Stegafoto: a lens which embeds audio and text inside images

Stegafoto is a Windows Phone Lens which enables the user to embed a piece of audio or text within the image. This article explains the theory used to embed the content (virtually "loss free") along with technical detail about the implementation.

17 Feb 2013

Introduction

Embedding text or audio within an image can make it easier for the photographer to vividly re-live the experience when browsing an image months after it has been taken. The technique used here is "fat-free" (does not increase the size of the image) and does not visibly distort or affect the quality of the image. Briefly put, the technique uses the principle of Steganography with a simple "even-odd" encoding scheme in the least significant bits of the pixels in the image.

The article explains how this result has been achieved at two levels. The first part is structured so that even someone with no programming experience should be able to get a feel for how it works - all you need is an open mind. The "Technical Details" parts that follow assume that the reader is familiar with C# and Javascript programming.

The video below shows this process:

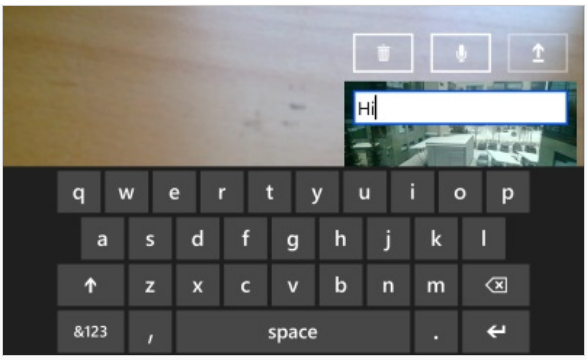
Walkthrough

In order to simplify the explanation, we first discuss how to embed *text* in the image (embedding audio is very similar and we discuss briefly below).

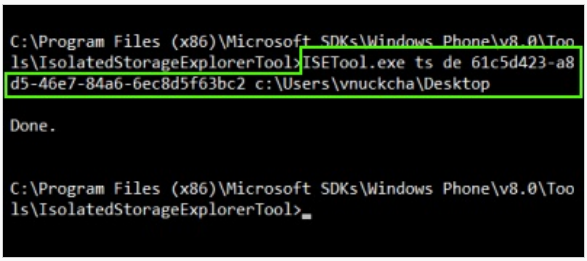
The case of embedding text into an image is demonstrated in the following steps:

Taking a picture with the Stegafoto lens.

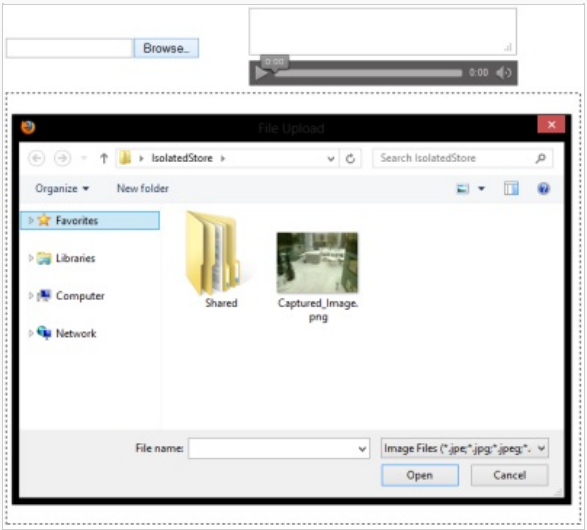
(C) Copyright Nokia 2013. All rights reserved.



