РАФАЭЛЬ МУН(RAPHAEL MUN)

Серия статей «Фильтры искусственного интеллекта лица в браузере»

УЧЕБНЫЕ РУКОВОДСТВА

Перевод: С. Кузнецов, 2024 г.

Articles: AI Face Filters in the Browser

Raphael Mun

2021

https://www.codeproject.com/Articles/instafluff#Article

Серия статей «Фильтры искусственного интеллекта лица в браузере»

Рафаэль Мун

2021

https://www.codeproject.com/Articles/instafluff#Article

Перевод: С. Кузнецов, 21.02.2024

Статья 4 «Создание виртуальных очков на лице в виде фильтра Snapchat-стиля в браузере с использованием библиотеки TensorFlow.js»

Статья 4 Создание виртуальных очков на лице в виде фильтра Snapchat-стиля в браузере с использованием библиотеки TensorFlow.js(Creating a Snapchat-Style Virtual Glasses Face Filter in the Browser with TensorFlow.js); https://www.codeproject.com/Articles/5293494/Creating-a-Snapchat-Style-Virtual-Glasses-Face-Fil) является статьей из серии статей Фильтры искусственного интеллекта лица в браузере(AI Face Filters in the Browser).

5 февраля 2021

В этой статье мы будем использовать ключевые лицевые точки(key facial points) для рендеринга 3D-модели фактически поверх видео из веб-камеры, для забавы в виде Добавленной реальности(Augmented Reality).

Здесь мы возвратимся к нашей цели - к созданию фильтра для лица в стиле Snapchat, используя предыдущие знания об отслеживании лиц, и добавим 3D-рендеринг с помощью библиотеки Three.js.

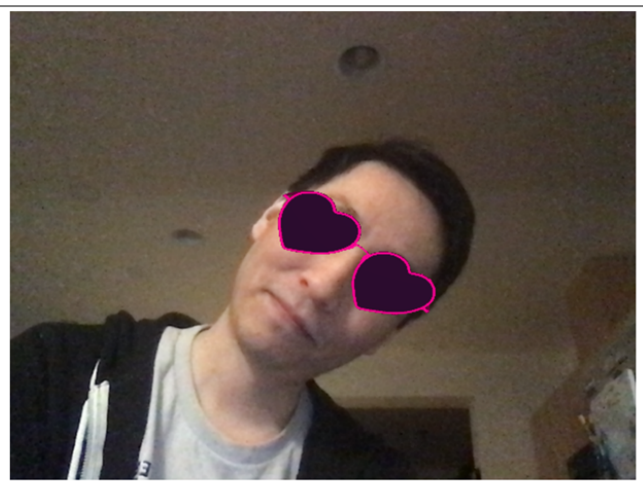
- **Загрузка кода и файлов - 565.6 KB**

Введение

Приложения, подобные приложению Snapchat, предлагают удивительное разнообразие фильтров лиц и линз, которые позволяют вам накладывать интересные эффекты на фотографии и видео. Если когда-либо вы «приделывали» себе виртуальные уши собаки или маскарадную шляпу, то знаете, это может быть забавно!

Задавались ли вы вопросом, как создать эти виды фильтров с нуля? Ну, теперь есть шанс научиться делать все в веб-браузере! В этой серии статей

мы собираемся показать, как в браузере создать фильтры в стиле **Snapchat**, обучить **AI**-модель понимать выражения лица, используя отслеживание лица и библиотеку **Tensorflow.js**.



(надпись под снимком экрана: **Загружено!(Loaded!)**)

Вы можете загрузить демонстрационный пример этого проекта. Возможно, для обеспечения производительности, вы будете должны в своем веб-браузере включить поддержку **Web**-графики **WebGL**. Также можно загрузить код и файлы для этой серии статей.

