A minimal raytracer

Aug 3, 2016

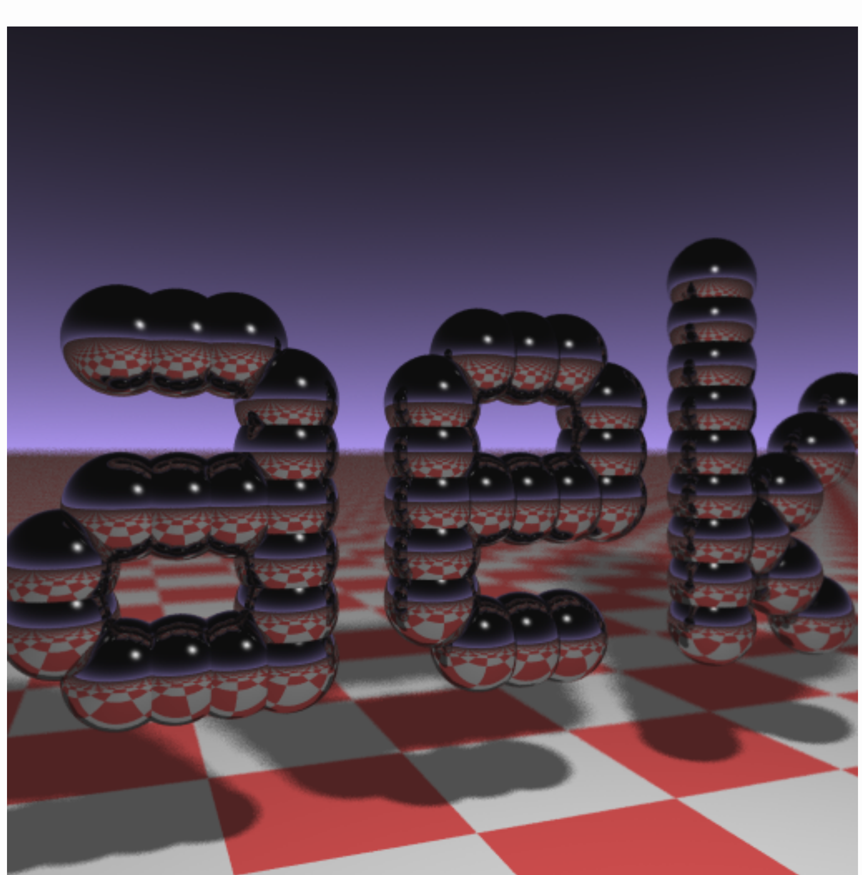
How my quest to make a business card sized raytracer got a little out of hand.

As an aside, this ended up being a very long post. If you get bored, you can just skip to the end where I've presented all of the version history side-by-side with the program outputs, or just go browse the github repo.

Update Aug 24, 2016: The cards have arrived!

Background

At some point during grad school (probably around mid-2009), I happened across Andrew Kensler's business card sized raytracer, which produces this amazingly beautiful image in just 1,337 "leet" bytes of C++ source:



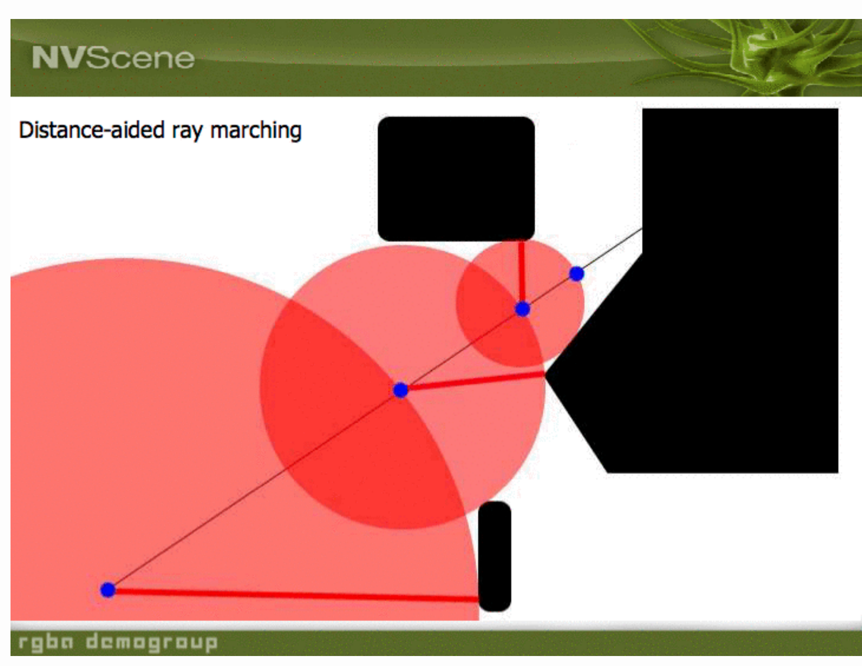
Aside from the obvious reflections, the program is notable for its depth-of-field blur and soft shadows.[1] When I learned that Kensler's program was itself an homage to a previous business card sized raytracer created by Paul Heckbert, I decided to create my own.

Objective

Like Kensler, I wanted my raytracer to write my name or initials; however, I thought I could improve the text rendering by using a vector font, as opposed to a raster font as Kensler's did. My goal was to use "ball-and-stick" letters composed of piecewise arcs and line segments to spell "mattz".

One immediate obstacle arose: the result of sweeping a 3D sphere along a 2D arc is a torus segment, and ray-torus intersection a bit hairy because it requires solving a quartic (fourth-order polynomial). Analytic solutions to general quartics exist, but they're complex enough to spill the program way over the size of a business card.

Fortunately, I had recently come across Inigo Quilez's nvscene 2008 presentation, Rendering Worlds With Two Triangles, which supplies a convenient dodge: raymarching, a.k.a. sphere tracing. Here's an illustration of the process, from iq's slides:



Sphere tracing works by computing or estimating the distance between the current point on the ray (initially the ray origin) and the nearest point in the scene, and advancing along the ray by that distance, as shown above. Although it may be slower than analytic raytracying, it is numerically robust and guaranteed to converge to an intersection as long as the ray starts in free space and the distance estimate is less than or equal to the true distance.[2]

While analytically intersecting a ray and torus is hairy, computing the exact distance between a point and a torus is a piece of cake, and it admits a straightforward solution to clamping the angle to obtain arcs of less than 360°.

