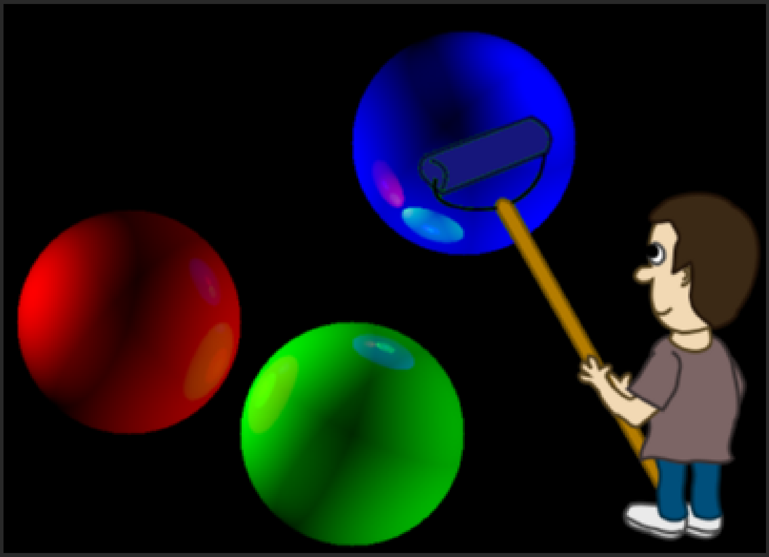
A Ray Tracer - Part 2

Submitted by PurpleAlien on Wed, 12/28/2011 - 01:17

Tags: C, Graphics, Mathematics, Programming, Ray Tracing

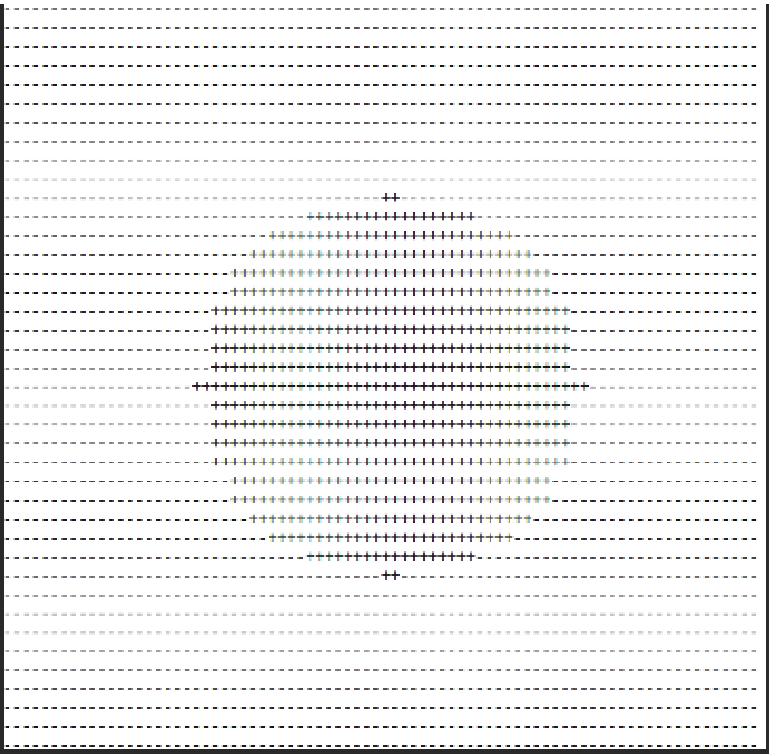
Last time we build up the basics of the raytracer: some essential math and the corresponding function implementations and structures in the C programming language. We will expand the ray tracer this time to generate actual graphics and to generate these images with multiple spheres, light sources, and reflections.



If you compiled and ran the example attached to the last article, you should have noticed a console output which looks like this:

Tag Cloud
Botnia Squid Computer Security SMS Lightning Efika Mathematics Electronics Fiinland Ray Tracing Genesi School Other UAV C Retro ARM Tracker Cryptography Computer Architecture MIPS Embedded Gaming Graphics Qt RoboCup First Post Network Linux Neon Future Electronics FTF Programming AEU San Antonio Finland Wireless Cloud Robots Hardware GPS

View mobile site



Not super exciting, I know, but it verifies the correctness of our ray/sphere intersect function. Let's take this one step further and create an actuall graphic which you can open in an image viewer.

Computer images are built up out of pixels (picture elements). Each pixel can represent any colour made up from the three primary colours red, green and blue. When we say that an image is '24-bit colour', we mean that each pixel is built up out of 3 values, each 8 bit long. In this way, we can let the red, green and blue component vary in value between 0 and 255 because $2^8 = 256$. With 24 bit colour, we have enough to represent over 16.7 million colours.

Let's start with something simple. Suppose we want to programatically create the Finnish flag:

The first thing we need to do is agree on the format we're going to use. There are many image formats such as jpeg, png, gif, etc. Most of these are quite complex to generate unless you use existing libraries. For this tutorial however, we will use something very simple: the ppm (portable pixmap) format. The ppm format is a simple uncompressed format from the Netpbm family of image formats. The PPM format has the following components (also called the header) needed for the file to be identified and displayed correctly by the image viewer:

- A "magic number" identifying the file type, in our case this will be "P6"
- Whitespace (any TAB/SPACE/NEWLINE)
- Width of the image, ASCII decimal representation
- Whitespace (any TAB/SPACE/NEWLINE)
- Height of the image, ASCII decimal representation
- Whitespace (any TAB/SPACE/NEWLINE)
- Maximum colour value, ASCII decimal representation
- Whitespace (single character, usually newline)
- Actual image data

Everything up until the image data can be written in C as follows:

```
fprintf(f, "P6 %d %d %d\n", width, height, 255);
```

Where 'f' is the pointer to an opened and writeable file. We will get back to this later. 'P6' in the PPM format indicates 'raw' PPM format. You can read more about the details in the links mentioned above.

So what is the image data composed of? It simply is the colour information per pixel, for all the pixels in the image. In other words it is the data represented by 3 bytes per pixel (8 bit per colour, 24 bit total) multiplied by the width and the height of the image. Therefor, a 24 bit image, measuring 800 pixels wide with a height of 600 pixels will contain 3 x 800 x 600 bytes of data, making 1.44MB of data. Interesting note: this is the amount of data you could store on a standard PC formatted double sided high density 3 1/2 inch floppy disk.

