Code Golfing Considered Harmful

Stefan Gustavson 2022-02-18, slight update 2025-05-06

If the concept of “code golf” is unfamiliar to you, it is an intellectual exercise among programmers to write code that is as *compact* as possible – to get a certain task done while using the least amount of characters in the source code. The practice is as old as programming, and akin to most other competitive activities invented by humans it imposes rules, some inherent to the programming language, and some decided upon in an often arbitrary manner. Also, like competitive sports, it really has nothing to do with real world activities, even if it happens to involve tools and techniques that do have “useful” applications. Note that I put “useful” in quotation marks, because code golfing can be very useful in a specific, narrow sense of the word. It is an intellectual exercise, performed for personal satisfaction and aesthetic value, and many human activities are performed for exactly those reasons and nothing else. That does not mean that they have no value – they can have *great* value. It does mean, however, that their value is limited in scope and should not be overstated.

Because I happen to be watching Olympic figure skating as I am writing this, let’s take that as an example of a competitive activity invented by humans. The obvious origin of figure skating is ice skating, which used to be a means for people in cold countries to travel quickly and efficiently along and across frozen bodies of water during winter. It is also a pleasant experience for many. Its pastime uses have now long overshadowed its practical applications, and someone skating on ice these days is almost invariably doing it for personal enjoyment, most likely for sports, on an indoors ice rink, regardless of season. Recreational ice skating has become something very different from getting from one location to another in winter.

Even with my very limited skill and experience, I enjoy ice skating, and I take considerable pleasure in seeing professional figure skaters perform acrobatic moves to music in what I consider one of the most enjoyable spectator sports invented by the human race. Seeing the joy of the performers, and being able to share their joy over a performance well executed both technically and aesthetically, is great entertainment to me and to countless others. However, the fact that someone can pull off a triple Lutz in a manner that is captivating to an audience of millions does not, for example, make me more or less inclined to vote for them in an election. The example is contrived, but not completely irrelevant: people with a background in sports do have an advantage when running for public office, in that they have name recognition and a fan base. Their former ability to compete and win at sports does say something about their general character, in that they must have had focus, determination and perseverance to succeed, but it says nothing about their skill in other specific areas. Their once impressive physical prowess, still present or not as the case may be, really has *nothing at all* to do with their ability to fulfill their duties as an elected representative.

Admittedly, the international figure skating community seems to be a bunch of very nice people. Only a select few claim the medals, but they are generously congratulating each other after fierce competitions, seemingly in complete earnest, and most participants claim that they are competing not only for the chance of winning, but just as much, often more, for their own enjoyment of doing something they excel at, and for performing in front of an audience. The ubiquitous final show after big competitions, and the attitude towards it – from the athletes as well as from the audience – tells a fascinating story of "stupid human tricks made into an art form", and it is a truly marvelous thing to watch. Figure skating is also a ridiculously inefficient and mostly ineffective way of making money, which means that these people are probably not greedy. I would trust such people any day over dishonest, scheming individuals who want political power for their own personal gain and are prepared to lie and cheat to get it, but *it doesn't take a figure skater to be a nice person*.

This, finally, leads me to the subject of code golfing.

I take no issue with people engaging in code golfing as an intellectual exercise, just as I take no issue with people who like to solve puzzles and play games – I'm most definitely one of those people myself. Human minds need to have fun at least now and then to work properly. I do, however, take issue with people who think that code golfing is anything more than a pastime puzzle, and I am sad to see when young and impressionable people who are talented but inexperienced programmers get confused and start engaging in code golfing before they learn how to write proper program code. Code golfing might have some marginal utility for experienced programmers, in that it encourages them to rethink old habits and approach problems from new and creative angles, but the activity as such leads absolutely nowhere. It should not be attempted by someone who is not already adept at programming, and *it doesn't take a code golfer to be an excellent programmer.*

To wrap things up with the figure skating analogy, and stretching it to its breaking point, it would be stupid and counterproductive to try to land a triple jump before you can keep your balance on a pair of skates, to let your figure skating training stop you from learning how to walk properly, or to tell anyone not wearing a pair of skates that they are doing things wrong, even when you are both standing on a concrete pavement rather than on an ice rink. This is, in essence, what some code golfers do.

