Fast QR code detection with GPU shading technology

Alberto Franco
University of Padua
⟨afranco87@gmail.com⟩

***Abstract*—The aim of the project is to create an application that detect QR[1] codes in a real-time fashion using GPU technology. The original idea was to make this application entirely run on mobile devices, due to some technical problems application has been developed for desktop machines in Java with the aid of OpenGL shading laguage GLSL. A future port for mobile systems (i.e. Android OS) should not be that hard.**

***Index Terms*—QR detection, Augmented Reality, Computer Vision, OpenGL, GLSL, Shader, Real-time computer graphics.**

I. Introduction

What we would like to explore is a different approach to detect QR codes into live stream of images. This idea is excellently implemented into a well-known library (ZXing, zebra crossing [5]) that detect and decode a QR code into an image or into a video/live stream. Many applications (such as QR droid or similar) relies onto this library to let user interact with QR codes.

Our approach is to move onto the GPU the detection of the code, so the calculation could be parallelized and take out by the graphic processing unit of a device. The idea is that at some point mobile GPUs will become faster than CPUs for such processing as just happend for desktop market. In that moment a similar approach will be faster than the one in CPU.

In this report we will present first an overview of the whole procedure, then we will dive into the various step. A section will explain how edge detection is performed and other sections will explain how interesting feature of QR codes are isolated. At the end of the day we will draw some conclusion on the research.

II. Our Approach

In this section we will present the approach used to detect QR codes in the scene. While the first steps of the processing are quite standard digital image processing techniques the end of our pipeline has been thought from scratch.

To get the edges we perform a simple Canny[3] edge detection as seen in [1] with GLSL through three passes. Each pass is rendered onto a framebuffer and stored in a texture to use it in the next step. Actually the framebuffer used is just one and different textures are stiched on it. Our passes to perform Canny edge detection are:

1) gaussian blur with convolution filter;
2) gradient calculation with sobel operator;

[1]Quick Response

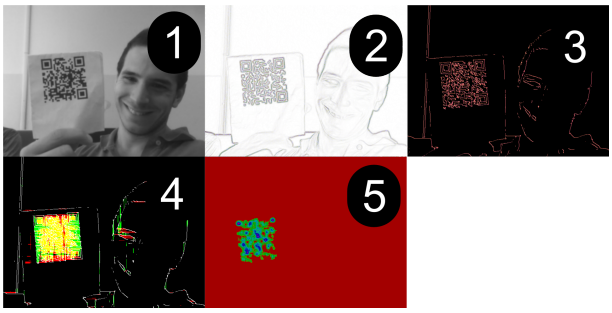

Fig. 1. The entire sequence of passes performed

3) canny edge detection.

After corners are detected further analysis and scoring of the areas of the image are performed. The idea is to try to detect the topology of the QR code, using particular features that are shape of QR.

III. Canny Edge Detection

We implemented a classical version of this algorithm, it is performed in a multi-stage fashion. It leds to a *thin edge* image as seen in picture. What is done is to read the stream of the webcam and copy it onto a texture then three shaders transform the original image. The first step is to blur the image, this is done to reduct noise (denoising) to perform a better gradient calculation to avoid to detect edges that are not such that but just sampling noise or small irrelevant features[4]. Then image

