raytracing-training

The goal of this lesson is to code a CPU raytracer. We will use CMake to compile the project and output a .ppm image file.

Using CMake to compile

On windows, install Cmake latest release, with PATH : https://cmake.org/download/ . It should already be installed at school. You can test it by opening a terminal and type `cmake`.

We will use Visual Studio Code to compile and build. Follow these steps:

- Make sure the C++ and CMake Tools extensions are installed.
- Create a folder for your project. Open the folder with Visual Code.
- In the folder, create a `main.cpp` file with a simple program.
- Create a `CMakeLists.txt` file at the root of your folder. This file is the CMake configuration It must contains:

```
cmake_minimum_required(VERSION 3.15)
project(ProjectName)

set(CMAKE_CXX_STANDARD 14)

add_executable(ProjectName main.cpp)
```

- Use Ctrl + Shift + P do open the command palette. Find and execute Cmake: Configure. It will create a `build` folder.
- When configure is done, press F7 to compile.
- Press F5 to ask for execution. Visual Code should tell you you must create a `launch.json` file. Do it choosing C++ Windows, if under windows.
- In the `launch.json` file, change the program line to fit your path. For instance: `"program": "build/Debug/ProjectName.exe",`
- You can add other file to compile by completing the `add_executable` command. For instance:

```
add_executable(OneWeekEnd main.cpp Vec3.h Ray.h Hittable.h Sphere.cpp Sphere.h HittableList.cpp HittableList.h)
```

- Run F5 again

Reading the ppm output

Our raytracing program won't display immediatly the generated image. It will rather create a .ppm file.

To read this file, add the PPM/PGM Viewer extension in Visual Code.

Raytracing course

You are supposed to use the first document of this serie. You can go further if you want.

Drive link to the course : HERE

Resources

Write a PPM file

```
#include <iostream>
#include <fstream>

int main() {
    std::ofstream output;
    output.open("output.ppm");
    output << "blabla" << "\n";
    ...
    output.close();
    return 0;
}
```

Vec3 class

You need this 3d vector class for the course. You can copy and paste it.

