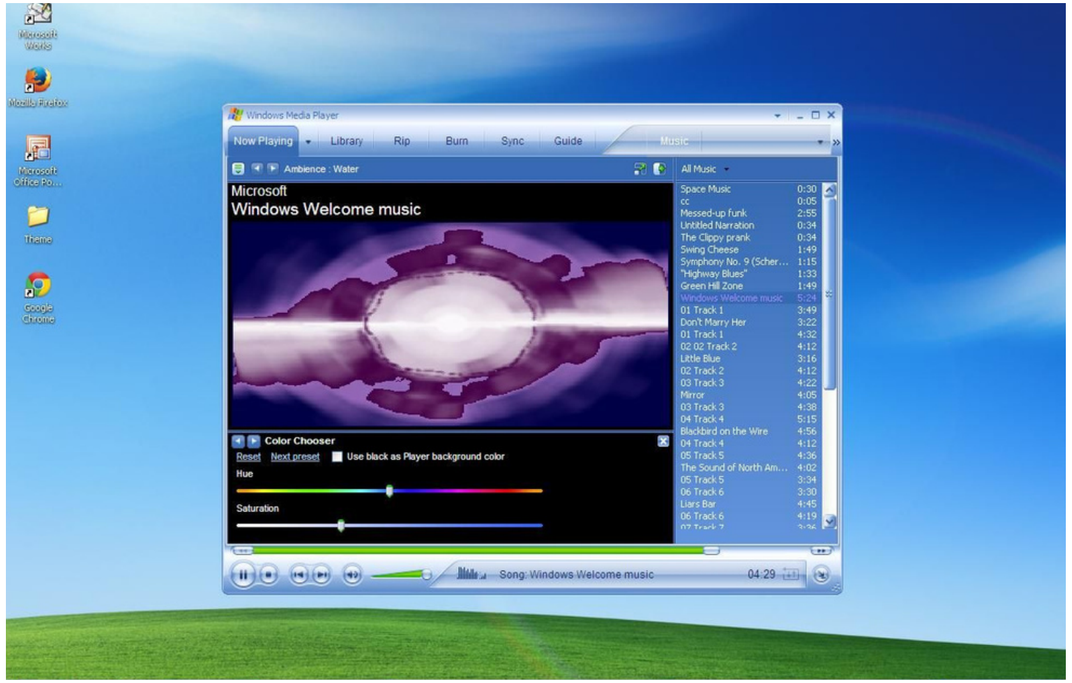
Three.js: синхронизация звука и графики

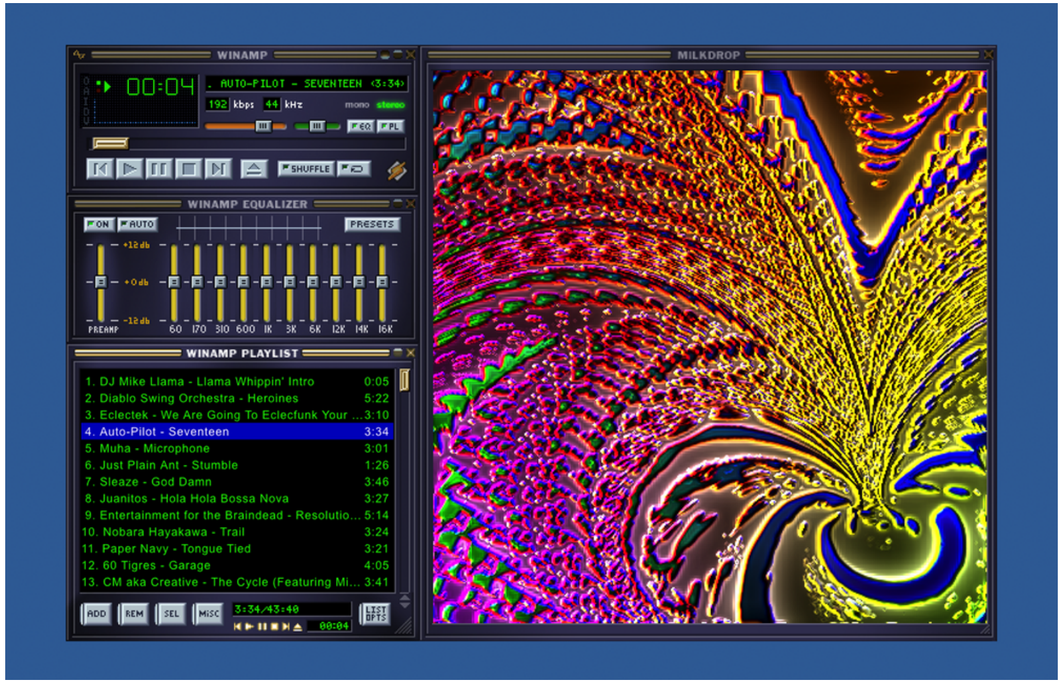


Губанова Виктория

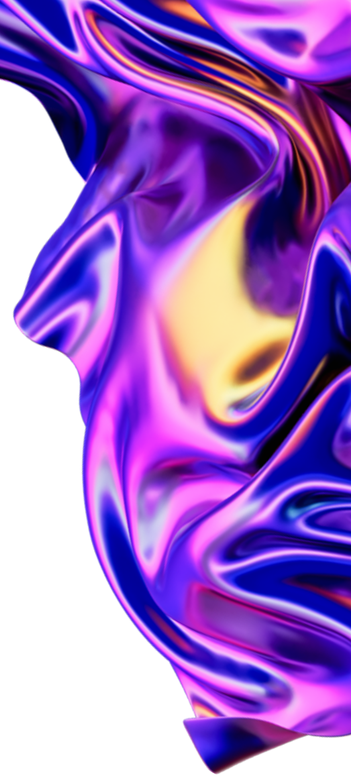
Tg.: @guvictory

Назад к истокам





План презентации



- Возможности Three.js для работы со звуком
- Анализ звука и математика
- Оптимизация производительности AudioAnalyser
- Анализ производительности
- Демо

Возможности Three.js

AudioListener и Audio

Основные объекты для воспроизведения звука.

AudioLoader

загружает аудиофайлы.

AudioAnalyser

анализирует частотный спектр аудио в реальном времени.

PositionalAudio

Расширяет Audio, добавляя поддержку позиционирования звука в 3D пространстве.