To open a PPM file, you need to install an appropriate viewer since Microsoft Windows does not come with PPM support by default. Linux should be able to open it out of the box. For Windows (and Mac) users, I've found these programs that should be fien for displaying PPM images:

For Windows: http://www.irfanview.com/

For Mac: http://www7a.biglobe.ne.jp/~ogihara/software/OSX/toyv-eng.html

To write a PPM file, we can write a function in C which looks like this:

```
/* Output data as PPM file */
void saveppm(char *filename, unsigned char *img, int width, int height){

    /* FILE pointer */
    FILE *f;

    /* Open file for writing */
    f = fopen(filename, "wb");

    /* PPM header info, including the size of the image */
    fprintf(f, "P6 %d %d %d\n", width, height, 255);

    /* Write the image data to the file - remember 3 byte per pixel */
    fwrite(img, 3, width*height, f);

    /* Make sure you close the file */
    fclose(f);

}
```

The pointer *img points to our image data, which consists of 3*width*height bytes of data, representing each pixel with its three colour components. Since each colour component can vary between 0 and 255, we can easly create the two colours needed for our flag. White can be represented with each component fully on, that is r=255, g=255, and b = 255. Similarly, the colour blue could be r=0, g=0, and b=128. This will give one shade of blue; you could use others by varying the value of each component.

The image data itself we can store in an array. Later on we can use dynamic memory allocation, but for now, a statically allocated array will do fine. Inside this array, we have to set the correct pixels white and blue in order to reporesent the Finnish flag. This is a matter of pretending you're in front of a canvas and drawing certain sections in a certain colour.

In C, we could do this as follows:

```
#define WIDTH  800
#define HEIGHT 500

int main(int argc, char *argv[]){

        unsigned char img[WIDTH * HEIGHT * 3];
        int i,j;

        for(i=0; i<HEIGHT; i++){
                for(j=0; j<WIDTH; j++){
                        if( (j>=250 && j<350) || (i>=200 && i<300) ){
                                img[(j + i*WIDTH)*3 + 0] = 0;
                                img[(j + i*WIDTH)*3 + 1] = 0;
                                img[(j + i*WIDTH)*3 + 2] = 128;
                        }else{
                                img[(j + i*WIDTH)*3 + 0] = 255;
                                img[(j + i*WIDTH)*3 + 1] = 255;
                                img[(j + i*WIDTH)*3 + 2] = 255;
                        }
                }

        }

        saveppm("image.ppm", img, WIDTH, HEIGHT);

return 0;
}
```

Within each iteration of the inner for loop, we draw one pixel. After one complete cycle of the outer for loop, one horizontal line is fully drawn. Within the inner for loop, we decide the colour each pixel will have, and set it's triplet (r,g,b) to the correct values. Whether a pixel becomes blue or wite is defined by the if statement in the inner for loop. It just checks if we are in the area of the flag that needs to be blue or white, calculated in advance knowing the width and height of the image. At the end, the image is written to a file, and you should be able to open it with an image viewer. The full version of this program is attached to this post (flag1.c).

Back to our ray tracer. We can now go back to the program we made last time and, instead of generating an ASCII output with a couple of printf() statements we can colour our sphere in any colour we want. We can also generate an image that is much larger than the 40x40 'image' we created before. As an example, let's build an 800x600 image with a red sphere on a black background. The resulting program is also attached (raytrace_sphere_1.c).

Hardly realistic, but it provides a start to make things look much better. The reason why it doesn't look realistic is because we're missing probably the most important element of a scene: "Lights, Camera, Action". We have the camera and the action (the sphere) - now all we need is light! First thing we want to do is change the way colour is assigned. After all, colour is a property of the object in question and should hence be assigned as such. We can change our code a little so that we have an element called 'colour' which contains the three colour components of the final shade. We can easily do this again with a structure:

```
/* Colour definition */
typedef struct{
        float red, green, blue;
}colour;
```

Why floats? We're going to need a way to represent the colour as a precentage. While one could calculate in absolute values of 0 to 255, it is easier to be able to say "colour this object with 50% red, 30% green and 10% blue". Floating points allow us to do that easily by specifying a value between 0.0 and 1.0, which represent the percentages.

As we said before, we need lights. Lights within this context have two things: a position in 3D space and an intensity per colour component. This we can represent as follows:

```
/* Light definition */
typedef struct{
        vector pos;
        colour intensity;
}light;
```

We want to be able to create a couple different materials and assign these to the sphere. A material in this context will be defines as an entity that can be assigned to an object, defining its reflectivity (0% - 100%) and diffuse colour. Diffuse colour will indicate the colour the object will reflect when being lit. Reflectivity will be used to determine how 'shiny' an object is and will act as a mirror for other objects. A material can be constructed as:

```
/* Material definition */
typedef struct{
    colour diffuse;
    float reflection;
}material;
```

Similarly, our sphere object can now become:

```
/* Sphere Primitive definition */
typedef struct{
        vector pos;
        float radius;
        int material;
}sphere;
```

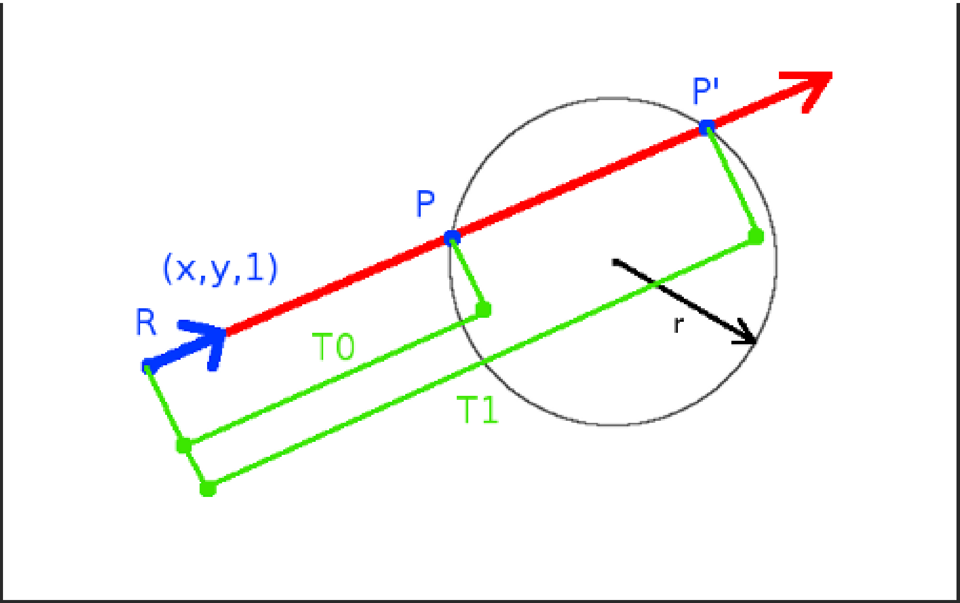
Here, material is the id which will be used to select the assigned material from an array of materials. The idea now, simplified, is to shoot our ray into the scene, and should it hit the sphere, bounce it off of the sphere (reflect) and check if it hits the lightsource. If it does, this point on the sphere is lit, otherwise it is not.