Initial development

So after playing a bit with distance functions and fonts, I whipped up version 0.1 of my business card raytracer, which produced this image:

The image was not so impressive (poor composition, no background), and the code was nearly twice as long as Kensler's, at 2,245 bytes[3]; however, it did successfully render my name in a proportional vector font with some nice Phong-like shading. The overall strategy for encoding the text led to a somewhat compact representation, and remained remarkably well-preserved throughout the entire development of the program even as the encoding got smaller and smaller.

```
float dst(vec p, vec* nn) {

  hit h(p);

  int data[] = {
    0x40426040,
    0x24824044,
    0xe38d2486,
    0x814a6048,
    0x42966329,
    0x632d814e,
    0x6031429e,
    0x60716331,
    0
  };

  for (uint8_t* c=(uint8_t*)data; *c; c+=2) {
    int o = c[0]>>5;
    float x = c[0]&31;
    float y = c[1]&31;
    int l = c[1]>>5;
    (o>=1&&o<=3)?seg(x,y,l*(o!=2),l*(o!=1),h):
      (o==4)?arc(v(x/2,y/2),1+0.5*(l>>2),-M_PI*((l>>1)&1),M_PI*(l&1),l
  }

  if (nn) { *nn = !(p-h.pc); }
  return sqrt(h.d2)-R;

}
```

The above snippet (with typenames expanded for clarity) implements the distance estimator, which iterates over each of the 16 strokes (line segments and arcs) that comprise the text "mattz". Each stroke is encoded in two bytes of the `data` array. The bytes include an opcode `o` that indicates line segment or arc, `x` and `y` coordinates for the endpoint of the line segment or center of the arc, and finally a data argument `l` that encodes line direction and slope, or arc angle limits.

The next few raytracer versions were focused on reducing the program's size. Between version 0.1 and version 0.6, the program was modified to

- jettison the ill-conceived templatized ray-marcher in 0.1
- totally remove the lighting code in 0.3, and replace it entirely in 0.5
- eliminate some unused vector class operator overloads
- make all identifiers single characters
- represent font bytecodes in decimal, rather than hex

Also, version 0.4 saw the the addition of comments – I guess groveling through all of the one-character identifiers became sufficiently burdensome to necessitate them.[4] By version 0.6, the program size was a "nearly-leet" 1,342 bytes, a reduction in size of about 70% from version 0.1x. The program output looked pretty much the same, though:

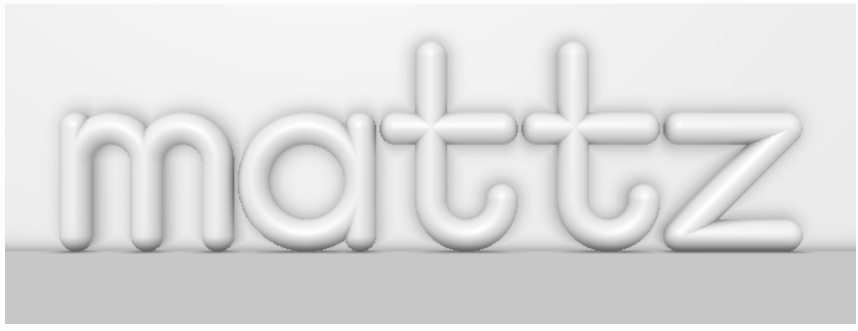
This was a decent milestone – proportional vector font in roughly the same space as Kensler's program; however, it was past time to do something about the basic appearance. The image has no background at all, and none of the hallmarks (reflections, shadows) that we have come to expect from ray tracers. To be honest, this looks like something you might get out of WordArt, not a badass raytracer.

It was time to up my game a little bit.

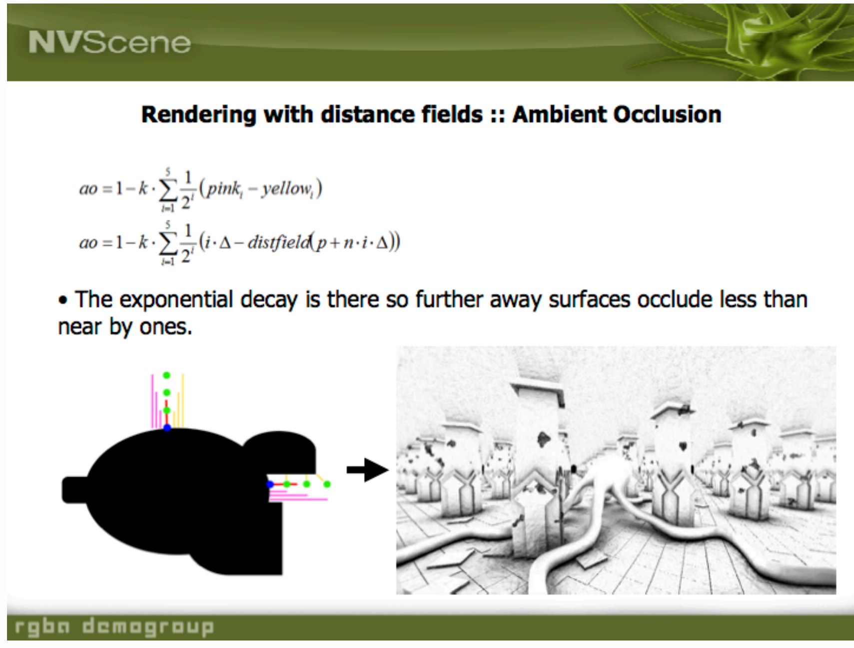
Getting serious

Let's talk about these version numbers for a second – they're a little dopey. I wasn't releasing the software to anyone (except showing it around to a few colleagues), but I nonetheless felt it was important to stash and label distinct versions to track how this crazy little program was evolving. This was in the days before I was a serious github convert, and an arbitrary numbering system was better than no system at all.

So in any event, I declared the next version to be 1.0 because it seemed like such a radical departure from the previous one. It had a back wall! And a floor! And something resembling shadowing!



Planes are just about the easiest primitive in the world to raymarch, so the walls were easy enough to add. To get the shadows, I was determined to add the ambient occlusion (AO) algorithm I had read about in iq's presentation:



In a nutshell, what this AO method does is to march along the normal vector near a ray intersection and darken areas where there is lots of other geometry nearby. It's totally fake, has no basis in reality, and looks amazingly convincing nonetheless.