Предполагается, что вы знакомы с языками **JavaScript** и **HTML** и имеете, по крайней мере, базовое понимание нейронных сетей. Если вы плохо знакомы с фреймворком **TensorFlow.js**, то рекомендуем прочитать статью «Начало работы с глубоким изучением в браузере с использованием фреймворка TensorFlow.js»**(«Getting Started With Deep Learning in Your Browser Using TensorFlow.js»; https://www.codeproject.com/Articles/5272760/Getting-Started-With-Deep-Learning-in-Your-Browser)**, которая является статьей из серии статей **Обнаружение касания лица с помощью Tensorflow.js(Face Touch Detection with Tensorflow.js)**

Если хотели бы увидеть больше того, что возможно в веб-браузере с помощью фреймворка **TensorFlow.js**, прочтите статьи из серии по **искусственному интеллекту(AI)**: «Собаки и пицца: машинное зрение в браузере с использованием TensorFlow.js»**(«Dogs and Pizza: Computer Vision in the Browser With TensorFlow.js»; https://www.codeproject.com/Articles/5272771/Dogs-and-Pizza-Computer-Visio**

n-in-the-Browser-With) и «Роботы чатов с помощью фреймворка TensorFlow.js»(Chatbots using TensorFlow.js.).

Ранее мы научились использовать искусственный интеллект(ИИ; AI) в веб-браузере для отслеживания лиц в режиме реального времени и применять глубокое обучение для обнаружения и классификации эмоций на лице. Следующий логический шаг должен был бы соединить эти две функциональности и увидеть, можем ли мы выполнить обнаруживать эмоции из веб-камеры в режиме реального времени. Давайте сделаем это!

Здесь мы возвратимся к нашей цели - к созданию в браузере фильтра для лица в стиле Snapchat. На сей раз мы будем использовать ключевые лицевые точки(key facial points) для рендеринга 3D-модели фактически поверх видео из веб-камеры, для забавы в виде Добавленной реальности(Augmented Reality).

Добавление 3D-графики с помощью библиотеки Three.js

Этот проект базируется на коде проекта отслеживания лиц, который мы создали в начале статей этой серии. Мы добавим код наложения 3D-сцены поверх исходного HTML-элемента холст canvas.

Библиотека Three.js, работу с 3D-графикой делает относительно простой, и поэтому мы собираемся использовать эту библиотеку для рендеринга виртуальных очков поверх наших лиц.

В верхней части страницы мы должны включить два файла скриптов, необходимые для добавления библиотеки Three.js и загрузчика файлов в GLTF-формате модели виртуальных очков, которую мы будем использовать:

JavaScript

```
<script
src="https://cdn.jsdelivr.net/npm/three@0.123.0/build/three.min.js"></scri
pt>
```

```
<script
src="https://cdn.jsdelivr.net/npm/three@0.123.0/examples/js/loaders/GLTFLoader.js"></script>
```

Чтобы сделать код простым и не беспокоиться о том, как поместить текстуру из веб-камеры в сцену, мы можем наложить дополнительный прозрачный **HTML**-элемент холст **canvas** и на нём нарисовать виртуальные очки. Мы будем использовать следующий код в **CSS**-таблице стиля, расположенной выше **HTML**-элемента тело **<body>**, и этот код помещает выходной **HTML**-элемент холст **canvas** в контейнер и в контейнер также добавляет накладываемый прозрачный **HTML**-элемент холст **canvas**.

HTML

```
<style>
    .canvas-container {
        position: relative;
        width: auto;
        height: auto;
    }
    .canvas-container canvas {
        position: absolute;
        left: 0;
        width: auto;
        height: auto;
    }
</style>
<body>
    <div class="canvas-container">
        <canvas id="output"></canvas>
        <canvas id="overlay"></canvas>
    </div>
    ...
</body>
```

Есть несколько переменных для сохранения в **3D**-сцене, и мы можем добавить вспомогательную функцию загрузки **3D**-модели из файлов в формате **GLTF**:

```
<style>
    .canvas-container {
```

```
        position: relative;
        width: auto;
        height: auto;
    }
    .canvas-container canvas {
        position: absolute;
        left: 0;
        width: auto;
        height: auto;
    }
</style>
<body>
    <div class="canvas-container">
        <canvas id="output"></canvas>
        <canvas id="overlay"></canvas>
    </div>
    ...
</body>
```

Теперь мы можем инициализировать все элементы в асинхронном блоке с модификатором **async**, начиная с размера накладываемого прозрачного **HTML**-элемента холст **canvas**, как это было сделано с выходным **HTML**-элементом холст **canvas**:

JavaScript

```
(async () => {
    ...

    let canvas = document.getElementById( "output" );
    canvas.width = video.width;
    canvas.height = video.height;

    let overlay = document.getElementById( "overlay" );
    overlay.width = video.width;
    overlay.height = video.height;

    ...
})();
```

