Grafika komputerowa - projekt

Autor:
Tymon Tobolski (181037)

Prowadzący:
Dr inż. Tomasz Kapłon

Wydział Elektroniki
III rok
Pn TP 08.15 - 11.00

9 stycznia 2012

1 Cel projektu

Celem projektu jest implementacja algorytmu rekursywnego śledzenia promieni (Recursive Ray Tracing).

2 Ray Tracing

Ray Tracing jest techniką generowania fotorealistycznych obrazów scen 3D wykorzystującą analize odbitych promieni świetlnych trafiających do obserwatora.

Dla każdego punktu rzutni wyprowadza się promień pierwotny o początku w położeniu obserwatora. Dla każdego takiego promienia wyszukiwany jest najbliższy punkt przecięcia z obiektami znajdującymi sie na scenie. Następnie wyznaczana jest jasność w tym punkcie dla każdego źródła światła zgodnie z modelem oświetlnia Phonga. Kolejnym krokiem algorytmu jest określenie odbitych promieni wtórnych i wykonanie dla każdego takiego promienia opisanych wyżej operacji. Obliczenia koloru dla danego punktu rzutni kończy się w momencie, gdy promienie nie przecinają już żadnych innych obiektów, lub osiągniety został maksymalny poziom rekurencji.

3 Opis programu

Pierwszym krokiem programu jest wczytanie danych opisujących scene z pliku tekstowego. Przykładowy plik opisu sceny znajduje się poniżej.

```
dimensions 400 400
background 0.3 0.3 0.3
global 0.1 0.1 0.1
sphere 0.7   3.0    0.0  -5.0  0.8 0.2 0.0  0.7 1.0 0.0  0.2 0.1 0.2  40
sphere 0.7  -3.0    0.0  -5.0  0.8 0.2 0.0  0.7 1.0 0.0  0.2 0.1 0.2  40
sphere 2.0   0.0    0.0  -3.0  0.8 0.1 0.0  0.8 0.1 0.0  0.2 0.1 0.2  40
sphere 2.0   0.0   -5.0  -3.0  0.8 0.2 0.0  0.0 0.7 1.0  0.2 0.1 0.2  40
sphere 2.0   0.0    5.0  -3.0  0.8 0.2 0.0  0.0 0.7 1.0  0.2 0.1 0.2  40
sphere 2.0  -5.0    2.5  -3.0  0.8 0.2 0.0  0.0 0.7 1.0  0.2 0.1 0.2  40
sphere 2.0  -5.0   -2.5  -3.0  0.8 0.2 0.0  0.0 0.7 1.0  0.2 0.1 0.2  40
sphere 2.0   5.0   -2.5  -3.0  0.8 0.2 0.0  0.0 0.7 1.0  0.2 0.1 0.2  40
sphere 2.0   5.0    2.5  -3.0  0.8 0.2 0.0  0.0 0.7 1.0  0.2 0.1 0.2  40
source       0.0    0.0  15.0  0.2 0.2 0.2  0.4 0.4 0.4  0.2 0.2 0.2
source      -5.0    0.0  10.0  0.2 0.2 0.2  1.0 0.0 1.0  0.3 0.3 0.1
```

```
source       5.0    0.0  10.0  0.2 0.2 0.2  1.0 0.0 1.0  0.3 0.3 0.1
source       5.0    0.0  12.0  0.2 0.2 0.2  0.0 1.0 1.0  0.4 0.5 0.3
source      -5.0    0.0  12.0  0.2 0.2 0.2  0.0 1.0 1.0  0.4 0.5 0.3
```

Takie dane są wczytywane do programu i zapisywane w odpowiednich strukturach.

```
// data structures
struct Sphere {
    float radius;
    float position[3];
    float specular[3];
    float diffuse[3];
    float ambient[3];
    float specularhinines;
};

struct Source {
    float position[3];
    float specular[3];
    float diffuse[3];
    float ambient[3];
};

// params
int dimensions[2];
float background[3];
float global[3];
vector<Sphere> spheres;
vector<Source> sources;

// read scene from file
void readFile(){
    fstream in("/Users/teamon/Downloads/scene.txt", ios :: in);
    string s;

    while(!in.eof()){
        in >> s;

        if(s == "dimensions"){
            in >> dimensions[0];
            in >> dimensions[1];
        } else if(s == "background"){
            TRIPLE(in >> background[i]);
        } else if(s == "global"){
            TRIPLE(in >> global[i]);
        } else if(s == "sphere"){
            Sphere sph;
            in >> sph.radius;
            TRIPLE(in >> sph.position[i]);
            TRIPLE(in >> sph.specular[i]);
            TRIPLE(in >> sph.diffuse[i]);
            TRIPLE(in >> sph.ambient[i]);
            in >> sph.specularhinines;
            spheres.push_back(sph);
        } else if(s == "source"){
```

```
            Source src;
            TRIPLE(in >> src.position[i]);
            TRIPLE(in >> src.specular[i]);
            TRIPLE(in >> src.diffuse[i]);
            TRIPLE(in >> src.ambient[i]);
            sources.push_back(src);
        }

    }

    in.close();
}
```