We now have to develop a way to use these newly defined entities. Let's start with building up a scene. We have several spheres we want to show (instead of just the one from the previous example). We also want to position some lightsources, since one lightsource will not be sufficient to provide realistic lighting. For now we will go through the process of positioning them one by one, but later on it would be easier to use a text file describing the scene to set this up. We will start with three materials, three lightsources and three spheres, all in their proper arrays. I'm not going to paste the code for that here since it's quite long and repetitive, but you can see the code in question, together with the next couple of topics in raytrace_sphere_1.c attached to this post.

The next thing we should do is to revise our ray-sphere intersection function. The way it is currently written it will only return true or false indicating intersection or not. We should however be able to find out exactly where this point of intersection is. Moreover, in case there are two intersection points, we want to know the one closest to the screen. That is the one that we actually see - the other one is on the other side of the sphere, and not visible. To accomplish this, we can modify the function as follows:

```
/* Check if the ray and sphere intersect */
bool intersectRaySphere(ray *r, sphere *s, float *t){

        bool retval = false;

        /* A = d.d, the vector dot product of the direction */
        float A = vectorDot(&r->dir, &r->dir);

        /* We need a vector representing the distance between the start of
         * the ray and the position of the circle.
         * This is the term (p0 - c)
         */
        vector dist = vectorSub(&r->start, &s->pos);

        /* 2d.(p0 - c) */
        float B = 2 * vectorDot(&r->dir, &dist);

        /* (p0 - c).(p0 - c) - r^2 */
        float C = vectorDot(&dist, &dist) - (s->radius * s->radius);

        /* Solving the discriminant */
        float discr = B * B - 4 * A * C;

        /* If the discriminant is negative, there are no real roots.
         * Return false in that case as the ray misses the sphere.
         * Return true in all other cases (can be one or two intersections)
         *  t, the solution to the equation represents the closest point
         * where our ray intersects with the sphere.
         */
        if(discr < 0)
                retval = false;
        else{
                float sqrtdiscr = sqrtf(discr);
                float t0 = (-B + sqrtdiscr)/(2);
                float t1 = (-B - sqrtdiscr)/(2);

                /* We want the closest one */
                if(t0 > t1)
                        t0 = t1;

                /* Verify t0 larger than 0 and less than the original t */
                if((t0 > 0.001f) && (t0 < *t)){
                        *t = t0;
                        retval = true;
                }else
                        retval = false;
        }

return retval;
}
```

The value 't' represents the length of the closest intersection. We can illustrate this as follows:



The value 't' is a scalar, and in order to find the vector $\vec{P}$, we have to write a function which takes this scalar and multiplies it with our vector $\vec{R}$ which is the direction of our ray. Multiplication of a vector with a scalar is just multiplying each component with the scalar. We know this already from part 1, in which we discussed parameterizing. Since the direction of our ray is always $(0, 0, 1)$, the scalar multiplication will result in a scaled vector representing the distance from the start of our ray to the point of impact on the sphere. Adding this scaled vector to the start position of our ray will give the vector $\vec{P}$. One thing I'd like to note here is the check whether t greater than 0, which is written as t0 > 0.001f in the code. The reason for using 0.001 instead of 0 has to do with floating point comparison and precision when representing floating point numbers. You can read more about this here.

We will need two additional functions in our arsenal to deal with the above: scaling and adding vectors:

```
/* Calculate Vector x Scalar and return resulting Vector*/
vector vectorScale(float c, vector *v){
        vector result = {v->x * c, v->y * c, v->z * c };
        return result;
}

/* Add two vectors and return the resulting vector */
vector vectorAdd(vector *v1, vector *v2){
        vector result = {v1->x + v2->x, v1->y + v2->y, v1->z + v2->z };
        return result;
}
```

Now that we have materials, spheres and lights, it's time to start working on the main objective. We need to simulate how light interacts with the spheres and the camera to build a realistic image. Each point on each sphere can be lit or unlit. The points that are lit can be lit with different intensities depending on how much light reaches the point in question. With our ray which we shoot through the screen will need to determine if it hits a point on one of the spheres, if light reaches this point. Furthermore, light reflects of off objects so from the impact point we reflect the ray and perhaps hit another sphere.

Our algorithm will look, simplified, like this:

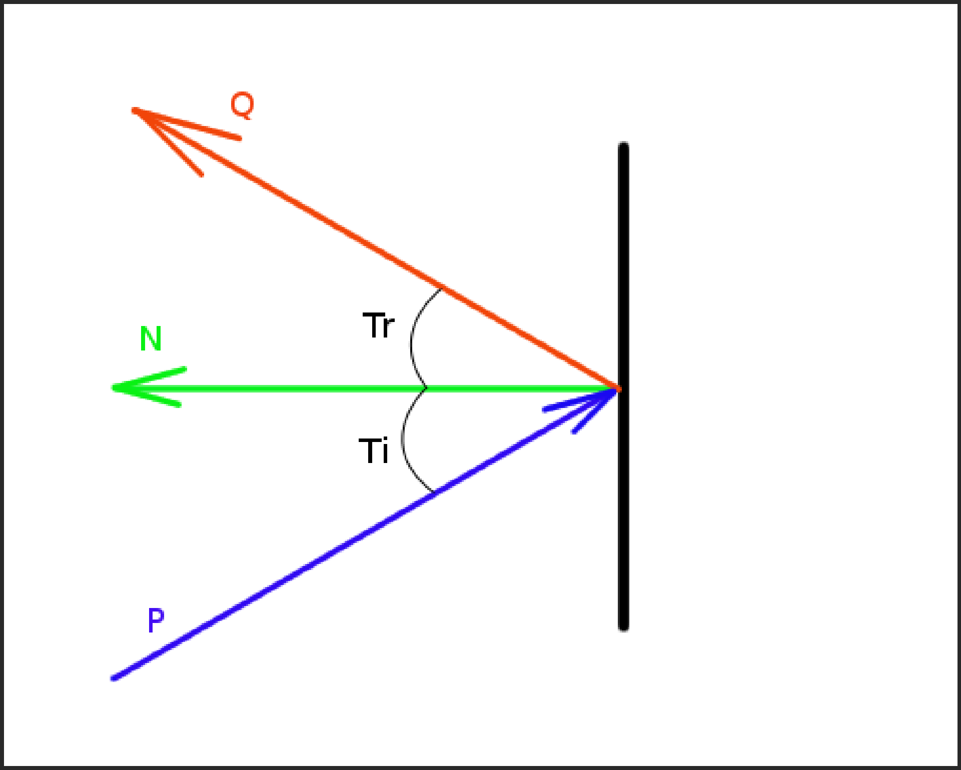
```
for(HEIGHT){
 for(WIDTH){
     do for every ray:
        - Find closest ray/sphere intersection:
          * Iterate over every sphere

        - Check if we reach a lightsource from this point
          * Iterate over every lightsource
          * Find right colour

        - Either go with reflected ray or go to next pixel
 }
}
```