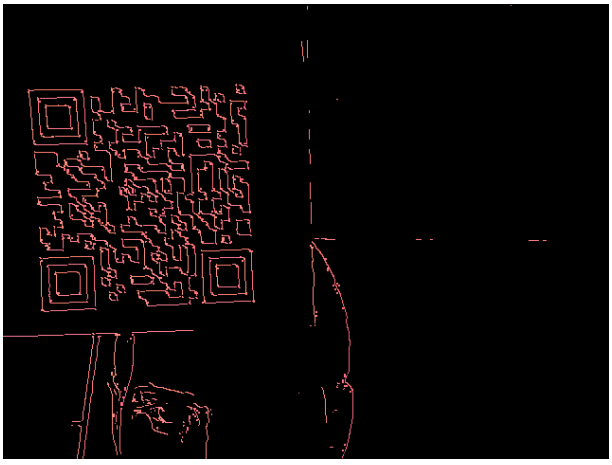

Fig. 2. Result of canny edge detection shader

gradient is calculated through sobel operator, this give us quite accuraccy in the calculation probably in mobile hardware we should change this two steps as they are too much heavy in calculation terms to be done entirely in classic fashion. This may be changed using two pass for image blurring and a less precise operator (such as Roberts' Cross or a simpler one) for gradient calculation. Finally canny edge detection are calculated with the following shader.

```
vec4 cannyEdge(vec2 coords) {
  vec4 color = texture2D(text, coords);
  color.z = dot(color.zw, unshift);
  // Thresholding
  if (color.z > threshold) {
    // Restore gradient directions.
    color.x -= 0.5;
    color.y -= 0.5;
    // Calculate offset direction
    vec2 offset = atanForCanny(color.y / color.x);
    offset.x *= texWidth;
    offset.y *= texHeight;
    // Get the two values
    vec4 forward  = texture2D(text, coords + offset);
    vec4 backward = texture2D(text, coords - offset);
    // Uncompress mag data
    forward.z  = dot(forward.zw, unshift);
    backward.z = dot(backward.zw, unshift);
    // Check maximum.
    if (forward.z >= color.z ||
      backward.z >= color.z) {
      return vec4(0.0, 0.0, 0.0, 1.0);
    } else {
      color.x += 0.5; color.y += 0.5;
      return vec4(1.0, color.x, color.y, 1.0);
    }
  }
  return vec4(0.0, 0.0, 0.0, 1.0);
}
```

Improvements from classic algorithm. We perform an eight direction search into the neighbor pixels to find local maximum. You may notice into the shader that the magnitude of the gradient is compress into two coordinates of the previous color. This is done to have more resolution in the calculation. Compression is done this way:

```
  const vec2 bitSh = vec2(256.0, 1.0);
  const vec2 bitMk = vec2(0.0, 1 / 256.0);

  vec2 pack = fract(amplitude * bitSh);
  pack -= pack.xx * bitMk;
```

and the decompression is done as you read in canny edge detection shader. We used this technique because float used for colors are 8 bit[2][6] float and to obtain more resolution we use a 16 bit float this way. Due to the same problem gradient directions are normalized onto the interval $[0, 1]$ and then the actual directions are restored. Using sobel operator we obtain values into the interval $[-4, 4]$ so we normalize them by adding four and dividing by eight. the actual direction of the gradient (this is what is interesting) is restored before any calculation.

In the end we have the result shown in picture, you may notice that there are artifacts in corner zone. This is due to 16 bit float compression, if the entire number would be preserved we don't have those artifacts but the compression leds to numerical losses and in those areas edge detection is not that precise as it would be with full 32 bit number[2].

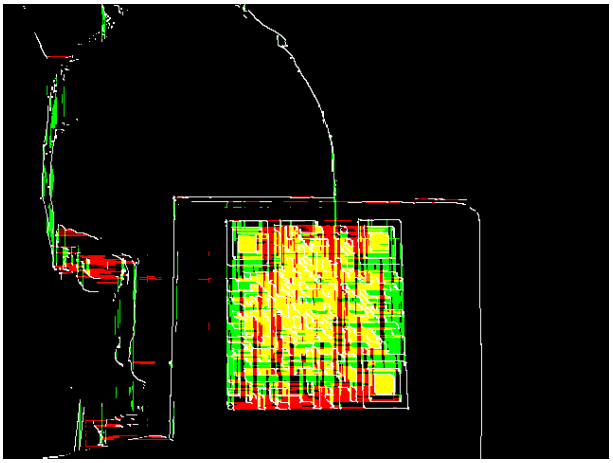

Fig. 3. Result of topology scoring

IV. QR Topology Scoring

The next step in our analysis of given image is to score areas that are possible to be into a QR code. Typically such a code has three marker for positioning the code that are black squares surrounded by a black border. Around the whole there is what is said *quiet zone* that is a wide white zone. We have focused on the positioning markers, in our thin edge image these markers are concentrical squares. In this phase we search pixel-wise for three edges on each directions. Search is done this way:

```
// Search up to window size or three edeges found
for (i = 1; i < windowSize; i++) {
  color = texture2D(texture,
              coords + searchDirection * i);
  if (color.x > 0.5) {
     // Edge found
     countLeft += 1;
     if (countLeft == 3) break;
  }
}
```

If borders are found horizontally or vertically we give a mark this way:

$$score(x, y) = \begin{cases} (1, 0) & \text{if horizontal check is true} \\ (0, 1) & \text{if vertical check is true} \\ (1, 1) & \text{if both checks are true} \end{cases}$$

In the end the image will have yellow zones where both test pass and green or red zone where just one of the two is passed. This is not sufficient to determinate where the QR code is, this first pass allow us to remove omogenous zones where is impossible to find QR codes. You may notice from the above picture that many area that are not in the code are higligted in red or green. These areas will be removed in the next step. The interesting thing in this pass is that the interesting areas are recognized with the pattern given, looking at the picture the center of positioning markers are fully highlighted in yellow, this is the result we want to achieve.