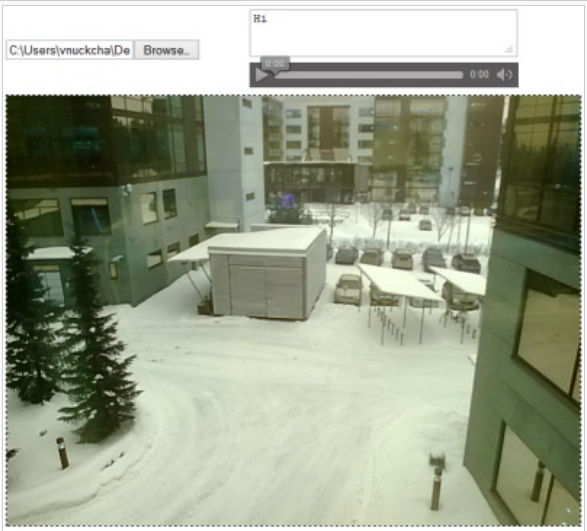
Entering a piece of text to be embedded inside the image. Text reads "Hi".



Transferring the stegafoto from the device to the PC in order to view it as I did not bother to implement an "upload-to-picture-service" feature.



Opening the picture from the web-browser via a loaded page with special Javascript code.



The captured image + the embedded text "Hi". *Note the greenish tint on the image comes from the 3rd party PNG encoder that i am using and not from the data fusion process.*

Also note, the method described below is one of the many ways of performing this task. While writing the application I used a test-driven approach coupled with rapid prototyping and this is why I ended up with this series of steps. I did not bother with re-factoring the algorithm in order to optimize the solution (e.g. implement a fault tolerant scheme or a picture upload functionality) as I am not interested in writing a product but, merely keen in proving the concept. Also, I used a 3rd party library Imagetools for Silverlight as the PNG encoder for saving the captured image stream to a file. I chose to use

 (C) Copyright Nokia 2013. All rights reserved.

Embedding the data

In order to explain the concept, let's use a real-life analogy: Let's say we would like to mix together two substances such as, sugar and water. One would go about by adding the two in a bowl and apply a process of stirring in order to do the mixing. In terms of an end result, if you were to inspect the bowl, it would look like the water has remained intact but, the sugar has vanished. The reason behind why the sugar "disappears" from the bowl is that there is a physical decomposition that splits the sugar particles to microscopic size so that at macroscopic level, they cannot be distinguished any more. It is important to realize here that neither has the sugar escaped the bowl nor, has it been transformed into something else. It is simply present in a different composition.

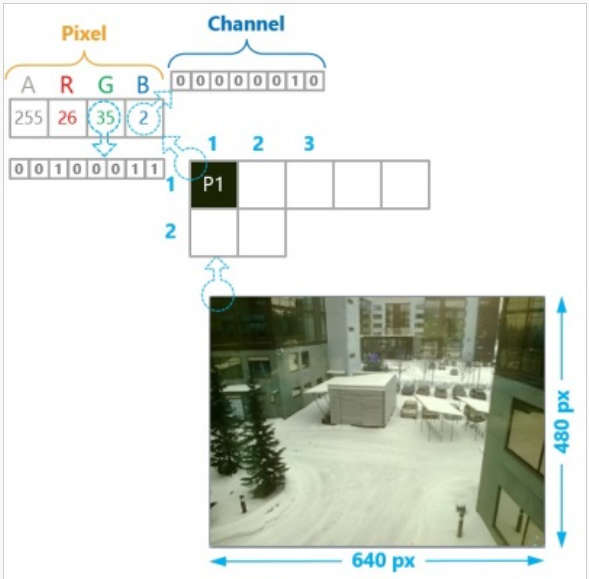
With this in mind, let's examine how we mix text (e.g. sugar) and image (e.g. water) together so that the text dissolves into the image. As in the above analogy, our "stirring" method reduces the characters of the text into a microscopic form so that they are indistinguishable among the pixels in the image. In computing terms, the atomic form to which we reduce the text is known as bit representation. As you may know, a bit is either number 1 (quantity of high value) or, number 0 (quantity of low value). So, generally speaking we convert the text and image into bits and mix the whole. You might wonder at this stage that if they are all bits, how can we later separate the character bits from the pixels bits in order to recover the embedded content. The answer to this question is that we need a systematic method (hence this article) of doing the "mixing" so that, at any point in time we can locate the position of character bits in the ocean of bits. By recognizing the positions, we can later simply access that bit and re-construct the embedded data. Note that unlike the above analogy where the volume of the mixture increases, in our case it does not. This is simply because we are replacing the bits and therefore not adding anything extra to the image. In order to understand this data embedding (mixing) process, we need to get some basics hammered down:

The anatomy of a piece of text:

In computing terms, a text is a sequence of characters known as a string. Example, the string *Hi* is the letter *H* followed by the letter *i*. At the bit (atomic) level, all characters (to simplify) are represented by a sequence of **8 bits known as a byte**. For instance, in the system (i.e. on the device), *H* is represented by the sequence 01001000 while *i* is represented by 01101001. See the technical detail sub-section for how to convert a string to its bit representation.

The anatomy of an image (pixel):

An image is made up of picture elements known as pixels. A pixel represent color information at a particular location in the image. So, when you are looking at an image on a computer screen, the image is simply a grid of colors. According to the ARGB color model, color is defined by 4 properties or, channels: Alpha which indicates the amount the transparency, Red which refers to the amount of red color, Blue which represents the amount of blue color and finally, Green which deals which green color. In combination, these 4 properties describe a particular color as well as how see-through it would be. In terms of implementing this model in images, a pixel uses a sequence of 4 bytes (or, 4 x 8 = 32 bits) to represent the color information at a given location. In other terms a byte for each channel or property. The figure below illustrates this explanation.



As it can be observed, the geography of an image can be viewed as a xy-coordinate system with the first pixel occupying the coordinate (1,1). Also, one can observe that the pixel is decomposed into 4 channels according to the ARGB color model. Finally, each channel is a byte long and is expressed as a sequence of 8 bits.

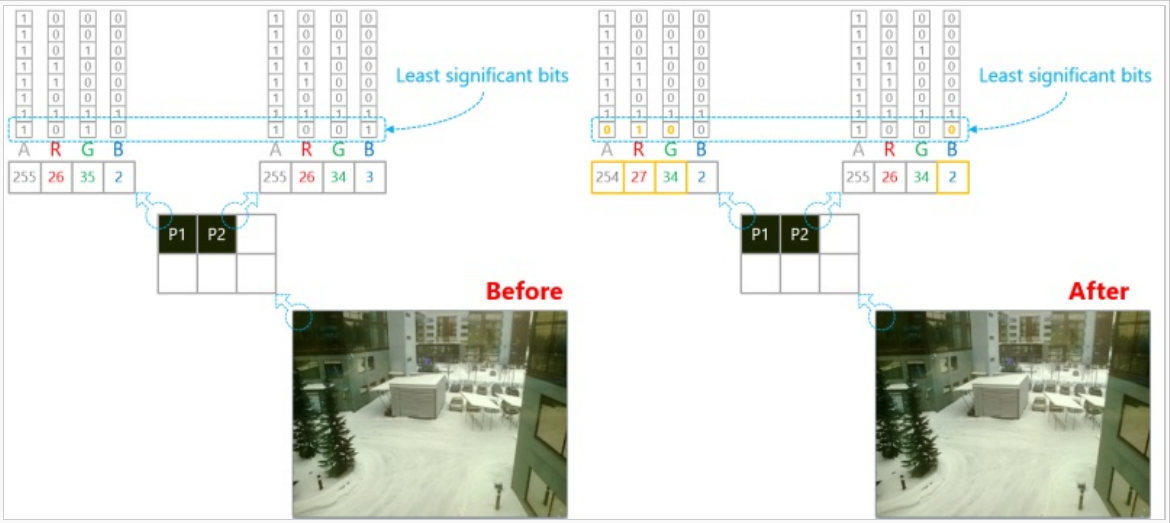
The "mixing" process:

If breaking the data into its bits representation is one aspect of the fusing (or, mixing process), then the other part constitute placing the bits of the text in pre-defined positions in the pixels so that they can be recovered later. We will refer to this process from now on as, encoding. So, the encoding method we are adopting is technically known as altering the least-significant bits of the channels of the pixels. In essence by doing so we are introducing the least amount of perturbation in the image.