W celu eliminacji często powtarzanego kodu - w tym wypadku trójpowtórzeniowej pętli `for` - wprowadzone zostało makro `TRIPLE`:

```
#define TRIPLE(f) for(int i=0; i<3; i++){ f; }
```

Poniżej znajduje się funkcja `draw()`, która dla każdego punktu rzutni generuje promień pierwotny i wywołuje algorytm śledzenia promieni.

```
// glowna funkcja rysujaca
void draw(){
    int  x, y;
    float xf, yf;
    float width_2 = dimensions[0]/2;
    float height_2 = dimensions[1]/2;

    glClear(GL_COLOR_BUFFER_BIT);
    glFlush();


    for (y = height_2; y > -height_2; y--){
        for (x = -width_2; x < width_2; x++){
            xf = (float)x/(dimensions[0]/starting_z);
            yf = (float)y/(dimensions[1]/starting_z);

            starting_point[0] =  xf;
            starting_point[1] =  yf;
            starting_point[2] =  starting_z;

            TRIPLE(color[i] = 0.0);

            trace(starting_point, starting_directions, 1);

            TRIPLE(
                if(color[i] == 0.0) color[i] = background[i];
                pixel[0][0][i] = min(255, color[i] * 255);
            );

            glRasterPos3f(xf, yf, 0);
```

```
            glDrawPixels(1, 1, GL_RGB, GL_UNSIGNED_BYTE, pixel);
        }
    }
    glFlush();
}
```

Funkcja `trace()` składa się z kilku etapów: sprawdzeniu czy promień przecina obiekt sceny, obliczeniu parametrów na podstaie modelu oświetlenia Phonga oraz rekurencyjne wywołanie funkcji `trace()` dla promienia odbitego.

```
void trace(float * p, float * v, int step){
    Sphere * sp = intersect(p, v);
    if(sp != NULL){
        normal(sp);
        reflect(v);
        phong(v, sp);
        TRIPLE(color[i] += inters_c[i]);

        if(step < TRACE_MAX) trace(inter, ref, step+1);
    }
}
```

Funkcja sprawdzająca przcięcie promienia z obiektem sceny (funkcja iteruje po wszystkich sferach):

```
Sphere * intersect(float * v1, float * v2){
    Sphere * s = NULL;
    float pre = FLT_MAX;
    float a, b, c, del, r;

    for(vector<Sphere>::iterator it=spheres.begin(); it < spheres.end(); it
        ++){
        Sphere sp = *it;
        float * spp = sp.position;

        a = v2[0] * v2[0] + v2[1] * v2[1] + v2[2] * v2[2];

        b = 2 * (v2[0] * (v1[0] - spp[0]) +
                 v2[1] * (v1[1] - spp[1]) +
                 v2[2] * (v1[2] - spp[2]));

        c = v1[0] * v1[0] + v1[1] * v1[1] + v1[2] * v1[2] -
        2*(spp[0] * v1[0] +
           spp[1] * v1[1] +
           spp[2] * v1[2]) +
        spp[0]*spp[0] + spp[1] * spp[1] + spp[2] * spp[2] -
        sp.radius * sp.radius;

        del = b*b - 4*a*c;
```

```
            if(del >= 0){
                r = (-b - sqrt(del))/(2*a);
                if(r > 0 && r < pre){
                    inter[0] = v1[0] + r*v2[0];
                    inter[1] = v1[1] + r*v2[1];
                    inter[2] = v1[2] + r*v2[2];
                    pre = sqrt((inter[0] - v1[0]) * (inter[0] * v1[0]) +
                              (inter[1] - v1[1]) * (inter[1] * v1[1]) +
                              (inter[2] - v1[2]) * (inter[2] * v1[2]));
                    s = &(*it);
                }
            }

        }

        return s;
    }
```