[2]To verify this fact a CPU edge detector has been developed with the exact same algorithm and the edges do not present artifacts with fully precise floats.

V. Consensus Scoring

Final step of the algorithm is the consensus scoring. We gather pixel-wise answer from previous step and calculate the consensus from neighbour pixels. Consense is evaluated in a 11x11 window using the following scores:

$$cons(x, y) = \begin{cases} 1 & \text{if answer is (1, 1)} \\ 1/8 & \text{if answer is (1, 0) or (0, 1)} \end{cases}$$

This operation is done for each pixel in the window and values are summed up and then normalized. This way of calculating the consensus led to some false positive in zones with lot of answer from vertical or horizontal test (i.e. green or red areas from input framebuffer). These false positive can be removed with a nonmaximal suppression with a decent threshold. In example application has been applied a threshold of 30 (where the maximum response is 11^2) and almost all zone with response that are not QR code are cutted off. To highlight better the variation of response calculated the final image is visualized using HSV (hue, saturation and value) color space, this does not allow to notice that the maximum of response are calculated in center of positioning marker so a gray scale print has been added to mark such fact.

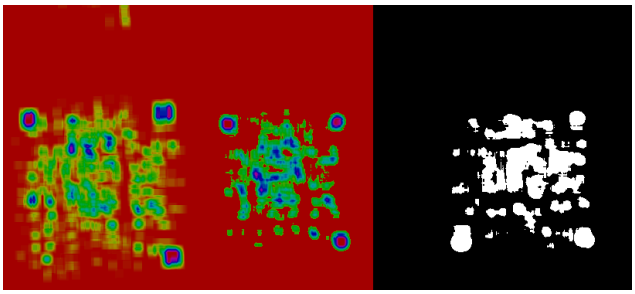

Fig. 4. Consensus left to right: normal, with nonmaximal suppression, gray scale

Potential pitfalls. This way of doing things has a number of potential pitfalls. First this approach detect and do not decode QR codes what would more interesting to do is to decode also codes to use informations stored into them.

VI. Final Image Processing

At the end of the whole step a final processing is done in order to put together the source image and the detection performed. This step is done simply searching for areas that has been detected and drawing a rectangle around peaks after nonmaximal suppression has been performed. In this step a bit of normalizing has to be done in order to correct center QR from source image to detection stream. Due to OpenGL texture clamp[6] some texture fetch are outside the image area and the pipeline returns value from the other side of the texture image. This fact leads to a shift of the image and has to be restored before rendering the final result.

Fig. 5. Webcam stream with QR detection.

VII. Conclusion and Future Works

This approach gives good results in detecting QR codes inside a live stream of images. There are some interference[3] but the overall code is detected with this technique.

The technique is not usable at current time for real case of detection and QR decoding due to his large amount of passes if compared with ZXing approach[5]. ZXing uses lot more resources, detect and decode QR with CPU processing on binarized image, searching for pattern onto it. A more efficent GPU pattern matching zxing-style could be implemented, for example in two passes can be do an image binarization and a pixel-wise search. At the end of the day, as usual, lot can be done in the future.

References

[1] James Fung, *Computer Vision on the GPU*. GPU Gems 2, Chapter 40. 2005, Pearson Education.
[2] R. S. Wright Jr., N. Haemel, G. Sellers, B. Lipchak, *OpenGL SuperBible*. 5th Edition, Addison-Wesley, 2011.
[3] J. Canny, *A Computational Approach To Edge Detection*. IEEE Trans. Pattern Analysis and Machine Intelligence, 8(6):679698, 1986
[4] R. C. Gonzales, R. E. Woods, *Digital Image Processing*. 2th Edition, Prentice-Hall, 2002.
[5] Zxing website - code.google.com/p/zxing/
[6] GLSL reference - www.opengl.org/documentation/glsl/

[3]The major responsable of interference are printed paper and objects that genereate lots of corner into canny edge detection.