Simulating the Colors of the Sky

This project contains the following files (right-click files you'd like to download):

skycolor.cpp

```
Simulating the color of the sky (Nishita model).
See "Display of The Earth Taking into Account Atmospheric Scattering" for more information.

Instructions to compile this program:
Download the acceleration.cpp and teapotdata.h file to a folder. Open a shell/terminal,
and run the following command where the files is saved:
clang++ -std=c++11 -o skycolor skycolor.cpp -O3
You can use c++ if you don't use clang++
Run with: ./skycolor. Open the resulting image (ppm) in Photoshop or any program reading
PPM files.
```

```cpp
#if defined(WIN32) || defined(_WIN32)
#include "stdafx.h"
#endif

#include <cassert>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <cmath>
#include <chrono>
#include <random>
#include <limits>

#ifndef M_PI
#define M_PI (3.14159265358979323846f)
#endif

const float kInfinity = std::numeric_limits<float>::max();

template<typename T>
class Vec3
{
public:
    Vec3() : x(0), y(0), z(0) {}
    Vec3(T xx) : x(xx), y(xx), z(xx) {}
    Vec3(T xx, T yy, T zz) : x(xx), y(yy), z(zz) {}
    Vec3 operator * (const T& r) const { return Vec3(x * r, y * r, z * r); }
    Vec3 operator * (const Vec3<T> &v) const { return Vec3(x * v.x, y * v.y, z * v.z);
    Vec3 operator + (const Vec3& v) const { return Vec3(x + v.x, y + v.y, z + v.z); }
```

```
    Vec3 operator + (const Vec3& v) const { return Vec3(x + v.x, y + v.y, z + v.z); }
    Vec3 operator - (const Vec3& v) const { return Vec3(x - v.x, y - v.y, z - v.z); }
    template<typename U>
    Vec3 operator / (const Vec3<U>& v) const { return Vec3(x / v.x, y / v.y, z / v.z);
    friend Vec3 operator / (const T r, const Vec3& v)
    {
        return Vec3(r / v.x, r / v.y, r / v.z);
    }
    const T& operator [] (size_t i) const { return (&x)[i]; }
    T& operator [] (size_t i) { return (&x)[i]; }
    T length2() const { return x * x + y * y + z * z; }
    T length() const { return std::sqrt(length2()); }
    Vec3& operator += (const Vec3<T>& v) { x += v.x, y += v.y, z += v.z; return *this;
    Vec3& operator *= (const float& r) { x *= r, y *= r, z *= r; return *this; }
    friend Vec3 operator * (const float&r, const Vec3& v)
    {
        return Vec3(v.x * r, v.y * r, v.z * r);
    }
    friend std::ostream& operator << (std::ostream& os, const Vec3<T>& v)
    {
        os << v.x << " " << v.y << " " << v.z << std::endl; return os;
    }
    T x, y, z;
};

template<typename T>
void normalize(Vec3<T>& vec)
{
    T len2 = vec.length2();
    if (len2 > 0) {
        T invLen = 1 / std::sqrt(len2);
        vec.x *= invLen, vec.y *= invLen, vec.z *= invLen;
    }
}

template<typename T>
T dot(const Vec3<T>& va, const Vec3<T>& vb)
{
    return va.x * vb.x + va.y * vb.y + va.z * vb.z;
}

using Vec3f = Vec3<float>;
```

The atmosphere class. Stores data about the planetory body (its radius), the atmosphere itself (thickness) and things such as the Mie and Rayleigh coefficients, the sun direction, etc.

```
class Atmosphere
{
public:
    Atmosphere(
        Vec3f sd = Vec3f(0, 1, 0),
        float er = 6360e3, float ar = 6420e3,
        float hr = 7994, float hm = 1200) :
        sunDirection(sd),
```

```
        sunDirection(sd),
        earthRadius(er),
        atmosphereRadius(ar),
        Hr(hr),
        Hm(hm)
    {}

    Vec3f computeIncidentLight(const Vec3f& orig, const Vec3f& dir, float tmin, float t

    Vec3f sunDirection;     // The sun direction (normalized)
    float earthRadius;      // In the paper this is usually Rg or Re (radius ground, ea
    float atmosphereRadius; // In the paper this is usually R or Ra (radius atmosphere)
    float Hr;               // Thickness of the atmosphere if density was uniform (Hr)
    float Hm;               // Same as above but for Mie scattering (Hm)

    static const Vec3f betaR;
    static const Vec3f betaM;
};

const Vec3f Atmosphere::betaR(3.8e-6f, 13.5e-6f, 33.1e-6f);
const Vec3f Atmosphere::betaM(21e-6f);

bool solveQuadratic(float a, float b, float c, float& x1, float& x2)
{
    if (b == 0) {
        // Handle special case where the the two vector ray.dir and V are perpendicular
        // with V = ray.orig - sphere.centre
        if (a == 0) return false;
        x1 = 0; x2 = std::sqrtf(-c / a);
        return true;
    }
    float discr = b * b - 4 * a * c;

    if (discr < 0) return false;

    float q = (b < 0.f) ? -0.5f * (b - std::sqrtf(discr)) : -0.5f * (b + std::sqrtf(dis
    x1 = q / a;
    x2 = c / q;

    return true;
}