In order to simplify the explanation, we are going to examine how the letter *H* is going to be dissolved into the image. As you may recall from the above explanation, letter *H* has been decomposed to the sequence 01101001. The algorithm behind Stegafoto employs all 4 channels for encoding and this means that for each character (which is 8 bits long) we are going to use (8 / 4 =) 2 pixels. Let's further assume that we will start the encoding starting with the first pixel (i.e. pixel at coordinate (1,1)). This means that we are going to need another pixel as well. For the simplicity and convenience, we are

(C) Copyright Nokia 2013. All rights reserved.

going to use the adjacent pixel. More explicitly, the pixel at coordinate (2, 1) instead of pixel (1, 2). In our schema below which illustrates the [illegible] referring to P1 and P2.



As it can be observed , if we panned out all the channels of the pixels and examine their least significant bits, we are effectively looking at a data space where we can store information. In the *before* part, if we were to decode that sequence of 8 least-significant bits we would get a character other than *H*. However, if we were to do the same with the representation in the *after* part, we would indeed get *H*. Note also the bits and bytes affected after the encoding process.

As you can recognize in the above illustration, if we consider the sequence of all least-significant bits in pixels P1 and P2, it is exactly the sequence of the binary representation of letter *H*. So, this is how we cleverly place the bits of the text so that we can easily recover them later. And at the same time make the least amount of disturbance in the color-space of the image. Without getting into much detail, we use the odd-even nature of the value of the channel in order to concretely make changes to the least-significant bits. If you want to know more see the technical details accompanying this section.

Recording audio (as in the above video) is the same process except that we have to convert the captured sound from the microphone into a binary sequence. This is done in the following manner:

1. Get the recorded sound bite as PCM data and apply the relevant header in order to convert it to WAV. The algorithm for converting PCM to WAV can be found here.
2. Next take WAV data as a sequence of bytes and convert it to Base64 encoding in order to get a string representation.
3. Take the string and pass it to the convert-to-binary method mentioned above.

Technical details

In this sub-section I am going to give code snippets and explanations (in the comments) on how the above algorithm has been implemented. So,

Converting a String to an array of bits (boolean):

```
private bool[] ConvertStringToBitArray(String str)
{
    bool[] bitArray = new bool[8 * str.Length];
    int j = 0;
    foreach (char c in str)
    {
        for (int i = 0; i < 8; i++)
        {
            bitArray[j + i] = (((c >> (7 - i)) & 0x00000001) == 1 ? true : false);
        }
        j += 8;
    }

    return bitArray;
}
```

Encoding a bit in a pixel:

```
/**
 * Method below is called from the following context:
 *
 * for (int i = 0; i < embeddedDataAsBitArray.Length; i++)
 * {
 *     // pngImage.Pixels[i] is referring to a channel in the pixel. E.g. when i%4 == 1 we are accessing the Red channel of the pixe
 *     pngImage.Pixels[i] = Encode(embeddedDataAsBitArray[i], pngImage.Pixels[i]);
 * }
 */
private byte Encode(bool bit, byte val)
{
    if (val % 2 == 1)
    {
        if (bit == false) // => byte is odd and we would like to write a 0
        {
            val--;
        }
    }
    else
    {
        if (bit == true) // => byte is even and we would like to write a 1
        {
            val++;
        }
    }
    return val;
}
```


(C) Copyright Nokia 2013. All rights reserved.

