

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.

```

0033  #if defined(WIN32) || defined(_WIN32)
0034  #include "stdafx.h"
0035  #endif
0036
0037  #include <cassert>
0038  #include <iostream>
0039  #include <fstream>
0040  #include <algorithm>
0041  #include <cmath>
0042  #include <chrono>
0043  #include <random>
0044  #include <limits>
0045
0046  #ifndef M_PI
0047  #define M_PI (3.14159265358979323846f)
0048  #endif
0049
0050  const float kInfinity = std::numeric_limits<float>::max();
0051
0052  template<typename T>
0053  class Vec3
0054  {
0055  public:
0056      Vec3() : x(0), y(0), z(0) {}
0057      Vec3(T xx) : x(xx), y(xx), z(xx) {}
0058      Vec3(T xx, T yy, T zz) : x(xx), y(yy), z(zz) {}
0059      Vec3 operator * (const T& r) const { return Vec3(x * r, y * r, z * r); }
0060      Vec3 operator * (const Vec3<T> &v) const { return Vec3(x * v.x, y * v.y, z * v.z); }
0061      Vec3 operator + (const Vec3<T> &u) const { return Vec3(x + u.x, y + u.y, z + u.z); }
0062      Vec3 operator - (const Vec3<T> &u) const { return Vec3(x - u.x, y - u.y, z - u.z); }
0063      Vec3 operator / (const T& r) const { return Vec3(x / r, y / r, z / r); }
0064      Vec3 operator / (const Vec3<T> &v) const { return Vec3(x / v.x, y / v.y, z / v.z); }
0065      Vec3 operator += (T r) { x += r; y += r; z += r; return *this; }
0066      Vec3 operator += (Vec3<T> &v) { x += v.x; y += v.y; z += v.z; return *this; }
0067      Vec3 operator -= (T r) { x -= r; y -= r; z -= r; return *this; }
0068      Vec3 operator -= (Vec3<T> &v) { x -= v.x; y -= v.y; z -= v.z; return *this; }
0069      Vec3 operator *= (T r) { x *= r; y *= r; z *= r; return *this; }
0070      Vec3 operator *= (Vec3<T> &v) { x *= v.x; y *= v.y; z *= v.z; return *this; }
0071      Vec3 operator += (T r) { x += r; y += r; z += r; return *this; }
0072      Vec3 operator += (Vec3<T> &v) { x += v.x; y += v.y; z += v.z; return *this; }
0073      Vec3 operator -= (T r) { x -= r; y -= r; z -= r; return *this; }
0074      Vec3 operator -= (Vec3<T> &v) { x -= v.x; y -= v.y; z -= v.z; return *this; }
0075      Vec3 operator /= (T r) { x /= r; y /= r; z /= r; return *this; }
0076      Vec3 operator /= (Vec3<T> &v) { x /= v.x; y /= v.y; z /= v.z; return *this; }
0077      Vec3 operator < (const Vec3<T> &u) const { return x < u.x && y < u.y && z < u.z; }
0078      Vec3 operator > (const Vec3<T> &u) const { return x > u.x && y > u.y && z > u.z; }
0079      Vec3 operator <= (const Vec3<T> &u) const { return x <= u.x && y <= u.y && z <= u.z; }
0080      Vec3 operator >= (const Vec3<T> &u) const { return x >= u.x && y >= u.y && z >= u.z; }
0081      Vec3 operator == (const Vec3<T> &u) const { return x == u.x && y == u.y && z == u.z; }
0082      Vec3 operator != (const Vec3<T> &u) const { return x != u.x || y != u.y || z != u.z; }
0083      Vec3 operator <= (T r) const { return x <= r && y <= r && z <= r; }
0084      Vec3 operator >= (T r) const { return x >= r && y >= r && z >= r; }
0085      Vec3 operator <= (Vec3<T> &u) const { return x <= u.x && y <= u.y && z <= u.z; }
0086      Vec3 operator >= (Vec3<T> &u) const { return x >= u.x && y >= u.y && z >= u.z; }
0087      Vec3 operator == (T r) const { return x == r && y == r && z == r; }
0088      Vec3 operator != (T r) const { return x != r || y != r || z != r; }
0089      Vec3 operator == (Vec3<T> &u) const { return x == u.x && y == u.y && z == u.z; }
0090      Vec3 operator != (Vec3<T> &u) const { return x != u.x || y != u.y || z != u.z; }
0091      Vec3 operator <= (T r) { x <= r; y <= r; z <= r; return *this; }
0092      Vec3 operator >= (T r) { x >= r; y >= r; z >= r; return *this; }
0093      Vec3 operator <= (Vec3<T> &u) { x <= u.x; y <= u.y; z <= u.z; return *this; }
0094      Vec3 operator >= (Vec3<T> &u) { x >= u.x; y >= u.y; z >= u.z; return *this; }
0095      Vec3 operator == (T r) { x == r; y == r; z == r; return *this; }
0096      Vec3 operator != (T r) { x != r; y != r; z != r; return *this; }