To get the colour and intensity of each pixel, we will apply Lambertian reflectance to model diffuse reflection whereby the light will be reflected in equally in all directions from the point where the light hits the object. This is a very simple model where: $I_D = \vec{L}.\vec{N}CI_L$. Here, $\vec{N}$ is the normal at the point of intersection, and $\vec{L}$ denotes the normalized vector from the point of intersecton to the light source. C represents the colour of the sphere and I_L is the intensity of the incoming light. The resulting I_D scalar represents the surface brightness at that point.

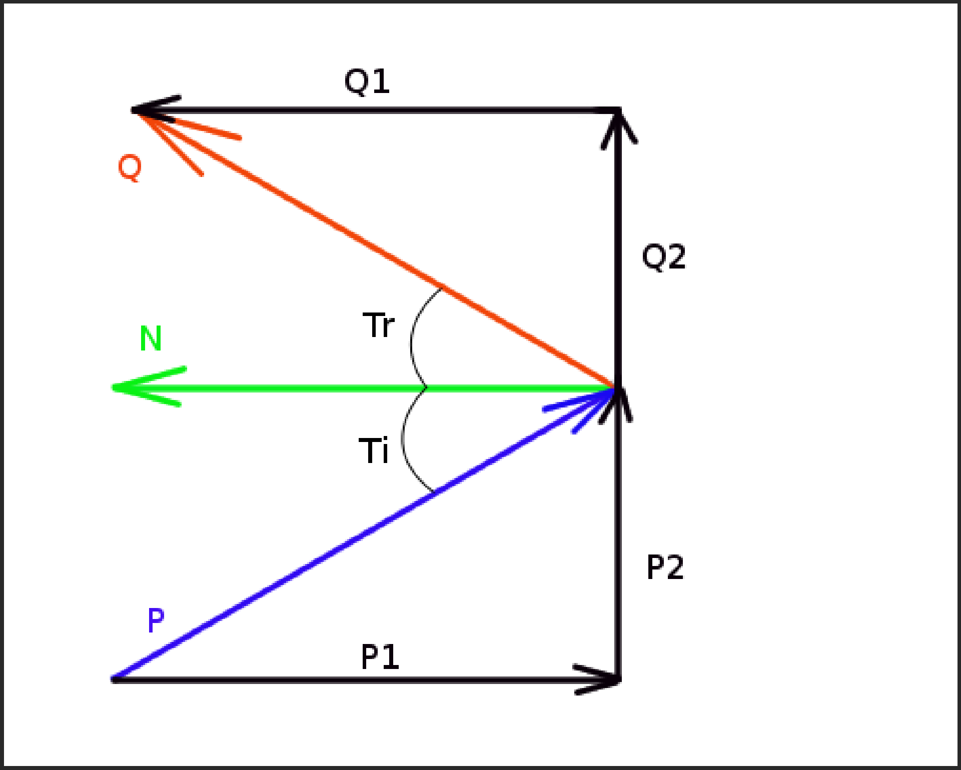
We will program this later as follows:

```
- For each lightsource that can be reached:
     * Calculate Lambert dot product with material reflection
     * Calculate each colour component, consisting of:
- Lambert dot product result
- Per colour intensity of the incoming light
```

We then need to get the reflection of our incoming ray and see if we hit another sphere and repeat the process. Getting the reflected ray is simple according to the law of reflection :



The angle between the incoming ray and the normal is equal to the angle between the reflected ray and the normal ($\theta_r = \theta_i$). The normal on a point on a sphere is a vector which can be constructed by drawing it starting at the center of the sphere and going through our point of intersection. Mathematically, we can determine this when we present the situation as in this drawing:



When we assume that $\vec{P}$ and $\vec{Q}$ are normalised, we know from basic trigonometry that:

$$|\vec{Q_1}| = cos(\theta_r) = cos(\theta_i) = |\vec{P_1}|$$

$$|\vec{Q_2}| = sin(\theta_r) = sin(\theta_i) = |\vec{P_2}|$$

From the drawing above, we can also see that:

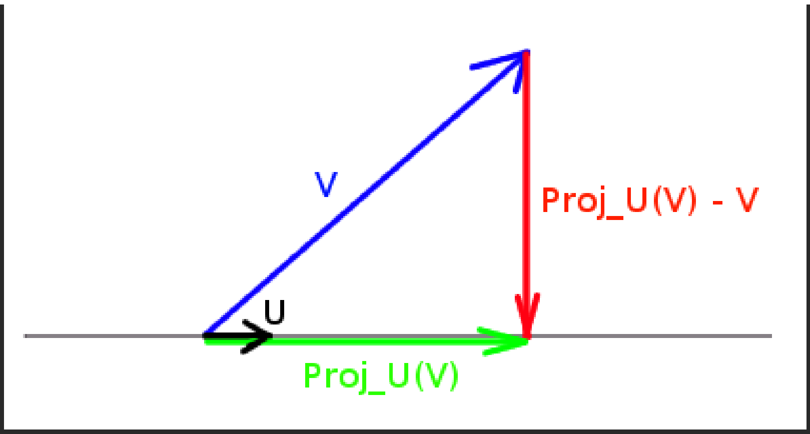
$$\vec{Q_1} = -\vec{P_1} \vec{Q_1} = \vec{Q_1}$$

After summation, we can get the direction:

$$\vec{Q} = \vec{Q_2} + \vec{Q_1} = \vec{P_2} - \vec{P_1}$$

We can work this further out. We know that: $\vec{Q} = \vec{P_2} - \vec{P_1}$.