```

A simple routine to compute the intersection of a ray with a sphere

```
bool raySphereIntersect(const Vec3f& orig, const Vec3f& dir, const float& radius, float
{
    // They ray dir is normalized so A = 1
    float A = dir.x * dir.x + dir.y * dir.y + dir.z * dir.z;
    float B = 2 * (dir.x * orig.x + dir.y * orig.y + dir.z * orig.z);
    float C = orig.x * orig.x + orig.y * orig.y + orig.z * orig.z - radius * radius;
```

```
    float C = orig.x * orig.x + orig.y * orig.y + orig.z * orig.z - radius * radius;

    if (!solveQuadratic(A, B, C, t0, t1)) return false;

    if (t0 > t1) std::swap(t0, t1);

    return true;
}

```

This is where all the magic happens. We first raymarch along the primary ray (from the camera origin to the point where the ray exits the atmosphere or intersect with the planetory body) For each sample along the primary ray, we then "cast" a light ray and raymarch along that ray as well. We basically shoot a ray in the direction of the sun.

```
Vec3f Atmosphere::computeIncidentLight(const Vec3f& orig, const Vec3f& dir, float tmin,
{
    float t0, t1;
    if (!raySphereIntersect(orig, dir, atmosphereRadius, t0, t1) || t1 < 0) return 0;
    if (t0 > tmin && t0 > 0) tmin = t0;
    if (t1 < tmax) tmax = t1;
    uint32_t numSamples = 16;
    uint32_t numSamplesLight = 8;
```

```
    uint32_t numSamplesLight = 8;
    float segmentLength = (tmax - tmin) / numSamples;
    float tCurrent = tmin;
    Vec3f sumR(0), sumM(0); // mie and rayleigh contribution
    float opticalDepthR = 0, opticalDepthM = 0;
    float mu = dot(dir, sunDirection); // mu in the paper which is the cosine of the ar
    float phaseR = 3.f / (16.f * M_PI) * (1 + mu * mu);
    float g = 0.76f;
    float phaseM = 3.f / (8.f * M_PI) * ((1.f - g * g) * (1.f + mu * mu)) / ((2.f + g *
    for (uint32_t i = 0; i < numSamples; ++i) {
        Vec3f samplePosition = orig + (tCurrent + segmentLength * 0.5f) * dir;
        float height = samplePosition.length() - earthRadius;
        // compute optical depth for light
        float hr = exp(-height / Hr) * segmentLength;
        float hm = exp(-height / Hm) * segmentLength;
        opticalDepthR += hr;
        opticalDepthM += hm;
        // light optical depth
        float t0Light, t1Light;
        raySphereIntersect(samplePosition, sunDirection, atmosphereRadius, t0Light, t1L
        float segmentLengthLight = t1Light / numSamplesLight, tCurrentLight = 0;
        float opticalDepthLightR = 0, opticalDepthLightM = 0;
        uint32_t j;
        for (j = 0; j < numSamplesLight; ++j) {
            Vec3f samplePositionLight = samplePosition + (tCurrentLight + segmentLength
            float heightLight = samplePositionLight.length() - earthRadius;
            if (heightLight < 0) break;
            opticalDepthLightR += exp(-heightLight / Hr) * segmentLengthLight;
            opticalDepthLightM += exp(-heightLight / Hm) * segmentLengthLight;
            tCurrentLight += segmentLengthLight;
        }
        if (j == numSamplesLight) {
            Vec3f tau = betaR * (opticalDepthR + opticalDepthLightR) + betaM * 1.1f * (
            Vec3f attenuation(exp(-tau.x), exp(-tau.y), exp(-tau.z));
            sumR += attenuation * hr;
            sumM += attenuation * hm;
        }
        tCurrent += segmentLength;
    }

```

We use a magic number here for the intensity of the sun (20). We will make it more scientific in a future revision of this lesson/code

```
    return (sumR * betaR * phaseR + sumM * betaM * phaseM) * 20;
}

void renderSkydome(const Vec3f& sunDir, const char *filename)
{
```

```
{
    Atmosphere atmosphere(sunDir);
    auto t0 = std::chrono::high_resolution_clock::now();
#if 1
```

Render fisheye

```
    const unsigned width = 512, height = 512;
    Vec3f *image = new Vec3f[width * height], *p = image;
    memset(image, 0x0, sizeof(Vec3f) * width * height);
    for (unsigned j = 0; j < height; ++j) {
        float y = 2.f * (j + 0.5f) / float(height - 1) - 1.f;
        for (unsigned i = 0; i < width; ++i, ++p) {
            float x = 2.f * (i + 0.5f) / float(width - 1) - 1.f;
            float z2 = x * x + y * y;
            if (z2 <= 1) {
                float phi = std::atan2(y, x);
                float theta = std::acos(1 - z2);
                Vec3f dir(sin(theta) * cos(phi), cos(theta), sin(theta) * sin(phi));
                // 1 meter above sea level
                *p = atmosphere.computeIncidentLight(Vec3f(0, atmosphere.earthRadius +
            }
        }
        fprintf(stderr, "\b\b\b\b\%3d%c", (int)(100 * j / (width - 1)), '%');
    }
#else
```