The bad news: adding the walls and all of that AO code took up lots of program space, and there was no way to fit onto a business card without cutting *something*. Switching to grayscale image rendering got rid of some excess code, but the new version was still a bit hefty, at 1,493 bytes.

By version 1.2 I had slimmed down the program to 1,461 bytes while addressing the bland composition. Even though 1.0 looked more "ray-tracy" than 0.6 due to walls and shadows, it still didn't look very "3D" because there was no strong perspective. I felt like the new camera pose did a much better job showcasing the soft shadows from AO, too.

The program size had shrunk to 1,242 bytes by the time I got to version 1.4, while leaving the output image nearly unchanged. One of the big wins was replacing the font table initializer to a character string instead of an `int` array. Here is the new, more-compact font encoding:

```
// from miniray_1.3.cpp
u d[] = "@`B@D@\x82$\x86$\x8d\xe3H`J\x81)c\x96""BN\x81-c\x9e""B1`1c
```

In time, I was able to compress this further by discovering an encoding which obviated the hexidecimal escapes and quotes; meanwhile, this was still a big improvement over the integer array. For 1.4, I also produced a thoroughly commented version of this revision, which I began to show to a larger audience.

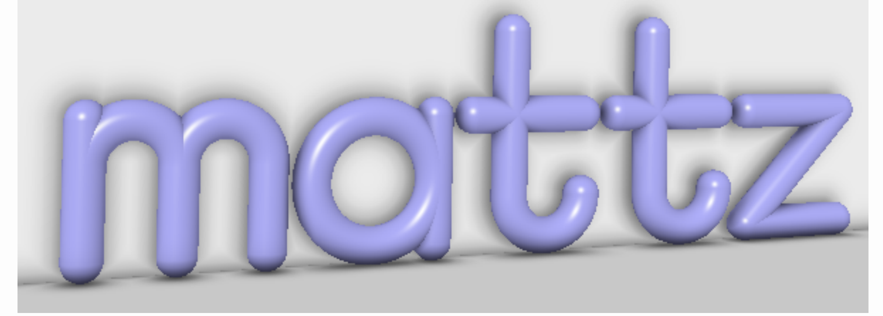
Going public

Sometime around September 2009, I posted a version of my program to the now-defunct ompf.org (whose demise is lamented here among other places), a web forum dedicated to the real-time raytracing community. Sadly, the forum thread has been lost to the ages (the Wayback Machine has the forum index page, but not the thread itself), so it's a bit hard to remember which version of the program first got posted.

My hazy recollection is that I posted something like the grayscale image above, and got feedback along the lines of "cool idea, needs color". A few changes enabled reducing the size enough to add RGB color back in, including:

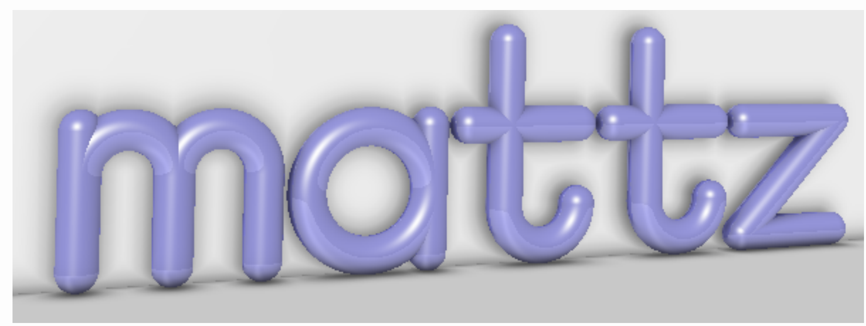
- further pruning the set of vector class operator overloads
- replacing the hit info class with a function
- adopting a font encoding requiring fewer hexidecimal escapes and quotes

Here is the output of the new version 2.0 (I figured adding in color merited bumping the major version number):



The color version ended up being barely larger than the smallest grayscale version, weighing in at just 1,290 bytes. This could definitely fit on a business card, and it looked pretty good, but it still lacked a staple raytracing feature: reflections.

In the run-up to version 2.4 I found a few more opportunites to shave off bytes, allowing me sufficient room to get reflections working. In the end, version 2.4 was just 1,296 bytes. The reflections in the output image are subtle, but the downward-sloping surfaces of the letters clearly pick up some extra white from the floor if you look closely:

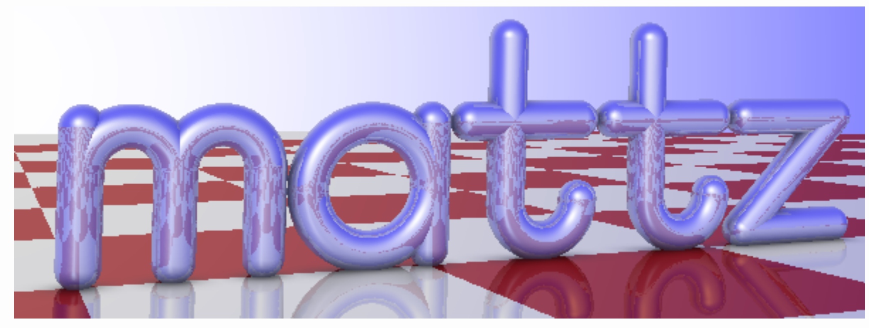


By this point, I had discovered a font encoding which was just about as compact as theoretically possible without implementing a fully general bitstream read across byte boundaries. It encoded the 16 strokes of the text “mattz” in just 32 bytes of C string constant, with exactly two bytes per stroke:

```
char B[]="BCJB@bJBHbJCE[FLL_A[FLMCA[CCTT`T"
```

A logical stopping point?

Based on feedback from ompf.org and friends, I decided to be a little less subtle with my newly-implemented reflections. What better way to do this than to cleave to the tried-and-true raytracing trope of the infinite checkerboard plane?