Converting a PCM to a WAV according to the algorithm found here:

```
Encoding ENCODING = System.Text.Encoding.UTF8;

// User has pressed the 'Record Audio' button:
private void RecordAudio(object sender, GestureEventArgs e)
{
    e.Handled = true;
    Debug.WriteLine("Recording Audio ...");
    if (_mic.State == MicrophoneState.Stopped)
    {
        Debug.WriteLine("Audio Sample Rate: {0}", _mic.SampleRate);
        _audioStream.SetLength(0);

         // Write a header to the stream so that we can have a WAV file:
         // This document was used to create header: https://ccrma.stanford.edu/courses/422/projects/WaveFormat/
        _audioStream.Write(ENCODING.GetBytes("RIFF"), 0, 4);
         // This will be filled later once the recording is done. I.e. we would know the size of data.
        _audioStream.Write(BitConverter.GetBytes(0), 0, 4);
         // WAVE is made up of 2 parts: Format (fmt ) which describes the audio data such as, channels,
         // bitrate, etc and then (data) which is the actual audio data.
        _audioStream.Write(ENCODING.GetBytes("WAVE"), 0, 4);
         // Writing the Format part:
        _audioStream.Write(ENCODING.GetBytes("fmt "), 0, 4);
         // This indicates the size of the 1st part that will follow this segment. 16 implies that audio is in PCM format.
        _audioStream.Write(BitConverter.GetBytes(16), 0, 4);
        _audioStream.Write(BitConverter.GetBytes((short)1), 0, 2);
        _audioStream.Write(BitConverter.GetBytes((short)1), 0, 2);
        _audioStream.Write(BitConverter.GetBytes(_mic.SampleRate), 0, 4);
        _audioStream.Write(BitConverter.GetBytes(_mic.SampleRate * BYTES_PER_SAMPLE), 0, 4);
        _audioStream.Write(BitConverter.GetBytes((short)BYTES_PER_SAMPLE), 0, 2);
        _audioStream.Write(BitConverter.GetBytes((short)BITS_PER_SAMPLE), 0, 2);
         // Writing the Data part:
        _audioStream.Write(ENCODING.GetBytes("data"), 0, 4);
         // The size of the data will be known once the recording is done.
        _audioStream.Write(BitConverter.GetBytes(0), 0, 4);
        _mic.Start();
        StopRecordingButton.Visibility = Visibility.Visible;
        RecordAudioButton.Visibility = Visibility.Collapsed;
    }
}

// User has pressed on the 'Stop Audio Recording' button:
private void StopRecording(object sender, GestureEventArgs e)
{
    e.Handled = true;
    Debug.WriteLine("Stop recording Audio ...");
    if (_mic.State == MicrophoneState.Started)
    {
        _mic.Stop();
        _audioStream.Flush();
        long endOfStream = _audioStream.Position;
        int streamLength = (int)_audioStream.Length;
        _audioStream.Seek(4, SeekOrigin.Begin); // Move the 'cursor' to the 1st place holder in the header of the WAVE format.
        _audioStream.Write(BitConverter.GetBytes(streamLength - 8), 0, 4); // Insert the size of the stream - the WAVE header part.
        _audioStream.Seek(40, SeekOrigin.Begin); // Move the 'cursor' to the 2nd place holder which is 36 bits away.
        _audioStream.Write(BitConverter.GetBytes(streamLength - 44), 0, 4);
        _audioStream.Seek(endOfStream, SeekOrigin.Begin);
        Debug.WriteLine("Recorded {0}s of audio", _mic.GetSampleDuration(streamLength));
        RecordAudioButton.Visibility = Visibility.Visible;
        StopRecordingButton.Visibility = Visibility.Collapsed;
        // Converting WAV into String format:
        _capturedHiddenData = System.Convert.ToBase64String(_audioStream.ToArray());
        if (!String.IsNullOrEmpty(_capturedHiddenData))
        {
            _capturedType = AUDIO_TYPE;
            ShareImageButton.IsEnabled = true;
        }
        else
        {
            _capturedType = UNDEFINED_TYPE;
            ShareImageButton.IsEnabled = false;
        }
    }
}
```

