
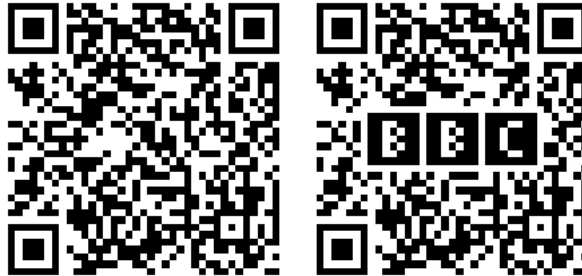


QArt Codes

Posted on Thursday, April 12, 2012. 

QR codes are 2-dimensional bar codes that encode arbitrary text strings. A common use of QR codes is to encode URLs so that people can scan a QR code (for example, on an advertising poster, [building roof](#), [volleyball bikini](#), [belt buckle](#), or [airplane banner](#)) to load a web site on a cell phone instead of having to “type” in a URL.

QR codes are encoded using [Reed-Solomon error-correcting codes](#), so that a QR scanner does not have to see every pixel correctly in order to decode the content. The error correction makes it possible to introduce a few errors (fewer than the maximum that the algorithm can fix) in order to make an image. For example, in 2008, [Duncan Robertson](#) took a QR code for “http://bbc.co.uk/programmes” (left) and introduced errors in the form of a BBC logo (right):



That's a neat trick and a pretty logo, but it's uninteresting from a technical standpoint. Although the BBC logo pixels look like QR code pixels, they are not contributing to the QR code. The QR reader can't tell much difference between the BBC logo and the Union Jack. There's just a bunch of noise in the middle either way.



Since the BBC QR logo appeared, there have been many imitators. Most just slap an obviously out-of-place logo in the middle of the code. This [Disney poster](#) is notable for being more in the spirit of the BBC code.

There's a different way to put pictures in QR codes. Instead of scribbling on redundant pieces and relying on error correction to preserve the meaning, we can engineer the encoded values to create the picture in a code with no inherent errors, like these:



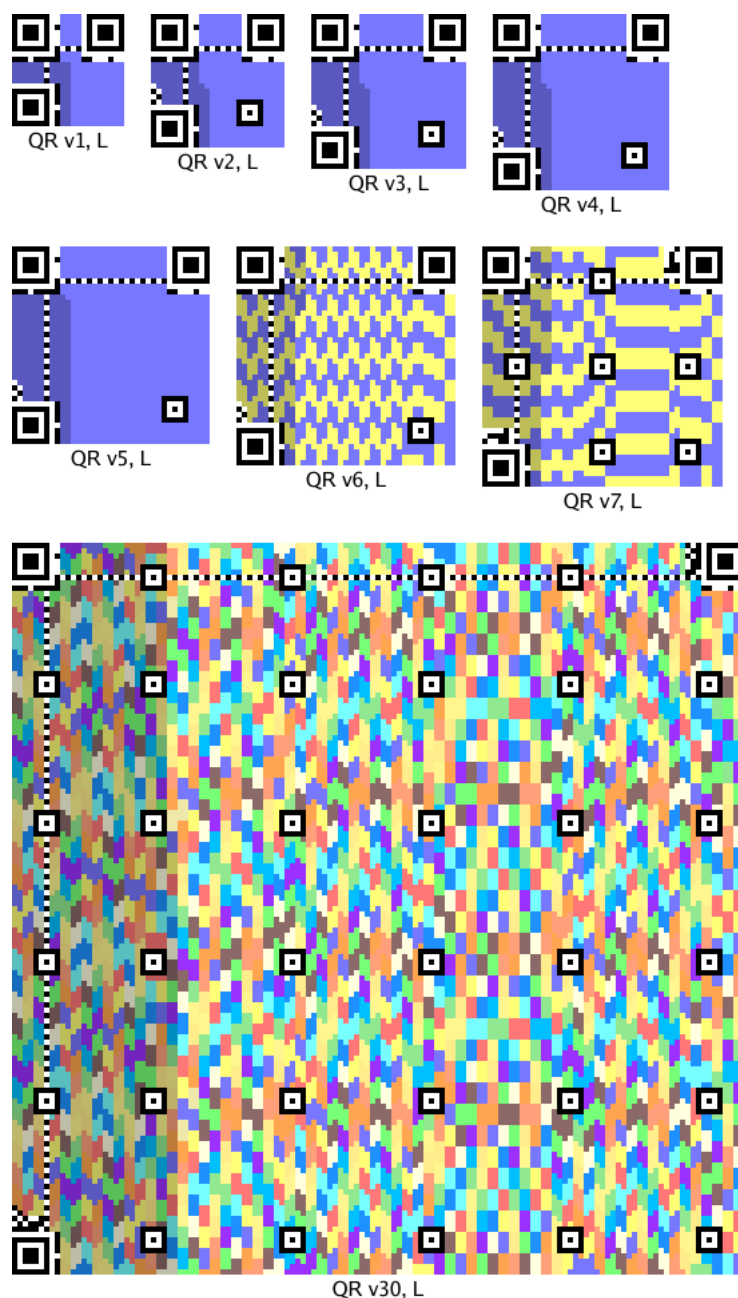
This post explains the math behind making codes like these, which I call QArt codes. I have published the Go programs that generated these codes at code.google.com/p/rsc and created a [web site for creating these codes](#).