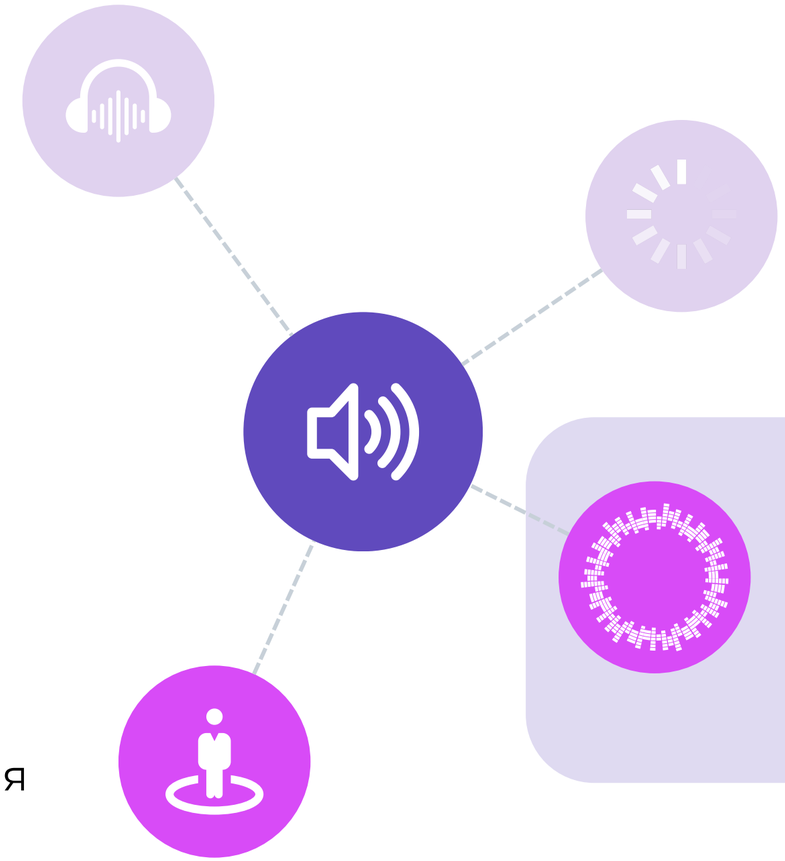
Возможности Three.js

AudioListener и Audio

Основные объекты для воспроизведения звука.

AudioLoader

загружает аудиофайлы.



AudioAnalyser

анализирует частотный спектр аудио в реальном времени.

PositionalAudio

Расширяет Audio, добавляя поддержку позиционирования звука в 3D пространстве.

AudioAnalyser

Разбитие на маленькие фрагменты

Обычно 1024 или 2048 сэмплов

AudioAnalyser

Разбитие на маленькие фрагменты

Обычно 1024 или 2048 сэмплов

Преобразование сигнала из временной области в частотную

Фрагменты проходят через быстрое преобразование Фурье

AudioAnalyser

Разбитие на маленькие фрагменты

Обычно 1024 или 2048 сэмплов

Преобразование сигнала из временной области в частотную

Фрагменты проходят через быстрое преобразование Фурье

Полученные частотных коэффициентов

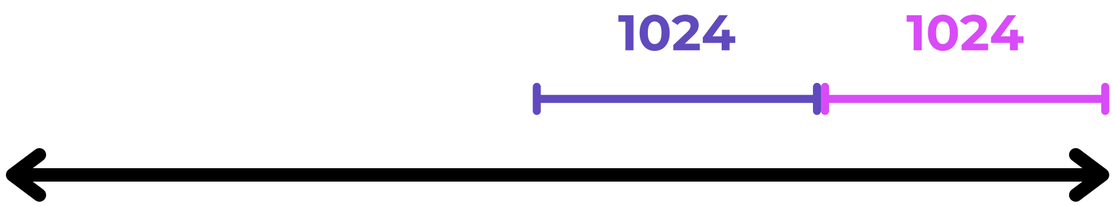
Интерпретируются как амплитуды (громкости)

Разбиение аудио на фрагменты

- Выбираем размер фрагмента - например, 1024
- Из потока аудиоданных выделяется первый фрагмент
- К этому фрагменту применяется БПФ, получаем спектр

Информация: Исходный аудиосигнал имеет частоту дискретизации 44100 Гц (44.1 кГц)

Разбиение аудио на фрагменты



- Выбираем размер фрагмента - например, 1024
- Из потока аудиоданных выделяется первый фрагмент
- К этому фрагменту применяется БПФ, получаем спектр
- Берётся следующий фрагмент аудиоданных, смещённый на 1024 относительно предыдущего

Преобразование в комплексные числа

$$x[n] \rightarrow X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j2\pi kn/N}$$

- x[n] - исходный сигнал, представленный отсчетами амплитуды в дискретные моменты времени n.
- X[k] - преобразованный сигнал, представленный комплексными коэффициентами для каждой гармоники с номером k.

Дискретное преобразование Фурье

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j2\pi kn/N}$$

- X[k] - результат ДПФ, комплексный коэффициент k-й гармоники.
- x[n] - исходный сигнал, представленный N отсчётами.