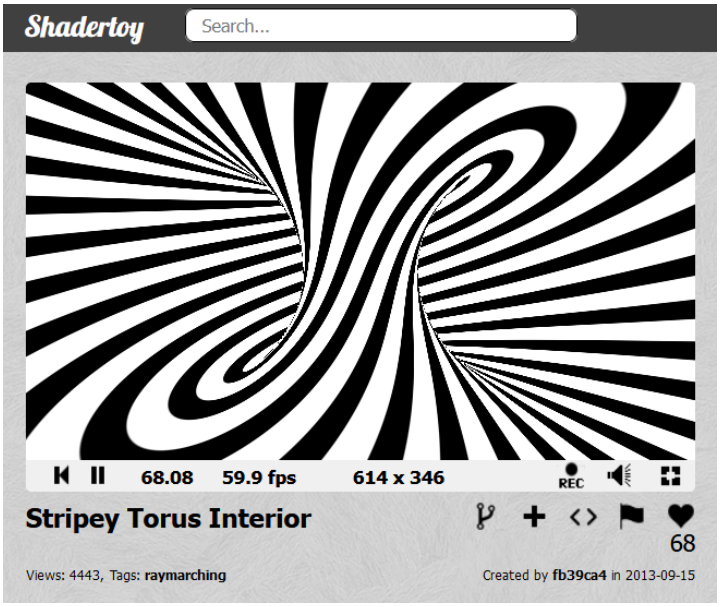
Real world programming is focused on getting things done in an efficient manner. Superficially, code golfing might seem somewhat focused on "efficiency" in terms of creating compact solutions, but the rules of the sport actually have very little to do with any meaningful definition of the word. The amount of characters in the source code is not only irrelevant to the task of efficient problem solving with programming – in most cases it is directly *contradictory* to it. It encourages a number of bad practices in coding, such as short and meaningless names for variables and functions, lack of spacing and structure, abuse of obfuscating quirks and undocumented features, and "cute code". The arbitrary, artificial constraints of keeping to the rules of the sport are not applicable to practical problem solving, and having the skill to be judged favorably by those rules has no strong bearing on your general skill as a programmer. It's not *completely* unrelated, as a successful code golfer needs to be smart and creative, but it carries the risk of letting the rules of the sport carry over to other problem solving tasks, either passively by an unrestrained force of habit or actively by misdirected ideals. In both cases, the result is directly *harmful*.

In order to make my point more clear, let me illustrate this with some examples. In the following, I will quote other people's work with credit, and I want to make it clear that I am not in any way criticizing them, or diminishing their highly successful, admirable and fun results from the spectator sport of code golfing. I am merely pointing out why it has absolutely nothing to do with general problem solving in programming, and how its rules invariably lead to an artificially constrained and convoluted solution. I can enjoy watching the results of code golfing, and on some level I would probably enjoy doing it myself, but I choose not to participate in it.

All examples are from Shadertoy, https://www.shadertoy.com/, because that's where I have had my most recent, still ongoing personal encounter with a hard-core code golfing sub-community. Now, code golfing is definitely not everything on Shadertoy. It is a great community for the exchange of creative ideas and sharing concrete, working code examples with an international and cross-disciplinary group of fellow enthusiasts in the field of computer graphics. However, code golfing is widely admired and encouraged in the general discussions on the site, and I think it has been allowed to proliferate to a point where it is becoming a detriment to the exchange of knowledge, and a significant barrier to entry for newcomers.

Example 1: Before and After

Code golfing often starts with a challenge: "Can you make this program smaller?" One such example among many is this fun demo, which displays an animated ray-marched image from the inside of a diagonally striped torus. A still image from the demo is shown below, followed by the code that creates it. The programming language is GLSL, OpenGL Shading Language, which is representative of many popular languages for code golfing: it has a C-like, quite flexible and potentially very compact syntax, and it has some unique quirks that require considerable experience and skill to utilize for golfing.