Besides the plane, the new version 3.0 picked up a blue sky which hackily fades to white along the plus and minus world -axis, giving a hint of a sun in the sky. Amazingly, I was able to keep the size to 1,295 bytes by:

- eliminating a few `sqrt` calls
- creating constants for a few commonly used quantities (like 0.5)
- eliminating the back wall

among other changes. Here is the compacted version of the final C++ business card raytracer that got uploaded to ompf.org, probably sometime in 2010 (the github version linked above actually has whitespace and comments):

```
#include <stdio.h>
#include <math.h>
#define O operator
#define E return
typedef float f;f H=.5,Z=.33,Y=Z+Z,I;struct v{f x,y,z;v(f a=0,f b=0,f
),y(b),z(c){}v O*(f s){E v(x*s,y*s,z*s);}f O%(v r){E x*r.x+y*r.y+z*r.
{v&t=*this;E t*pow(t%t,-H);}v O+(v r){E v(x+r.x,y+r.y,z+r.z);}v O-(v
+r*-1;}}L=!v(-1,1,2),W(1,1,1),F(Y,Y,1),P,C,M,N;f U(f a){E a<0?0:a>1?1
c,v m){f d=(P-c)%(P-c);if(d<I){C=c;I=d;M=m;}}f D(v p){f x=0;I=99;P=p;
"BCJB@bJBHbJCE[FLL_A[FLMCA[CCTT`T",*b;for(b=B;*b;++b){x+=*b/4&15;int
*++b&7,y=*b/8&7;v k(x,y),d(a*(o&1),o/2*a);if(o)Q(k+d*U((p-k)%d/(d%d))
f r=H*(a&1)+1,t=atan2(p.y-y*H,p.x-x*H),P=M_PI,l=-P*(a/4&1),u=P*(a/2&1
:t>u?u:t;Q(k*H+v(cos(t),sin(t))*r,F);}}N=v(0,1);Q(p-N*(p%N+.9),W);if(
int((p.x+64)/8)^int((p.z+64)/8))&1)M=Y;N=P-C;E sqrt(I)-.45;}v R(v o,v
n,p;f u=0,l=0,i=0,a=1,k=d.z*d.z;while(u<97)if((l=D(o+d*(u+=l)))*l<.00
!N;o=o+d*u;while(++i<6)a-=U(i/5-D(o+n*.2*i))/pow(2,i);p=p*(U(n%L)*Z+Y
p=p*Y+R(o+n*.1,d-n*2*(d%n),z-1)*Z;u=pow(U(n%!(L-d)),40);E p*(1-u)+W*u
,1);}int main(){f y=-111;puts("P6 600 220 255");while(++y<110)for(f x=
300;++x){v p=R(v(-2,4,25),!((!v(5,0,2)*x-!v(-2,73)*y)*.034+v(10.25,-2
*255;putchar(p.x);putchar(p.y);putchar(p.z);}}
```

This program seemed like the total realization of my goals for a business card raytracer. As small as Kensler's, implementing its own new effects (vector font, soft shadows), while still looking absolutely "ray tracey" (checkerboard floor: check; reflections out the wazoo: check). It amazed me that every time I thought I had reached the absolute minimum size possible, a few more size optimizations had occured to me. At this point, I was ready to shelve the project as a success, and move on.

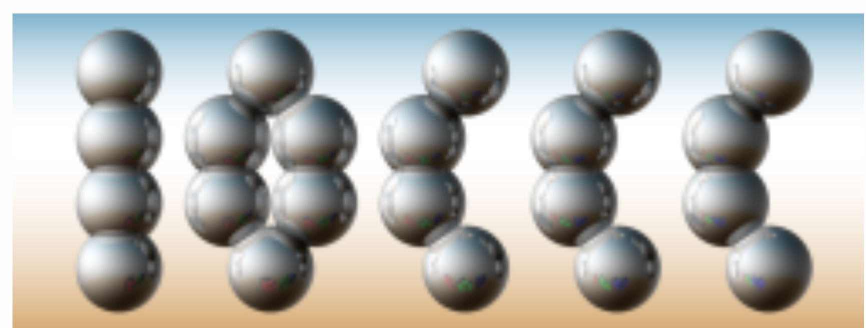
That is, until I saw the announcement for the 20th IOCCC appear on Slashdot.org.

Gearing up for IOCCC

For years, I had been following the International Obfuscated C Code Contest, or IOCCC. According to the contest organizers, its goals are:

> - *To write the most Obscure/Obfuscated C program within the rules.*
> - *To show the importance of programming style, in an ironic way.*
> - *To stress C compilers with unusual code.*
> - *To illustrate some of the subtleties of the C language.*
> - *To provide a safe forum for poor C code. :-)*

I'm not exactly sure when I became familiar with the IOCCC, but Fabrice Bellard, a programmer I particularly admired during grad school, won once with a tiny C compiler, notably producing the first contest entry capable of compiling itself. There had even been a previous tiny raytracer entry which the judges liked well enough to adopt to render an IOCCC logo:



Soon after I got to grad school, the IOCCC went on hiatus. There was a competition in 2006, but none again until the 2011 announcement linked above. So at the long-awaited return of the contest, I realized I had the basis for a pretty solid entry, but there was just one problem: my program was written in C++, and the IOCCC is a C-only competition.

What else could I do but port the entire thing to C? The 3.0 version above is fairly C-like, but it still uses a couple of C++ language features like constructors and operator overloading. After some wrangling, I managed to come up with version 4.0 which produces byte-for-byte identical output to the 3.0 program above. The program size, however, increased to 1,422 bytes.

Fortunately, the size increase wasn't particularly problematic. Moving from a business card to the IOCCC actually opened up some headroom for program size because the 2011 official rules specified[5]

> *2) The size of your program source must be <= 4096 bytes in length. The number of characters excluding whitespace (tab, space, newline, formfeed, return), and excluding any ; { or } immediately followed by whitespace or end of file, must be <= 2048.*

According to this metric, the 4.0 version of my program had a "contest length" of 1,351 bytes, leaving lots of space to add more functionality. I decided to try something a little ambitious. Writing a message in a proportional-width vector font inside a tiny raytracer is pretty cool, but wouldn't it be better to be able to write *any* message in a proportional-width vector font inside a tiny raytracer?