Также должны быть установлены и настроены **рендер(renderer)**, **сцена(scene)** и **камера(camera)**, но не волнуйтесь, если вы не знакомы с **3D**-перспективой и с математикой камеры. Этот код просто помещает камеру

сцены в то место, которое сделало бы ширину и высоту видео из веб-камеры соответствующим координатам 3D-пространства:

JavaScript

```
(async () => {
    ...

    // Загрузка модели обнаружения признаков лица
    // Load Face Landmarks Detection
    model = await faceLandmarksDetection.load(
        faceLandmarksDetection.SupportedPackages.mediapipeFacemesh
    );

    renderer = new THREE.WebGLRenderer({
        canvas: document.getElementById( "overlay" ),
        alpha: true
    });

    camera = new THREE.PerspectiveCamera( 45, 1, 0.1, 2000 );
    camera.position.x = videoWidth / 2;
    camera.position.y = -videoHeight / 2;
    camera.position.z = -( videoHeight / 2 ) / Math.tan( 45 / 2 );
    // расстояние до z должно быть тангенсом(fov / 2)
    // distance to z should be tan( fov / 2 )

    scene = new THREE.Scene();
    scene.add( new THREE.AmbientLight( 0xcccccc, 0.4 ) );
    camera.add( new THREE.PointLight( 0xffffff, 0.8 ) );
    scene.add( camera );

    camera.lookAt( { x: videoWidth / 2, y: -videoHeight / 2, z: 0,
isVector3: true } );

    ...
})();
```

Мы должны только, внутри функции отслеживания лица `trackFace`, добавить одну строку кода для рендеринга сцены поверх вывода отслеживания лица:

JavaScript

```
async function trackFace() {
    const video = document.querySelector( "video" );
```

```
    output.drawImage(
        video,
        0, 0, video.width, video.height,
        0, 0, video.width, video.height
    );
    renderer.render( scene, camera );

    const faces = await model.estimateFaces( {
        input: video,
        returnTensors: false,
        flipHorizontal: false,
    });

    ...
}
```

Последняя процедура, решения этой проблемы прежде, чем отобразить виртуальные объекты на нашем лице, должна загрузить **3D**-модель виртуальных очков. Мы нашли пару очков в форме сердца под маркой Heart Glasses by Maximkuzlin on SketchFab. Не стесняйтесь загрузить и использовать другой объект.

Вот код того, как мы можем загрузить объект и добавить его к сцене, прежде чем мы вызовем функцию отслеживания лица **trackFace**:

```
glasses = await loadModel( "web/3d/heart_glasses.gltf" );
scene.add( glasses );
```

Размещение виртуальных очков на отслеженном лице

Теперь наступает самое забавное: примерим наши виртуальные очки.

Маркированные аннотации, представленные **TensorFlow**-моделью отслеживания лиц, включают координату середины расстояния между глазами **midwayBetweenEyes**, где **X**- и **Y**-координаты отображаются на экране, и **Z**-координата добавляет глубину к экрану. Это делает размещение очков в наших глазах довольно простым.

Мы должны инвертировать **Y**-координату, потому что в **2D**-системе координат экрана положительная **Y**-ось указывает вниз, а в **3D**-системе

координат `Y`-ось указывает вверх. Мы также вычитаем расстояние камеры или глубину от значения `Z`-координаты для получения надлежащего расстояния в сцене.

JavaScript

```
glasses.position.x = face.annotations.midwayBetweenEyes[ 0 ][ 0 ];
glasses.position.y = -face.annotations.midwayBetweenEyes[ 0 ][ 1 ];
glasses.position.z = -camera.position.z +
                face.annotations.midwayBetweenEyes[ 0 ][ 2 ];
```

Теперь мы должны вычислить ориентацию и масштаб очков. Это становится возможным, как только мы выясним направление `"вверх"("up")` относительно нашего лица, указывающее на макушку нашей головы, и какое расстояние между глазами.

