FranzPlot's User Manual

August 5, 2019

This software has been developed to help students understand and visualize geometries created by manipulating points, parametric curves and parametric surfaces. Because of this reason it does not feature any advanced graphics capabilities one would expect from a modern 3D software (including shadows, reflections and such), but it will hopefully help you reason about all the non-trivial concepts of the most common parametric objects.

Chapter 1

Installation

The software does not require any installation, you can simply download it and unzip in a folder of your choice. Keep in mind that both Windows and OS X block executables downloaded from the internet, therefore you will need to right-click on the executable and chose the "Open" option. This will ask if the software is safe to execute, and this confirmation only needs to be given once. You can skip the rest of this chapter if you have no need of compiling from sources.

1.1 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.

Under OS X, the code can been compiled using Apple's version of Clang (the default system compiler). Some additional tools like CMake are however required, the developer suggests the use of the homebrew package manager for fetching all the dependencies.

1.1.1 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. 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.

To guided compilations via provided scripts, 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.

1.1.2 Deployment

The procedure to deploy the executable is slightly different depending on the target OS. If you used the provided scripts, both the Linux and OS X builds should be standalone executables with no external dependencies. The application might be named “dcs” instead of “franzplot”, depending on the version of the sources you are using.

On Windows, to help mitigate some bugs in the OpenGL Intel driver that is used on most laptops with integrated Intel HD graphics, the build process uses SDL2 and Google’s ANGLE software. When distributing your executable, you have to also bundle it with the libraries provided in the `win_deploy` subfolder: `libglesv2.dll`, `libegl.dll`, `libwinpthread-1.dll`, `sdl2.dll`. It is still possible to distribute a single `.exe` file by using a packager. The tool that we used for such a task is called NSIS: Nullsoft Scriptable Install System, which is freely available under an open source license.

To create a standalone launcher, follow those steps:

- install the NSIS software (`https://nsis.sourceforge.io/Main_Page`)
- compile the software as per the previous section
- move the compiled executable inside the `win_deploy` folder; **make sure to rename it to franzplot.exe**
- right click on the `create_launcher.nsi` file and chose “compile script” from the menu
- check that the created launcher works properly by moving it to a different folder and executing it

Chapter 2

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 other nodes, simply by dragging a link between them using the mouse. While the same output can be used as input to multiple nodes, the opposite is an error: you cannot feed data coming from 2 outputs into one input.

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.

2.1 Data types

Before we dive in the user interface, it is important to briefly talk about all the different Data types that the user will encounter while using the software.

2.1.1 Geometry

The most important data type is the **Geometry**. This represents a generic object, it might be a point (0D), a curve (1D), a surface (2D) 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 modified geometry into a Renderer node to visualize the result on screen. As we will see later on, parametric transformations and sampling operations can change the number of dimension of a Geometry object. As an example, applying a parametric transform to a given curve may turn it into a surface.

2.1.2 Vector

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 point in the 3D space. If we were to write such vector in homogeneous coordinates, its w component is **zero**.

2.1.3 Matrix

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 section 3.1 to know more about the different matrices that can be created.

2.1.4 Time Transform

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

2.1.5 Interval and Value

Both Interval and Value data are related to parameters. An **Interval** defines a name for a parameter and the interval on which it lives. Please note that this is always to be interpreted as a closed interval. A **Value** on the other hand contains the parameter name with a specified value; this information can be used in a Sample Parameter node or in a matrix.

2.2 Notes

Assembling a graph does not have a 1:1 correspondency with writing procedural code (e.g.: typical C++ code). You are only describing a series of objects and a set of operations that manipulate those objects, but you do not have to to define the order in which the computations will be executed. The order of execution will be decided by the software by looking at how nodes depend from each other.