Extracting embedded data

In this section I am going to discuss the method of retrieving the embedded data from the image. For this purpose, I imagined that a probable context where the user would do that would be while browsing the photos hosted on a web-service. In that sense, the viewer application would be a web-browser. Having said that, the photos could be hosted locally (as in my examples above) and hence the web-browser would thus be the ideal tool for all situations. In an earlier experiment, I implemented the "decoding" with the basic <canvas> element but it was not so successful. It turns out that the get-pixels function (getImageData) of the <Canvas> object returns premultiplied alpha pixels which for us means that the embedded data is destroyed. Some vendors provide flags to turn off the premultiplier property but, this means that we have to have almost bespoke solutions for each browsers and more than often turning off the flags does not even help. So, then I decided to take the WebGL route and this was much simpler (I tested solution on Firefox and Chrome and the script worked flawlessly without any alterations.). Note that, Internet Explorer does not support WebGL but, an equivalent solution can be cooked with Silverlight 5. You can verify if your browser supports WebGL by visiting the can I use website.

The algorithm for this part of the solution is equally simple:

1. First prepare your canvas so that it can leverage the WebGL APIs.
2. Get a reference to the resource (i.e. URI to the image) and render it as a texture on the canvas.
3. Then, read the pixels of the texture (**not** off the canvas via the getImageData() method) and create a binary sequence out of the ARGB channels of the pixels based on the even-or-odd nature of those values.
4. Finally, convert the sequence of bits into a String and process according to the type of data we are dealing with. If the type of data is:
    1. pure text then display it somewhere (e.g. step 5 in the Introduction of this article).
    2. is audio, prepend "data:audio/wav; base64," to the data and set that string as source to the <Audio> element. As the WAV data was encoded in Base64 by the Lens and media elements in HTML support data-urls, subsequently when the user presses on the play button, the sound comes through.

 (C) Copyright Nokia 2013. All rights reserved.

Technical details

Because this solution permits the user to embed either text or audio, the system must be able to distinguish one from the other in order to deliver the appropriate experience to the user. This issue is tackled by simply hardening the format in which embedded data is encoded. For the sake of the demo a very simple scheme by prepending a header to the data before it is encoded. The header has the following structure: ST#<type_of_data>#<length_of_actual_data_in_bytes>#. For example, in the case described in the Introduction of the article, the encoded data is: ST#T#2#Hi. With this in mind, let's have a look at some code snippets to see how the above algorithm has been implemented. So,