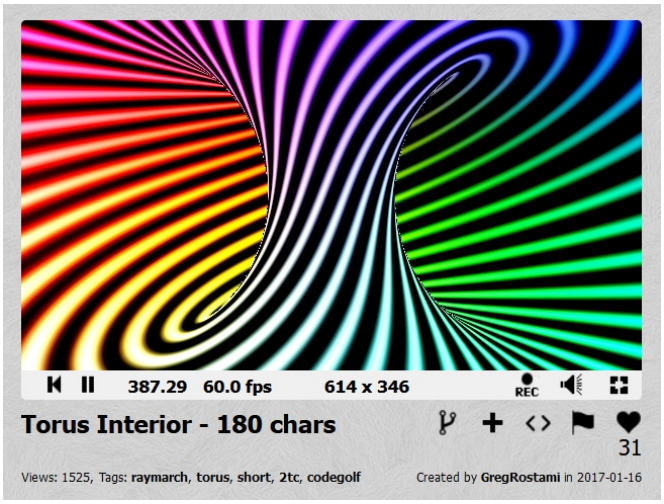



```
// 1151 chars - Original shader by fb39ca4 on Shadertoy

const float PI = 3.1415926536;

const int MAX_PRIMARY_RAY_STEPS = 64; // decrease this if it runs slow

vec2 rotate2d(vec2 v, float a) {
                   return vec2(v.x * cos(a) - v.y * sin(a), v.y * cos(a) + v.x *
sin(a));
}

float sdTorus( vec3 p, vec2 t ) {
  vec2 q = vec2(length(p.xz)-t.x,p.y);
  return length(q)-t.y;
}

float distanceField(vec3 p) {
  return -sdTorus(p, vec2(4.0, 3.0));
}
```

```
vec3 castRay(vec3 pos, vec3 dir, float treshold) {
  for (int i = 0; i < MAX_PRIMARY_RAY_STEPS; i++) {
    float dist = distanceField(pos);
    //if (abs(dist) < treshold) break;
    pos += dist * dir;
  }
  return pos;
}

void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
  vec4 mousePos = (iMouse / iResolution.xyxy) * 2.0 - 1.0;
  vec2 screenPos = (fragCoord.xy / iResolution.xy) * 2.0 - 1.0;
  vec3 cameraPos = vec3(0.0, 0.0, -3.8);

  vec3 cameraDir = vec3(0.0, 0.0, 0.5);
  vec3 planeU = vec3(1.0, 0.0, 0.0) * 0.8;
  vec3 planeV = vec3(0.0, iResolution.y / iResolution.x * 1.0, 0.0);
  vec3 rayDir = normalize(cameraDir + screenPos.x * planeU + screenPos.y * planeV);

  //cameraPos.yz = rotate2d(cameraPos.yz, mousePos.y);
  //rayDir.yz = rotate2d(rayDir.yz, mousePos.y);

  //cameraPos.xz = rotate2d(cameraPos.xz, mousePos.x);
  //rayDir.xz = rotate2d(rayDir.xz, mousePos.x);

  vec3 rayPos = castRay(cameraPos, rayDir, 0.01);

  float majorAngle = atan(rayPos.z, rayPos.x);
  float minorAngle = atan(rayPos.y, length(rayPos.xz) - 4.0);

  float edge = mod(8.0 * (minorAngle + majorAngle + iTime) / PI, 1.0);
  float color = edge < 0.7 ? smoothstep(edge, edge+0.03, 0.5) : 1.0-smoothstep(edge,
edge+0.03, 0.96);
  //float color = step(mod(8.0 * (minorAngle + majorAngle + iTime) / PI, 1.0), 0.5);
  //color -= 0.20 * step(mod(1.0 * (minorAngle + 1.0 * majorAngle + PI / 2.0) / PI, 1.0),
0.2);

  fragColor = vec4(color);
}
```

The code is not exactly a shining example of coding for longevity and clarity, but for a short demo written for the enjoyment of a small community – a piece of code to watch, learn from, experiment with and build upon – it's a good example of reasonably well written code that serves its purpose. There is a clear structure, proper spacing, relevant symbolic names, and at least one comment on the role of a key parameter for performance. The majority of the "comments" are just commented-out lines of code, but they are informative to the relevant audience: programmers. It shows some options for experimentation, and the author decided to keep them on display in the published code.

Now, this piece of code is short and sweet by any normal standards, but it became the subject of a code golf challenge. The result is shown below. The image is even more complex in that it adds some color, but the code is now so terse and full of coding tricks that it is, for all practical purposes, *indecipherable*.




```
// 180 chars - iq and Fabrice not only fix a bug, they kill another 13 chars:

void mainImage(out vec4 o, vec2 u) {
  vec3 R = iResolution;
  for ( o.z++; R.z++ < 64. ; )
    o += vec4((u+u-R.xy)/R.x,1,0)*(length(vec2(o.a=length(o.xz)-.7,o.y) )-.5);
  o += sin( 21.* ( atan(o.y,o.w) - atan(o.z,o.x) - iTime ) ); }
```

