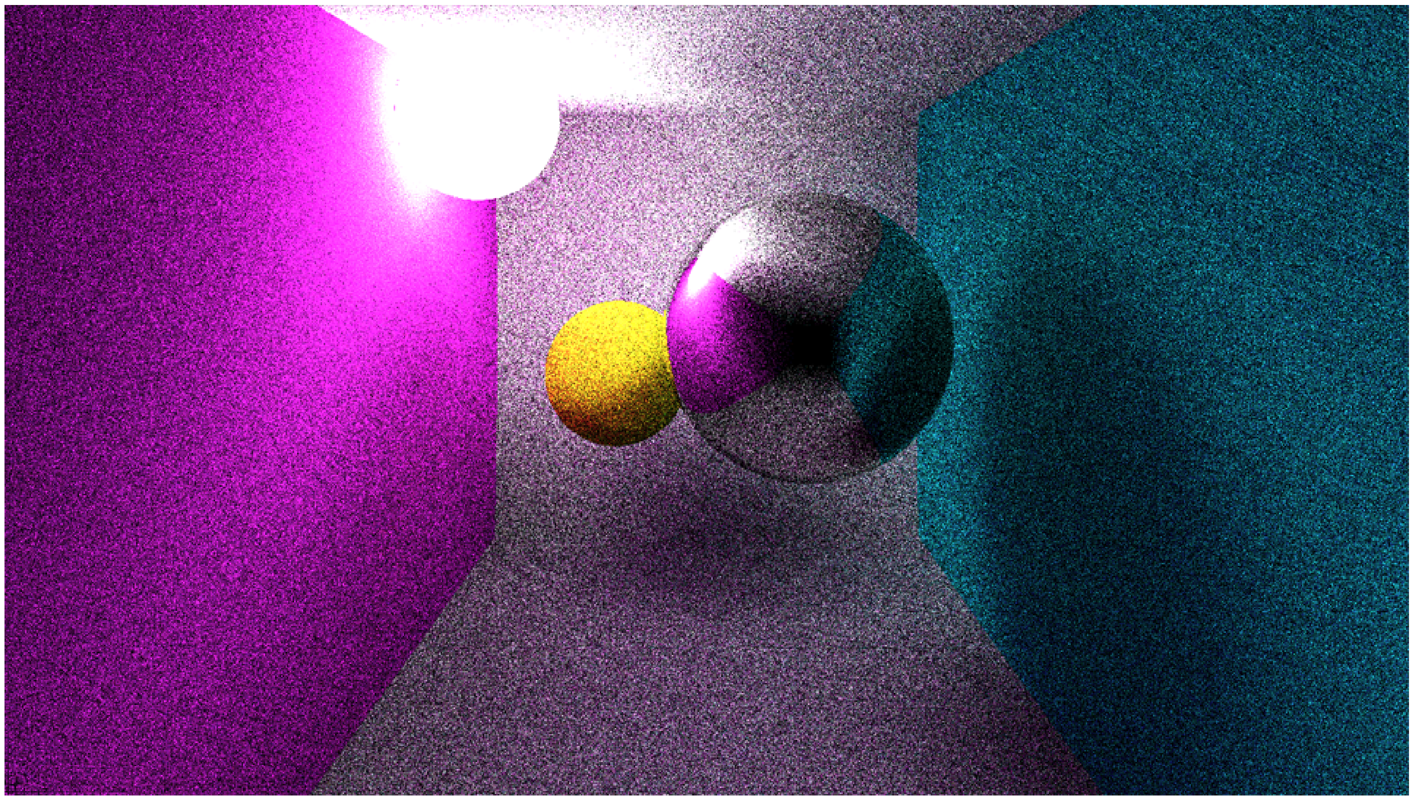
Making of Kornell Box

This is a short making of my Kornell Box intro, that got the 3rd place in Assembly 2013 1k intro competition. It is a raytracer that calculates 4 light bounces around a box. All light in the scene comes from the bright white sphere.



Reaching 1k and below

Getting to 1k with this intro was quite hard. The first versions of the fully functional raytracer were around 1600 bytes. From there I got it to around 1300 bytes fairly fast, but after that it came really hard to slice off even few bytes. However, I had quite few breakthrough such as moving away from Closure Compiler to manually tinyfied Javascript code, moving canvas creation to HTML and removing all functions from the shader, leaving only a single main loop. I also managed to squeeze my pseudorandom number generator into *fract(sin(seed += .7)*seed)*. Obviously it is pretty bad prng but it does its job for this intros purposes.

The images you see below are all versions of the raytracer when it was below 1024 bytes, however all of them were static, nothing was animated yet. I got as far as to add glass material with refractions and caustics seen in the last image. Unfortunately, the glass material was a bit buggy and nothing was animated, so I had to abandon hopes of getting the glass material into the final intro. I dropped that and replaced it with better colors, animated spheres and switching between the version with artifacts and the one without.