0097      Vec3 operator == (Vec3<T> &u) { x == u.x; y == u.y; z == u.z; return *this; }
0098      Vec3 operator != (Vec3<T> &u) { x != u.x; y != u.y; z != u.z; return *this; }
0099      Vec3 operator <= (T r) { x <= r; y <= r; z <= r; return *this; }
0100      Vec3 operator >= (T r) { x >= r; y >= r; z >= r; return *this; }
0101      Vec3 operator <= (Vec3<T> &u) { x <= u.x; y <= u.y; z <= u.z; return *this; }
0102      Vec3 operator >= (Vec3<T> &u) { x >= u.x; y >= u.y; z >= u.z; return *this; }
0103      Vec3 operator == (T r) { x == r; y == r; z == r; return *this; }
0104      Vec3 operator != (T r) { x != r; y != r; z != r; return *this; }
0105      Vec3 operator == (Vec3<T> &u) { x == u.x; y == u.y; z == u.z; return *this; }
0106      Vec3 operator != (Vec3<T> &u) { x != u.x; y != u.y; z != u.z; return *this; }
0107      Vec3 operator <= (T r) { x <= r; y <= r; z <= r; return *this; }
0108      Vec3 operator >= (T r) { x >= r; y >= r; z >= r; return *this; }
0109      Vec3 operator <= (Vec3<T> &u) { x <= u.x; y <= u.y; z <= u.z; return *this; }
0110      Vec3 operator >= (Vec3<T> &u) { x >= u.x; y >= u.y; z >= u.z; return *this; }
0111      Vec3 operator == (T r) { x == r; y == r; z == r; return *this; }
0112      Vec3 operator != (T r) { x != r; y != r; z != r; return *this; }
0113      Vec3 operator == (Vec3<T> &u) { x == u.x; y == u.y; z == u.z; return *this; }
0114      Vec3 operator != (Vec3<T> &u) { x != u.x; y != u.y; z != u.z; return *this; }
0115      Vec3 operator <= (T r) { x <= r; y <= r; z <= r; return *this; }
0116      Vec3 operator >= (T r) { x >= r; y >= r; z >= r; return *this; }
0117      Vec3 operator <= (Vec3<T> &u) { x <= u.x; y <= u.y; z <= u.z; return *this; }
0118      Vec3 operator >= (Vec3<T> &u) { x >= u.x; y >= u.y; z >= u.z; return *this; }
0119      Vec3 operator == (T r) { x == r; y == r; z == r; return *this; }
0120      Vec3 operator != (T r) { x != r; y != r; z != r; return *this; }
0121      Vec3 operator == (Vec3<T> &u) { x == u.x; y == u.y; z == u.z; return *this; }
0122      Vec3 operator != (Vec3<T> &u) { x != u.x; y != u.y; z != u.z; return *this; }
0123      Vec3 operator <= (T r) { x <= r; y <= r; z <= r; return *this; }
0124      Vec3 operator >= (T r) { x >= r; y >= r; z >= r; return *this; }
0125      Vec3 operator <= (Vec3<T> &u)
```

```

000     vec3 operator + (const vec3& v) const { return vec3(x + v.x, y + v.y, z + v.z); }
061     Vec3 operator - (const Vec3& v) const { return Vec3(x - v.x, y - v.y, z - v.z); }
062     template<typename U>
063     Vec3 operator / (const Vec3<U>& v) const { return Vec3(x / v.x, y / v.y, z / v.z); }
064     friend Vec3 operator / (const T r, const Vec3& v)
065     {
066         return Vec3(r / v.x, r / v.y, r / v.z);
067     }
068     const T& operator [] (size_t i) const { return (&x)[i]; }
069     T& operator [] (size_t i) { return (&x)[i]; }
070     T length2() const { return x * x + y * y + z * z; }
071     T length() const { return std::sqrt(length2()); }
072     Vec3& operator += (const Vec3<T>& v) { x += v.x, y += v.y, z += v.z; return *this; }
073     Vec3& operator *= (const float& r) { x *= r, y *= r, z *= r; return *this; }
074     friend Vec3 operator * (const float&r, const Vec3& v)
075     {
076         return Vec3(v.x * r, v.y * r, v.z * r);
077     }
078     friend std::ostream& operator << (std::ostream& os, const Vec3<T>& v)
079     {
080         os << v.x << " " << v.y << " " << v.z << std::endl; return os;
081     }
082     T x, y, z;
083 };
084
085 template<typename T>
086 void normalize(Vec3<T>& vec)
087 {
088     T len2 = vec.length2();
089     if (len2 > 0) {
090         T invLen = 1 / std::sqrt(len2);
091         vec.x *= invLen, vec.y *= invLen, vec.z *= invLen;
092     }
093 }
094
095 template<typename T>
096 T dot(const Vec3<T>& va, const Vec3<T>& vb)
097 {
098     return va.x * vb.x + va.y * vb.y + va.z * vb.z;
099 }
100
101 using Vec3f = Vec3<float>;
102
103