Нормализация коэффициентов

$$X_{\text{norm}}[k] = 20 \cdot \log\left(\frac{|X[k]|}{|X_{\text{max}}|}\right)$$

- X_norm[k] - нормализованное значение коэффициента k-й гармоники.
- X[k] - исходный коэффициент k-й гармоники, полученный в результате ДПФ.

Нормализация коэффициентов

$$X_{\text{norm}}[k] = 20 \cdot \log\left(\frac{|X[k]|}{|X_{\text{max}}|}\right)$$

коэффициент для перевода в логарифмический масштаб децибел

- X_norm[k] - нормализованное значение коэффициента k-й гармоники.
- X[k] - исходный коэффициент k-й гармоники, полученный в результате ДПФ.

Расчет амплитуды диапазонов

$$A[m] = \sum_{k \in \text{diap } m} X_{\text{norm}}[k]$$

- A[m] - результирующая амплитуда частотного диапазона номер m.
- X_norm[k] - нормализованный коэффициент Фурье для гармоники номер k.

А кто из вас использовал AudioAnalyzer?

Давайте перейдем и проголосуем, кто использовал, а кто нет 👀

Оптимизация производительности AudioAnalyser

Использование Web Workers

+ Позволит распараллелить вычисления

- Накладные расходы на передачу данных

Оптимизация алгоритмов

+ Уменьшит количество операций

- Возможности ограничены
- Может быть сложно

Векторизация вычислений

+ Эффективное использование SIMD => Высокий потенциал ускорения

- Может быть сложно

Оптимизация производительности AudioAnalyser

Использование Web Workers

+ Позволит распараллелить вычисления

- Накладные расходы на передачу данных

Оптимизация алгоритмов

+ Уменьшит количество операций

- Возможности ограничены
- Может быть сложно

Векторизация вычислений

+ Эффективное использование SIMD => Высокий потенциал ускорения

- Может быть сложно

Векторизация вычислений

Преимущества векторизации:

- Ускорение за счет параллельности
- Эффективное использование ресурсов CPU
- Масштабируется на многоядерные системы

Основная идея: выполнять операции не с отдельными элементами данных, а сразу с векторами (массивами) элементов.

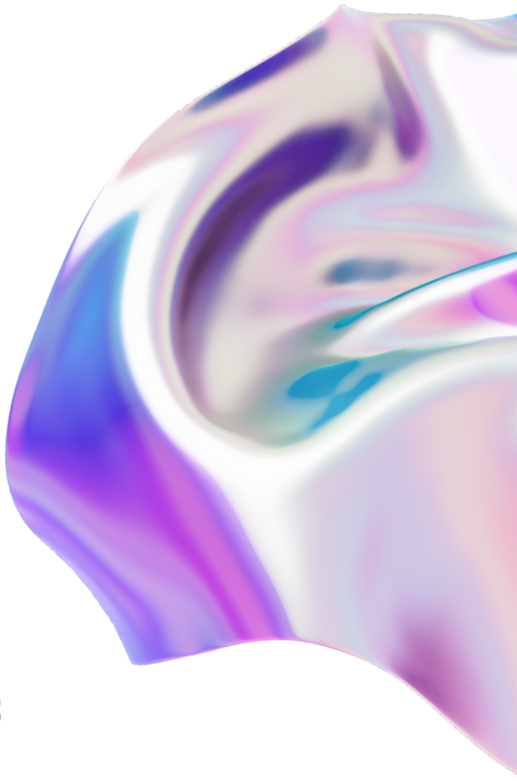
Векторизация вычислений

Можно векторизовать:

- ДПФ, переписав на векторных операциях
- Вычисление амплитуд частотных полос
- Другие участки кода math.js