Preparing <Canvas> to use WebGL:

```
var _canvas = null;
var _gl = null;
var _shaderProgram = null;

// Creates the shader based on the ID of the shader description found in the DOM:
function GetShader(id) {
  var shader;

  var shaderScriptNode = document.getElementById(id);
  if (!shaderScriptNode) {
    throw "Could not find a shader script descriptor with ID [" + id + "]";
  }

  // Walk down the node and construct the shader script:
  var script = "";
  var currChild = shaderScriptNode.firstChild;
  while(currChild) {
    if(currChild.nodeType == currChild.TEXT_NODE) {
      script += currChild.textContent;
    }
    currChild = currChild.nextSibling;
  }

  // Identify the type of shader (Vertex or Fragment):
  if (shaderScriptNode.type == "x-shader/x-vertex") {
    shader = _gl.createShader(_gl.VERTEX_SHADER);
  } else if (shaderScriptNode.type == "x-shader/x-fragment") {
    shader = _gl.createShader(_gl.FRAGMENT_SHADER);
  } else {
    throw "Could not find a valid shader-type descriptor";
  }

  // Load the script into the shader object and compile:
  _gl.shaderSource(shader, script);
  _gl.compileShader(shader);
  if (!_gl.getShaderParameter(shader, _gl.COMPILE_STATUS)) {
    throw "Compilation error in script [" + id + "]: " + _gl.getShaderInfoLog(shader);
  }
  return shader;
}

function CreateShaderProgram(vsId, fsId) {
  var vs = GetShader(vsId);
  var fs = GetShader(fsId);
  var shaderProgram = _gl.createProgram();
  _gl.attachShader(shaderProgram, vs);
  _gl.attachShader(shaderProgram, fs);
  _gl.linkProgram(shaderProgram);

  if(!_gl.getProgramParameter(shaderProgram, _gl.LINK_STATUS)) {
    throw "Unable to create shader program with provided shaders."
  }
  _gl.useProgram(shaderProgram);
  return shaderProgram;
}

function InitWebGL(canvasId, VertexShaderScriptId, FragmentShaderScriptId) {
  _canvas = document.getElementById(canvasId);

  if (!_canvas) {
    throw "Could not locate a canvas element with id '" + canvasId + "'";
  } else {
    try {
      _gl = _canvas.getContext("webgl") || _canvas.getContext("experimental-webgl");
      console.log("Created WebGL context ...");
      _gl.pixelStorei(_gl.UNPACK_PREMULTIPLY_ALPHA_WEBGL, false);
      _gl.pixelStorei(_gl.UNPACK_COLORSPACE_CONVERSION_WEBGL, false);
      _shaderProgram = CreateShaderProgram(VertexShaderScriptId, FragmentShaderScriptId);
      console.log("Created Shader Program ...");
    } catch (e) {
      _gl = null;
      throw "Err: WebGl not supported by this browser.";
    }
  }
}

// Initializing the canvas called 'WorkingArea':
function Initialize() {
  try {
    InitWebGL("WorkingArea", "ImgVertexShader", "ImgPixelShader");

    // Set the canvas dimensions in the Shader Program (Vertex Shader):
    _gl.uniform2f(_gl.getUniformLocation(_shaderProgram, "uCanvasRes"), _canvas.width, _canvas.height);

    // Create a buffer for the Texture Coordinate:
    _gl.bindBuffer(_gl.ARRAY_BUFFER, _gl.createBuffer());
    _gl.bufferData(_gl.ARRAY_BUFFER, new Float32Array([0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0]), _gl.STATIC_DRAW
    var texCoordLocation = _gl.getAttribLocation(_shaderProgram, "aTextureCoord");
    _gl.enableVertexAttribArray(texCoordLocation);
    _gl.vertexAttribPointer(texCoordLocation, 2, _gl.FLOAT, false, 0, 0);

    // Create a texture inorder to load image into it later:
    _gl.bindTexture(_gl.TEXTURE_2D, _gl.createTexture());
    _gl.texParameteri(_gl.TEXTURE_2D, _gl.TEXTURE_WRAP_S, _gl.CLAMP_TO_EDGE);
    _gl.texParameteri(_gl.TEXTURE_2D, _gl.TEXTURE_WRAP_T, _gl.CLAMP_TO_EDGE);
    _gl.texParameteri(_gl.TEXTURE_2D, _gl.TEXTURE_MIN_FILTER, _gl.NEAREST);
    _gl.texParameteri(_gl.TEXTURE_2D, _gl.TEXTURE_MAG_FILTER, _gl.NEAREST);

    // Create a buffer for the rectangle that will "host" the texture:
    _gl.bindBuffer(_gl.ARRAY_BUFFER, _gl.createBuffer());
    var positionLocation = _gl.getAttribLocation(_shaderProgram, "aVertexPosition");
    _gl.enableVertexAttribArray(positionLocation);
    _gl.vertexAttribPointer(positionLocation, 2, _gl.FLOAT, false, 0, 0);
  } catch (e) {
```