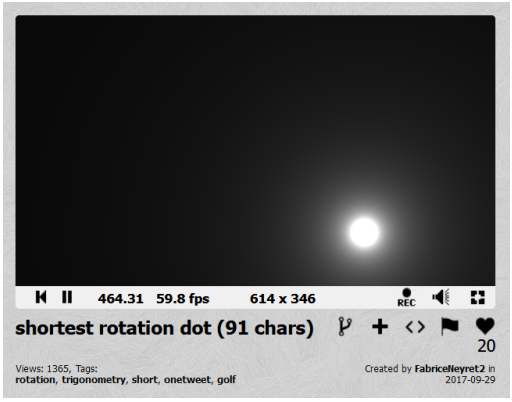
While it is truly *amazing* that these 180 characters compile to code that generates the same image, and a more fancy version at that, it serves no purpose other than being the shortest sequence of characters to generate this particular image in this particular programming language on this particular platform – so far. Code golfing is often a never-ending search for ways to shave off a character or two from existing solutions, much like competitors in physical sports try to set new records. This is effectively "write-only" code, constructed with considerable effort not to be readable, fast, understandable, or any other criteria that would have any meaning in other circumstances, but *compact* at all costs, in the very narrow sense of "using as few characters as possible for the source code". It is the result of an extreme sport, meant to provide enjoyment and mutual collaborative admiration in a small community, not to generate useful program code. This "world record" of sorts has no value outside of the sport. It's not something anyone in their right mind would even try when solving a real programming task, and it takes tremendous effort for anyone but an elite competitor in GLSL code golf to decipher it. A piece of code that requires this much work to even make sense of what it *does*, and how, is something that would never be acceptable as part of an actual software product. It is impossible to read, and therefore impossible to modify, to debug, even to document properly, and it would be dangerous to make any other code dependent on it. To make things worse, "golfed code" is notoriously brittle, prone to breaking as a result of even minor changes to language version or subtle platform differences, simply because it is written with no consideration to such criteria. It's a piece of code aimed exclusively at using as few characters as possible *right now*, with no regard to longevity, readability, stability, compatibility, even *ability*, as it is very specifically solving this problem and this problem alone. Attempts to modify it for any other purpose might prove utterly futile, and it is probably easier, faster and better in every sense to start over from scratch.

Example 2: Rotating dot

The example above is typical, and telling, but despite its brevity it's a little too complicated to examine and explain in detail here. To demonstrate how much effort it takes to understand golfed code, I will instead present a shorter example with more specific comments on the idiosyncracies. Once again, I stress the point

that this is *not* criticism of the author of this particular line of code, Fabrice Neyret. He is a very nice and smart person whom I enjoy communicating with over at Shadertoy, he has done a lot of great work over the time of a long career and continues to do great work when he is not code golfing, and I am more than happy to see him enjoy himself, and entertain and engage others, with his impressive skill in the sport.

The (eternally ongoing) challenge in this case is to draw a dot rotating in 2-D with as few characters as possible. Several variants have been presented over the years, and this is one of the shortest:




```
void mainImage(out vec4 O, vec2 U) {
    vec2 R = iResolution.xy;
    O += .1 / length( (U+U-R)/R.y - sin( iTime  + vec2(33,0) ) ) - O;
}
```

Let us expand this to use more descriptive variable names and split the one-liner by using some intermediate variables, more like it would have been written if this hadn't been a puzzle game:

```
void mainImage(out vec4 color, vec2 position) { // Was: O, U
    vec2 resolution = iResolution.xy; // was: R
    vec2 cos_sin = sin( iTime  + vec2(33,0) );
    vec2 center = (position+position-resolution)/resolution.y; // (U+U-R)/R.y
    color += 0.1 / length(center - cos_sin) - color;
}
```

The choice of the single upper-case character `O` for a variable name is unnecessarily confusing and easily mixed up with the digit `0`. This naming convention for the output parameter has become standard practice for golfing on Shadertoy, for *no good reason at all*, but that is a minor problem.

Looking at the expanded code, a few things stand out as odd. A rotation requires you to evaluate the sine and cosine functions for a time-dependent angle, but there's only a call to the `sin()` function. It operates on the globally defined variable `iTime`, which contains the number of seconds since the program started, added to a two-element vector of integers, `vec2(33,0)`. In GLSL, adding a vector and a single value adds the value to each element of the vector and produces a vector as the result. This means that we are evaluating `sin(vec2(iTime+33, iTime))`, which is equivalent to `vec2(sin(iTime+33), sin(iTime))`. Now, the sine function is periodic with period 2π. GLSL has no predefined symbolic constant for the value of π, but the integer 33 just so happens to be quite close to 10.5π. This makes the expression almost equivalent to `vec2(sin(iTime+PI/2, sin(iTime))`, which is `vec2(cos(iTime, sin(iTime))`. The problem here is the word "almost". There is a slight error in the approximation, about 1%, because $33 \approx 10.50422\pi$, so we are effectively approximating $1/2$ with 0.50422. The error might not be noticeable for this particular application, but it would not be good enough in the general case, and deliberately introducing a numerical error of 1% for no other reason than to avoid typing a few decimals would be completely unacceptable in real software development. Furthermore, sine and cosine are very frequently