Using orthogonal projection:



We know that $\vec{proj_U}(V) = a\vec{U}$ for some scalar a. Since $\vec{proj_U}(V) - \vec{V}$ is orthogonal to $\vec{U}$, we have that $(a\vec{U} - \vec{V}).\vec{U} = 0$.

Multiply out the dot product and solving for a gives:

$$a\vec{U}.\vec{U} - \vec{V}.\vec{U} = 0$$

$$a = \vec{V}.\vec{U}/\vec{U}.\vec{U}.$$

The result is that:

$$\vec{proj_U}(V) = (\frac{\vec{U}.\vec{V}}{\vec{U}.\vec{U}})\vec{U}$$

From the definition of the dot product, we know that $\vec{U}.\vec{U} = |\vec{U}|^2$.

Applying this to our situation gives that, with $\vec{N}$ our Normal:

$$\vec{P}1 = \frac{\vec{P}.\vec{N}}{|\vec{N}|^2}\vec{N}$$

Knowing that $|\vec{N}| = 1$

This gives $\vec{P}1 = (\vec{P}.\vec{N})\vec{N}$

Coming back to: $\vec{Q} = \vec{P_2} - \vec{P_1}$, we can solve this as follows:

$$\vec{Q} = [\vec{P} - (\vec{P}.\vec{N})\vec{N}] - (\vec{P}.\vec{N})\vec{N})$$

This can be solved further as:

$$\vec{Q} = \vec{P} - 2(\vec{P}.\vec{N})\vec{N})$$

That's it - we now have our reflected vector and continue to the next ray trace loop with this vector as a start.

The full C code can be found in the raytrace_sphere_2.c file attached. The main raytrace loop will continue to iterate over multiple reflections until a certain level (otherwise it might run forever). We can further enhance this program by adding shadows to the final image. Calculating shadows is pretty easy: one just has to trace the 'lightRay' found before determining the colour back to each object. If it intersects with another sphere, the point in question sits in the shadow of said sphere. The solution to this is in the attached raytrace_sphere_3.c.

Attachments

flag.c

flag.c **1.4 KB**

raytrace_sphere_1.c
raytrace_sphere_1.c **3.14 KB**

raytrace_sphere_2.c
raytrace_sphere_2.c **6.97 KB**

raytrace_sphere_3.c
raytrace_sphere_3.c **7.22 KB**

Tweet 7 Share 36 Share 2 Bookmarks 0 Share 306

Add new comment

COMMENTS

doubt

Submitted by Anonymous on Mon, 09/23/2019 - 20:12.

Hi, thank you so much for making ray tracing easy to understand!

I was wondering... how can I add more spheres to the code raytrace_sphere_2.c

Thank you !!!!

reply

Hi, Glad you liked it! To add

Submitted by PurpleAlien on Mon, 09/23/2019 - 21:01.

Hi,

Glad you liked it!

To add a sphere, have a look at the place in main() where you will find 'sphere spheres[3];' You can add one there (3 becomes 4), and after that set some coordinates for it and assign material:

```
        spheres[3].pos.x = 100;
        spheres[3].pos.y = 100;
        spheres[3].pos.z = 0;
        spheres[3].radius = 50;
        spheres[3].material = 1;
```

reply

Hi, thank you very for

Submitted by Anonymous on Mon, 09/23/2019 - 22:14.

Hi, thank you very for answering!

These are the changes I made to the code but its not outputting any file, I am very sorry for the inconvenience but I just started to learn about coding and raytracing :)