```

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

```
109 class Atmosphere
110 {
111 public:
112     Atmosphere(
113         Vec3f sd = Vec3f(0, 1, 0),
114         float er = 6360e3, float ar = 6420e3,
115         float hr = 7994, float hm = 1200) :
116         sunDirection(sd) {
```

```

115     sunDirVec3f(su,
116     earthRadius(er),
117     atmosphereRadius(ar),
118     Hr(hr),
119     Hm(hm)
120     {}
121
122
123 Vec3f computeIncidentLight(const Vec3f& orig, const Vec3f& dir, float tmin, float tmax)
124
125 Vec3f sunDirection;    // The sun direction (normalized)
126 float earthRadius;     // In the paper this is usually Rg or Re (radius ground, ea
127 float atmosphereRadius; // In the paper this is usually R or Ra (radius atmosphere)
128 float Hr;              // Thickness of the atmosphere if density was uniform (Hr)
129 float Hm;              // Same as above but for Mie scattering (Hm)
130
131 static const Vec3f betaR;
132 static const Vec3f betaM;
133 };
134
135 const Vec3f Atmosphere::betaR(3.8e-6f, 13.5e-6f, 33.1e-6f);
136 const Vec3f Atmosphere::betaM(21e-6f);
137
138 bool solveQuadratic(float a, float b, float c, float& x1, float& x2)
139 {
140     if (b == 0) {
141         // Handle special case where the the two vector ray.dir and V are perpendicular
142         // with V = ray.orig - sphere.centre
143         if (a == 0) return false;
144         x1 = 0; x2 = std::sqrtf(-c / a);
145         return true;
146     }
147     float discr = b * b - 4 * a * c;
148
149     if (discr < 0) return false;
150
151     float q = (b < 0.f) ? -0.5f * (b - std::sqrtf(discr)) : -0.5f * (b + std::sqrtf(discr));
152     x1 = q / a;
153     x2 = c / q;
154
155     return true;
156 }
157

```

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

```

161 bool raySphereIntersect(const Vec3f& orig, const Vec3f& dir, const float& radius, float& t)
162 {
163     // They ray dir is normalized so A = 1
164     float A = dir.x * dir.x + dir.y * dir.y + dir.z * dir.z;
165     float B = 2 * (dir.x * orig.x + dir.y * orig.y + dir.z * orig.z);
166     float C = orig.x * orig.x + orig.y * orig.y + orig.z * orig.z - radius * radius;