used together like this – so frequently that many GPU models can evaluate both functions in parallel by a single assembly-level instruction. However, no compiler is going to make the decision for us to replace a call to $\sin(a+0.50422\pi)$ with $\cos(a)$, because they are *not* the same. Thus, this clever trick for golfing ends up forcing the compiler to generate code for two separate calls to the sine function for two different angles, requiring *more* instructions and possibly more than twice as much effort as if it had been asked to evaluate a sine/cosine pair for the same angle in one go. The trick is *counterproductive* in every sense except one: it takes a few characters less to type it than `vec2(cos(iTime), sin(iTime))`. Note that the "savings" made in this case depends on Shadertoy's chosen name for the global variable `iTime`. Had the time variable instead been named `t`, it would have changed the rules of the game, and it would have been almost as short to write `vec2(cos(t),sin(t))` (19 characters) as `sin(t+vec2(33,0))` (17 characters). The rules for any variant of code golfing are *arbitrary* and *artificial* constraints.

The next line, `vec2 center = (position+position-resolution)/resolution.y`, is also weird. Adding a number to itself is equivalent to multiplying it by 2, and at a glance you might think an addition is faster to execute than a multiplication. However, the multiplication is by a constant 2, which is a simple bit-shift in integer arithmetic, and in floating point arithmetic it is reduced to incrementing the exponent by one, which amounts to a single 7-bit addition. Adding a number to itself forces the compiler to issue an addition instruction (unless it has an expression analyzer good enough to recognize the pattern `U+U` and replace the addition with a multiplication), and an addition is *more* work than an increment of the exponent, because it involves more bits and two operations in sequence: after the addition of the two 23-bit mantissas you need to detect the need for a renormalization, causing a bit-shift of the mantissa and an increase of the exponent. This might not take longer to execute in the heavily pipelined arithmetic operations of a modern GPU, but again, the golfing's obfuscated expression `U+U` runs the risk of producing *less* efficient executable code than if we had simply written `2.0*U` – or `2.*U` if you prefer to golf it. One or two characters were saved in the source code, where it really doesn't matter at all, at the possible expense of execution speed, which matters *a lot*.

The last line is a strange animal as well: `color += 0.1 / length(center - sincos) – color`.

First we add something to `color`, and then we subtract `color` from the total. This effectively cancels out the addition with `color` inherent to the `+=` operator, so why use it in the first place? Why not just write `color = 0.1 / length(center – sincos)` and be done with it, using several characters less? The answer is that we are assigning a single value for the grayscale to an RGB vector, and that is not allowed in GLSL. Adding a single value *to* a vector, however, is permitted, so `color += 0.1` works, while `color = 0.1` would be a type mismatch error. The subtraction at the end, `- color`, would serve the same purpose by itself. It's not completely unreasonable to assume that a modern optimizing compiler could find this pattern, `color += x – color`, and optimize away both the addition and the subtraction, but it is by no means guaranteed, because the expressions `color += x – color` and `color = vec3(x)` are not quite equivalent. The value of the output parameter `out vec4 color` is formally undefined when we enter the function, and using `+=` as the first operation on an undefined value has an undefined result. Many current GLSL implementations initialize `out` variables to zero, but some do not, so this is a potential source of platform-dependent bugs. The `- color` at the end seems to subtract anything that was in memory as we entered – in fact, `x -= x` is frequently used as a golfed form of, say, `x = vec3(0.0)` – but keep in mind that this is floating point arithmetic. A random large value for `color` at entry would swallow the small value that is added to it by truncation, and a possible random bit pattern in `color` at entry might not even be a legal representation of a number. This could generate an illegal result from the first operation, which would propagate to an illegal value as the final result. If the uninitialized variable happened to have the value NaN, the result would definitely be NaN in accordance with IEEE floating point specifications. The code golfing once again ends up generating *less compact* executable code that takes *longer* to run, which is hard to understand *and* prone to breaking. To top it off, this has a potential hidden source of bugs in its nonstandard treatment of the fourth ("alpha") component of the output color: instead of setting it to 1.0 as required by the