Render from a normal camera

```
    const unsigned width = 640, height = 480;
    Vec3f *image = new Vec3f[width * height], *p = image;
    memset(image, 0x0, sizeof(Vec3f) * width * height);
    float aspectRatio = width / float(height);
    float fov = 65;
    float angle = std::tan(fov * M_PI / 180 * 0.5f);
    unsigned numPixelSamples = 4;
    Vec3f orig(0, atmosphere.earthRadius + 1000, 30000); // camera position
```

```
    Vec3f orig(0, atmosphere.earthRadius + 1000, 30000); // camera position
    std::default_random_engine generator;
    std::uniform_real_distribution<float> distribution(0, 1); // to generate random flo
    for (unsigned y = 0; y < height; ++y) {
        for (unsigned x = 0; x < width; ++x, ++p) {
            for (unsigned m = 0; m < numPixelSamples; ++m) {
                for (unsigned n = 0; n < numPixelSamples; ++n) {
                    float rayx = (2 * (x + (m + distribution(generator)) / numPixelSamp
                    float rayy = (1 - (y + (n + distribution(generator)) / numPixelSamp
                    Vec3f dir(rayx, rayy, -1);
                    normalize(dir);
```

Does the ray intersect the planetory body? (the intersection test is against the Earth here not against the atmosphere). If the ray intersects the Earth body and that the intersection is ahead of us, then the ray intersects the planet in 2 points, t0 and t1. But we only want to comupute the atmosphere between t=0 and t=t0 (where the ray hits the Earth first). If the viewing ray doesn't hit the Earth, or course the ray is then bounded to the range [0:INF]. In the method computeIncidentLight() we then compute where this primary ray intersects the atmosphere and we limit the max t range of the ray to the point where it leaves the atmosphere.

```
                    float t0, t1, tMax = kInfinity;
                    if (raySphereIntersect(orig, dir, atmosphere.earthRadius, t0, t1) &
                        tMax = std::max(0.f, t0);
```

The *viewing or camera ray* is bounded to the range [0:tMax]

```
                    *p += atmosphere.computeIncidentLight(orig, dir, 0, tMax);
                }
            }
            *p *= 1.f / (numPixelSamples * numPixelSamples);
        }
        fprintf(stderr, "\b\b\b\b%3d%c", (int)(100 * y / (width - 1)), '%');
    }
#endif
```

```
#endif
    std::cout << "\b\b\b\b" << ((std::chrono::duration<float>)(std::chrono::high_resolu
    // Save result to a PPM image (keep these flags if you compile under Windows)
    std::ofstream ofs(filename, std::ios::out | std::ios::binary);
    ofs << "P6\n" << width << " " << height << "\n255\n";
    p = image;
    for (unsigned j = 0; j < height; ++j) {
        for (unsigned i = 0; i < width; ++i, ++p) {
#if 1
            // Apply tone mapping function
            (*p)[0] = (*p)[0] < 1.413f ? pow((*p)[0] * 0.38317f, 1.0f / 2.2f) : 1.0f -
            (*p)[1] = (*p)[1] < 1.413f ? pow((*p)[1] * 0.38317f, 1.0f / 2.2f) : 1.0f -
            (*p)[2] = (*p)[2] < 1.413f ? pow((*p)[2] * 0.38317f, 1.0f / 2.2f) : 1.0f -
#endif
            ofs << (unsigned char)(std::min(1.f, (*p)[0]) * 255)
                << (unsigned char)(std::min(1.f, (*p)[1]) * 255)
                << (unsigned char)(std::min(1.f, (*p)[2]) * 255);
        }
    }
    ofs.close();
    delete[] image;
}

int main()
{
#if 1
```

Render a sequence of images (sunrise to sunset)

```
    unsigned nangles = 128;
    for (unsigned i = 0; i < nangles; ++i) {
        char filename[1024];
        sprintf(filename, "./skydome.%04d.ppm", i);
        float angle = i / float(nangles - 1) * M_PI * 0.6;
        fprintf(stderr, "Rendering image %d, angle = %0.2f\n", i, angle * 180 / M_PI);
        renderSkydome(Vec3f(0, cos(angle), -sin(angle)), filename);
    }
#else
```

Render one single image

```
    float angle = M_PI * 0;
    Vec3f sunDir(0, std::cos(angle), -std::sin(angle));
    std::cerr << "Sun direction: " << sunDir << std::endl;
    renderSkydome(sunDir, "./skydome.ppm");
#endif

    return 0;
}
```

```
}
```