```

```

166     Vec3f C = orig.x * orig.x + orig.y * orig.y + orig.z * orig.z;
167
168     if (!solveQuadratic(A, B, C, t0, t1)) return false;
169
170     if (t0 > t1) std::swap(t0, t1);
171
172     return true;
173 }
174

```

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 planetary 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.

```

181 Vec3f Atmosphere::computeIncidentLight(const Vec3f& orig, const Vec3f& dir, float tmin,
182 {
183     float t0, t1;
184     if (!raySphereIntersect(orig, dir, atmosphereRadius, t0, t1) || t1 < 0) return 0;
185     if (t0 > tmin && t0 > 0) tmin = t0;
186     if (t1 < tmax) tmax = t1;
187     uint32_t numSamples = 16;
188     uint32_t numSamplesLight = 8;

```

```

188     uint32_t numSamplesLight = 0;
189     float segmentLength = (tmax - tmin) / numSamples;
190     float tCurrent = tmin;
191     Vec3f sumR(0), sumM(0); // mie and rayleigh contribution
192     float opticalDepthR = 0, opticalDepthM = 0;
193     float mu = dot(dir, sunDirection); // mu in the paper which is the cosine of the ar
194     float phaseR = 3.f / (16.f * M_PI) * (1 + mu * mu);
195     float g = 0.76f;
196     float phaseM = 3.f / (8.f * M_PI) * ((1.f - g * g) * (1.f + mu * mu)) / ((2.f + g * g) * (1.f + mu * mu));
197     for (uint32_t i = 0; i < numSamples; ++i) {
198         Vec3f samplePosition = orig + (tCurrent + segmentLength * 0.5f) * dir;
199         float height = samplePosition.length() - earthRadius;
200         // compute optical depth for light
201         float hr = exp(-height / Hr) * segmentLength;
202         float hm = exp(-height / Hm) * segmentLength;
203         opticalDepthR += hr;
204         opticalDepthM += hm;
205         // light optical depth
206         float t0Light, t1Light;
207         raySphereIntersect(samplePosition, sunDirection, atmosphereRadius, t0Light, t1Light);
208         float segmentLengthLight = t1Light - t0Light;
209         float opticalDepthLightR = 0, opticalDepthLightM = 0;
210         uint32_t j;
211         for (j = 0; j < numSamplesLight; ++j) {
212             Vec3f samplePositionLight = samplePosition + (tCurrentLight + segmentLengthLight * 0.5f) * dir;
213             float heightLight = samplePositionLight.length() - earthRadius;
214             if (heightLight < 0) break;
215             opticalDepthLightR += exp(-heightLight / Hr) * segmentLengthLight;
216             opticalDepthLightM += exp(-heightLight / Hm) * segmentLengthLight;
217             tCurrentLight += segmentLengthLight;
218         }
219         if (j == numSamplesLight) {
220             Vec3f tau = betaR * (opticalDepthR + opticalDepthLightR) + betaM * 1.1f * (opticalDepthM + opticalDepthLightM);
221             Vec3f attenuation(exp(-tau.x), exp(-tau.y), exp(-tau.z));
222             sumR += attenuation * hr;
223             sumM += attenuation * hm;
224         }
225         tCurrent += segmentLength;
226     }
227

```

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

```

232     return (sumR * betaR * phaseR + sumM * betaM * phaseM) * 20;
233 }
234
235 void renderSkydome(const Vec3f& sunDir, const char *filename)
236 {

```

```

237     Atmosphere atmosphere(sunDir);
238     auto t0 = std::chrono::high_resolution_clock::now();
239     #if 1

Render fisheye

243     const unsigned width = 512, height = 512;
244     Vec3f *image = new Vec3f[width * height], *p = image;
245     memset(image, 0x0, sizeof(Vec3f) * width * height);
246     for (unsigned j = 0; j < height; ++j) {
247         float y = 2.f * (j + 0.5f) / float(height - 1) - 1.f;
248         for (unsigned i = 0; i < width; ++i, ++p) {
249             float x = 2.f * (i + 0.5f) / float(width - 1) - 1.f;
250             float z2 = x * x + y * y;
251             if (z2 <= 1) {
252                 float phi = std::atan2(y, x);
253                 float theta = std::acos(1 - z2);
254                 Vec3f dir(sin(theta) * cos(phi), cos(theta), sin(theta) * sin(phi));
255                 // 1 meter above sea level
256                 *p = atmosphere.computeIncidentLight(Vec3f(0, atmosphere.earthRadius +
257             }
258         }
259         fprintf(stderr, "\b\b\b\b\b%3d%c", (int)(100 * j / (width - 1)), '%');
260     }
261     #else

