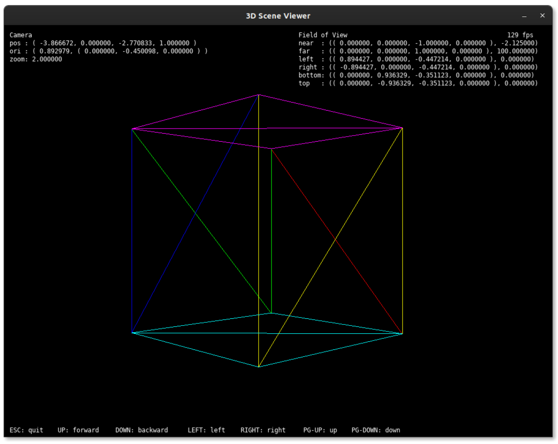
FMJ–Mathematical Tools: Lab Project #1

Due Tuesday, 16 November 2021 23:59 (UTC+2)

The lab projects of FMJ–Mathematical Tools will lead you to create your own “Rasterizer” from scratch in C++. A rasterizer is a software that takes an image described in a vector format as input and converts it into pixels so that it can be printed or displayed on the screen. This will be done in several steps. Each step corresponds to one lab project. When finished (after the last lab project), the final product will look like the picture below.



```
Camera
pos : ( -3.866672, 0.000000, -2.770833, 1.000000 )
ori : ( 0.892979, ( 0.000000, -0.450098, 0.000000 ) )
zoom: 2.000000

Field of View                                                  129 fps
near  : (( 0.000000, 0.000000, -1.000000, 0.000000 ), -2.125000)
far   : (( 0.000000, 0.000000, 1.000000, 0.000000 ), 100.000000)
left  : (( 0.894427, 0.000000, -0.447214, 0.000000 ), 0.000000)
right : (( -0.894427, 0.000000, -0.447214, 0.000000 ), 0.000000)
bottom: (( 0.000000, 0.936329, -0.351123, 0.000000 ), 0.000000)
top   : (( 0.000000, -0.936329, -0.351123, 0.000000 ), 0.000000)

ESC: quit   UP: forward   DOWN: backward   LEFT: left   RIGHT: right   PG-UP: up   PG-DOWN: down
```

Creating a rasterizer from scratch means that you will program every detail of a software that creates 3D images. The software will be able to show 3D scenes on the screen and also provide keyboard commands to move the camera on the scene, thus permitting the user to see the scene from every angle. The aim of such a project is twofold: (1) to help you understand the theoretical material thought on the course and (2) to give you a detailed idea of all the steps video games, simulators and other software use to render 3D images.

Template

You need to start your work in an organised way in order to be able to continue it in the next lab projects. On the website of the discipline, you will find a template corresponding to this lab project for download. You are free to choose a name for your software. Once you do that, create a directory with that name (or acronym) and, inside it, extract the files of the template. The template contains the directory hierarchy below:

```
<software_name>/
   |
   +- bin/         (executables)
   +- build/       (objects --- .o)
   +- src/         (implementation --- .h and .cpp)
   +- test/        (test routines --- .h and .cpp)
   +- .gitignore
   +- makefile
   +- README.md
```

The contents of each subdirectory and file is as follows:

- `bin`: Location of the binary executable files created when the project is compiled, including the executables created to test your implementation.
- `build`: Location of the temporary object files created by the compiler.
- `src`: Location of the source files (`.h` and `.cpp`) of your implementation.
- `test`: Location of the source files (`.h` and `.cpp`) of the test routines.
- `.gitignore`: If you use git (optional), this file prevents git from unnecessarily pushing temporary files to the server.
- `makefile`: Script used to compile the software and test routines.
- `README.md`: Text file in mark down format. You have to fill up this file with relevant information to whoever downloads your software. This file must contain, at least the description of the software (a paragraph is enough), the name of its author, and instructions of how to compile, test and use the software.

  The readme is also the place to explain eventual limitations and bugs of your implementation. If your implementation has a known bug, take the time to explain it in the readme. Since part of the evaluation will be done automatically, a good explanation in the readme my prevent your work from receiving a very bad grade because of a small mistake!