Чтобы вычислить направление `"вверх"("up")`, мы можем использовать вектор из точки середины расстояния между глазами `midwayBetweenEyes`, которую что мы использовали для очков, вместе с отслеженной точкой для нижней части носа, и затем нормализуем его длину следующим образом:

JavaScript

```
glasses.up.x = face.annotations.midwayBetweenEyes[ 0 ][ 0 ] -
                  face.annotations.noseBottom[ 0 ][ 0 ];
glasses.up.y = -( face.annotations.midwayBetweenEyes[ 0 ][ 1 ] -
                  face.annotations.noseBottom[ 0 ][ 1 ] );
glasses.up.z = face.annotations.midwayBetweenEyes[ 0 ][ 2 ] -
                  face.annotations.noseBottom[ 0 ][ 2 ];
const length = Math.sqrt( glasses.up.x ** 2 + glasses.up.y ** 2 +
                  glasses.up.z ** 2 );
glasses.up.x /= length;
glasses.up.y /= length;
glasses.up.z /= length;
```

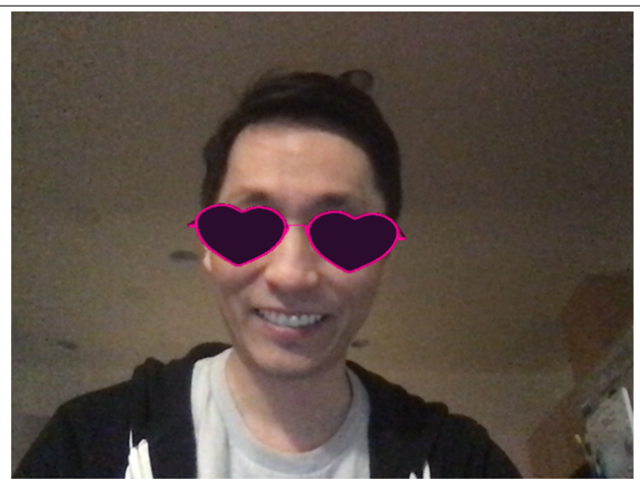
Чтобы получить относительный размер головы, мы можем вычислить расстояние между глазами:

JavaScript

```
const eyeDist = Math.sqrt(
    ( face.annotations.leftEyeUpper1[ 3 ][ 0 ] -
face.annotations.rightEyeUpper1[ 3 ][ 0 ] ) ** 2 +
    ( face.annotations.leftEyeUpper1[ 3 ][ 1 ] -
face.annotations.rightEyeUpper1[ 3 ][ 1 ] ) ** 2 +
    ( face.annotations.leftEyeUpper1[ 3 ][ 2 ] -
face.annotations.rightEyeUpper1[ 3 ][ 2 ] ) ** 2
);
```

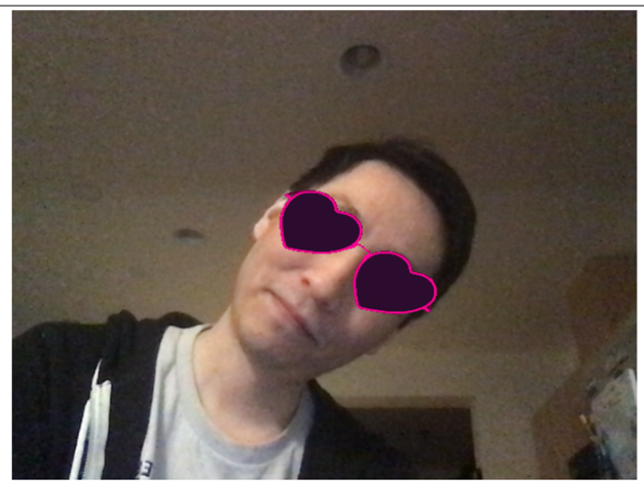
Наконец, мы масштабируем очки на основе значения расстояния между глазами **`eyeDist`** и ориентируем очки по **`Z`**-оси, используя угол между вектором **`"вверх"("up")`** и **`Y`**-осью, и вот и все!

Выполните свой код и проверьте результат.



Loaded!

(надпись под снимком экрана: **`Загружено!(Loaded!)`**)



Loaded!

(надпись под снимком экрана: **`Загружено!(Loaded!)`**)

Loaded!

