Graphics for ROOT 7

Olivier Couet - April 2017

- Introduction
- Graphics Model
- Requirements
- Interactive Editing
- ROOT 6 functionalities we want to keep.. or not

Introduction

The current ROOT **Graphical User Interface** (GUI) need to be **rethink** in the context of ROOT 7 for at least **three main reasons**:

1. It is very **OS-specific.**
   - Dead end on MacOs: Mac apps cannot embed ROOT windows
   - Dead end on Windows: rely on gdk which is very slow compare to what is now available.
2. Need to **reduce** the code to be **maintained.**
3. **Remote GUI** is required. Base the GUI in a web browser

To fulfil these requirements, moving to a **web based GUI** is the solution.

These changes in the GUI imply that **ROOT 7 Graphics** must be "**on the web**" also.

==> As ROOT 7 is already breaking some backward compatibilities with ROOT 6, graphics for ROOT 7 is not tied to any specific model. Therefore we are free to base it on modern **web based visualisation techniques**.

==> In the past we very reluctant to base graphics on external tools (like Qt) not being sure of their lifetime. These days "Web Graphics" is supported by a very large community. Therefore relying on tools like D3 is possible.

Graphics model

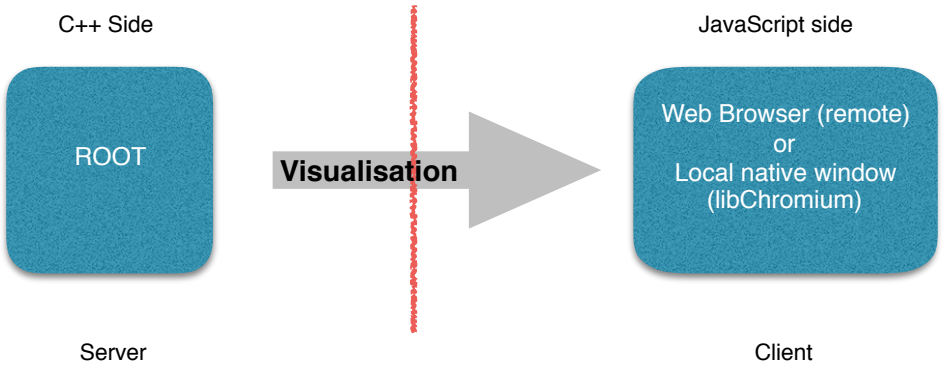
The basic ideas for future ROOT 7 graphics:

- **ROOT 7** is running as usual and **generates the graphics display list** by running user macro or plotting objects. **This is the "server side"** or **"C++ side"**.

- Instead of being rendered by the usual C++ painters, the **graphics** display list is **sent to a client** which can be remote (Web Browser) or local (for instance a TCanvas based on libChromium). **Each object in the display list has a painter on the client side.**

==> This **Client Side** is written in JavaScript and does the rendering thanks to systems like D3 for SVG rendering or three.js for WebGL rendering.

The following figure summarise the model:



Graphics model

- This model allows to be **independent** from **any local graphics backend** (X11 or Cocoa) as the final rendering is done via standards like SVG (Scalable Vector Graphics) and/or WebGL.

- It allows to have **remote display for free** as graphics can be rendered in web browser thanks to the http protocol.

- For **local display** the JavaScript rendering might be performed in a **local canvas** via libraries like libChromium.

==> A prototype of a such system already exists. It was started by *Bertrand Bellenot* and *Sergey Linev*.
It is called "**JSROOT**".

The initial goal of **JSROOT** was to read objects (histogram, graphs, canvas etc …) in ROOT files and display them in a web browser using JavaScript.

This first approach is now working and **allows to browse objects in ROOT files**. It is **also used by the SWAN project**. Once displayed the objects can be manipulated in the web browser (zoomed, scaled, etc…).

Requirements

General requirements

- **Multi-display** (multiple view of multiple objects) support.
- The **same object** can be displayed **in multiple pads**, or even multiple times in the same pad.
- **Keep the two sides in sync:**
  - Transfer of objects to the browser, to allow **client state**.
  - The server **send delta** of what has changed. Delta is on object level ("pad"), not on property level.
  - **A communication package is needed** for context menus as well as even handling / reaction (e.g. `TGraph::AddPoint` at mouse click position).
- We need to allow dedicated **I/O for the graphics representation**; default ROOT I/O might be the default (see slide on “graphics output files")..