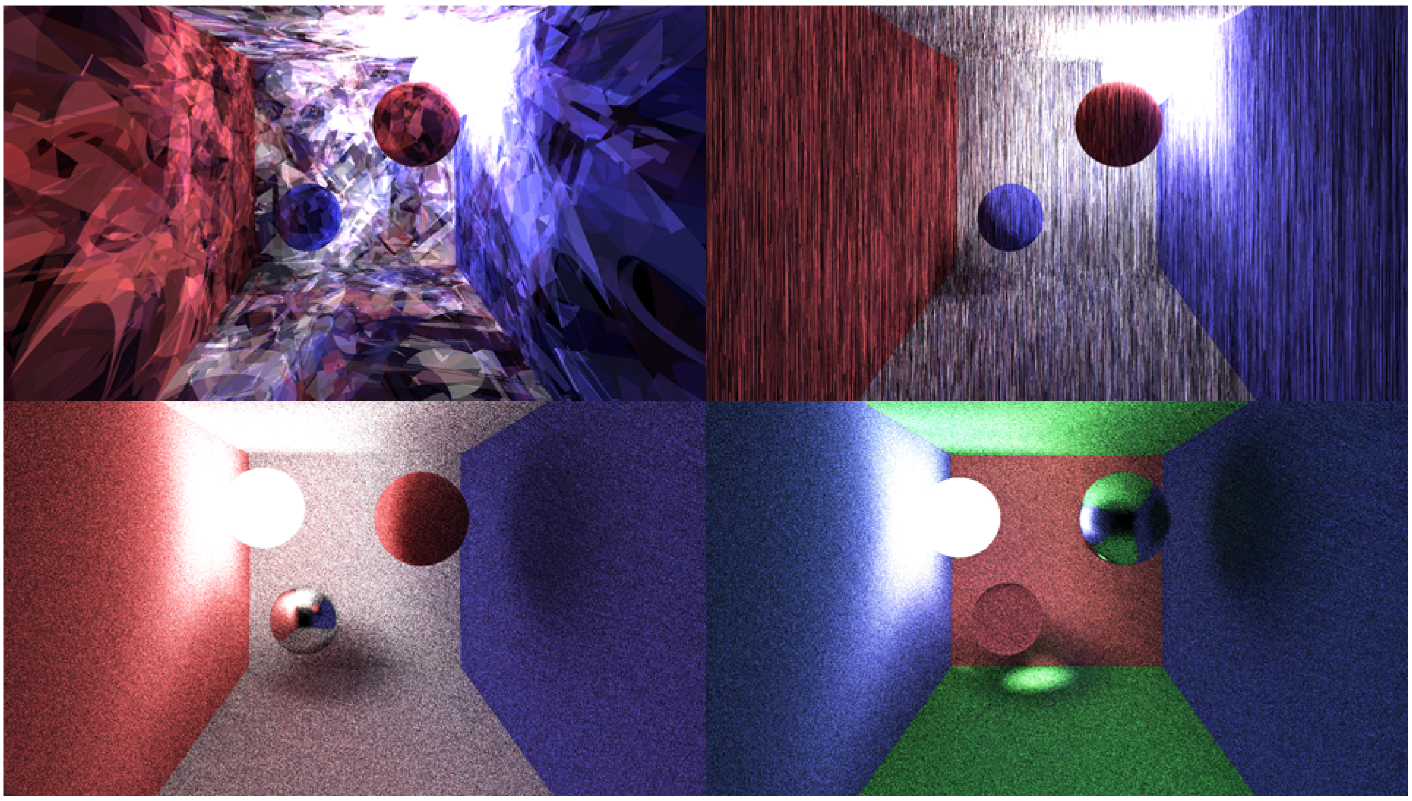
Tools

Google Closure Compiler was used at some point to minify the Javascript, but the final entry is entirely hand crafted to get it as small as possibly, optimizing things as using same variable names as within the fragment shader that does the actual work. This way it compresses into slightly smaller space.

Shader Minifier was used to minify the GLSL shader, I did not do any editing by hand. It is possible that some bytes might be gained through manual editing, but I doubt more than that.

jsexe was used to compress the final code into a PNG image and adding the html/javascript bootstrapper that decompresses the PNG and evaluates the javascript.

Notepad++ was used for writing all the code and with Hex editor plugin, for adding a canvas element to jsexefied file.



Javascript

In this intro Javascript is used only for setting up WebGL with absolutely minimum amount of code. It does not do anything except calculate time and update the WebGL view 10 times per second. Large part of the javascript code is not my own but taken in bits and pieces from around the web (p01 etc.).