Background

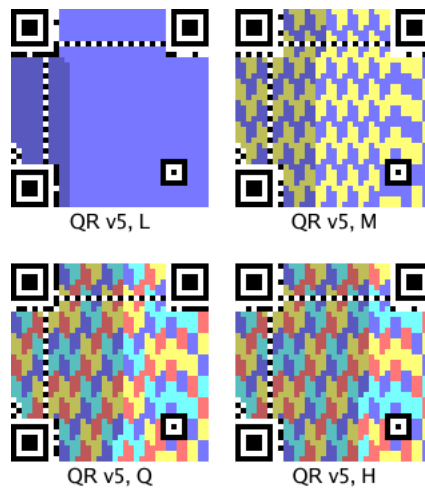
For error correction, QR uses Reed-Solomon coding (like nearly everything else). For our purposes, Reed-Solomon coding has two important properties. First, it is what coding theorists call a *systematic code*: you can see the original message in the encoding. That is, the Reed-Solomon encoding of “hello” is “hello” followed by some error-correction bytes. Second, Reed-Solomon encoded messages can be XOR'ed: if we have two different Reed-Solomon encoded blocks b_1 and b_2 corresponding to messages m_1 and m_2 , $b_1 \oplus b_2$ is also a Reed-Solomon encoded block; it corresponds to the message $m_1 \oplus m_2$. (Here, \oplus means XOR.) If you are curious about why these two properties are true, see my earlier post, [Finite Field Arithmetic and Reed-Solomon Coding](#).

QR Codes

A QR code has a distinctive frame that help both people and computers recognize them as QR codes. The details of the frame depend on the exact size of the code—bigger codes have room for more bits—but you know one when you see it: the outlined squares are the giveaway. Here are QR frames for a sampling of sizes:

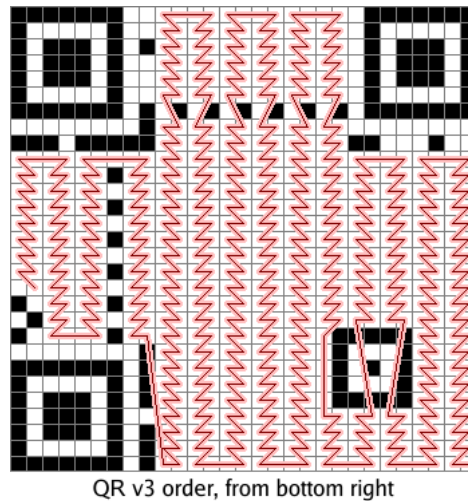


The colored pixels are where the Reed-Solomon-encoded data bits go. Each code may have one or more Reed-Solomon blocks, depending on its size and the error correction level. The pictures show the bits from each block in a different color. The L encoding is the lowest amount of redundancy, about 20%. The other three encodings increase the redundancy, using 38%, 55%, and 65%.