Thank You

```
/* A simple ray tracer */

#include <stdio.h>
#include <stdbool.h> /* Needed for boolean datatype */
#include <math.h>

#define min(a,b) (((a) < (b)) ? (a) : (b))

/* Width and height of out image */
#define WIDTH  800
#define HEIGHT 600

/* The vector structure */
typedef struct{
      float x,y,z;
}vector;

/* The sphere */
typedef struct{
        vector pos;
        float  radius;
    int material;
}sphere;

/* The ray */
typedef struct{
        vector start;
        vector dir;
}ray;

/* Colour */
typedef struct{
    float red, green, blue, orange;
}colour;

/* Material Definition */
typedef struct{
    colour diffuse;
    float reflection;
}material;

/* Lightsource definition */
typedef struct{
    vector pos;
    colour intensity;
}light;
```

```
/* Subtract two vectors and return the resulting vector */
vector vectorSub(vector *v1, vector *v2){
    vector result = {v1->x - v2->x, v1->y - v2->y, v1->z - v2->z };
    return result;
}

/* Multiply two vectors and return the resulting scalar (dot product) */
float vectorDot(vector *v1, vector *v2){
    return v1->x * v2->x + v1->y * v2->y + v1->z * v2->z;
}

/* Calculate Vector x Scalar and return resulting Vector*/
vector vectorScale(float c, vector *v){
        vector result = {v->x * c, v->y * c, v->z * c };
        return result;
}

/* Add two vectors and return the resulting vector */
vector vectorAdd(vector *v1, vector *v2){
        vector result = {v1->x + v2->x, v1->y + v2->y, v1->z + v2->z };
        return result;
}

/* Check if the ray and sphere intersect */
bool intersectRaySphere(ray *r, sphere *s, float *t){

    bool retval = false;

    /* A = d.d, the vector dot product of the direction */
    float A = vectorDot(&r->dir, &r->dir);

    /* We need a vector representing the distance between the start of
     * the ray and the position of the circle.
     * This is the term (p0 - c)
     */
    vector dist = vectorSub(&r->start, &s->pos);

    /* 2d.(p0 - c) */
    float B = 2 * vectorDot(&r->dir, &dist);

    /* (p0 - c).(p0 - c) - r^2 */
    float C = vectorDot(&dist, &dist) - (s->radius * s->radius);

    /* Solving the discriminant */
    float discr = B * B - 4 * A * C;

    /* If the discriminant is negative, there are no real roots.
     * Return false in that case as the ray misses the sphere.
     * Return true in all other cases (can be one or two intersections)
     * t represents the distance between the start of the ray and
     * the point on the sphere where it intersects.
     */
    if(discr < 0)
        retval = false;
    else{
        float sqrtdiscr = sqrtf(discr);
        float t0 = (-B + sqrtdiscr)/(2);
        float t1 = (-B - sqrtdiscr)/(2);

        /* We want the closest one */
        if(t0 > t1)
            t0 = t1;

        /* Verify t1 larger than 0 and less than the original t */
        if((t0 > 0.001f) && (t0 < *t)){
            *t = t0;
            retval = true;
        }else
            retval = false;
    }

return retval;
}

/* Output data as PPM file */
void saveppm(char *filename, unsigned char *img, int width, int height){
    /* FILE pointer */
    FILE *f;

    /* Open file for writing */
    f = fopen(filename, "wb");

    /* PPM header info, including the size of the image */
    fprintf(f, "P6 %d %d %d\n", width, height, 255);

    /* Write the image data to the file - remember 3 byte per pixel */
    fwrite(img, 3, width*height, f);

    /* Make sure you close the file */
    fclose(f);
}

int main(int argc, char *argv[]){

    ray r;

    material materials[4];
    materials[0].diffuse.red = 1;
    materials[0].diffuse.green = 0;
    materials[0].diffuse.blue = 0;
    materials[0].diffuse.orange = 0;
    materials[0].reflection = 0.2;

    materials[1].diffuse.red = 0;
    materials[1].diffuse.green = 1;
    materials[1].diffuse.blue = 0;
    materials[1].diffuse.orange = 0;
    materials[1].reflection = 0.5;

    materials[2].diffuse.red = 0;
    materials[2].diffuse.green = 0;
    materials[2].diffuse.blue = 1;
    materials[2].diffuse.orange = 0;
    materials[2].reflection = 0.7;
```

```
materials[3].diffuse.red = 0;
materials[3].diffuse.green = 0;
materials[3].diffuse.blue = 0;
materials[3].diffuse.orange = 1;
materials[3].reflection = 0.9;



sphere spheres[4];
spheres[0].pos.x = 50;
spheres[0].pos.y = 30;
spheres[0].pos.z = 0;
spheres[0].radius = 100;
spheres[0].material = 0;

spheres[1].pos.x = 200;
spheres[1].pos.y = 400;
spheres[1].pos.z = 0;
spheres[1].radius = 100;
spheres[1].material = 1;

spheres[2].pos.x = 500;
spheres[2].pos.y = 140;
spheres[2].pos.z = 0;
spheres[2].radius = 100;
spheres[2].material = 2;

spheres[3].pos.x = 100;
spheres[3].pos.y = 100;
spheres[3].pos.z = 0;
spheres[3].radius = 50;
spheres[3].material = 1;

light lights[4];

lights[0].pos.x = 0;
lights[0].pos.y = 240;
lights[0].pos.z = -100;
lights[0].intensity.red = 1;
lights[0].intensity.green = 1;
lights[0].intensity.blue = 1;
lights[0].intensity.orange = 1;

lights[1].pos.x = 3200;
lights[1].pos.y = 3000;
lights[1].pos.z = -1000;
lights[1].intensity.red = 0.6;
lights[1].intensity.green = 0.7;
lights[1].intensity.blue = 1;
lights[1].intensity.orange = 0.8;

lights[2].pos.x = 600;
lights[2].pos.y = 0;
lights[2].pos.z = -100;
lights[2].intensity.red = 0.3;
lights[2].intensity.green = 0.5;
```

```
lights[2].intensity.blue = 1;
lights[2].intensity.orange = 0.4;

lights[3].pos.x = 500;
lights[3].pos.y = 0;
lights[3].pos.z = -200;
lights[3].intensity.red = 0.3;
lights[3].intensity.green = 0.5;
lights[3].intensity.blue = 1;
lights[3].intensity.orange = 1;



/* Will contain the raw image */
unsigned char img[3*WIDTH*HEIGHT];

int x, y;
for(y=0;y<HEIGHT;y++){
  for(x=0;x<WIDTH;x++){

      float red = 0;
      float green = 0;
      float blue = 0;
      float orange = 0;

      int level = 0;
      float coef = 1.0;

      r.start.x = x;
      r.start.y = y;
      r.start.z = -2000;

      r.dir.x = 0;
      r.dir.y = 0;
      r.dir.z = 1;

      do{
        /* Find closest intersection */
        float t = 20000.0f;
        int currentSphere = -1;

        unsigned int i;
        for(i = 0; i < 4; i++){
          if(intersectRaySphere(&r, &spheres[i], &t))
            currentSphere = i;
        }
        if(currentSphere == -1) break;

        vector scaled = vectorScale(t, &r.dir);
        vector newStart = vectorAdd(&r.start, &scaled);

        /* Find the normal for this new vector at the point of intersection */
        vector n = vectorSub(&newStart, &spheres[currentSphere].pos);
        float temp = vectorDot(&n, &n);
        if(temp == 0) break;

        temp = 1.0f / sqrtf(temp);
        n = vectorScale(temp, &n);
```

```
                /* Find the material to determine the colour */
                material currentMat = materials[spheres[currentSphere].material];

                /* Find the value of the light at this point */
                unsigned int j;
                for(j=0; j < 4; j++){
                    light currentLight = lights[j];
                    vector dist = vectorSub(&currentLight.pos, &newStart);
                    if(vectorDot(&n, &dist) <= 0.0f) continue;
                    float t = sqrtf(vectorDot(&dist,&dist));
                    if(t <= 0.0f) continue;

                    ray lightRay;
                    lightRay.start = newStart;
                    lightRay.dir = vectorScale((1/t), &dist);

                    /* Calculate shadows */
                    bool inShadow = false;
                    unsigned int k;
                    for (k = 0; k < 4; ++k) {
                        if (intersectRaySphere(&lightRay, &spheres[k], &t)){
                            inShadow = true;
                            break;
                        }
                    }
                    if (!inShadow){
                        /* Lambert diffusion */
                        float lambert = vectorDot(&lightRay.dir, &n) * coef;
                        red += lambert * currentLight.intensity.red * currentMat.diffuse.red;
                        green += lambert * currentLight.intensity.green *
currentMat.diffuse.green;
                        blue += lambert * currentLight.intensity.blue * currentMat.diffuse.blue;
                    }
                }
                /* Iterate over the reflection */
                coef *= currentMat.reflection;

                /* The reflected ray start and direction */
                r.start = newStart;
                float reflect = 2.0f * vectorDot(&r.dir, &n);
                vector tmp = vectorScale(reflect, &n);
                r.dir = vectorSub(&r.dir, &tmp);

                level++;

            }while((coef > 0.0f) && (level < 15));

            img[(x + y*WIDTH)*3 + 0] = (unsigned char)min(red*255.0f, 255.0f);
            img[(x + y*WIDTH)*3 + 1] = (unsigned char)min(green*255.0f, 255.0f);
            img[(x + y*WIDTH)*3 + 2] = (unsigned char)min(blue*255.0f, 255.0f);
            img[(x + y*WIDTH)*3 + 3] = (unsigned char)min(blue*255.0f, 255.0f);

        }
    }

    saveppm("x6.ppm", img, WIDTH, HEIGHT);
```

```
return 0;
}
```