(надпись под снимком экрана: **Загружено!(Loaded!)**)

Финишная черта

Прежде чем мы перейдем к следующей статье этой серии, давайте покажем полный код:

HTML

```
<html>
    <head>
        <title>Создание виртуальных очков на лице в виде фильтра
Snapchat-стиля</title>
        <script
src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@2.4.0/dist/tf.min.js"><
/script>
        <script
src="https://cdn.jsdelivr.net/npm/@tensorflow-models/face-landmarks-detect
ion@0.0.1/dist/face-landmarks-detection.js"></script>
        <script
src="https://cdn.jsdelivr.net/npm/three@0.123.0/build/three.min.js"></scri
pt>
        <script
src="https://cdn.jsdelivr.net/npm/three@0.123.0/examples/js/loaders/GLTFLo
ader.js"></script>
    </head>
    <style>
        .canvas-container {
            position: relative;
            width: auto;
            height: auto;
        }
        .canvas-container canvas {
            position: absolute;
```

```
            left: 0;
            width: auto;
            height: auto;
        }
    </style>
    <body>
        <div class="canvas-container">
            <canvas id="output"></canvas>
            <canvas id="overlay"></canvas>
        </div>
        <video id="webcam" playsinline style="
            visibility: hidden;
            width: auto;
            height: auto;
            ">
        </video>
        <h1 id="status">Загрузка/Loading...</h1>
        <script>
        function setText( text ) {
            document.getElementById( "status" ).innerText = text;
        }

        function drawLine( ctx, x1, y1, x2, y2 ) {
            ctx.beginPath();
            ctx.moveTo( x1, y1 );
            ctx.lineTo( x2, y2 );
            ctx.stroke();
        }

        async function setupWebcam() {
            return new Promise( ( resolve, reject ) => {
                const webcamElement = document.getElementById( "webcam" );
                const navigatorAny = navigator;
                navigator.getUserMedia = navigator.getUserMedia ||
                        navigatorAny.webkitGetUserMedia ||
                        navigatorAny.mozGetUserMedia ||
                        navigatorAny.msGetUserMedia;
                if( navigator.getUserMedia ) {
                    navigator.getUserMedia( { video: true },
                        stream => {
                            webcamElement.srcObject = stream;
                            webcamElement.addEventListener( "loadeddata",
                          resolve, false );
                        },
                    error => reject());
                }
                else {
                    reject();
                }
            });
        }
```

```
let output = null;
let model = null;
let renderer = null;
let scene = null;
let camera = null;
let glasses = null;

function loadModel( file ) {
    return new Promise( ( res, rej ) => {
        const loader = new THREE.GLTFLoader();
        loader.load( file, function ( gltf ) {
            res( gltf.scene );
        }, undefined, function ( error ) {
            rej( error );
        } );
    });
}

async function trackFace() {
    const video = document.querySelector( "video" );
    output.drawImage(
        video,
        0, 0, video.width, video.height,
        0, 0, video.width, video.height
    );
    renderer.render( scene, camera );

    const faces = await model.estimateFaces( {
        input: video,
        returnTensors: false,
        flipHorizontal: false,
    });

    faces.forEach( face => {
        // Рисуем ограничивающий прямоугольник вокруг лица
        // Draw the bounding box
        const x1 = face.boundingBox.topLeft[ 0 ];
        const y1 = face.boundingBox.topLeft[ 1 ];
        const x2 = face.boundingBox.bottomRight[ 0 ];
        const y2 = face.boundingBox.bottomRight[ 1 ];
        const bWidth = x2 - x1;
        const bHeight = y2 - y1;
        drawLine( output, x1, y1, x2, y1 );
        drawLine( output, x2, y1, x2, y2 );
        drawLine( output, x1, y2, x2, y2 );
        drawLine( output, x1, y1, x1, y2 );

        glasses.position.x =
                  face.annotations.midwayBetweenEyes[ 0 ][ 0 ];
        glasses.position.y =
```

```
            -face.annotations.midwayBetweenEyes[ 0 ][ 1 ];