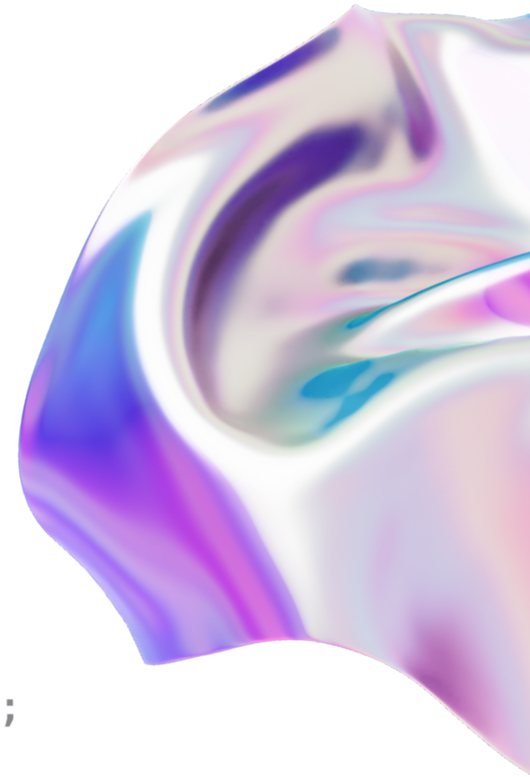
ДПФ без векторизации

```
function fft(buffer) {

  const n = buffer.length;
  const spectrum = [];

  for (let k = 0; k < n; k++) {

    // Вычисление спектра по формуле
    let sum = 0;
    for (let t = 0; t < n; t++) {
      let coeff = buffer[t] * windows[t] * Math.exp(-2j * Math.PI * k * t / n);
      sum += coeff;
    }

    spectrum[k] = sum;

  }

  return spectrum;

}
```



ДПФ без векторизации

```
function fft(buffer) {

  const n = buffer.length;
  const spectrum = [];

  for (let k = 0; k < n; k++) {

    // Вычисление спектра по формуле
    let sum = 0;
    for (let t = 0; t < n; t++) {
      let coeff = buffer[t] * windows[t] * Math.exp(-2j * Math.PI * k * t / n);
      sum += coeff;
    }

    spectrum[k] = sum;

  }

  return spectrum;

}
```



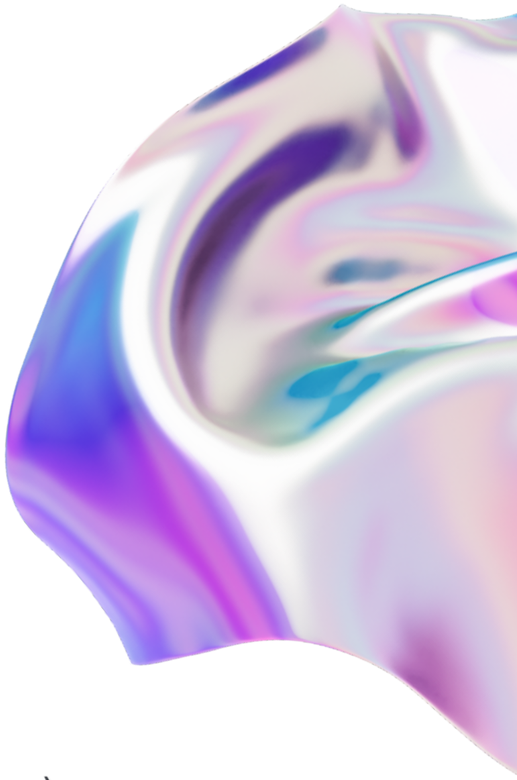
массив коэффициентов оконной функции

Оконные функции

Применяются для уменьшения эффектов утечки спектра при расчете дискретного преобразования Фурье.