You may also add a `LICENCE` file if you like.

This is the structure for this lab project. More files and directories will be added to this on the next labs.

Aline: A Linear Algebra Library

In this lab project, your job is to create a library containing the most fundamental data structures and functions for the implementation of the rasterizer, i.e., vectors and matrices.

You will create the namespace `aline` wherein all data structures and functions of the library will be defined. Each main data structure must have its own header file (`.h`), implementing the interface of the data structure, and an optional implementation file (`.cpp`). Both files are to be put in the `src` directory. (You are allowed to create more source files if you like.)

In this lab project you will create two classes: `Vector` and `Matrix`. Thus, at least two header files `vector.h` and `matrix.h` must be created.

Vectors

`class Vector<T,N>`
Defines a fixed size sequence of N values of type T, where N and T are class parameters. You must, at least, define the following elements:

- **Constructors:**
  - `Vector()`
    Constructs a vector filled with zeros.
  - `Vector( std::initializer_list<Vector<T,N>> )`
    Constructs a vector with the values given as arguments. Fills up with zeros if less than `N` arguments are given. Throws `runtime_error` if more than `N` arguments are given.

  - `Vector( const Vector<T,N> & )`
    Constructs a copy of the vector given as argument.

- **Methods:**
  - `T Vector<T,N>::at( size_t ) const`
    The element indexed by the given argument. Throws `runtime_error` if the index is out of range.
  - `T Vector<T,N>::operator[]( size_t ) const`
    Subscripting (the as `at()`, but does not throw an exception).
  - `T & Vector<T,N>::operator[]( size_t )`
    Subscripting permitting assignment.
  - `Vector<T,N> & Vector::operator+=( const Vector<T,N> & )`
    Vector addition and assignment.

- **Functions:**
  - `Vector<T,N> cross( const Vector<T,N> &, const Vector<T,N> & )`
    The cross product of two vectors. Uses only the first 3 elements (zero the others in the result). Throws `runtime_error` if the vectors have less than 3 elements.
  - `T dot( const Vector<T,N> &, const Vector<T,N> & )`
    The dot product of two vectors.
  - `bool isnan( const Vector & )`
    Tests if the vector contains `NAN` (not a number) values. (It is sometimes useful when the result of a computation does not exist, e.g. division by zero).
  - `bool is_unit( const Vector<T,N> & )`
    Tests if the vector is a unit vector.
  - `bool nearly_equal( const Vector<T,N> &, const Vector<T,N> & )`
    Tests if two vectors contain nearly equal values. Two values are nearly equal when they are very close, with respect to their magnitudes. For example, 1.0000001 can be considered nearly equal to 1, whereas 1.234 is not nearly equal to 1.242. However, because of their magnitude, 67329.234 can be considered nearly equal to 67329.242.[1]
  - `T Vector<T,N>::norm()`
    The norm (magnitude) of the vector.
  - `bool operator==( const Vector<T,N> &, const Vector<T,N> & )`
    Tests if two vectors contain the same values.
  - `bool operator!=( const Vector<T,N> & v1, const Vector<T,N> & v2 )`
    Test if two vectors contain different values.
  - `std::ostream operator<<( std::ostream &, const Vector<T,N> v )`
    Output operator.
  - `Vector<T,N> operator+( const Vector<T,N> &, const Vector<T,N> & )`
    The sum of two vectors.
  - `Vector<T,N> operator-( const Vector<T,N> & )`
    The negation of a vector.

[1]For more information about that, you may consult, e.g., `https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/`.