```

Render from a normal camera

```

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

```

```

272     Vec3f orig(0, atmosphere.earthRadius * 1000, 50000); // camera position
273     std::default_random_engine generator;
274     std::uniform_real_distribution<float> distribution(0, 1); // to generate random flc
275     for (unsigned y = 0; y < height; ++y) {
276         for (unsigned x = 0; x < width; ++x, ++p) {
277             for (unsigned m = 0; m < numPixelSamples; ++m) {
278                 for (unsigned n = 0; n < numPixelSamples; ++n) {
279                     float rayx = (2 * (x + (m + distribution(generator)) / numPixelSam
280                     float rayy = (1 - (y + (n + distribution(generator)) / numPixelSam
281                     Vec3f dir(rayx, rayy, -1);
282                     normalize(dir);

```

Does the ray intersect the planetary 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, t_0 and t_1 . But we only want to compute the atmosphere between $t=0$ and $t=t_0$ (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.

```

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

```

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

```

299         *p += atmosphere.computeIncidentLight(orig, dir, 0, tMax);
300     }
301 }
302 *p *= 1.f / (numPixelSamples * numPixelSamples);
303 }
304 fprintf(stderr, "\b\b\b\b\b%3d%c", (int)(100 * y / (width - 1)), '%');
305 }
306 #endif

```



```

307     std::cout << "\b\b\b\b" << ((std::chrono::duration<float>)(std::chrono::high_resolu
308     // Save result to a PPM image (keep these flags if you compile under Windows)
309     std::ofstream ofs(filename, std::ios::out | std::ios::binary);
310     ofs << "P6\n" << width << " " << height << "\n255\n";
311     p = image;
312     for (unsigned j = 0; j < height; ++j) {
313         for (unsigned i = 0; i < width; ++i, ++p) {
314             #if 1
315                 // Apply tone mapping function
316                 (*p)[0] = (*p)[0] < 1.413f ? pow((*p)[0] * 0.38317f, 1.0f / 2.2f) : 1.0f -
317                 (*p)[1] = (*p)[1] < 1.413f ? pow((*p)[1] * 0.38317f, 1.0f / 2.2f) : 1.0f -
318                 (*p)[2] = (*p)[2] < 1.413f ? pow((*p)[2] * 0.38317f, 1.0f / 2.2f) : 1.0f -
319             #endif
320                 ofs << (unsigned char)(std::min(1.f, (*p)[0]) * 255)
321                 << (unsigned char)(std::min(1.f, (*p)[1]) * 255)
322                 << (unsigned char)(std::min(1.f, (*p)[2]) * 255);
323             }
324         }
325     ofs.close();
326     delete[] image;
327 }
328
329 int main()
330 {
331     #if 1

```

Render a sequence of images (sunrise to sunset)

```

335     unsigned nangles = 128;
336     for (unsigned i = 0; i < nangles; ++i) {
337         char filename[1024];
338         sprintf(filename, "./skydome.%04d.ppm", i);
339         float angle = i / float(nangles - 1) * M_PI * 0.6;
340         fprintf(stderr, "Rendering image %d, angle = %0.2f\n", i, angle * 180 / M_PI);
341         renderSkydome(Vec3f(0, cos(angle), -sin(angle)), filename);
342     }
343     #else

```

Render one single image

```

347     float angle = M_PI * 0;
348     Vec3f sunDir(0, std::cos(angle), -std::sin(angle));
349     std::cerr << "Sun direction: " << sunDir << std::endl;
350     renderSkydome(sunDir, "./skydome.ppm");
351 #endif
352
353     return 0;
354 }

```