- Прямоугольное окно - все коэффициенты равны 1
- Окно Хэмминга - коэффициенты подобраны для уменьшения утечек
- Окно Ханна - еще более эффективное подавление утечек

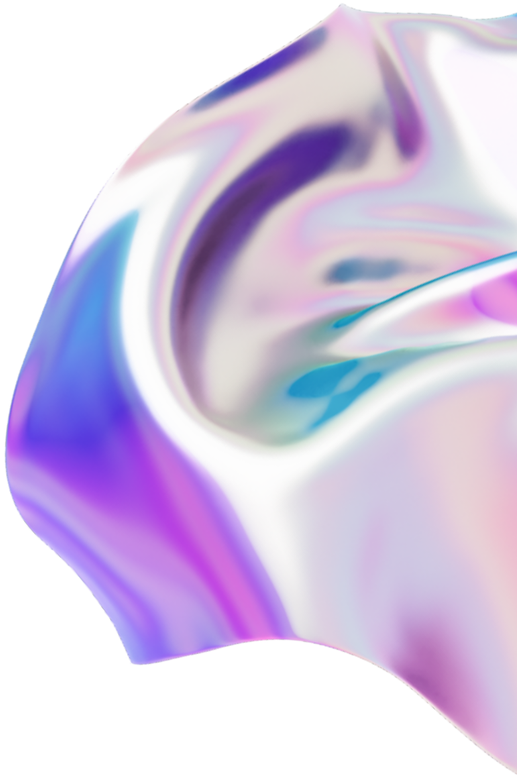
Векторизация для ДПФ



```
const vecSize = 4;

function vectorFFT(buffer) {
  const n = buffer.length;
  const spectrum = [];

  for (let k = 0; k < n; k+=vecSize) {
    const vecBuffer = buffer.slice(k, k+vecSize);
    const vecSpectrum = [];

    for (let t = 0; t < n; t++) {
      let sums = [];
      for (let v = 0; v < vecSize; v++) {
        let coeff = vecBuffer[v] * windows[t] * Math.exp(-2j * Math.PI * (k + v) * t / n);
        sums[v] += coeff;
      }
      vecSpectrum.push(sums);
    }
    spectrum.push(...vecSpectrum);
  }
  return spectrum;
}
```

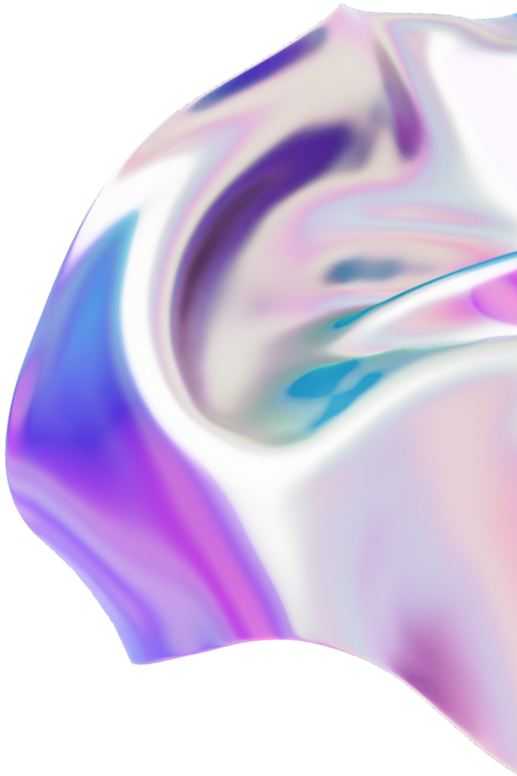
Амплитуды без векторизации

```
function getAmplitudes(spectrum) {

  const amplitudes = [];

  for (let k = 0; k < spectrum.length; k++) {

    let sum = Math.abs(spectrum[k]);
    amplitudes.push(sum);

  }

  return amplitudes;

}
```