- `Vector<T,N> operator-( const Vector<T,N> &, const Vector<T,N> & )`
  The subtraction of two vectors.
- `Vector<T,n> operator*( const T &, const vector<T,N> & )`
  The product of a scalar and a vector.
- `Vector<T,n> operator*( const vector<T,N> &, const T & )`
  The product of a vector and a scalar.
- `Vector<T,n> operator*( const vector<T,N> &, const Vector<T,N> & )`
  The dot product of two vectors.
- `Vector<T,n> operator/( const Vector<T,N> & v, const T & s )`
  The division of a vector by a scalar (same as the multiplication by $\frac{1}{s}$).
- `T Vector<T,N>::sq_norm()`
  The square of the norm (magnitude) of the vector.
- `std::string to_string( const Vector<T,N> & )`
  A string representation of a vector.
- `Vector<T,N> unit_vector( const Vector<T,N> & v )`
  The vector normalized.

Matrices

`class Matrix<T,M,N>`
Defines a fixed size $M \times N$ (M lines per N columns) matrix of values of type T, where M, N and T are class parameters. You must, at least, define the following elements:

- **Constructors:**
  - `Matrix()`
    Constructs a matrix filled up with zeros.
  - `Matrix( std::initializer_list<Vector<T,N>> )`
    Constructs a matrix with the vectors given as arguments. Each vector is one line of the matrix.
  - `Matrix( const Matrix<T,M,N> & )`
    Constructs a copy of the matrix given as argument.
- **Methods:**
  - `at( size_t )`
    The line indexed by the given argument. Throws `runtime_error` if the index is out of range.
  - `at( size_t, size_t )`
    The element indexed by the given arguments. Throws `runtime_error` if the index is out of range.
  - `Vector<T,N> Matrix<T,M,N>::operator[]( size_t i ) const`
    Subscripting.
  - `Vector<T,N> & Matrix<T,M,N>::operator[]( size_t i )`
    Subscripting permitting assignment.

  - `Matrix<T,M,N> & Matrix<T,M,N>::operator+=( const Matrix<T,M,N> & )`
    Matrix addition and assignment.

- **Functions:**
  - `bool inverse( const Matrix<T,M,N> & )`
    The inverse of a matrix.
  - `bool isnan( const Matrix<T,M,N> & )`
    Tests if a matrix contains NAN (not a number) values.
  - `bool operator==( const Matrix<T,M,N> &, const Matrix<T,M,N> & )`
    Tests if two matrices contain the same values.
  - `bool operator!=( const Matrix<T,M,N> &, const Matrix<T,M,N> & )`
    Tests if two matrices contain different elements.
  - `std::ostream & operator<<( std::ostream, const Matrix<T,M,N> & )`
    Output operator.
  - `Matrix<T,M,N> operator+( const Matrix<T,M,N> &, const Matrix<T,M,N> & )`
    The sum of two matrices.
  - `Matrix<T,M,N> operator-( const Matrix<T,M,N> & )`
    The negation of a matrix.
  - `Matrix<T,M,N> operator-( const Matrix<T,M,N> &, const Matrix<T,M,N> & )`
    The subtraction of two matrices.
  - `Matrix<T,M,N> operator*( const T &, const Matrix<T,M,N> & )`
    The product of a scalar and a matrix.
  - `Matrix<T,M,N> operator*( const Matrix<T,M,N> &, const T & )`
    The product of a matrix and a scalar.
  - `Vector<T,M> operator*( const Matrix<T,M,N> &, const Vector<T,N> & )`
    The product of a matrix and a vector.
  - `Matrix<T,M,O> operator*( const Matrix<T,M,N> &, const Matrix<T,N,O> & )`
    The product of two matrices.
  - `Matrix<T,M,N> operator/( const Matrix<T,M,N> & m, const T & s )`
    The division of a matrix by a scalar (same as the multiplication by $\frac{1}{s}$).
  - `Matrix<T,M,N> to_string( const Matrix<T,M,N> & )`
    A string representation of a matrix.
  - `Matrix<T,M,N> transpose( const Matrix<T,M,N> & )`
    The transpose of a matrix.

Tests

You should verify if your library is working correctly by compiling and running the test routines provided in the directory `test`. These tests are provided to help you know how your library will be tested and evaluated by your instructor. The evaluation of your work will be done using a larger set of tests though. You are of course allowed to add more tests to the test routines provided as well as make your own test routines in order to be sure that your implementation of the library is correct.