(By the way, you can read the redundancy level from the top pixels in the two leftmost columns. If black=0 and white=1, then you can see that 00 is L, 01 is M, 10 is Q, and 11 is H. Thus, you can tell that the QR code [on the T-shirt in this picture](#) is encoded at the highest redundancy level, while [this shirt](#) uses the lowest level and therefore might take longer or be harder to scan.

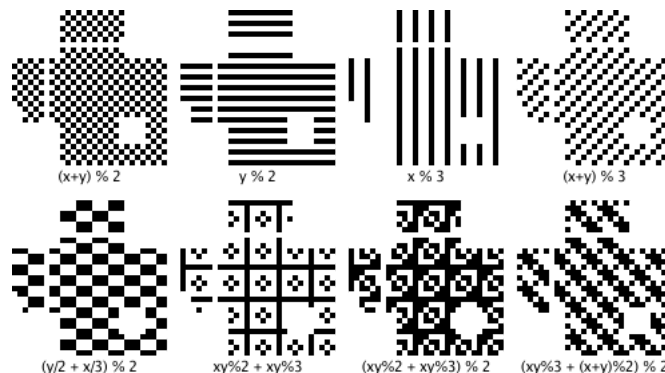
As I mentioned above, the original message bits are included directly in the message's Reed-Solomon encoding. Thus, each bit in the original message corresponds to a pixel in the QR code. Those are the lighter pixels in the pictures above. The darker pixels are the error correction bits. The encoded bits are laid down in a vertical boustrophedon pattern in which each line is two columns wide, starting at the bottom right corner and ending on the left side:



We can easily work out where each message bit ends up in the QR code. By changing those bits of the message, we can change those pixels and draw a picture. There are, however, a few complications that make things interesting.

QR Masks

The first complication is that the encoded data is XOR'ed with an obfuscating mask to create the final code. There are eight masks:



An encoder is supposed to choose the mask that best hides any patterns in the data, to keep those patterns from being mistaken for framing boxes. In our encoder, however, we can choose a mask before choosing the data. This violates the spirit of the spec but still produces legitimate codes.

QR Data Encoding

The second complication is that we want the QR code's message to be intelligible. We could draw arbitrary pictures using arbitrary 8-bit data, but when scanned the codes would produce binary garbage. We need to limit ourselves to data that produces sensible messages. Luckily for us, QR codes allow messages to be written using a few different alphabets. One alphabet is 8-bit data, which would require binary garbage to draw a picture. Another is numeric data, in which every run of 10 bits defines 3 decimal digits. That limits our choice of pixels slightly: we must not generate a 10-bit run with a value above 999. That's not complete flexibility, but it's close: 9.96 bits of freedom out of 10. If, after encoding an image, we find that we've generated an invalid number, we pick one of the 5 most significant bits at random—all of them must be 1s to make an invalid number—hard wire that bit to zero, and start over.

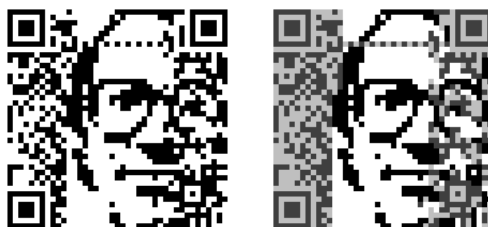
Having only decimal messages would still not be very interesting: the message would be a very large number. Luckily for us (again), QR codes allow a single message to be composed from pieces using different encodings. The codes I have generated consist of an 8-bit-encoded URL ending in a # followed by a numeric-encoded number that draws the actual picture:

http://swtch.com/pjw/#123456789...