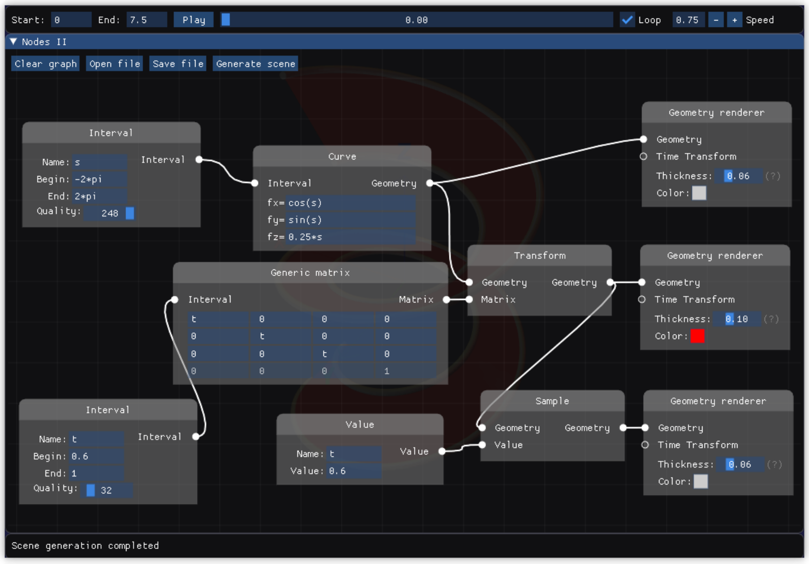
Another important aspect of the node programming approach is that all the operations start from some *sources* (i.e: geometries and parameters) and *flow* to the graph to reach a *sink* (i.e: visualization). It is an error to create a loop in the graph because it would break this flow and there would be no clear way on how to run all the needed computations. As an example, a loop might be created by using the output of a transformation as its own input.

Chapter 3

User Interface

The user interface consists of a top bar, a status bar in the bottom, the Scene Viewport and the Node Graph Editor with the Graph Menu Bar. When the software is started the Node Graph Editor is shown, and it can be minimized by clicking on a small white triangle on its top left corner; this will reveal the Scene Viewport behind it. The top bar and the status bar are always visible.

3.1 Node Graph Editor



Start: 0 End: 7.5 0.00 ✓ Loop 0.75 Speed

Interval — Name: s Begin: -2*pi End: 2*pi Quality: 248

Curve — fx= cos(s) fy= sin(s) fz= 0.25*s

Generic matrix:

t	0	0	0
0	t	0	0
0	0	t	0
0	0	0	1

Interval — Name: t Begin: 0.6 End: 1 Quality: 32

Value — Name: t Value: 0.6

Geometry renderer — Thickness: 0.06 Color: ⬜

Geometry renderer — Thickness: 0.10 Color: 🟥

Geometry renderer — Thickness: 0.06 Color: ⬜

Scene generation completed

The Node Graph Editor is where most of the work is done. Each node represents an operation that creates or modifies data. Besides the inputs and the output, many nodes also have internal fields in which the user can write an expression or choose a value for a given property (e.g. a function, or "$pi/4$" for an angle).

The user can perform the following:

- To add a node, simply right-click on empty space inside the node graph editor; a popup menu will open to display all the available node types, grouped by categories. See the following section for a detailed explaination of all menu entries.
- To move a node around, left click on its header and drag it to its new position.
- You can select a group of nodes by left-clicking on empty space and dragging the mouse, or by clicking on nodes while holding the `ctrl` key, to quickly move all of them.
- To remove a node, right-click on its header and select "delete" from the dropdown menu.
- To connect two nodes, hover over an input or output. When its label turns blue, click it and drag with the mouse to the label you want to connect to.
- To remove a connection, double-click it or hover and press the `delete` key
- The pan the graph either right-click on empty space and drag with the mouse, or use the mouse scroll (defaults to vertical pan, hold `shift` for horizontal pan).
- The user can zoom in or out by holding the `ctrl` key and using the mouse scroll. Clicking the mouse scroll (a.k.a. `Mouse3 button`) will reset the zoom level.

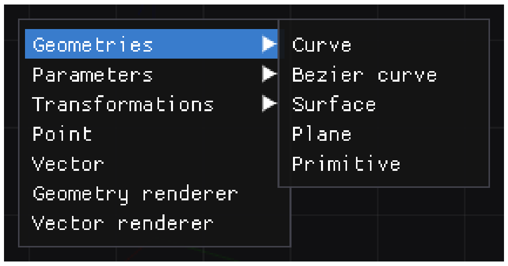


Figure 3.1: The right-click menu, with the "Geometry" submenu open

3.1.1 Geometry submenu

The geometry submenu contains the following nodes:

Curve

Inputs:	Interval
Fields:	fx, fy, fz
Output:	1D Geometry object

This node is used to create a parametric curve from the input parameter Interval and the given expressions for fx, fy and fz. Simple to use, just make sure that the expressions are using the correct parameter name (i.e. the same name used by the input Interval)

Bezier

Inputs:	2, 3 or 4 Points (0D Geometry objects)
Fields:	Quality
Output:	1D Geometry object

This node uses its inputs as control points for a Bezièr curve. The degree of the curve depends on how many inputs have been connected (empty inputs are ignored). The Quality field determines the quality of the numerical approximation of the ideal Bezièr curve.