Going big

I decided to create an *entire font* that represented a reasonable fraction of the ASCII character set, including all of the lowercase letters, all ten digits, and a handful of punctuation marks. Like the original “mattz” characters, these were composed of strokes made of line segments and arcs. To assist in this, I wrote a substantial C++ program, encode.cpp, for generating the font data. It automatically searched the space of possible bit-level encodings of the characters to make sure that not only was the font data string as short as possible (no extended ASCII chars or quotes to escape), but the C expressions used to decode it were compact as well.[6] Here is its final output:

!"#$%&()*+,-
./0123456789@a
bcdefghijklmno
pqrstuvwxyz

After implementing anti-aliasing via 2x2 supersampling as a warm-up in version 4.1, I shoved the new font into version 4.2. Running the program with no arguments and piping output to a PPM like this:

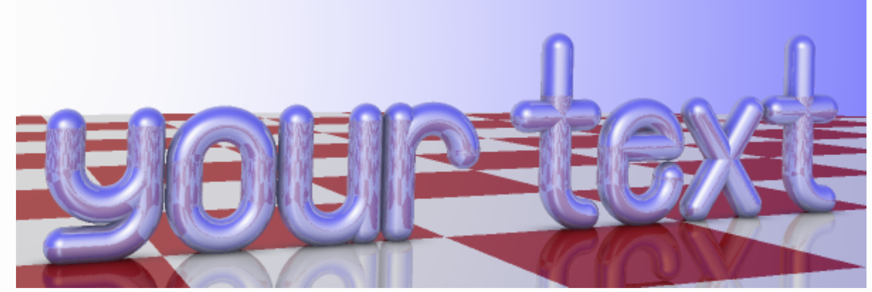
```
./miniray_4.2 > image.ppm
```

would generate this image with the new default text “ioccc 2011”:

but running with an additional argument like this:

```
./miniray_4.2 "your text" > image.ppm
```

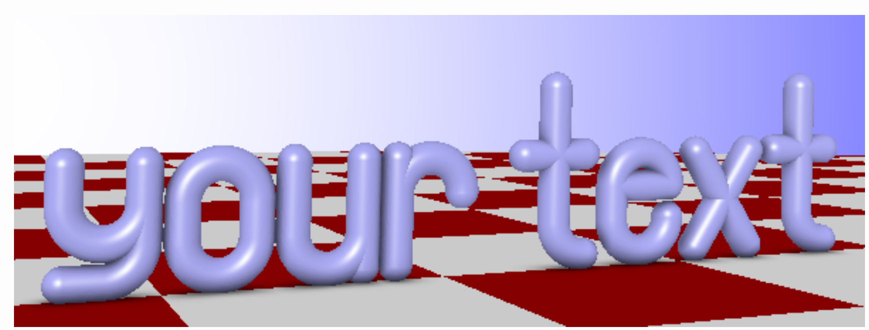
would render the string provided on the command line, like so:



The final new feature for 4.2 was a preview mode. If the user specified a second command line argument, like so:

```
./miniray_4.2 "your text" -preview > image.ppm
```

then the image was generated with supersampling and reflections disabled:



This preview mode resulted in a *much* faster render time, and although it was billed to the IOCCC judges as a feature, it also aided development, as I could make changes to the program and not have to wait ages (thanks to 2x2 supersampling) to see if I had completely broken the renderer.

The code had gotten much more complex, as it now had to:

- count command line arguments to decide which text to display
- look up the glyph for each text character and arrange the strokes in each glyph into the scene
- support two different rendering modes

Despite all of that, version 4.2 still fit into the contest requirements, with a total program size of 2,058 bytes and a contest length of just 1,982, still under the 2,048 byte limit. Incidentally, this is the version of the source code where comments were totally abandoned. I think it was just too hard to keep detailed explanations in place as I kept transforming the program.

I was pretty sure I had a solid IOCCC contender, but I felt it could bear some additional obfuscation.

Increasing inscrutability

My next goal was to make it not just difficult, but nigh-impossible for a competent C

programmer to interpret the program. Versions 4.3 through 4.5 went a long way towards accomplishing this by

- eliminating *all* local variables – every variable is either a program global or a function argument.
- undertaking major re-organizations to shorten the code – throughout this project I consistently found the shorter the program, the more impenetrable.
- tweaking the encoding a bit more, and hoisting the PPM header `"P6 600 220 255"` into the giant unreadable string at the top of the file.
- eliminating all control flow structures aside from `for` and the ternary operator `?:`.
- abusing the hell out of the comma operator to cram multiple statements inside `for` loop initializers, conditions, and counters.

Along the way, I also tweaked the camera parameters and deepened the shadow intensity, which provided better contrast where the letters met the ground, as shown here:

When I was reading over version 4.5 this week, I was amused to encounter this comment:

```
/*
  _abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
   iiivc fvfccfFiivfivif fttiFvvFPiFffcvivvFvFiPiFFvifv
   *                          *
  */
```

…written to help me remember which identifiers had which data types. Capital `F` is for function, lowercase `f` for float, `i` int, etc. I don't recall what the asterisks in the bottom row are for, but it is notable that there are only three valid one-character identifiers left! I also hid a cat in the definition of the 3D vector struct:

```
typedef struct { x c,a,t; } y;
```

I'm pretty sure that 4.5 was the version initially submitted to the IOCCC; however, the contest judges found a few characters (`"`, `:`, and `/`) which caused rendering artifacts. The final modification (from 4.5 to 4.6) was just a bugfix to correct the issue.