The leading URL is the first data encoded; it takes up the right side of the QR code. The error correction bits take up the left side.

When the phone scans the QR code, it sees a URL; loading it in a browser visits the base page and then looks for an internal anchor on the page with the given number. The browser won't find such an anchor, but it also won't complain.

The techniques so far let us draw codes like this one:



The second copy darkens the pixels that we have no control over: the error correction bits on the left and the URL prefix on the right. I appreciate the cyborg effect of Peter melting into the binary noise, but it would be nice to widen our canvas.

Gauss-Jordan Elimination

The third complication, then, is that we want to draw using more than just the slice of data pixels in the middle of the image. Luckily, we can.

I mentioned above that Reed-Solomon messages can be XOR'ed: if we have two different Reed-Solomon encoded blocks b_1 and b_2 corresponding to messages m_1 and m_2 , $b_1 \oplus b_2$ is also a Reed-Solomon encoded block; it corresponds to the message $m_1 \oplus m_2$. (In the notation of the [previous post](#), this happens because Reed-Solomon blocks correspond 1:1 with multiples of $g(x)$. Since b_1 and b_2 are multiples of $g(x)$, their sum is a multiple of $g(x)$ too.) This property means that we can build up a valid Reed-Solomon block from other Reed-Solomon blocks. In particular, we can construct the sequence of blocks b_0, b_1, b_2, \dots , where b_i is the block whose data bits are all zeros except for bit i and whose error correction bits are then set to correspond to a valid Reed-Solomon block. That set is a [basis](#) for the entire vector space of valid Reed-Solomon blocks. Here is the basis matrix for the space of blocks with 2 data bytes and 2 checksum bytes:



The missing entries are zeros. The gray columns highlight the pixels we have complete control over: there is only one row with a 1 for each of those pixels. Each time we want to change such a pixel, we can XOR our current data with its row to change that pixel, not change any of the other controlled pixels, and keep the error correction bits up to date.

So what, you say. We're still just twiddling data bits. The canvas is the same.

But wait, there's more! The basis we had above lets us change individual data pixels, but we can XOR rows together to create other basis matrices that trade data bits for error correction bits. No matter what, we're not going to increase our flexibility—the number of pixels we have direct control over cannot increase—but we can redistribute that flexibility throughout the image, at the same time smearing the uncooperative noise pixels evenly all over the canvas. This is the same procedure as Gauss-Jordan elimination, the way you turn a matrix into row-reduced echelon form.

This matrix shows the result of trying to assert control over alternating pixels (the gray columns):

[illegible]

The matrix illustrates an important point about this trick: it's not completely general. The data bits are linearly independent, but there are dependencies between the error correction bits that mean we often can't have every pixel we ask for. In this example, the last four pixels we tried to get were unavailable: our manipulations of the rows to isolate the first four error correction bits zeroed out the last four that we wanted.

In practice, a good approach is to create a list of all the pixels in the Reed-Solomon block sorted by how useful it would be to be able to set that pixel. (Pixels from high-contrast regions of the image are less important than pixels from low-contrast regions.) Then, we can consider each pixel in turn, and if the basis matrix allows it, isolate that pixel. If not, no big deal, we move on to the next pixel.

Applying this insight, we can build wider but noisier pictures in our QR codes:



The pixels in Peter's forehead and on his right side have been sacrificed for the ability to draw the full width of the picture.

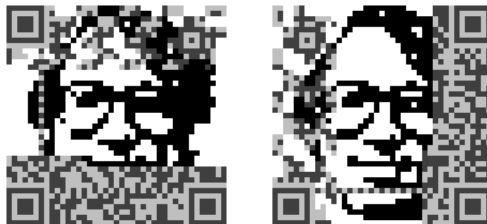
We can also choose the pixels we want to control at random, to make Peter peek out from behind a binary fog:



Rotations

One final trick. QR codes have no required orientation. The URL base pixels that we have no control over are on the right side in the canonical orientation, but we can rotate the QR code to move them to other edges.
