Surface

Inputs:	2 Intervals
Fields:	fx, fy, fz
Output:	2D Geometry object

Similarly to the curve one, this node can be used to create a parametric surface from two Intervals and the given expressions for fx, fy and fz. Both intervals are required and the user should use both parameters: if one parameter is never used in the expressions, no visual output will be produced, since the final result will be a degenerate surface.

Plane

Inputs:	1 point (0D Geometry), 1 Vector
Fields:	none
Output:	Mesh Geometry object

This node takes one point and a normal vector to create a special representation of a plane. In particular, the representation consists of an evely-spaced grid centered about the point given as input.

Please note the output is **not** a 2D Geometry, but a Mesh Geometry; this means that you will not be allowed to apply a parametric transformation or a sample operation to a it, only a constant-valued transformation can be applied. This is because the purpose of this node is to have something useful for visualization purposes, not to have a built-in parametric plane.

Primitive

Inputs:	none
Fields:	Kind (a drop-down menu), Size
Output:	Mesh Geometry object

This node allows the user to create a basic primitive of the given Kind (Cube, Sphere, Cylinder, Cone, Pyramid or Dice). The Size fields allows adjusting the primitive dimensions. See the question mark next to the slider for a description on how this value effects the output.

Just like the Plane node, the output is a Mesh Geometry, which is useful for quick visualization or similar purposes. Parametric transformations on these objects are **not** allowed.

3.1.2 Parameters submenu

This submenu contains all the nodes related to parameters

Interval

Inputs:	none
Fields:	Name, Begin, End, Quality
Output:	Interval

This node defines a parameter. By assigning a name, you will then be able to use this parameter in expressions for parametric curves, surfaces and matrices. Begin and End define the closed range in which the parameter lives, while Quality allows the user to choose how many discretization points will be used when creating a curve or a surface out of this parameter.

Value

Inputs:	none
Fields:	Name, Value
Output:	Parameter Value

In a similar way to the Interval node, the Value node lets you define a parameter and assign a specific value to it. You can use it either as a "variable" to be used as input for a Matrix (e.g. define *theta* and then use it as the angle inside the matrix expressions) or as an input to the Sample Parameter node.

Sample Parameter

Inputs:	1D or 2D Geometry object, Parameter Value
Fields:	none
Output:	0D or 1D Geometry object

The parameter sampling operation allows you to "downgrade" the dimension of a parametric Geometry object, by fixing a parameter to the value given as input. Mostly useful for obtaining u or v curves.

In order for the operation to succeed, the Geometry must have a parameter with the same name as to the one used in the Value input (e.g. if a surface is $S(p,q)$, the input value cannot be named r, only p or q will be accepted). The value must also be inside the parameter's range.

Please note that 1D Geometry created with the Bezièr curve uses a hidden parameter name, and cannot therefore be sampled.

3.1.3 Transformations submenu

This submenu contains all nodes used to define transformation matrices and to apply them to Geometry objects.

Generic Matrix

Inputs:	Parameter (either Interval or Value, optional)
Fields:	matrix elements
Output:	Matrix object

This node is used to define a generic Matrix, by writing one expression for each matrix element. If an Interval is given as an input, then the output matrix will be a parametric one. Note that due to the nature of the software, there is no way to define a projection matrix, since the last row of the matrix is fixed to $[0,0,0,1]$ and cannot be modified.

As usual, if you provide a parameter as an input, make sure you will be actually using it inside the expressions to prevent any kind of visualization issue.

Rotation Matrix

Inputs:	none
Fields:	Axis (drop down menu), Angle
Output:	Matrix object

This node allows the user to quickly define a matrix which represents a rotation of given Angle (in radians) around the chosen Axis.

Translation Matrix

Inputs:	Vector
Fields:	none
Output:	Matrix Object

Similarly to the previous one, this node allows the user to quickly define a translation matrix given the input Vector.

Transform

Inputs:	Geometry object, Matrix object
Fields:	none
Output:	Geometry object

This node takes a matrix and applies it to a Geometry, producing a transformed Geometry. If the input matrix was a parametric matrix, then a parametric transformation will be applied.

Please note that a parametric transformation is not "blindly" applied to an object; instead the parameter names will be read and the operation will be performed accordingly. As an example, consider a circle on the xy plane parametrized in s: if we apply a parametric translation with value $[0,0,s]$, the output will be a circular helix.

Time Transform

Inputs:	none
Fields:	matrix elements
Output:	Time Transform object

This node creates a Time Transform object, read about Data Types in section 2.1 for more informations. All the matrix element expressions may contain the parameter t, as in “time”.