Without any fancy layout, the final version came in at 1,870 bytes with a contest length of 1,808; however, with some insertion of additional whitespace and formatting, the final submission still fit comfortably under the size limit, and looked like this:

```
#include <stdio.h>
#include <math.h>
#define E return
#define S for
char*J="LJFFF%7544x^H^XXHZZXHZ]]2#( #@@DA#(.@@%(0CAaIqDCI$IDEH%P@T@qL%
I%KBPBEP%CBPEaIqBAI%CAaIqBqDAI%U@PE%AAaIqBcDAI%ACaIaCqDCI%(aHCcIpBBH%I
AaIqB%AAaIqBEH%AAPBaIqB%PCDHxL%H@hIcBBI%E@qJBH#C@@D%aIBI@D%E@QB2P#E@'(
%C@qJBH%AAaIqBAI%C@cJ%"   "cJ"   "CH%C@qJ%aIqB1I%PCDI`I%BAaICH%KH+@'JI
3P%H@ABhIaBBI%P@S@PC#",     *j   ,*e;typedef float x;x U(x a){E a<0?0
typedef struct{x c,a,t;      }   y;y W={1,1,1},Z={0,0,0},B[99],P,C,M,I
;y G(x t,x a,x c){K.c=t   ;      K.t=c; K.a=a;E K;}int T=-1,b=0,r,F=-1
nt)=putchar,X=40,z=5,o,   a,     c,t=0   ,n,R;y A(y a,y b,x c){E G(a.c
+c*b.a,b.t*c+a.t);}x H=   .5,Y   =.66     ,I,l=0,q,w,u,i,g;x O(y a,y b
b.t+b.c*a.c+a.a*b.a;}x Q(){E A(P,M,T       ),O(K,K)<I?C=M,I=q:0;}y V(y
a,pow(O(a,a),-H));}x D(y p){S(I=X,P         =p,b=T; M=B[++b],p=B[M.c+=
++b],b<=r;Q())M=p.t?q =M_PI*H,w=atan2(   P.a-M.a,P.c-M.c)  /q,o=p.c-2,
o+a,w=q*(w>t+H*a?o:    w>t?t:w<o-H*a?t   :w<o?o:w),A(      M,G(cos(w),
1):A(M,p,U(O(A(P,M,T)    ,p)/O(p,p)));   M=P;M.a=-         .9;o=P.c/8-
/8+8; M=Q        ()?o&1    ?G(Y,0,0):W   :G(Y,Y,1);E       sqrt
int main(   int    L,char    **k){ S(e   =L>1?1[z=    0,   k]:J   ;*e
++e)S(o=a   =0,j   =J+9;(c=    *++j)&&   !(o&&c<    X&&(q=l+=w)   );o
32,b++[B]   =G(q   +=*j/8&3,*    j&7,0   ),B[r    =b++]=G((c/8&
T:1), (c&   7)+    1e-4,o>2),1:    (o    =(a    =(c-=X)<0?w=c+6   ,t=
?0:m(c),a        ):*++j)==((*e|32        )    ^z)&&1[j]-X));S(z   =3*
F<110;)S(L=-301;p=Z,++L<300;m(              p.c),m(p.a),m(p.t))S(c=T;-
=G(-4,4.6,29),d=V(A(A(A(Z,V(G(5,0         ,2)),L+L+c/2),V(G(2,-73,0)),
(30.75,-6,-75),20)),g=R=255-(n=z)*64;    R*n+R;g*=H){S(u=i=R=0;!R&&94>
A(h,d,i)));R=i<.01);S(N=V(A(P,C,               T)),q=d.t*d.t,s=M,u=1;-
U(i/3-D(A(h,N,i/3)))/pow(                             2,i));s=R?i=pow
M=V(G(T,1,2)),d,T))))                                     ,X),p=A(p,W,
O(N,M))*H*Y+Y,g*=                                             n--?Y-Y:
q,q,1); p=A(p,s                                                 ,g*u),
);d=A(d,N,-2*O                                                     (d,
```

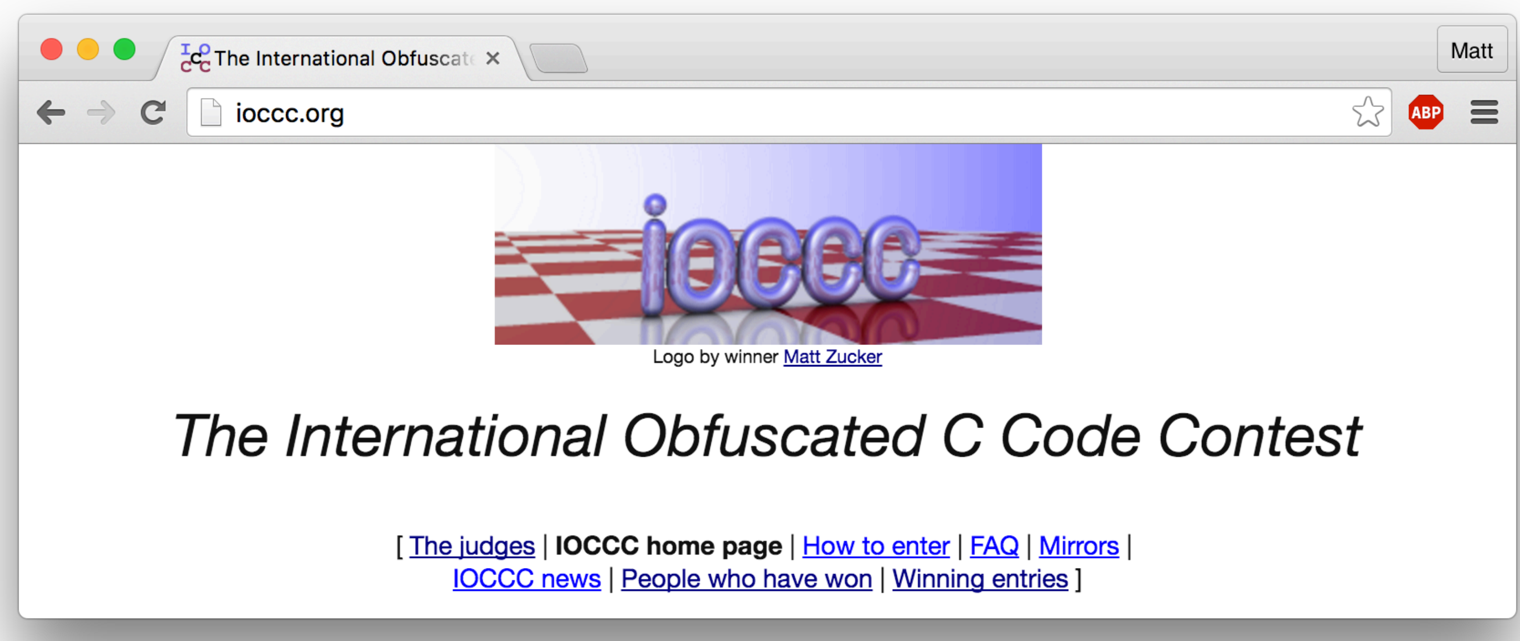
It's supposed to represent a canonical raytracing diagram, where an incident ray with direction hits a surface with normal and reflects in direction .

Victory and aftermath

On February 2, 2012, I was ecstatic to receive an email with the subject line **IOCCC 2011: Your program has won the 20th ICCCC.** My little raytracer would join the other entries on the Previous IOCCC winners page. Clearly, the final round of obfucsations had impressed the judges, who incredulously noted:

> *This entry uses no local variables. None! At! All!*

They also used the program to render a logo graphic for the IOCCC webpage, where it has remained to this day (no doubt to be replaced as soon as someone develops an even cooler obfuscated graphics program that spells out the contest name, I suppose). Here's a current screenshot:



Looking back, this IOCCC entry is probably my favorite program I've ever written. It just packs so much *stuff* into such a tiny and inscrutable space, and its output looks both fun and tangible, like a shiny sculpture you'd want to touch. Furthermore, being an IOCCC winner has helped me build up a healthy stockpile of programmer cred with my colleagues.