(C) Copyright Nokia 2013. All rights reserved.

```
    alert(e);
  }
}
```

Defining the shader objects with HLSL so that the image can be rendered correctly as a texture:

```
<script id="ImgPixelShader" type="x-shader/x-fragment">
  precision mediump float;
  uniform sampler2D uImage;
  varying vec2 vTextureCoord;

  void main() {
    gl_FragColor = texture2D(uImage, vTextureCoord);
  }
</script>
<script id="ImgVertexShader" type="x-shader/x-vertex">
  attribute vec2 aVertexPosition;
  attribute vec2 aTextureCoord;
  uniform vec2 uCanvasRes;
  varying vec2 vTextureCoord;

  void main() {
    // The coordinate system is a different geometry to how we usually treat images. In an image the "origin" of the coordinate
    // system is on the top-left corner. In this system, the origin is at the 'center' with a [-1, 1] range. Hence, we must perform
    // the following transformation below in order to calibrate things. The end result is a coordinate system called the Clip-Space
    // coordinate system with the origin at the bottom-left.
    vec2 inCSCoordPos = ((aVertexPosition/uCanvasRes) * 2.0) - 1.0;
    gl_Position = vec4(inCSCoordPos * vec2(1, -1), 0, 1);
    vTextureCoord = aTextureCoord;
  }
</script>
```

Extractring the embedded data from the loaded texture:

```
function ConvertBitArrayToString(bitArr) {
  var str = "";
  for (var i=0; i<bitArr.length; i+=8) {
    var val = 0;
    for (var j=0, shiftCtr=7; j<8; j++, shiftCtr--) {
      val += (bitArr[i+j] << shiftCtr);
    }
    str += String.fromCharCode(val);
  }
  return str;
}

function ReadLine(lineNumber) {
  var bitArray = new Array();
  var pixelArrayInRGBA = new Uint8Array(4 * _canvas.width);
  _gl.readPixels(0, lineNumber, _canvas.width, 1, _gl.RGBA, _gl.UNSIGNED_BYTE, pixelArrayInRGBA);
  for(var i = 0; i < pixelArrayInRGBA.length; i++) {
    bitArray[i] = pixelArrayInRGBA[i] % 2;
  }
  return ConvertBitArrayToString(bitArray);
}

function DecodeAsStegaFotoImage() {
  console.log("Decoding as a StegaFoto ...");
  // Because the origin is at the bottom-left in clipspace coordinate, this means that 1st pixel
  // row of the image is actually at the very bottom of the canvas:
  var lineCounter = _canvas.height - 1;
  var line = ReadLine(lineCounter);
  var indexOfDelimeter = line.indexOf(_MESSAGE_DELIM);
  if (indexOfDelimeter > 0) {
    var arrParts = line.substring(0, indexOfDelimeter).split(":");
    var lengthOfData = parseInt(arrParts[2], 10);
    var typeOfEmbeddedData = arrParts[1];
    var data = "";
    var fromIdx = indexOfDelimeter + _MESSAGE_DELIM.length;
    var numberOfLinesToRead = ((lengthOfData * 2) + fromIdx) % _canvas.width;
    if(numberOfLinesToRead > 0) {
      for(var i=1; i <= numberOfLinesToRead; i++) {
        line += ReadLine(lineCounter - i);
      }
    }
    data = line.slice(fromIdx, fromIdx + lengthOfData);
    // 'T' means that data must be interpreted as pure text. While 'A' implies that data must be treated as a the data part of data-
    if (typeOfEmbeddedData == "T") {
      document.getElementById("EmbeddedMessage").value = data;
    } else if (typeOfEmbeddedData == "A") {
      document.getElementById("AudioPlayer").src = "data:audio/wav; base64," + data;
    }
  }
}
```

Downloads

Warning: The code provided with this article is a prototype "proof of concept". It is fragile - and will crash if you record audio for much more than 4-5 seconds.

The source code can be downloaded from here: Media:Stegafoto SRC.zip

Conclusion

I hope that the way I have structured the article makes the approach clear for both programmers and non-programmers. Hopefully, the videos and images (schemas) support your understanding on how the Stegafoto app works.

(C) Copyright Nokia 2013. All rights reserved.

 (C) Copyright Nokia 2013. All rights reserved.