3.1.4 Point

Inputs:	none
Fields:	x, y, z
Output:	0D Geometry object

A very simple node to create a point. The implicit w coordinate is set to 1

3.1.5 Vector

Inputs:	none
Fields:	x, y, z
Output:	Vector object

A very simple node to create a Vector. As described in the Data Types section, a Vector is to be interpreted as a direction, and the implicit w coordinate is set to 0.

3.1.6 Geometry Renderer

Inputs:	Geometry, Time Transform (optional)
Fields:	Thickness, Color
Output:	none

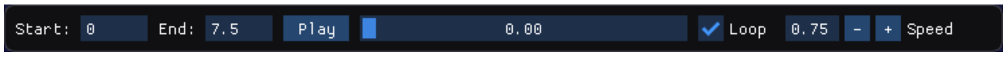
This node is the one responsible for taking a Geometry object and rendering it to the screen. Every frame the the Time Transform is applied to it by evaluating the transformation with the current value of t. The user can choose what Color to use for any Geometry, while the Thickness value only affects the display of 1D geometries (curves).

3.1.7 Vector Renderer

Inputs:	Point, Vector
Fields:	Thickness, Color
Output:	none

This node is used to display a Vector, by representing it as an arrow. Since a Vector is only a direction, the user must also provide the application Point (i.e: the "tail") of the vector. Just like with the Geometry Renderer, one can choose a color and the thickness of the arrow.

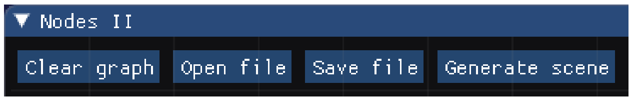
3.2 Top Bar



As it was explained before, any Geometry Renderer can have a Time Transform attached to it. This transformation contains a parameter named t that changes in real time while the user interacts with the rendered scene, as opposed to all other parameters that are evaluated only once when the scene is generated. The Top Bar gives the user control over the value of this parameter t. It contains the following:

- Start: the expression for the beginning of the parameter interval. To use the constant π, write it as `pi`.
- End: the expression for the end of the parameter interval.
- Play/Pause: by clicking "Play" the parameter t will increment as time passes, while by clicking "Pause" the parameter will stay frozen to the value selected by the user.
- nameless slider: the user can click and drag the slider to manually assign a value to t
- Loop checkbox: as the name implies, if this box is ticked the animation will loop (i.e. every time t reaches the end of the interval, it will be rewinded to the start)
- Speed: defines how fast t advances when "Play" is clicked. Defaults to 0.75, meaning that 1 second of wall clock time will advance t by 0.75. The user can either write down the speed or use the `+`/`-` signs to change it by small steps. Negative speeds are allowed!

3.3 Graph Menu Bar



The Graph Menu Bar contains 4 buttons, which function is easy to guess by their labels:

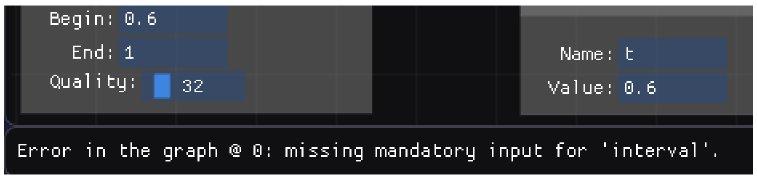
- Clear Graph: deletes all nodes and connections.
- Open File: clears the graph and loads a previously saved file.
- Save File: saves the current graph to a file.
- Generate Scene: when the user is done modifying the graph, clicking this will trigger the generation of the scene.

Clearing the graph, opening and saving a file will display a dialog asking to confirm the operation. Please note that when saving a file the extension `.toml` will be automatically added if the name does not contain it. If any name conflict arises, then the user will be asked whether the software should overwrite the file or abort the operation.

3.4 Scene Viewport

After the scene is generated, the Graph Editor is minimized and the scene containing the generated geometries is shown. Navigation is very easy: left-click and drag to rotate the view and use the mouse scroll or numpad's "+/-" to control zoom. Currently it is not possible to pan around the scene. The scene contains some colored lights placed in specific points in the attempt to make it easier to understand what shapes the user is looking at. However this also means that some combinations of shape and colors might produce dark spots in some geometries, try using a different color if this happens.

3.5 Status Bar



```
Error in the graph @ 0: missing mandatory input for 'interval'.
```

Figure 3.2: The status bar reporting an error in the user-created graph

The status bar is an often overlooked part of the User Interface, probably due to its location at the very bottom of the window. It is however extremely useful because when the software detects a mistake in the graph, it will notify the user with a (hopefully) useful message to help understand where the error is. It will also report if scene generation and file operations were successful or not.