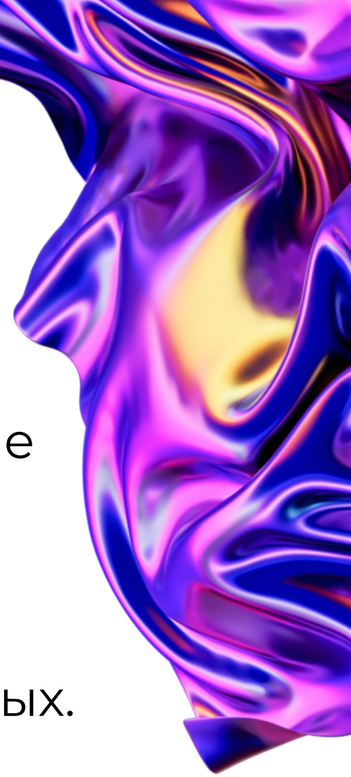


Векторизация амплитуд

```
function vectorGetAmplitudes(spectrum) {

  const amplitudes = [];
  const vecSize = 4;

  for (let k = 0; k < spectrum.length; k+=vecSize) {

    const vecSpectrum = spectrum.slice(k, k+vecSize);
    let sum = 0;

    for (let v = 0; v < vecSize; v++) {
      sum += Math.abs(vecSpectrum[v]);
    }

    amplitudes.push(sum);

  }

  return amplitudes;

}
```



А что если использовать SIMD?



SIMD (Single Instruction Multiple Data) - это тип инструкций процессора, которые позволяют выполнять одну операцию сразу над несколькими данными.

Основные характеристики SIMD:

- Параллельные вычисления - одна инструкция применяется к вектору данных.
- Данные упаковываются в регистры процессора и обрабатываются за один такт.
- Типичный размер вектора данных - 128, 256 или 512 бит.
- Поддержка арифметических, логических и других операций.

Векторизация для ДПФ с SIMD

```
function vectorFFT(buffer) {
  const n = buffer.length;
  const spectrum = [];
  const vecSize = 4;

  for (let k = 0; k < n; k+=vecSize) {
    // Загрузка входных данных в векторы
    let real = SIMD.Float32x4.load(buffer, k);
    let imag = SIMD.Float32x4.load(buffer, k+vecSize);

    // Вычисление спектра
    for (let t = 0; t < n; t++) {
      let coeff = SIMD.Float32x4.mul(real, windows[t]);
      coeff = SIMD.Float32x4.sub(coeff, imag);
      imag = SIMD.Float32x4.add(coeff, imag);
      real = coeff;

    }
    spectrum.push(real, imag);
  }
  return spectrum;
}
```

Векторизация для амплитуд с SIMD

```
function vectorGetAmplitudes(spectrum) {
  const amplitudes = [];
  for (let k = 0; k < spectrum.length; k+=8) {
    let real = SIMD.Float32x4.load(spectrum, k);
    let imag = SIMD.Float32x4.load(spectrum, k+4);

    let sum = SIMD.Float32x4.add(
      SIMD.Float32x4.mul(real, real),
      SIMD.Float32x4.mul(imag, imag)
    );
    amplitudes.push(SIMD.Float32x4.sum(sum));
  }
  return amplitudes;
}
```

Сравнение скорости работы (1)