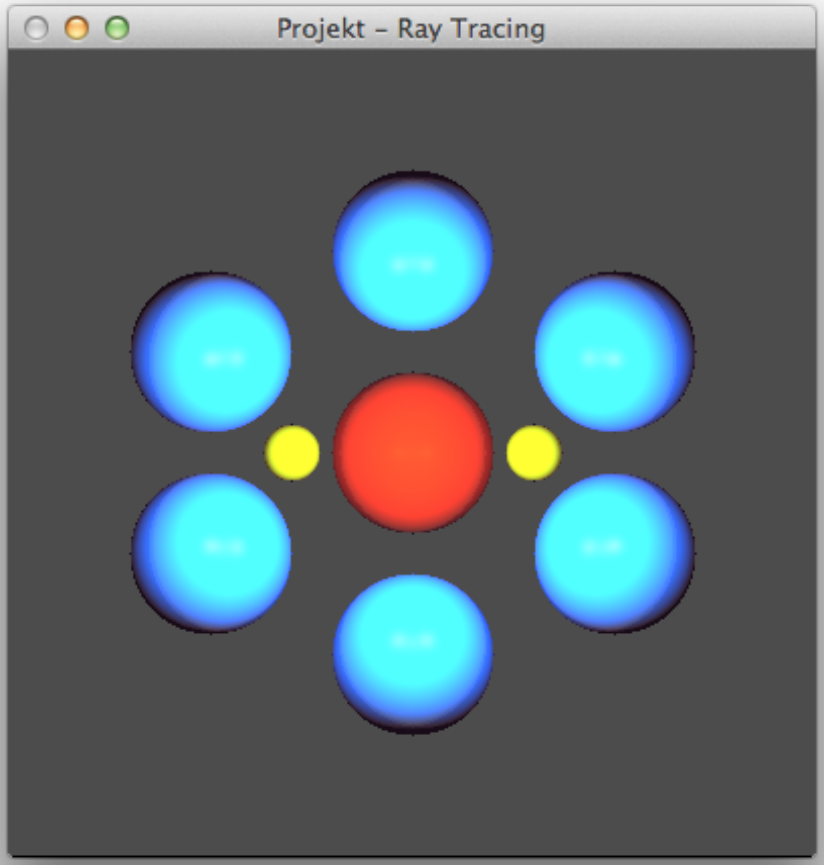
Obliczenie koloru z modeu oświetlenia Phonga:

```
    void phong(float * v, Sphere * sp){
        float light_vec[3];
        float reflection_vector[3];
        float viewer_v[3];
        float nl, vr;

        TRIPLE(
            viewer_v[i] = -v[i];
            inters_c[i] = 0;
        )

        for(vector<Source>::iterator it=sources.begin(); it!= sources.end(); it
            ++){
            Source src = *it;
            float * srcp = src.position;
            TRIPLE(light_vec[i] = srcp[i] - inter[i])

            normalization(light_vec);
            nl = prod(light_vec, vec_n);
            TRIPLE(reflection_vector[i] = 2*nl*vec_n[i] - light_vec[i])

            normalization(reflection_vector);
            vr = prod(reflection_vector, viewer_v);

            if(vr < 0) vr = 0;

            if(nl > 0){
                float x = sqrt((srcp[0] - inter[0])*(srcp[0] - inter[0]) +
                               (srcp[1] - inter[1])*(srcp[1] - inter[1]) +
                               (srcp[2] - inter[2])*(srcp[2] - inter[2]));
                TRIPLE(
                    inters_c[i] += (1.0/(PHONG_A + PHONG_B*x + PHONG_C*x*x)) *
                    (sp->diffuse[i]*src.diffuse[i]*nl + sp->specular[i]*src.
                        specular[i]*pow(vr, sp->specularhinines)) +
                    sp->ambient[i]*src.ambient[i];
                )
```

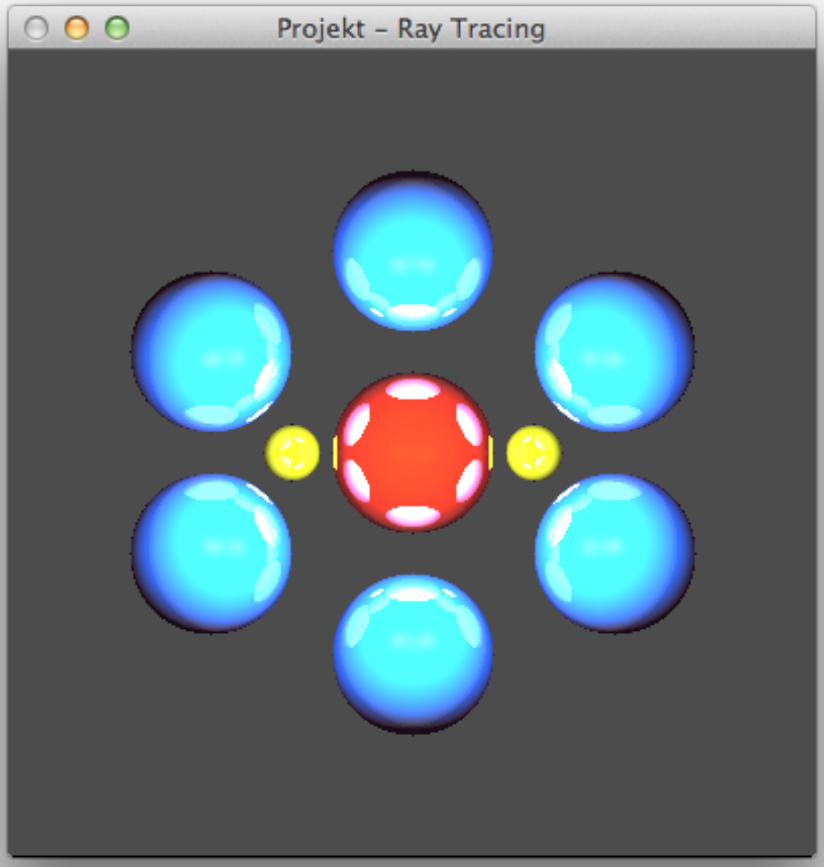
```
            } else {
                TRIPLE(inters_c[i] += sp->ambient[i]*global[i])
            }
        }
39
    }
```