Just this week, in preparation for this blog entry, I ended up producing a heavily commented version of the final 4.6 code. It took me hours to finish – after nearly five years, I had forgotten exactly how gnarly the program is. I *wrote* the dang thing, and even I was having trouble analyzing all of the re-used global variables, comma operators, and nested ternary operators. Annotating the program definitely made me appreciate how nuts I was do write it in the first place.

I remember enjoying writing up the tongue-in-cheek author's comments to accompany the entry, and I've definitely enjoyed writing up this more thorough retrospective of the project as well. So, thanks to the IOCCC organizers for "providing a safe forum for poor C code"!

Postscript: printed card design

Update Aug 17, 2016: Working on this post finally prompted me to get business cards printed with the program on the back. After a few extra tweaks, I came up with a C++ edition that fits really nicely on a standard 3.5”x2” business card. Here it is, front and back:

Matt Zucker

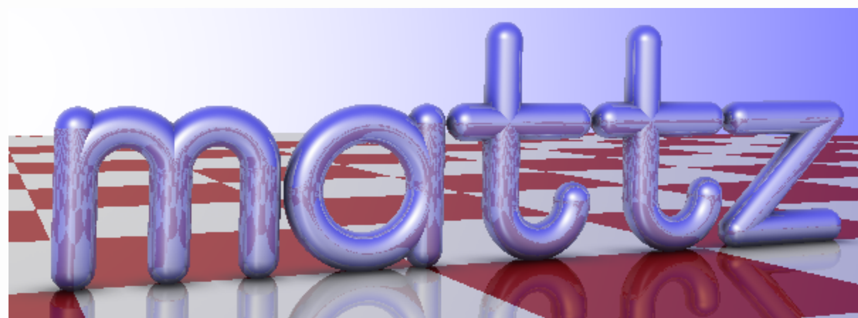
Associate Professor of Engineering

Swarthmore College
500 College Avenue
Swarthmore, PA USA 19081

http://swarthmore.edu/NatSci/mzucker1
mzucker1@swarthmore.edu

```
#include <stdio.h> // ./card > mattz.ppm # see https://goo.gl/JM9c2P
typedef double f;f H=.5,Y=.66,S=-1,I,y=-111;extern"C"{f cos(f),pow(f
,f),atan2(f,f);}struct v{f x,y,z;v(f a=0,f b=0,f c=0):x(a),y(b),z(c)
{}f operator%(v r){return x*r.x+y*r.y+z*r.z;}v operator+(v r){return
v(x+r.x,y+r.y,z+r.z);}v operator*(f s){return v(x*s,y*s,z*s);}}W(1,1
,1),P,C,M;f U(f a){return a<0?0:a>1?1:a;}v _(v t){return t*pow(t%t,-
H);}f Q(v c){M=P+c*S;f d=M%M;return d<I?C=c,I=d:0;}f D(v p){I=99;P=p
;f l,u,t;v k;for(const char*b="BCJB@bJBHbJCE[FLL_A[FLMCA[CCTT`T";*b;
++b){k.x+=*b/4&15;int o=*b&3,a=*++b&7;k.y=*b/8&7;v d(o%2*a,o/2*a);!o
?l=a/4%2*-3.14,u=a/2%2*3.14,d=p+k*-H,t=atan2(d.y,d.x),t=t<l?l:t>u?u:
t,Q(k*H+v(cos(t),cos(t-1.57))*(a%2*H+1)):Q(k+d*U((p+k*S)%d/(d%d)));}
return M=Q(v(p.x,-.9,p.z))?(int(p.x+64)^int(p.z+64))/8&1?Y:W:v(Y,Y,1
),pow(I,H)-.45;}v R(v o,v d,f z){for(f u=0,l=1,i=0,a=1;u<97;u+=l=D(o
+d*u))if(l<.01){v p=M,n=_(P+C*S),L=_(v(S,1,2));for(o=o+d*u;++i<6;a-=
U(i/3-D(o+n*i*.3))/pow(2,i));p=p*(U(n%L)*H*Y+Y)*a;p=z?p*Y+R(o+n*.1,d
+n*-2*(d%n),z-1)*H*Y:p;u=pow(U(n%_(L+d*S)),40);return p+p*-u+W*u;}z=
d.z*d.z;return v(z,z,1);} int main(){for(puts("P6 600 220 255");++y<
110;)for(f x=-301;P=R(v(-2,4,25),_(_(v(5,0,2))*++x+_(v(-2,73))*-y+v(
301,-59,-735)),2)*255,x<300;putchar(P.z))putchar(P.x),putchar(P.y);}
```