reply

Hi, It seems to work for

Submitted by PurpleAlien on Tue, 09/24/2019 - 06:47.

Hi,

It seems to work for me:

```
$ ls
simple.c
$ gcc -o simple simple.c -lm
$ ./simple
$ ls
simple  simple.c  x6.ppm
```

What are you seeing on your end?

reply

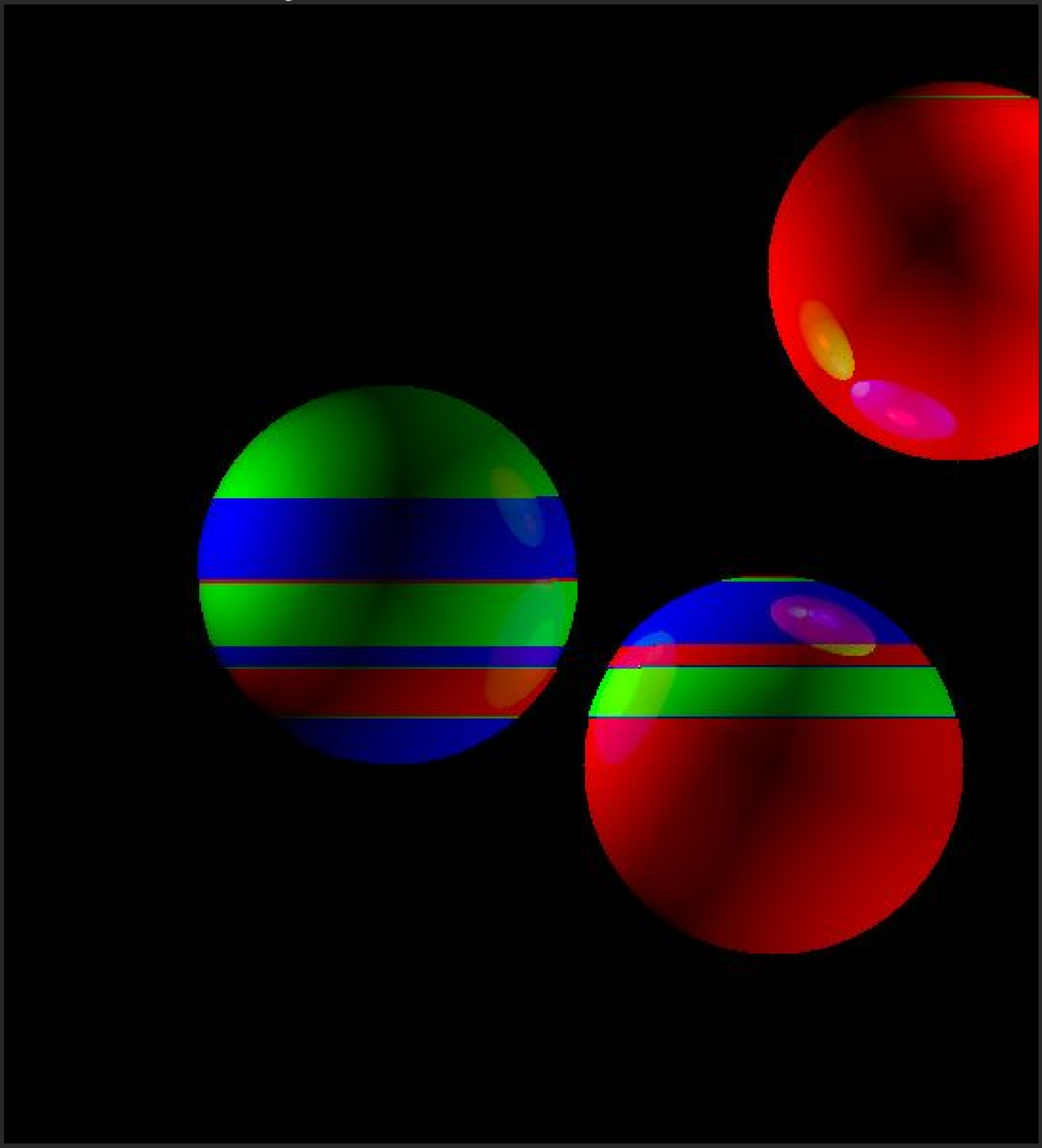
Ray tracing bug ?

Submitted by Anonymous on Sat, 05/20/2017 - 16:40.

Hi, thanks for your awesome course !

I used Visual Studio to compile the code and during the course I ran into two problems : the first is that setting a too high width and height (even with 800x600, and above) triggers a stack overflow error, i solved that by having a dynamicly allocated array for img.

The second is a visual bug :



I get the same result when I compile raytrace_sphere_2.c as it is in Visual Studio and Code::Blocks. Do you have an idea of where it comes from ? Thanks

reply

Possible solution

Submitted by Anonymous on Sat, 05/27/2017 - 20:48.

Hi

I had the same issue with colors on windows. I believe the issue is related to how windows write data to files. For me the issue was fixed by changing the file opening flag from "w" to "wb". Hope this helps!

reply

I ran across this same issue

Submitted by Anonymous on Tue, 09/12/2017 - 06:01.

I ran across this same issue and were pulling hairs... I couldn't understand why my colors came out swizzled. Heck, I even wrote three pixels (red, green, blue) and not even that was correct! Apparently \n is not compiled the way we want under Windows... If analysing the file under a hexeditor like HxD, the compiler insert \n as 0D 0A instead of just 0A on Windows. This adds one byte to the initial fwrite, causing all r,g,b values to be offset by one. Instead of getting [r,g,b] as one pixel, we now get something horrible similar to [b,g,r].