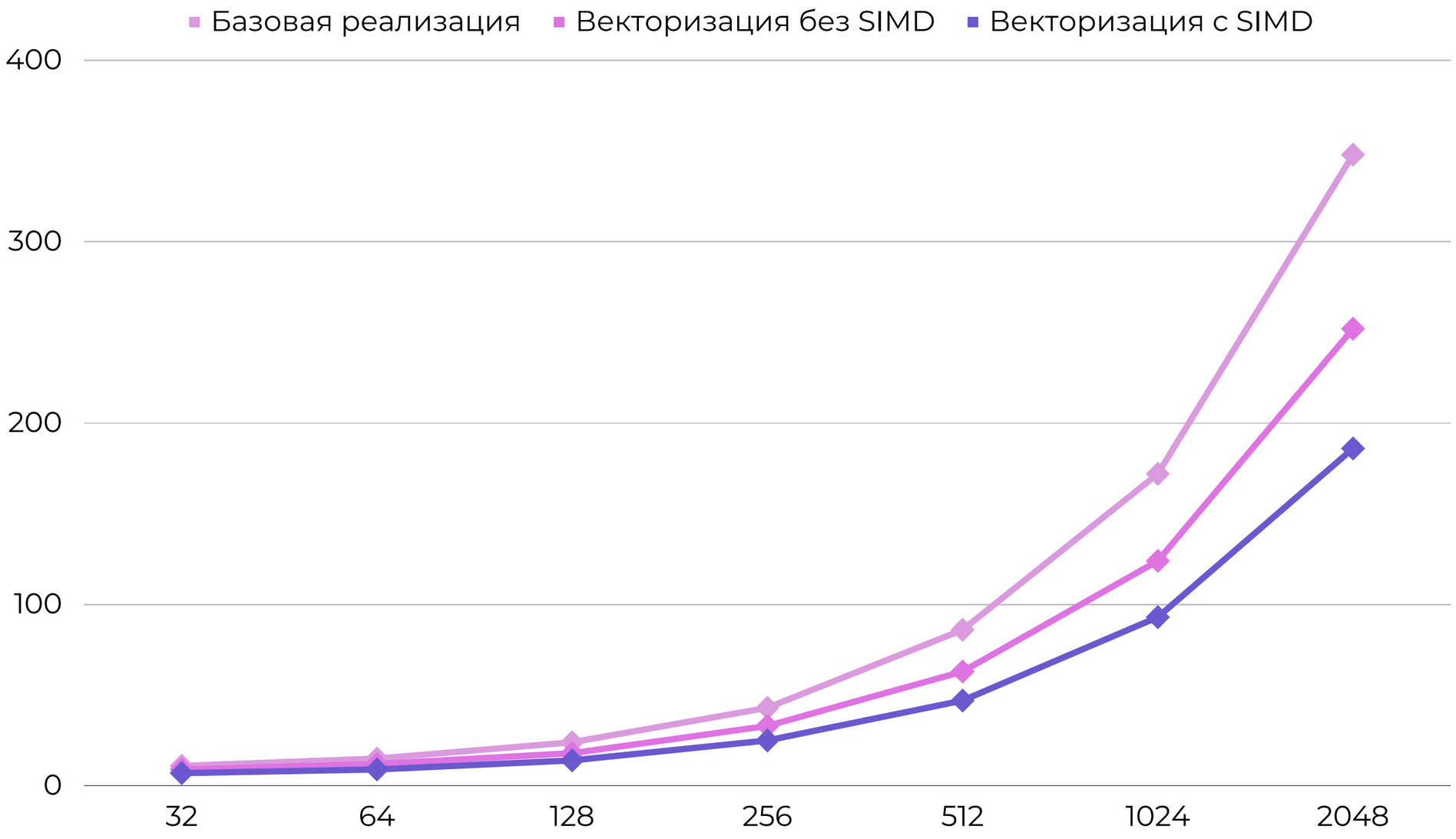
Размер fft	Базовая реализация AudioAnalyzer (мс)	Векторизированная реализация без SIMD (мс)	Векторизированная реализация с SIMD (мс)
32	11	9	7
64	15	12	9
128	24	18	14
256	43	33	25

Сравнение скорости работы (2)

Размер fft	Базовая реализация AudioAnalyzer (мс)	Векторизированная реализация без SIMD (мс)	Векторизированная реализация с SIMD (мс)
512	86	63	47
1024	172	124	93
2048	348	252	186

Какие результаты?

Векторизация без SIMD дает прирост производительности в ~1.2-1.3 раза.
С применением SIMD - до 1.5-1.9 раз ускорения.


Базовая реализация
Векторизация без SIMD
Векторизация с SIMD
400
300
200
100
0
32
64
128
256
512
1024
2048

Вопросы?

Email

guvictory.log@gmail.com

Telegram

@guvictory

Мой контент в Дзене