Requirements

Object identifier (ObjectID)

Objects are identified thanks to a **unique ObjectID** known on the client and server side to be sure any object changes on one side can be properly reflected on the other.

- ***Per pad****, which can be inside a pad which is inside a canvas.*
- ***Canvas*** *will have global unique* ***canvas ID***
- ***ObjectID*** *is unique counter, kept in each pad, assigned at drawable creation. (To find an object in a pad with a given ID we'll have to do a linear search).*

Canvas versus browser

- **Locally**, Create a new TCanvas by creating a **native window** embedding a Chromium's canvas. We need also to run in **batch mode** (libChromium should help also).
  Remotely, display a URL that people can click to spawn a remote browser.

- By default, new canvas will open new tab / browser window

- A canvas needs to be embeddable (like TRootEmbeddedCanvas)

Requirements

Graphics output on files

Two main kind graphics output images are required:

1. **Vector graphics** output like **PDF**, **PostScript**, **SVG** and **Latex**. In ROOT 6 vector graphics formats are implemented by native ROOT classes not relying on any external libraries. They are the exact clone of what is visible on screen.
2. **Bitmap output** like **png**, **gif** (also animated gif), **jpeg tiff** etc .. In ROOT 6 the bitmap outputs are implemented thanks to the libAfterImage (ImageMagick can also be consider to replace it) library which has been included in the ROOT distribution.

This kind of output represent a **large fraction of the graphics produced by ROOT**. Usually ROOT runs in batch mode without any graphics displayed on screen. **Massive productions of images** can be launched as bitmap or vectors graphics images.

SVG output: unlike PDF or PS, produces only one image per file (like for the bitmap output). It can be used to include **high quality** zoomable **plots** in **web pages**.

Latex output: it consist of **PGF/TikZ vector graphics** output which is the format required to include graphics in LaTeX documents to **have exactly the same look and feel** as the rest of the **LaTeX document**.

Some solutions need to be found in the ROOT 7 context:

- The client side generates SVG graphics.
- Some converter are needed to other formats like PDF.
- The rendering of Math formulae, done with MathJax seems fine with PDF output; if needed we can use MathJax4SVG
- Batch mode PNG output is assumed to be fine...

Requirements

2D vs 3D

- **2D** is rendered as **SVG**.
- **3D** scenes produced by Eve are rendered using **WebGL**.

  ==> That's two different systems. Question: Can we use only 3D, even for 2D ? may be yes (we do it now in ROOT 6 with the OpenGL backend).

three.js allows to generate SVG; the problem is that it is relatively slow and cannot be used for interactive rendering (lack of basic support for SVG in three.js interactive tools).

For **now** , assume **both 2D and 3D canvases are needed**.

Requirements

Coordinates systems

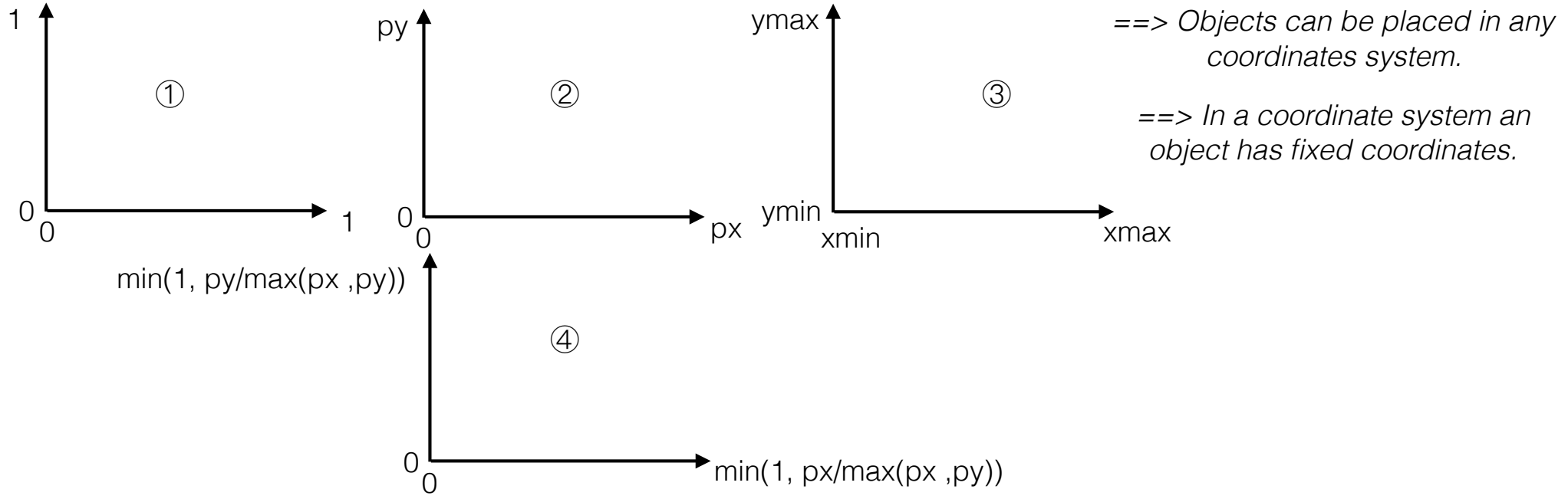
Coordinate systems are :