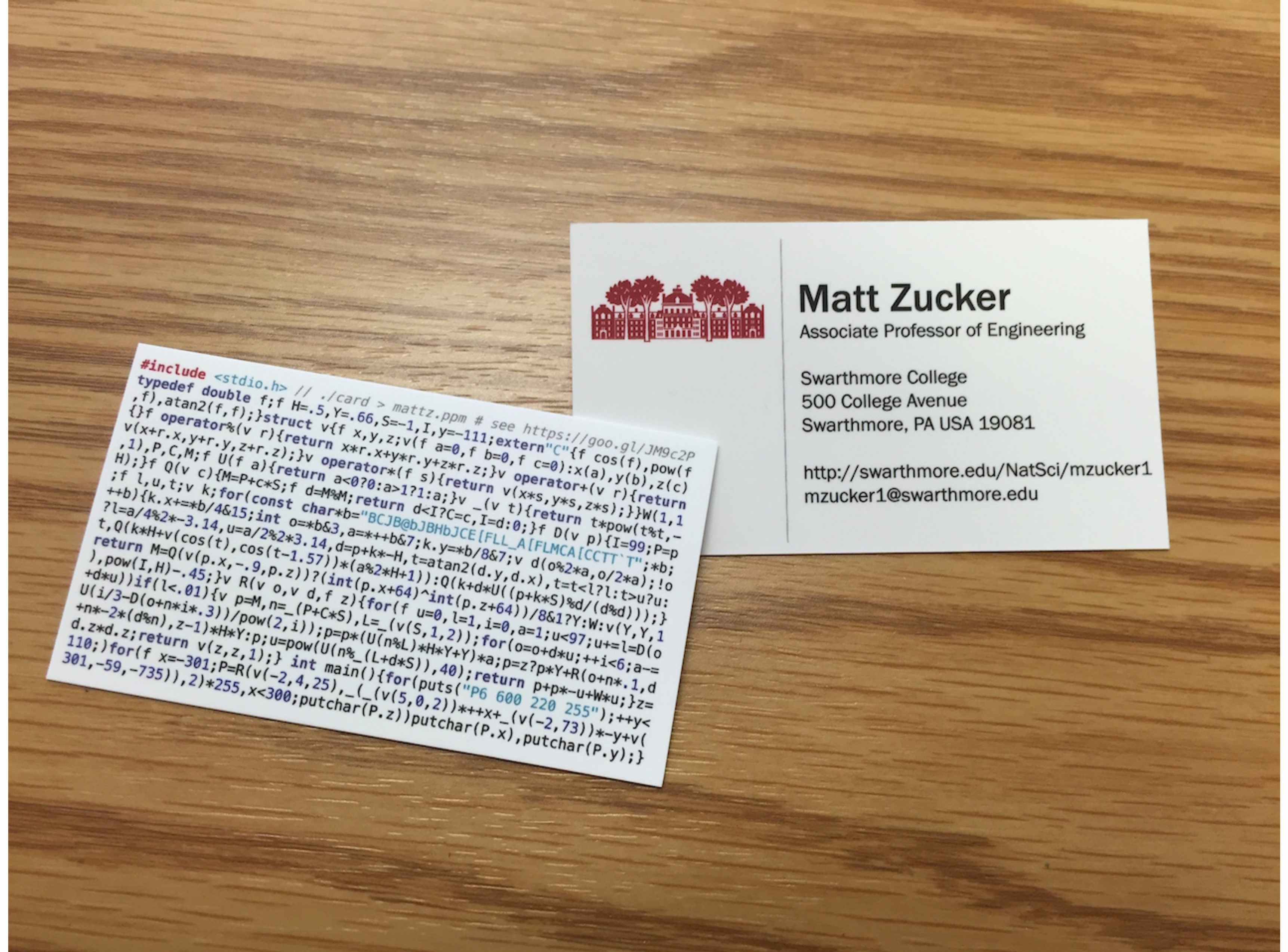
The URL in the top right points to this page, of course. The code produces this image:



What’s printed is version 3.1, with a bit of extra formatting. Reducing the size from 3.0 involved changing a couple `while` statements to `for` statements, replacing `if` with `?:`, and aggressively removing parentheses and brackets. To eliminate the whitespace at the top, I had to get rid of some preprocessor directives – this involved manually declaring functions from `<math.h>`, a dirty move that I am nonetheless particularly proud of. I also back-ported the increased shadow intensity from 4.3, which I thought looked nicer.

The cards have been ordered and should arrive before September!

Update Aug 24, 2016: The cards have arrived – here’s the actual, tangible, meatspace version:



MOO.com did a great job with the print, would order from them again. They are sturdy and vibrant, the only minor issue is the cardstock feels a bit slick on both sides despite the matte finish. Next time I might choose one rung higher on the menu of finishing options (I selected the lowest one).

Appendix: Version history

Version	Language	Size[3]	Color?	Reflections	AA	Usertext?	Output
0.1	C++	2,245	✓				
0.2	C++	1,945	✓				
0.3	C++	1,490	✓				
0.4	C++	1,378	✓				
0.5	C++	1,484	✓				
0.6	C++	1,342	✓				

1.0	C++	1,493				
1.1	C++	1,423				
1.2	C++	1,461				
1.3	C++	1,332				
1.4	C++	1,242				
2.0	C++	1,290	✓			
2.1	C++	1,278	✓			
2.2	C++	1,280	✓			
2.3	C++	1,267	✓			
2.4	C++	1,296	✓	✓		
3.0	C++	1,295	✓	✓		
3.1	C++	1,247	✓	✓		
4.0	C	1,422	✓	✓		
4.1	C	1,493	✓	✓	✓	
4.2	C	2,058	✓	✓	✓	✓
4.3	C	1,923	✓	✓	✓	✓
4.4	C	1,883	✓	✓	✓	✓
4.5	C	1,861	✓	✓	✓	✓
4.6	C	1,870	✓	✓	✓	✓

Comments

Comments are closed, see here for details.

Greg Popovitch · 2016-Aug-18
Very cool article, thanks for writing it! I'd love to get one of your business cards :-)

Crackity Jones · 2016-Aug-18
Presumably the mystery asterisks indicate that the identifiers are of type int* and vector* respectively?

edit: vector* rather than void*

> **Matt Zucker** · 2016-Aug-18
> Mystery asterisks in which version of the program?
>
> > **jleader** · 2016-Aug-18
> > I think Crackity is referring to the asterisks in the 3rd line of the identifier-to-datatype mapping comment from version 4.5
> >
> > > **Matt Zucker** · 2016-Aug-19
> > > Oh -- right, that totally makes sense. Mystery solved!

Amin Shah Gilani · 2016-Aug-18
Great read :)

pmsc · 2016-Aug-18
ompf still lives! Go to ompf2.com - it's a lot quieter than it used to be but many of the same people are there.

> **Matt Zucker** · 2016-Aug-19
> Thanks – too bad they don't have an archive of the old forum posts tho.

---

1. Fabien Sanglard did an admirably thorough analysis of Kensler's program in 2013, long after the 2011 IOCCC. ↩

2. These days, raymarching/sphere tracing forms the basis of the vast majority of the shaders on Shadertoy, which was founded and popularized by none other than Quilez himself. I have a bit of a presence there, too. ↩

3. Not the size of the raw file; C/C++ file sizes reported here are computed by the format_and_count.py utility included in the github repository. ↩ ↩2

4. Sadly, the comments didn't last throughout the entire program's development, which

meant I needed to re-discover a large amount of program functionality while writing this post. ↩

5. If this seems complicated, the 2013 rules introduced an "size tool", a standalone C program used to compute the official length of entries. ↩

6. That is, all else being equal, a binary shift right of some integer `i` by 3 bits is preferable to a shift right by 4 bits, because the former can be expressed as `i/8` and the latter must be written as `i/16` or `i>>4`, both of which are one byte longer. ↩