This software has been developed to help students understand and visualize geometries created by manipulating points, parametric curves and parametric surfaces.

1 Installation

The software does not require any installation, you can simply download it and unzip in a folder of your choice.

2 Compiling from sources

If you have access to the sources you can compile them under Linux and OS X quite easily, since we provide with some scripts to setup everything in a mostly automated way.

Under Windows, the code has been developed and compiled under MSYS2, which is a software distro and building platform for Windows. What that means is you have a linux-like shell environment and package manager, with a mingw64-based compiler toolchain. The generated executables will run on any 64-bit Windows machine, provided you bundle a few `dll` files.

2.1 Building instructions

Both the software and its dependencies are written in C++and use CMAKE as a build system. You can simply [git clone] all the dependencies, cmake install them and be good to go. However, if you want a more guided way of building everything from scratch, you can start by cloning a repository that contains a series of scripts to help you in the process: [git cmake_scripts]. Those will download and build all the parts with the needed cmake flags, using a temporary build folder so that your system root does not get polluted with any cmake install used in the process.

For compiling you will need the following tools:

- a recent compiler with C++14 support
- `cmake`
- the `ninja-build` build system
- `pkg-config`

You will also need some development libraries. For Debian-based linux, the list of packages is as follows:

- `libglfw3-dev`
  - or, as an alternative, all of the following:
    `libx11-dev libxrandr-dev libxinerama-dev libxcursor-dev libxi-dev`
- `libgl1-mesa-dev`
- `libgtk-3-dev`

Please note that the dependency names might be slightly different depending on your distribution.

The list of operations is as follows:

- `git clone http://github.com/francesco_cattoglio/cmake_scripts.git`
- run the `./cmake_scripts/all_git_downloads.sh` script
- `mkdir tmp_folder`
- export environment var: `export CMAKE_BUILDS_PREFIX=/full/path/to/tmp_folder`
- launch the `./cmake_scripts/build_everything.sh` script
  - this will trigger the individual build scripts for each dependency
- launch the final executable in the `./dcs/build` folder

Since the build script makes use of relative paths, please **do not change folder** before launching it.

If something goes wrong, instead of calling the `build_everything` script, you can build and install dependencies one at a time and keep the build scripts as references for the needed cmake flags.

2.2 Deployment

To deploy the executable, the procedure is a bit different, depending on which OS you are on. For Linux, everything is really simple since all of the libraries are compiled in as static libs.

On Windows, ...

On OS X, ...

3 User Interface

3.1 Basic concepts

The central element of the user interface is the node graph editor. Inside it you create and link nodes of different type to assemble the scene containing all the computed objects.

There are several different kind of nodes, each with a different functionality and a different number of inputs, but only one output. This output contains some data that can be used as an input to one or more nodes, simply by linking to them. While the same output can be used as input to multiple nodes, the opposite is an error: you cannot use data coming from 2 outputs into the same input. The data used in the graph is of one of the following types:

- `Geometry`
- `Vector`
- `Matrix`
- `Time Transform`
- `Interval`
- `Value`

One can only connect an output to an input of the same type. For example, it would not make sense to pass an **Interval** to a node that requires a **Matrix** input.

3.2 Data types

3.2.1 Geometry type

The most important data type is **Geometry**. This represents a generic geometric object, it might be a point, a curve, a surface or a pre-fabricated mesh. Most of the time we start by creating a geometry of some sort, then manipulate it with transform functions, and in the end plug the result into a Renderer node that is the one that turns our data into something that we visualize on screen. As we will see later on, parametric transformations and sampling operations can change the geometric dimension of a Geometry object. As an example, a given curve can become a surface by applying a parametric transform to it.

3.2.2 Vector type

As the name implies, the data contained in this type is just a vector with user-chosen x, y and z coordinates. There is however a very important difference between a vector and a point: a vector is **not** a Geometry and can **only** be used to build a translation Matrix, as a plane normal or for visualization purposes. This is because the vector is to be intended as a direction, not as a generic coordinate in the 3D space. If we write a vector in homogeneous coordinates, its w component is **always zero**.

3.2.3 Matrix type

This data contains a 4x4 matrix which should be interpreted as a transformation written in homogeneous coordinates. The matrix can be either parametric or a constant-valued one. See the nodes section of this chapter to know more about the different matrices that can be created.

3.2.4 Time Transform

This is a very specific 4x4 matrix with an implicit parameter named t as in *time*. By writing out time-dependent expressions, the user can attach some animation to a rendered Geometry object. This can be useful for example to show the movement of a point along a curve, or to display how a transformation progressively deforms an object into a final shape.

3.2.5 Interval and Value

Both **Interval** and **Value** data deals with parameters: the first contains the parameter and its interval boundaries, while the second contains the parameter and a specific value in its interval, to be used in a **Sample parameter** node.

3.3 Notes

Assembling a graph does not have a 1:1 correspondency with writing procedural code (e.g.: classical C code). You are only describing a series of objects and a set of operations that manipulate those objects. The software will then compute the correct order in which the operations that you defined are to be executed to produce the final output.