glasses.position.z = -camera.position.z +
            face.annotations.midwayBetweenEyes[ 0 ][ 2 ];

// Вычислите Вектор, направленный вверх,
// используя позицию глаз и нижнюю часть носа
// Calculate an Up-Vector using the eyes
// position and the bottom of the nose
glasses.up.x = face.annotations.midwayBetweenEyes[ 0 ]
            [ 0 ] - face.annotations.noseBottom[ 0 ][ 0 ];
glasses.up.y = -( face.annotations.midwayBetweenEyes[ 0 ]
            [ 1 ] - face.annotations.noseBottom[ 0 ][ 1 ] );
glasses.up.z = face.annotations.midwayBetweenEyes[ 0 ]
            [ 2 ] - face.annotations.noseBottom[ 0 ][ 2 ];
const length = Math.sqrt( glasses.up.x ** 2 +
            glasses.up.y ** 2 + glasses.up.z ** 2 );
glasses.up.x /= length;
glasses.up.y /= length;
glasses.up.z /= length;

// Масштаб к размеру головы
// Scale to the size of the head
const eyeDist = Math.sqrt(
    ( face.annotations.leftEyeUpper1[ 3 ][ 0 ] -
      face.annotations.rightEyeUpper1[ 3 ][ 0 ] ) ** 2 +
    ( face.annotations.leftEyeUpper1[ 3 ][ 1 ] -
      face.annotations.rightEyeUpper1[ 3 ][ 1 ] ) ** 2 +
    ( face.annotations.leftEyeUpper1[ 3 ][ 2 ] -
      face.annotations.rightEyeUpper1[ 3 ][ 2 ] ) ** 2
        );
        glasses.scale.x = eyeDist / 6;
        glasses.scale.y = eyeDist / 6;
        glasses.scale.z = eyeDist / 6;

        glasses.rotation.y = Math.PI;
        glasses.rotation.z = Math.PI / 2 -
                    Math.acos( glasses.up.x );
    });

    requestAnimationFrame( trackFace );
}

(async () => {
    await setupWebcam();
    const video = document.getElementById( "webcam" );
    video.play();
    let videoWidth = video.videoWidth;
    let videoHeight = video.videoHeight;
    video.width = videoWidth;
    video.height = videoHeight;
```

```
let canvas = document.getElementById( "output" );
canvas.width = video.width;
canvas.height = video.height;

let overlay = document.getElementById( "overlay" );
overlay.width = video.width;
overlay.height = video.height;

output = canvas.getContext( "2d" );
output.translate( canvas.width, 0 );
output.scale( -1, 1 ); // Зеркалируем
output.fillStyle = "#fdffb6";
output.strokeStyle = "#fdffb6";
output.lineWidth = 2;

// Загрузка модели обнаружения признаков лица
// Load Face Landmarks Detection
model = await faceLandmarksDetection.load(
    faceLandmarksDetection.SupportedPackages.mediapipeFacemesh
);

renderer = new THREE.WebGLRenderer({
    canvas: document.getElementById( "overlay" ),
    alpha: true
});
```

```
camera = new THREE.PerspectiveCamera( 45, 1, 0.1, 2000 );
camera.position.x = videoWidth / 2;
camera.position.y = -videoHeight / 2;
camera.position.z = -( videoHeight / 2 ) / Math.tan( 45 / 2 );
 // расстояние до z должно быть тангенсом(fov / 2)
 // distance to z should be tan( fov / 2 )

scene = new THREE.Scene();
scene.add( new THREE.AmbientLight( 0xcccccc, 0.4 ) );
camera.add( new THREE.PointLight( 0xffffff, 0.8 ) );
scene.add( camera );

camera.lookAt( { x: videoWidth / 2,
 y: -videoHeight / 2, z: 0, isVector3: true } );
 // Очки с сайта по адресу https://sketchfab.com/3d-models/
 heart-glasses-ef812c7e7dc14f6b8783ccb516b3495c
 // Glasses from
https://sketchfab.com/3d-models/heart-glasses-ef812c7e7dc14f6b8783ccb516b3
495c
glasses = await loadModel( "web/3d/heart_glasses.gltf" );
scene.add( glasses );

setText( "Загружено!/Loaded!" );

trackFace();
```

```
        })();
        </script>
    </body>
</html>
```

Что далее? Что, если мы также добавим обнаружение эмоции на лице?

Полагаете ли вы, что все это возможно в одной веб-странице? Добавляя 3D-объекты к функциональности отслеживания лица в реальном времени, мы вызвали волшебство камеры прямо в веб-браузере. Вы могли бы подумать, “Но очки в форме сердца существуют в реальной жизни...” И это - правда и поэтому, а что, если мы создадим что-то действительно волшебное, такое как шляпа …, которая знает, как мы чувствуем?

Давайте в следующей статье создадим волшебную шляпу для обнаружения эмоций на лице и посмотрим, можем ли мы сделать невозможное возможным с более глубоким использованием библиотеки TensorFlow.js!

Эта статья является статьей из серии статей **Фильтры искусственного интеллекта лица в браузере(AI Face Filters in the Browser)**.