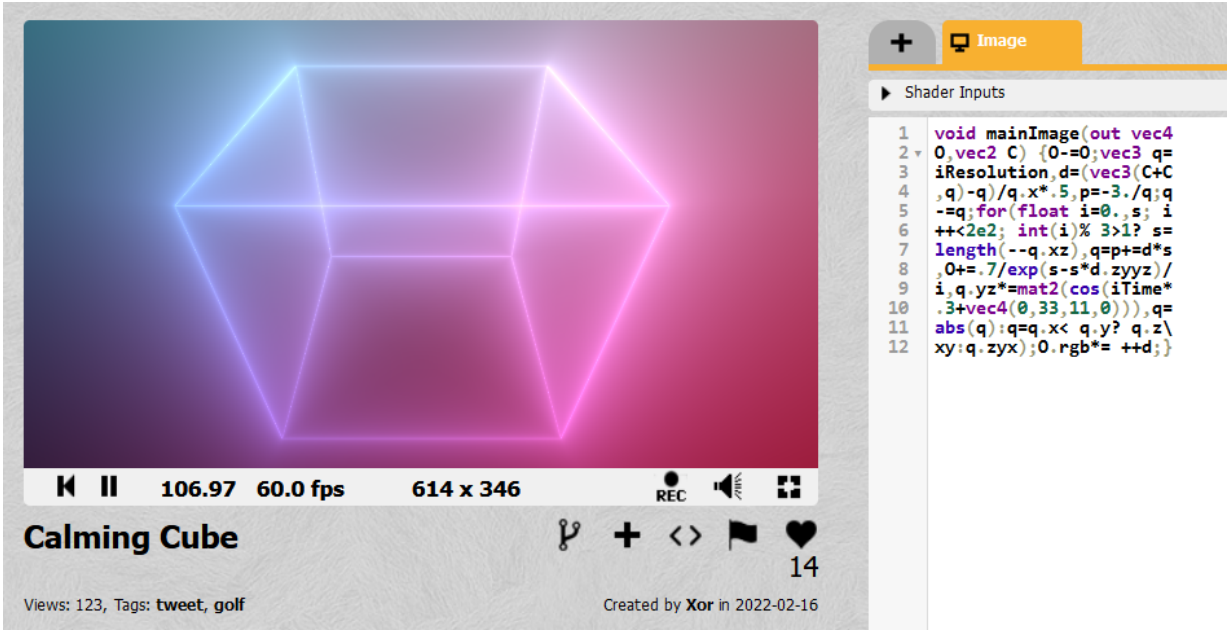
specification, it is set equal to the other three values. This does not make any difference in the current implementation of Shadertoy, but it violates the API specification and might stop working in future updates. Playing *by* the rules has now slid into the dangerous territory of *playing* the rules instead of honoring their intent. In sports, this is considered cheating.

The total "savings" of using this weird and borderline illegal statement is also marginal by any other measure than the weird rules of code golfing. Typing `O=vec3(x)` would have required only three characters more than `O+=x-O`.

Finally, a note on `0.1 / length()`: This is used to create a blurry "glow" around a white dot against a black background, and it relies on the output color being clamped to 1.0, so that the dot is flat white within a radius of 0.1 units from the rotated position `center-sincos`. This is far from obvious, and in circumstances outside of code golfing this expression would require a clear and concise comment to explain what it does. (Comments are, of course, completely shunned in golfing, as they would be a "waste of characters".)

Conclusion

I hope this has convinced you that code golfing is harmful if it is ever taken outside the sports arena. With that being said, I will readily admit that I sometimes look at some of the results from code golfing on Shadertoy and decipher them as a fun puzzle. One final and impressive example, demonstrating the bad practice of writing "cute code", is this little snippet, which amazingly enough draws a rotating 3-D cube.



The code for the cube is "formatted" as a square of characters. This kind of demo tickles a programmer's humor. It's fun. I *like* it. I can even see beauty in it. But I absolutely hate to see any of these bad habits make their way into *actual* programming, and I hate to see people confusing code golfing skills with useful programming skills. They often come together, but they are only very distant relatives, and with that analogy, the notorious code golfer is the shifty-eyed second cousin who claims to be a carpenter but whom you would not even trust with assembling an IKEA chair, because it might end up impaling you after his decision not to use some "unnecessary" screws, fail to properly tighten the few screws he used because that's "not his style", and to use saliva instead of glue "because it's basically the same thing".

And with that, I will now crawl back into my old guy's corner. Thanks for reading.