Unix and non-Unix operating systems handles return and carriage return differently. If we read the reference page for fopen we find the following quote in regards to the "w" flag (which treats the file as a text file): "*Text files* are files containing sequences of lines of text. Depending on the environment where the application runs, some special character conversion may occur in input/output operations in *text mode* to adapt them to a system-specific text file format." http://www.cplusplus.com/reference/cstdio/fopen/

In our scenario, that means our \n becomes 0D 0A instead of 0A when using the "w"-flag under windows, which in turn offsets all color values... The solution for this tutorial is to either change the fopen flag to "wb", to ensure no OS specific conversions happen, or change \n to \r. I suggest updating the tutorial to use "wb" by default considering that we are indeed writing a binary image file.

reply

Re: I ran across this same issue

Submitted by PurpleAlien on Tue, 09/12/2017 - 06:28.

Hi,

Thanks for reporting that. I've made the change in the tutorial. Looking back at the code I see that the Github version has this set. I think that it's something I just wrote wrong when writing this up.

Johan.

reply

I forgot to say thank you for

Submitted by Anonymous on Tue, 09/12/2017 - 21:44.

I forgot to say thank you for creating the tutorial. It helped me better understand ray tracing. Keep up the good work!

reply

Re: I forgot to say thank you for

Submitted by PurpleAlien on Tue, 09/12/2017 - 21:53.

Thanks - I'm glad you enjoyed it and that it could be of help!

Johan.

reply

Re: Possible solution

Submitted by PurpleAlien on Sat, 05/27/2017 - 21:55.

Yes, that is most likely the reason for the issue. It probably makes sense to do that on all platforms anyway.

Thanks for commenting!

reply

Re: Ray tracing bug ?

Submitted by PurpleAlien on Sat, 05/20/2017 - 17:54.

Hi,

I never tested the code with Visual Studio, or on Windows at all for that matter. I had some people report that it worked fine, but that's a few years ago. That stack overflow might be something Windows related, probably because the stack is 1MB by default - have a look here: https://msdn.microsoft.com/en-us/library/windows/desktop/ms686774(v=vs.85).aspx Quote: "The default stack reservation size used by the linker is 1 MB".

As for that visual bug, do you get the same result with my code on Github?

Johan.

reply

just a tiny bit more help needed

Submitted by Anonymous on Sun, 04/16/2017 - 17:08.

Hello !

First of all thank you for this awesome tutorial , it is greatly helping me in understanding how computer graphic works, and i am currently trying to use it to make my own raytracer. I cannot thank you enought for the progress i've made so far thanks to that tutorial.

However, as a pretty below average user in both math and programming , there is still a lot of things i fail to understand, i was wondering if by any chance you could provide and exemple code for just one other shape adapted to that tutorial (i have tried looking at the github link you provided in the comments but it is far too advanced for me) just so i can compare to the difference with treating a sphere and ... something else ^^.

it's been a verry long time so i'm not sure if you still follow this, but i guess it's worth a try.

Thanks again !

reply

RE: just a tiny bit more help needed

Submitted by PurpleAlien on Sun, 04/16/2017 - 18:02.

Hi,

Thanks - I'm glad you liked the tutorial!

If you look at the collideRayTriangle() function in the Github repo here: https://github.com/PurpleAlien/Raytracer/blob/master/triangle.c that should give you an idea. The reason for using a collideRayTriangle() is that 3D objects are often made up of polygons (triangles in principle), and thus we can right away use much more complex 3D objects when we can use those. This is the 3D spaceship you get when you run that code.

You can also find other intersections between a ray and primitive. for example a cone and a ray. For that, let me point you to this document: https://www.geometrictools.com/Documentation/IntersectionLineCone.pdf

There are other possibilities of course, but a sphere and a cone are probably the easiest when it comes to primitives. Once you understand the ray/triangle intersection, you can visualize complex 3D objects like the one in my Github code. Naturally, none of this is really optimized for best performance - but there are lots of tutorials, books and so on written on that topic. I'm sure you'll come across those as you proceed.

Hope that helps!

Johan.

reply

it works !

Submitted by Anonymous on Tue, 04/25/2017 - 16:14.

Thank you verry much ! i finally got it to work with triangles , cylindres and cones , thank to the code you provided above !

Thanks a lot for this great tutorial !

reply

Re: it works !

Submitted by PurpleAlien on Tue, 04/25/2017 - 16:42.

Glad to hear that - have fun! :)

Johan.

reply

thanks

Submitted by Anonymous on Mon, 02/09/2015 - 11:50.

Thanks for this nice tutorial. Now my goal is to be able to move the observation point. A friend told me that I have to use another projection type than orthogonal projection.
Gonna look into it.
I'll also try to get acquainted with SDL in order to make the scene animated. I was able to do it just in the terminal but since later we made ppm I was no longer able to animate the scene.

reply

Re: thanks

Submitted by PurpleAlien on Mon, 02/09/2015 - 16:23.

Hi.

Glad you liked the tutorial!. You can find the final, expanded version of the raytracer here: https://github.com/PurpleAlien/Raytracer. This one also adds conic projection, so it can give you an idea on how to proceed with different projection methods.

Johan.

reply

It has been a year...

Submitted by Anonymous on Sat, 02/27/2016 - 23:53.

..now but I got back into this work and got the spheres moving using SDL, in a standalone window, kind of my first 3D animation.
The link to the video of it is here: https://vid.me/nTFS

SDL is really nice to do graphics for the first time. Here I am not using OpenGL with SDL, but it is something you can do (and should do). The spheres are supposed to move a lot faster and I was able to determine that the bottleneck is not SDL, so it must be the computations behind the display of the spheres. This is interesting so I will be looking into how to make it lighter or how to draw other things.

But so far, great success, thanks for your awesome tutorial.

reply

It has been a year...

Submitted by PurpleAlien on Sun, 02/28/2016 - 00:01.

Nice! So glad to see this tutorial is still useful for people.

Your speed issues are most definitely because of the ray tracing code. Ray tracing is very compute intensive, and there is still a lot of research in the field of 'real time ray tracing' (Google is your friend). The code in this tutorial was never optimized since I wanted to keep the conversion from the math to the code as straight forward as possible (to support a few courses I taught). You can definitely improve things and speed things up a lot!

Keep having fun!

Johan.

reply