Further Information

All the source code for this post, including the web server, is at code.google.com/p/rsc/source/browse/qr. If you liked this, you might also like [Zip Files All The Way Down](#).

Acknowledgements

Alex Healy pointed out that valid Reed-Solomon encodings are closed under XOR, which is the key to spreading the picture into the error correction pixels. Peter Weinberger has been nothing but gracious about the overuse of his binary likeness. Thanks to both.

Comments? Please join the [Google+ discussion](#).

-  [Gustavo Niemeyer](#) (3 years ago) Wow, this is brilliant. I had missed the original post with the background somehow. Great read, thank you.
-  [Aron Swab](#) (3 years ago) I am always making Qr codes for sites I am apart of; this might have to be a new challenge of mine.
-  [Ralph Corderoy](#) (3 years ago) +[Peter Weinberger](#), the W in AWK and the face in that QR code above, is here on G+ in case any want to add him to circles. He's posted little publicly so far but perhaps a wider audience may encourage more. :-)
-  [Jeff Wendling](#) (3 years ago) I especially liked how you used the fragment to smuggle in the arbitrary data. Wonderful.
-  [JP Sugarbroad](#) (3 years ago) Have you considered using uppercase to reduce the fixed region?
-  [stan whyte](#) (3 years ago) let me suggest great QR tool at <http://www.qrhacker.com/> and if you need a hosted business card (where you can point your great QR code) <http://mybest.tel>
-  [Russ Cox](#) (3 years ago) +[JP Sugarbroad](#), I did look into that, but I wasn't thrilled about the [HTTP://ALL.CAPS](#). Maybe most damning, that alphabet doesn't have a #, so you have to do server side tricks to arrange for the URL to be valid. It would work, but I've found using short URLs to be easier and look nicer.
-  [Brad Parks](#) (3 years ago) Brilliant! And I love the deep dive into how they work. Thank you.
-  [Carrieanne Rowland](#) (3 years ago) if u look at the code properly, it looks like a mans face with glasses on!!
-  [Jason Phoenix](#) (3 years ago) Thank you! Informative and highly useful for those of us that desire some aspect of human-readability along with our QR.
-  [Alejandro Velarde](#) (3 years ago) awesome!
-  [Ken Ferry](#) (3 years ago) Wow. When you posed the xkcd code, I assumed you were relying on the error correction to bail you out. Very cool!
-  [Christopher Bulle](#) (3 years ago) that is well clever, thank you very much.
-  [Marko Veelma](#) (3 years ago) I have done something like this manually and it is damn hard to stay in allowed error limits. It is cool that somebody took this topic and show us how to make systematic changes without affecting data.
-  [Alexander Gallego](#) (3 years ago) You should revisit the idea of the technical magazine you once posted on your blog. I'd buy a subscription !
-  [Brock French](#) (3 years ago) Happy to have stumbled upon this before the 503. Great work.
-  [Russ Cox](#) (3 years ago) Un-503'ed, sorry about that.
-  [Russ Cox](#) (3 years ago) Yes, if the image you upload has transparent pixels then they get treated as don't care. Also, areas of low contrast in the image get treated as "don't care as much".
-  [Mark Berry](#) (3 years ago) The lack of # in the Alphanumeric (ALL CAPS) alphabet doesn't seem to be a deal breaker - you just have to encode the # in byte encoding mode (of course you lose some of the gain by having to insert a mode switch code), but saving 2.5 bits per character in the URL means the break even point is reached even with short URLs.

Of course, QArt codes are mostly about how it looks, and it would be interesting to see if deliberately using variant/non optimal encoding of the URL portion (the bit you state you have no control over) could actually give a better image overall - for example placing an encoding mode indicator (plus the length value of 1) between each character in the URL would give lots of zero bits, which might look better.

