# FaceTheremin

In this project we’ll be using brand new UWP APIs for image processing and low latency audio. There are some massive improvements in UWP for 3rd party developers working with media APIs such as camera and audio playback:

<https://msdn.microsoft.com/en-us/library/windows/apps/mt203788.aspx>

If you haven’t yet had time to checkout the related //BUILD 2015 sessions, you definitely should. Here are just a few:

**A Studio in the Palm of Your Hand: Developing Audio and Video Creation Apps for Windows 10**

<https://channel9.msdn.com/Events/Build/2015/3-634>

**Camera: Developing Powerful Camera Apps**

<https://channel9.msdn.com/Events/Build/2015/2-730>

**Developing Audio and Video Apps**

<https://channel9.msdn.com/Events/Build/2015/3-747>

To inspire you to experiment more with these new media APIs, we decided to make something very simple, but at the same time highly visual and entertaining. Presenting: FaceTheremin!

You’ve probably seen people play Guitar Hero or similar games at parties countless times. It’s a lot of fun! If you’re a real geek and developer, you can make a similar entertainment system for a party in just few hours. All you need is Windows 10 PC with a web camera. How?

We’re going to use face detection on a live video stream from a camera to trigger musical events. The easiest way to do this is to consider the video frame as a grid of cells where each cell corresponds to a unique sound. If our app detects a face in a given cell, it should trigger a sound. With the new FaceTracker API you can track a collection of faces at the same time. Make sure to read MSDN documentation on FaceTracker:

<https://msdn.microsoft.com/en-us/library/windows/apps/windows.media.faceanalysis.facetracker.aspx>

For a complete and comprehensive example of face tracking, you should take a look at CameraFaceDetection from the official UWP sampler repository:

<https://github.com/Microsoft/Windows-universal-samples/tree/master/Samples/CameraFaceDetection>

There are multiple options for making sounds available to developers, including real-time tone generation, but the simplest way is to have a collection of readymade MP3 or WAV audio files. The traditional option for hassle-free playback of pre-recorded media in Windows Runtime is MediaElement. If you need really low latency playback, you used to go with XAudio2 API which is C++ based and quite complex. But now we have an excellent new Audio Graph API:

<https://msdn.microsoft.com/en-us/library/windows/apps/mt203787.aspx>

Audio Graph API is very easy to use, supports compressed audio formats and uses Windows 10’s low-latency audio pipeline.

Let’s look at the actual app making mechanics!

Create a new Blank App (Universal Windows) project, add CaptureElement for streaming video from a webcam and TextBlock for status of the app. By the way, it’s more natural if the video image is inverted horizontally, so you can see yourself a mirror image of yourself.

<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">

<Grid.RowDefinitions>

<RowDefinition/>

<RowDefinition Height="Auto"/>

</Grid.RowDefinitions>

<Grid RenderTransformOrigin="0.5,0.5">

<Grid.RenderTransform>

<ScaleTransform ScaleX="-1"/>

</Grid.RenderTransform>

<CaptureElement x:Name="StreamingElement"

HorizontalAlignment="Stretch"

VerticalAlignment="Stretch"

Stretch="Fill">

</CaptureElement>

</Grid>

<RelativePanel Grid.Row="1" Margin="12">

<TextBlock Text="{x:Bind StatusText, Mode=OneWay}"

RelativePanel.AlignRightWithPanel="True"

RelativePanel.AlignVerticalCenterWithPanel="True"/>

</RelativePanel>

</Grid>

In order to render a live video preview from the webcam, we should use MediaCapture with StreamingCaptureModeVideo mode and set it as a source for the CaptureElement.

var settings = new MediaCaptureInitializationSettings {StreamingCaptureMode = StreamingCaptureMode.Video};

\_mediaCapture = new MediaCapture();

await \_mediaCapture.InitializeAsync(settings);

\_videoProperties = \_mediaCapture.VideoDeviceController.GetMediaStreamProperties(MediaStreamType.VideoPreview) as VideoEncodingProperties;

StreamingElement.Source = \_mediaCapture;

await \_mediaCapture.StartPreviewAsync();

Once we can see ourselves, it’s time to detect faces. We need some kind of loop that runs a face detection function every N milliseconds. Timer is the easiest way to do this, especially in a “toy” app like this. For every “tick”, MediaCapture provides us with a VideoFrame, which we can pass to the FaceTracker to get a list of DetectedFaces.

using (var previewFrame = new VideoFrame(inputPixelFormat, (int)\_videoProperties.Width, (int)\_videoProperties.Height))

{

await \_mediaCapture.GetPreviewFrameAsync(previewFrame);

IList<DetectedFace> faces;

var faces = await \_faceTracker.ProcessNextFrameAsync(previewFrame);

Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () => {

ProcessFaceCells(previewFrameSize, faces);

});

}

Now, with faces detected, it’s time to play some sounds. We chose three different instruments: drums, piano chords and some synth tones. Each type of sound has two columns with 8 variations each. AudioGraph connects several input nodes (they can be generated, played from files or MIDI) to one or more output nodes (in our case speakers). First of all, we need to initialize AudioGraph and create an output node:

var settings = new AudioGraphSettings(AudioRenderCategory.Media);

var result = await AudioGraph.CreateAsync(settings);

if (result.Status == AudioGraphCreationStatus.Success)

{

\_audio = result.Graph;

var outputResult = await \_audio.CreateDeviceOutputNodeAsync();

AudioDeviceOutputNode audioDeviceOutputNode = null;

if (outputResult.Status == AudioDeviceNodeCreationStatus.Success)

{

audioDeviceOutputNode = outputResult.DeviceOutputNode;

}

}

We have a separate StorageFile and AudioFileInputNode for each sound. After initialization, we can add an output node and store sounds in two-dimension array:

var instruments = new[] {"snd\_2", "snd\_1", "synth\_2", "synth\_1", "drum\_2", "drum\_1"};

var storageFiles = new StorageFile[CellsRowsCount, CellsColumnsCount];

for (var i = 0; i < instruments.Length; i++)

{

await LoadStorageFiles(storageFiles, i, instruments[i]);

}

for (var y = 0; y < CellsRowsCount; y++)

{

for (var x = 0; x < CellsColumnsCount; x++)

{

var inputResult = await \_audio.CreateFileInputNodeAsync(storageFiles[y, x]);

if (inputResult.Status == AudioFileNodeCreationStatus.Success)

{

var audioFileInputNode = inputResult.FileInputNode;

audioFileInputNode.Stop();

audioFileInputNode.AddOutgoingConnection(audioDeviceOutputNode);

\_audioFileInputNodes[y, x] = audioFileInputNode;

}

}

}

We know the coordinates of the detected faces, so we know in which cells they are and can finally start audio playback:

foreach (var audioFileInputNode in newCells.Select(x => \_audioFileInputNodes[x.Y, x.X]))

{

audioFileInputNode.Reset();

audioFileInputNode.Start();

}

That’s it! You can invite your friends to help you make a new hit or at least have tons of fun!