If you put this through jsexe and add the canvas element with a hex editor to the resulting file, you will get the final entry i submitted to Assembly 2013 intro competition which is exactly 1024 bytes.

In the source code I also ponder between creating canvas for webgl context outside or inside the Javascript. I ended up doing it HTML. However hybrid approach would have been better (as I found out after deadline). By creating canvas element in HTML and then setting the style in Javascript I managed to squeeze the intro into 1011 bytes.

```
// This adds 70 bytes to file size, it is added with Hex editor after PNG packing
// It creates correctly sized and positioned canvas element that is required for webgl context
//<canvas id=F width=1280 height=720 style=position:fixed;top:0;left:0;>

// Optionally this could be used, but it usually takes more than 70 even thought its compressed
// unlike the HTML version above, which is simply added to final file as plain text
//c=document.createElement("canvas");
//document.body.appendChild(c);
//s=c.style;s.position="fixed";s.left=s.top=0;c.width=1280;c.height=720;

// Get webgl context and shorten all its function names with clever regex, this way we never have to
// full names of the functions which are pretty long
for(k in g=F.getContext("experimental-webgl"))
    g[k.match(/^..|[A-V]/g).join("")]=g[k];

// Save some space by not having to write g. in front of every webgl command
with(g){
    // Compile shaders
    for(a=crP(t=r=2);t;coS(s),atS(a,s))
        shS(s=crS(35634-t),
        // Fragment shader which does all the work and draws the scene
        // Since this is unreadable, I suggest you look at f1k.c instead which is the same thing,
        // but before going through Shader Minifier
        --t?"precision highp float;uniform float F;void main(){vec3 v=vec3(0);float i,f,s=.7<sin(F*"
        +".2)?F:gl_FragCoord.r*gl_FragCoord.g;for(int r=0;r<100;r++){vec3 g,n,c=vec3(1),d=vec3(0,.7"
        +",.1),b=normalize(gl_FragCoord.rbg/1440.-vec3(.44,-.2,.25)-d);for(int m=0;m<4;m++){vec3 k="
        +"vec3(0,10.25,0);i=100.;for(int e=0;e<8;e++){vec3 o=vec3(.2*cos(F),.2*sin(F),.34-F*.004),a"
        +"=k;f=.005;if(e<7)a=vec3(0,.7,.1).ggb,o=vec3(.12*cos(F*.7+3.),.12*sin(F*.7+3.),sin(F*.12)*"
        +".12);if(e<6)a=vec3(1),o=vec3(.12*cos(F*.7),.12*sin(F*.7),-sin(F*.12)*.12),f=.012;if(e<5)a"
        +"=vec3(0,.7,.1).ggg,o=k=-k.brg,f=100.;if(e==1)a=vec3(0,.7,.1).bgg;if(e==4)a=vec3(0,.7,.1)."
        +"gbg;float t,l=dot(b,o-d),C=l*l-dot(o-d,o-d)+f;if(0.<C)if(.0001<(t=l-sqrt(C)))if(t<i)i=t,n"
        +"=a,g=o;}if(i==100.)break;if(n==k){v+=12.*c;break;}d+=b*i;c*=n;g=normalize(d-g);b=n==vec3("
        +"1)?reflect(b,g):sign(dot(k=normalize(vec3(fract(sin(s+=.7)*s)-.5,fract(sin(s+=.7)*s)-.5,f"
        +"ract(sin(s+=.7)*s)-.5)),g))*k;}}gl_FragColor=vec4(.02*v*sign(166.-F),1);}"

        // Vertex shader, sadly we can't live without it
        :"attribute vec4 a;void main(){gl_Position=a;}");

    // Setup triangle on which the shader is applied
    veAP(enVAA(biB(34962,crB())),
        2,
        5126,
        liP(a),
        usP(a),
        buD(34962,new Float32Array([1,1,1,-3,-3,1]),35044)
    );

    // Main loop, hard coded to 10 FPS with 100 interval
    setInterval(
        // The shader only takes 1 parameter, time, as input
        // We have to use full function name "uniform1f" because there are too many collsions for
        // this function name in our renamer regex, however we use the word "uniform" in our GLSL
        // shader declaration and combine this with compression and using full name for this
        // particular function is actually cheap sizewise
        'g.drA(4,g.uniform1f(g.geUL(a,"F"),r+=.2),3)',
        100)
};
```

Shader

This is the WebGL fragment shader that, when put through Shader Minifier, produces the garbled code shown in the Javascript above.