1. **Normalized Coordinates** (% of canvas size). It allows to position an object or define the size of a text in % of the canvas. This is a convenient way for user to place objects independently of the user coordinates.

2. A Canvas has its size in **Pixels Coordinates** (on the client), possibly different for each view; can have initial server provided size, but in the end the size is defined by each client independently.

3. **User Coordinates** (currently provided by TFrame), can have multiple, each defining their own coordinate system and mapping to normalized device coordinates, log scales; use by drawing “into” them.

4. **Normalized Device Coordinates**: This system keeps the aspect ratio of primitives when the canvas aspect ratio changes. Typically it allows an arrow head to alway be a triangle. This coordinate system is mainly for **internal use**.



==> Objects can be placed in any coordinates system.

==> In a coordinate system an object has fixed coordinates.

Interactive Editing

Object selection

- Selection is "powerpoint" like. Objects can be selected using a **lasso** or a multiple selection with mouse click while the **CTRL key** is pressed.
- Selected objects can then be **grouped**, **copy/pasted** etc …The interactive part of this operation occurs **on the client side**.

Object insertion

- With mouse clicks; equivalent of the ROOT 6 "tool bar".
- This operation occurs **on the client side**.
- Once an object is fully created it is sent back **to the server side**.

Interactive Editing

Grouping

Objects can be grouped.

- The grouped objects are removed from the original pad and moved into a new overlay pad.
- The initial pad will contain the new pad.
- The relative position of the contained objects stays fixed.

Objects can only be grouped with objects in the same coordinate system.

Copy/Cut & Paste

On the client side object or group of objects can be cut/copied and pasted.

Once the operation is completed, the new created object are sent back to the server.

If an object is cut or deleted it is also reflected to the server side.

Usual shortcuts should be available `CTRL-C`, `CTRL-X` and `CTRL-V`.

Interactive Editing

Object ordering (occlusion)

Objects are stored on the server side in a display list.

In 2D they are displayed "in order”. Therefore the first object in the list will be displayed first and will be in the background of the resulting image (this is only valid in 2D of course).

As the last drawn object may hide the previously drawn objects, a push/pop mechanism should be provided to interactively reorder objects and pads.

In 3D objects' positions are fully defined in the 3D space.

Undo

Actions coming from interactive clients should be undoable. Changes coming from C++ side not necessarily

The C++ side should take care of pushing undo actions, based on events it gets (e.g. “SetTitle” should push the previous title).

ROOT 6 functionalities we want to keep .. or not

TExec

In ROOT 6, `TExec` is a utility class that can be used to execute C++ code when the canvas display list is traversed (or the list of functions of an histogram). By itself this class is a “No-Op” class until the C++ code attached to it is defined by the user program.

A `TExec` object, "`exec`", will be placed in the list of pad primitives as soon as the command `exec.Draw()` is performed.

When the pad is rendered ("painted" in the ROOT jargon), the `TExec::Paint` function is called. This function will execute the specified C++ code attached to `TExec`.

In the new system it should reacts on `pad::update()`. The TExec’s code should be run on the C++ side

What we might lose ?

PDF/TikZ vector graphics output (for LaTeX output, class TTeXDump), can we use inkscape's SVG
=> PDF + Latex? (this is not completely equivalent as this approach generates two files)