I also see that there is an end of message encoding mode. How do decoders cope if they encounter this before the data has been exhausted? At the moment you produce your image as

URL + # + switch to numeric mode + data that shows image but is interpreted as a non existent anchor

Why not try

URL + end of message mode indicator + binary data of image that is hopefully simply ignored

(Since encoding necessarily uses a variable number of bits that do not necessarily exactly fit the square format of the code, then that end of message mode indicator would appear to be needed in all data streams, so the question is do any decoders go wrong if they encounter such a code "unrealistically early" in the data stream?)

From your description, the QR code is actually

URL + image data + error correction


I know that it wasn't the point of your article, but seeing as we don't care about the image data being interpreted correctly when decoded (in your original case since it's just an anchor that is not sent to the server) could you not simply consider this as


URL + # + image data + error stuff for the URL + error stuff for the image

Since the error stuff for the image is correcting data that we don't care about beyond it's visual appearance, why not simply make that bit of the error stuff hold whatever you want - it will "correct" data that you are effectively discarding anyway.

 **Erik Westermann** (3 years ago) Can someone explain how to get this working under OS X? I installed Go and got the latest version of the QR generator, yet when I try to build (using command: go build qr.go png.go) I always get the same error:

qr.go:15:2: import "code.google.com/p/rsc/qr/coding": cannot find package


 **Mathieu Lonjaret** (3 years ago) +**Erik Westermann** How did you acquire qr.go? if you just set your \$GOPATH and then do 'go get code.google.com/p/rsc/qr' it should fetch everything needed.

 **Mathieu Lonjaret** (3 years ago) Nice read, and fun to play with. Unfortunately, I was unable to make a pretty enough looking gopher QArt (with the running one, not just the face). It was just too much pixelated... I haven't tweaked the code though, just tried directly on <http://research.swtch.com/qr/draw>


 **Chris Edwards** (3 years ago) Amazing. I am going to write an article on my QR code blog directing everyone to check out this article. Very cool!

 **Jeffrey Baker** (3 years ago) Bonus mission: animated GIF which is the same valid QR code on every frame.

 **Lucas Brito Arruda** (3 years ago) Amazing!


 **Kevin Baker** (3 years ago) Nice work extending the image into the Reed-Solomon data! If you violate the spec even more, and use oversized padding data to encode the image instead of a real data section, you can do away with the actual encoding being represented to the user and eliminate the URL cruft.


This is the approach I took at <http://www.qrpixel.com> . Maybe I will look into reusing the ECC after all...


 **Mark Grimm** (3 years ago) Any chance of changing the app to allow user-defined parsing besides a # sign? I'm interested in experimenting with other data formats like contact info and being able to change the syntax to ignore the garbage data in different formats would be nice. Maybe include a way to move the garbage data to the front of the entered text as well?

 **Russ Cox** (3 years ago) +**Mark Berry**, to respond to your excellent points:

1. (zeros) you definitely get patterns in the QR code if you arrange for all zeros, but they're not single color patterns, because of the XOR mask being applied.
2. (post-terminator patterns) Indeed, that sounds like what +**Kevin Baker**'s qrpixel.com is doing very nicely. Technically that's a violation of the spec, which prescribes a specific repeating 2-byte fill, but if all the readers you care about don't check the fill, then I agree it produces much nicer codes.
3. (less-than-perfect RS blocks) That's an interesting idea, but it is complicated by a few things. First, the default code size on the site is only a single RS block. Second, even if you use multi-block sizes, the bytes from the multiple blocks are interlaced before being laid out in the code, so the last RS block is not a contiguous region in the image. Third (and credit to +**Alex Healy** for this), if two RS encodings are distance 2d apart, then you run the risk of creating a code that is exactly halfway between them, which a decoder might reject as ambiguous. Also, there exist byte sequences that are >d corrections away from any valid RS block, and a decoder would be unable to handle those either.