Vec3.h

```
#ifndef VEC3_H
#define VEC3_H

#include <iostream>
#include <cmath>
#include <cstdlib>

class Vec3 {
public:
    Vec3() {}

    Vec3(float e0, float e1, float e2) {
        e[0] = e0;
        e[1] = e1;
        e[2] = e2;
    }

    inline float x() const { return e[0]; }

    inline float y() const { return e[1]; }

    inline float z() const { return e[2]; }

    inline float r() const { return e[0]; }

    inline float g() const { return e[1]; }

    inline float b() const { return e[2]; }

    inline const Vec3 &operator+() const { return *this; }

    inline Vec3 operator-() const { return Vec3(-e[0], -e[1], -e[2]); }

    inline float operator[](int i) const { return e[i]; }

    inline float &operator[](int i) { return e[i]; }

    inline Vec3 &operator+=(const Vec3 &v2);

    inline Vec3 &operator-=(const Vec3 &v2);

    inline Vec3 &operator*=(const Vec3 &v2);

    inline Vec3 &operator/=(const Vec3 &v2);

    inline Vec3 &operator*=(const float t);

    inline Vec3 &operator/=(const float t);

    inline float length() const { return sqrt(e[0] * e[0] + e[1] * e[1] + e[2] * e[2]); }

    inline float squaredLength() const { return e[0] * e[0] + e[1] * e[1] + e[2] * e[2]; }

    inline void makeUnitVector();

    float e[3];
};

inline std::istream &operator>>(std::istream &is, Vec3 &t) {
    is >> t.e[0] >> t.e[1] >> t.e[2];
    return is;
}

inline std::ostream &operator<<(std::ostream &os, const Vec3 &t) {
    os << t.e[0] << " " << t.e[1] << " " << t.e[2];
    return os;
}
```

```
inline void Vec3::makeUnitVector() {
    float k = 1.0 / sqrt(e[0] * e[0] + e[1] * e[1] + e[2] * e[2]);
    e[0] *= k;
    e[1] *= k;
    e[2] *= k;
}

inline Vec3 operator+(const Vec3 &v1, const Vec3 &v2) {
    return Vec3(v1.e[0] + v2.e[0], v1.e[1] + v2.e[1], v1.e[2] + v2.e[2]);
}

inline Vec3 operator-(const Vec3 &v1, const Vec3 &v2) {
    return Vec3(v1.e[0] - v2.e[0], v1.e[1] - v2.e[1], v1.e[2] - v2.e[2]);
}

inline Vec3 operator*(const Vec3 &v1, const Vec3 &v2) {
    return Vec3(v1.e[0] * v2.e[0], v1.e[1] * v2.e[1], v1.e[2] * v2.e[2]);
}

inline Vec3 operator/(const Vec3 &v1, const Vec3 &v2) {
    return Vec3(v1.e[0] / v2.e[0], v1.e[1] / v2.e[1], v1.e[2] / v2.e[2]);
}

inline Vec3 operator*(float t, const Vec3 &v) {
    return Vec3(t * v.e[0], t * v.e[1], t * v.e[2]);
}

inline Vec3 operator/(Vec3 v, float t) {
    return Vec3(v.e[0] / t, v.e[1] / t, v.e[2] / t);
}

inline Vec3 operator*(const Vec3 &v, float t) {
    return Vec3(t * v.e[0], t * v.e[1], t * v.e[2]);
}

inline Vec3 &Vec3::operator+=(const Vec3 &v) {
    e[0] += v.e[0];
    e[1] += v.e[1];
    e[2] += v.e[2];
    return *this;
}

inline Vec3 &Vec3::operator*=(const Vec3 &v) {
    e[0] *= v.e[0];
    e[1] *= v.e[1];
    e[2] *= v.e[2];
    return *this;
}

inline Vec3 &Vec3::operator/=(const Vec3 &v) {
    e[0] /= v.e[0];
    e[1] /= v.e[1];
    e[2] /= v.e[2];
    return *this;
}

inline Vec3 &Vec3::operator-=(const Vec3 &v) {
    e[0] -= v.e[0];
    e[1] -= v.e[1];
    e[2] -= v.e[2];
    return *this;
}

inline Vec3 &Vec3::operator*=(const float t) {
    e[0] *= t;
    e[1] *= t;
    e[2] *= t;
    return *this;
}

inline Vec3 &Vec3::operator/=(const float t) {
```

```
    float k = 1.0f / t;

    e[0] *= k;
    e[1] *= k;
    e[2] *= k;
    return *this;
}

inline Vec3 unitVector(Vec3 v) {
    return v / v.length();
}

inline float dot(const Vec3 &v1, const Vec3 &v2) {
    return v1.e[0] * v2.e[0]
           + v1.e[1] * v2.e[1]
           + v1.e[2] * v2.e[2];
}

inline Vec3 cross(const Vec3 &v1, const Vec3 &v2) {
    return Vec3(v1.e[1] * v2.e[2] - v1.e[2] * v2.e[1],
                v1.e[2] * v2.e[0] - v1.e[0] * v2.e[2],
                v1.e[0] * v2.e[1] - v1.e[1] * v2.e[0]);
}

inline Vec3 reflect(const Vec3& v, const Vec3& n) {
    return v - 2 * dot(v, n) * n;
}

inline bool refract(const Vec3& v, const Vec3& n, float niOverNt, Vec3& refracted) {
    Vec3 uv = unitVector(v);
    float dt = dot(uv, n);
    float discriminant = 1.0 - niOverNt * niOverNt * (1 - dt * dt);
    if(discriminant > 0) {
        refracted = niOverNt * (uv - n * dt) - n * sqrt(discriminant);
        return true;
    }
    return false;
}

inline float schlick(float cosine, float refractionIndex) {
    float r0 = (1 - refractionIndex) / (1 + refractionIndex);
    r0 = r0 * r0;
    return r0 + (1 - r0) * pow((1 - cosine), 5);
}

#endif
```