4 Uzyskane rezultaty

Poniżej znajdują sie wyniki działania programu dla różnego maksymalnego poziomy rekurencji.



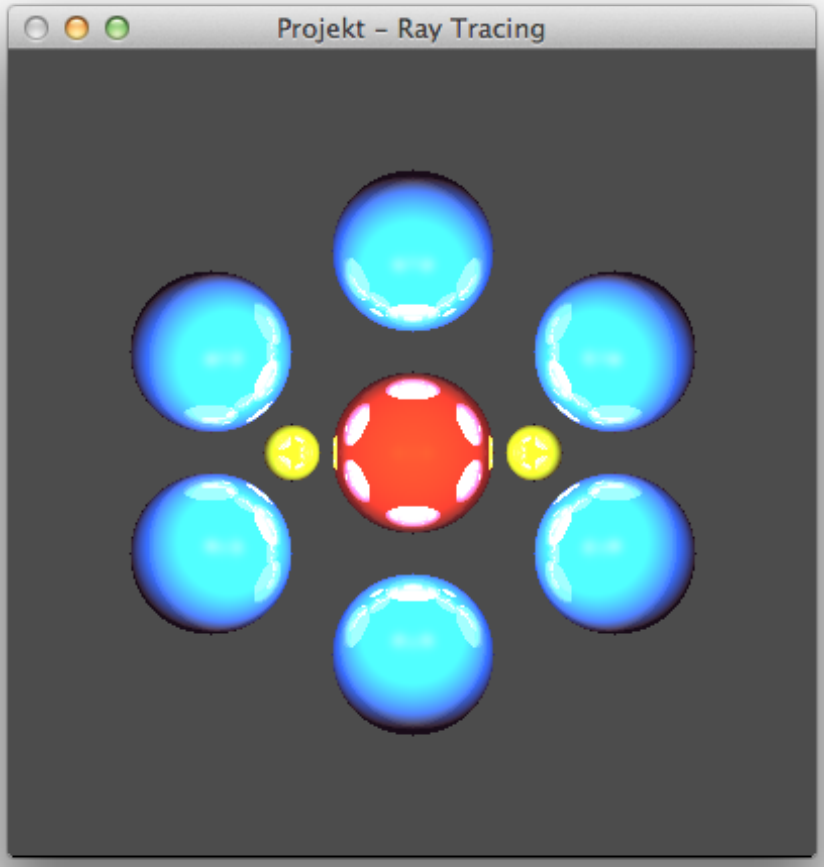
Rysunek 1: TRACE_MAX = 1



Rysunek 2: TRACE_MAX = 2

Jak widać na rysunkach jeden poziom algorytmu Ray Tracing daje stosunkowo słabe rezultaty. Dodanie jednego poziomu rekurencji znacznie poprawia realizm oświetlenia sceny. Kolejny poziom nieznacznie poprawia wygląd odbić na powierzchni sfer.

Dla większych wartości maksymalnego poziomu rekurencji uzyskany obraz był identyczny co dla $TRACE\_MAX = 3$.



Rysunek 3: TRACE_MAX = 3

5 Wnioski

Algorytm rekursywnego śledzenia promieni pozwala na uzyskanie realistycznie oświetlonej sceny. Mimo uproszczonego modelu interakcji światła z otoczeniem metoda ta daje stosunkowo dobre rezultaty. Problemem może być wydajność algorytmu, który wymaga sporej ilości obliczeń. Ilość operacji jest zależna od rozdzielczości obrazu, liczby obiektów, a także ilości źródeł światła. Niebanalnym zadaniem jest też określenie czy promień przecina dany obiekt. W przypadku sfer sprowadza sie to do rozwiązania stosunkowo prostego równiania jednak przy barziej złożonych obiektach takie sprawdzenie może się okazać trudnych i czasochłonnym (zarówno dla programisty jak i procesora) zadaniem.