 **Russ Cox** (3 years ago) +**Kevin Baker** Very nice site. Since the XOR of the data+checksum pixels in two QR codes produces another valid QR code, it would actually be plausible to do live editing of the code. After encoding a particular URL, the server could send down, for each pixel that can be isolated, the pixel values that have to be flipped along with it, and then in the browser when the user clicks a pixel the local JS could update the relevant pixels to preserve the correctness of the code.

 **Russ Cox** (3 years ago) +**Mark Grimm**, I'm not planning to work on the editor anymore, but the code is available if you want to play with it (link in the post).


 **Kevin Baker** (3 years ago) +**Russ Cox** Yeah, I was initially quite worried about being exactly to spec but I spent a lot of time testing with reader apps on different platforms and none of them seemed to verify the correctness of the padding. Also I found a code from qr-designer.de featured on the QR code Wikipedia article so I figured it would be ok as they use a similar method commercially.

I really like the idea about live editing, it would result in a much better user experience! I am also testing a few different masks on generation as well to have the resulting image have similar finder pattern / filled block avoidance as normal QR codes, but there's no reason that couldn't be done client side as well.

The generator that is there now is just the output of a quick messy hacked up python script to get something minimal up. I guess I need to start finding time to work on a v2.0.


 **Muhammad Hakim Asy'ari** (3 years ago) I clone your qart project (from googlecode) and try it in the local appengine but failed to compile with error message: bad import "syscal"
any idea how to make it work ?

thanks

 **Russ Cox** (3 years ago) cd ../code.google.com/p/rsc/app
../mkapp

Then point app engine at 'tmp'.

The mkapp script generates a directory tree named 'tmp' that will have the necessary packages and only those.

 **Martin Strauss** (3 years ago) Not sure I completely followed the discussion about isolating pixels. Is the idea to mark a set of pixels that we will insist on setting a certain way, in which case the other pixels are determined so that the overall codeword is correct?


In any case, here are some more suggestions. Instead of marking single pixels as don't care, find pairs of neighboring black and white pixels that we don't mind swapping. This might arise, say, in a textured grayscale part of the image. Or consider a line with slope 1/2, where the rendered heights might be (0,+,1,+,2,+,3,+), and where each plus represents a height that needs to be equal to one of its neighbors, but we don't care which. Artistically, swapping neighboring black and white pixels may be better than changing a pixel.

Swapping neighboring opposite-valued pixels amounts to XORing with a 2-pixel rectangle. After the artist marks sufficiently-many such rectangles (in a live-editing session), the software needs to find a valid R-S codeword w supported on the rectangles, i.e., w is zero off of the rectangles and each rectangle is either 11 or 00.


This can potentially be repeated at coarser scales.


Finally, if the artist is willing to shift the image up/down left/right, into, 4 up/down choices and 4 left/right choices, that's 2 bits each. The choice of mask gives another 3 bits and the rotation another 2 bits, for 9 bits total, which can be used to match parity with zero degradation of the (shifted) image. The point is that rotation not only moves the uncontrolled bits

to the side we want, but gives us effectively another independent chance to hide the parity checks unobtrusively, since we are effectively trying to code a new (rotated) image instead of the original. 9 bits is not much, but it adds up...

 [Martin Strauss](#) (3 years ago) Another possibility is as follows. Start with a relatively low-resolution image. Replace each white pixel in the original image with a 2-by-2 square of three white and one black pixel, and replace each black pixel in the original image with a 2-by-2 square of three black and one white pixel. Then make a qr-code from the result. Which orientation to use in each 2-by-2 superpixel will be left up to the encoder, which can choose the orientation to encode parity checks. So, in the final image, each 2-by-2 rectangle takes 4 bits in the qr-code and provides one bit of image and two bits of parity check. This is plenty of parity check rate to encode the image and the URL.

Of course, there are other schemes for handling grayscale. What would make sense?