Some fun trivia can be calculated from the for loops below. We shoot 921 600 000 rays into the scene per second which in theoretical worst case scenario means GPU has to calculate 36 864 000 000 ray to sphere intersections each second. Since most rays terminate before 5 bounces are up the real amount is probably closer to 20 billion.

```
precision highp float;

// Takes time as parameter
uniform float F;

void main() {
    vec3 result = vec3(0);

    // When seed is constant, time in this case, you get massive artifacts, using screen space
    // coordinates multiplied gives smooth noise, varying between these two gives the funky effect
    float distance, radius, seed = .7 < sin(F*.2) ? F : gl_FragCoord.x*gl_FragCoord.y;

    // We shoot 100 rays per pixel to smoothen out noise
    for (int j=0; j<100; j++) {
        vec3 spherePos;
        vec3 mat;
        vec3 tint = vec3(1);
        vec3 rayO = vec3(0,.7,.1);

        // Calculate ray direction based on screen space coordinates
        vec3 rayD = normalize(gl_FragCoord.xzy/1440. - vec3(.44,-.2,.25) - rayO);

        // WebGL doesn't support recursion in shaders, so we use for loop to calcualte multiple
        // light bounces in the scene
        for (int a=0; a<4; a++) {
            // Wall positions, we swap axis to get different sphere positions
            vec3 walls = vec3(0, 10.25,0);
            distance=100.;
            for (int k=0; k<8; k++) { // Go through all spheres
                // Setup sphere and materials
                vec3 sphere = vec3(.2 * cos(F), .2 * sin(F), .34 - F*.004); // Default to light
                // Just use some random vector to detect lights, m=x, if (m==x) light
                vec3 m = walls;
                radius=.005;
                if (k<7) // Diffuse sphere
                    m = vec3(0,.7,.1).yyz,
                    sphere=vec3(.12 * cos(F*.7+3.), .12 * sin(F*.7+3.),sin(F*.12)*.12);
                if (k<6) // Mirror sphere
                    m = vec3(1),
                    sphere=vec3(.12 * cos(F*.7), .12 * sin(F*.7),-sin(F*.12)*.12),
                    radius = .012;
                if (k<5) // Walls
                    m = vec3(0,.7,.1).yyy,
                    // Here we do walls=-walls.zxy which basically over 5 iterations gives us all
                    // 5 walls you see, left, right, top, bottom and back wall
                    sphere=walls=-walls.zxy,
                    radius = 100.;
                if (k==1)
                    m = vec3(0,.7,.1).zyy;
                if (k==4)
                    m = vec3(0,.7,.1).yzy;
                // Intersect sphere
                float d, b=dot(rayD, sphere - rayO),
                    det=b*b-dot(sphere - rayO, sphere - rayO)+radius;
                // Check if we hit something
                if (.0 < det)
                    if (.0001 < (d=b-sqrt(det)))
                        if (d < distance) {
                            // If we did, set the materials, sphere position and hit distance
                            distance = d;
                            mat = m;
                            spherePos = sphere;
                        }
            }
            if (distance == 100.) // Ray missed scene, it bounces no more
                break;
            if (mat == walls) { // We hit light, add tinted color to result
                result += 12. * tint;
                break;
            }
            rayO += rayD * distance; // Calculate intersection point
            tint *= mat; // Apply color

            // Store normal of the intersection into spherePos
            spherePos = normalize(rayO - spherePos);

            // Decide if material is white which means its mirror or something else then its diffuse
            rayD = mat == vec3(1) ?
```

```
                // Reflection
                reflect(rayD, spherePos) :
                // Diffuse
                // fract(sin(seed += .7)*seed) is our random function
                // We create random vector then check if it is facing same way as normal,
                // if not then we flip it
                sign(dot(walls=normalize(vec3(fract(sin(seed += .7)*seed)-.5,
                                                fract(sin(seed += .7)*seed)-.5,
                                                fract(sin(seed += .7)*seed)-.5)),
                                    spherePos))*walls;
            }
        }
        // Set color, if time is over 166. then set color to black, otherwise some artifacts keep
        // appearing from inaccurate sphere interscetions that show even after the light has moved
        // outside of the box, sign(166. - F) returns 1 when F is below 166. and -1 when its above, so
        // multiplying color with that gives us the effect we want
        gl_FragColor = vec4(.02 * result * sign(166. - F), 1);
}
```

Thoughts

This was my second real time production, first one being Highway 4k. This started as a technical experiment to see if raytracer can fit into 1 kilobyte and the results you see above. With some additional size optimizations, perhaps it might also be possible to add glass material with refractions.

Credits

These are not comprehensive credits but I hope I remembered to name most of the people who shared their code on the web and thus helped create this intro.

- http://www.bitsnbites.eu/?p=112
- http://daeken.com/superpacking-js-demos
- http://www.p01.org/
- http://www.ctrl-alt-test.fr/
- http://pouet.net/prod.php?which=59298
- http://www.kevinbeason.com/smallpt/