 [Russ Cox](#) (3 years ago) The blog post is about how to control individual pixels. Once you know you can control individual pixels, there are many things you can do. It's not true that you can just lose one out of every 2x2 pixel square and still reconstruct the image. That's not how things are laid out: a quorum of the Reed-Solomon *bytes* (not bits) have to be read correctly, and the bytes may not line up with what you want to change.

 [Martin Strauss](#) (3 years ago) First, I should say that I really enjoyed the original post!


I'm not suggesting that we lose one pixel out of each 2x2 square and hope to recover.

My claim is the following. Suppose we have an image made up entirely of 2x2 squares with 3/1 black to white or white to black. Then it is possible to rotate the 2x2 squares independently so that the result will be a valid Reed-Solomon codeword (at modest error correction level). Furthermore, if we have such an image AND a URL, we can rotate the 2x2 squares in the image only, leaving the URL alone, and get a valid Reed-Solomon codeword.


So we would lose control of a non-constant 2x2 square by letting it rotate out of our control, rather than losing control of a single pixel by having it flip. I'm proposing that we might as well lose control of all the 2x2 squares this way, though there are other things one can do.


There *is* a bit/byte issue with what I wrote above. I think we still get something, though a somewhat weaker than I claimed. I'll check again. (We need the 2x2 squares to span multiple bytes.)

The advantage over the original is that there would no longer be explicit parity checks in the left few columns. The image degradation would be limited to one pixel per 2x2 square. This may be similar to degrading the image at random pixels as in the blog, except that this method would encode each original pixel by a 2x2 square with uniform average intensity of 1/4 for white and 3/4 for black.

 [Adam Lewis](#) (2 years ago) This is brilliant, but I can't use it because of the # + switch to numeric mode after the URL. Can you / How can I generate an Art QR Code with just the URL and not the stream of numbers after?

 [Patrick Van Renterghem](#) (2 years ago) Great post. Are there on-line web sites that generate Q-Art codes ?


 [Jayme Christensen](#) (2 years ago) Is there any reason the image data can't be grayscale on the final qr code?

 [Russ Cox](#) (2 years ago) There are lots of things you could do, like gray or color or just changing pixels, but that would violate the QR spec. The goal here was to work within the bounds of the spec.

 [Alexander Walther](#) (2 years ago) There's another nice solution at <http://www.qrpxel.com/> where you can draw pixels. Looks like it's even possible to get a url without #anchorstreamthing


 [Rene Hermenau](#) (2 years ago) Hi Russ,

that's a great resource. Thank you for sharing it to us. If you're interested, take a look at <http://www.free-qr-code.net/design-qr-code.html>. That design qr code generator works similar, but uses images as background images and modifies the foreground color of the code, so that it fits and looks like a handcrafted design code :)

 [Laurent Chener](#) (about a year ago) Hi Russ,

First of all, thank you very much for sharing this, so well explained! I'm myself playing with QRcodes (<http://www.qrcodebox.com>) and thanks to you I now also discover the "go" language :-). But about that I downloaded the last go for windows, and did "go get code.google.com/p/rsc/qr/" to start feeding my brain with your lines, but then, well, I don't know what's next :-/. What are the commands I should run under go for windows to start the web app?

Thanks again!


 [Vinothkumar G](#) (about a year ago) Hi All

This is a fantastic work. I felt it more interesting and am trying to implement the same with C++ . I am bit confused with the logic behind this work. If possible can any one please share the algorithm to convert the image to numbers at the end of the URL ??? .Awaiting for some positive response !!

for
(EG) [http://swtch.com/qr#\"267874727342436226429939649810714342040213724436313379488430015917416174994196691682682838225386007902207599130498835478788244314682674](http://swtch.com/qr#\)
I meant the numbers proceeding after #

Thanks Again !!

 [Russ Cox](#) (about a year ago) The code is in <https://code.google.com/p/rsc/source/browse/qr/web/play.go>.

 [Mahad Azad](#) (5 months ago) This is great. I also made a tool for qr codes a very advanced one at <http://www.qrunched.com> its also available for purchase. Have look. I hope you will like it