ROOT 7 Graphics

Olivier Couet (CERN EP-SFT)

Introduction

ROOT 6 **GUI** is showing its age and need to be **rethough** in the context of ROOT 7, because:

➔ It is very ”**OS-specific**”

➔ The need to **reduce** the amount of code to be actively **maintained**

➔ A **Remote/web GUI** is required.

These changes in the GUI will trigger that **ROOT Graphics** must be “**on the web**” also.

➔ Like the rest of ROOT 7, graphics can be reinvent based on 20 years of experience.

➔ In the past we were very reluctant to base graphics on external tools (like Qt) not being sure of their lifetime. These days "Web Graphics” is based on widely used technologies (SVG and WebGL) supported by a very large community (D3.js, THREE.js).

Graphics model (1)

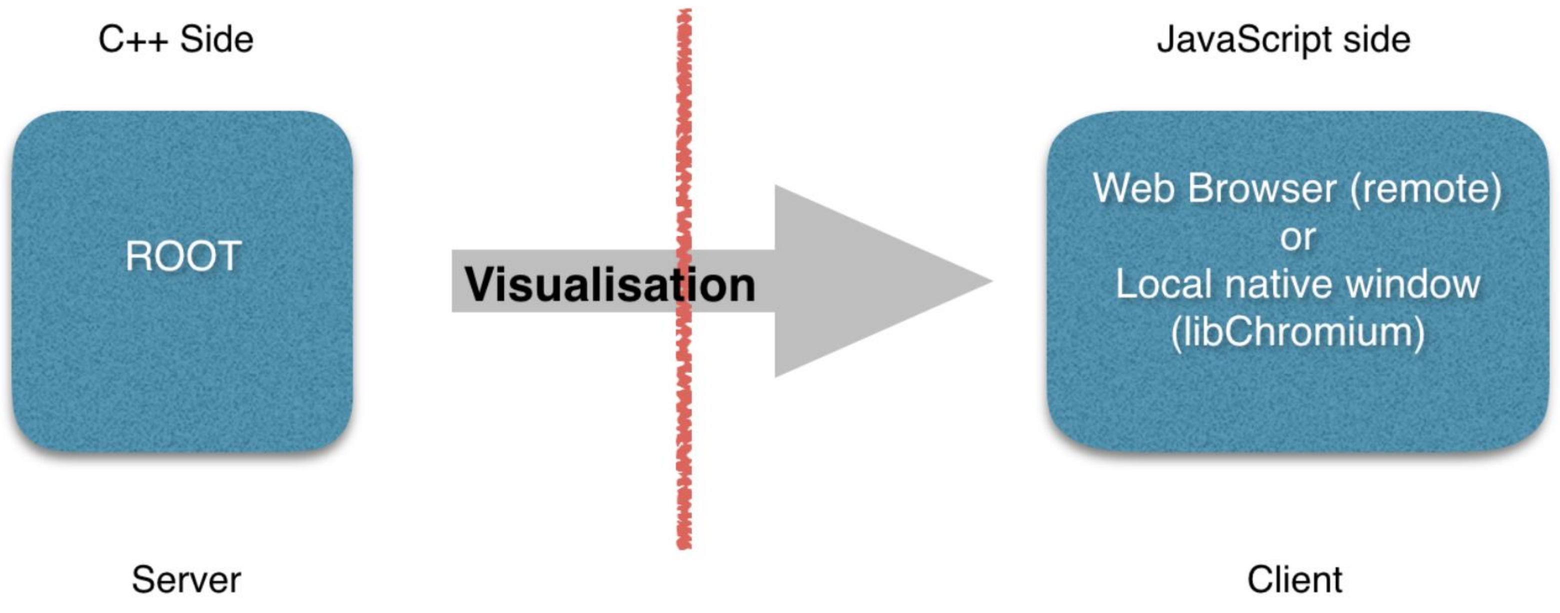
Basic ideas for future ROOT 7 graphics:

- **ROOT 7** is running as usual and **generates the graphics display list** by drawing objects.

  ➔ **This is the "server side"** or **"C++ side".**

- The **graphics** display list is **sent to a client** which can be remote (Web Browser) or local (libChromium).

  ➔ **This is the "client side"** or **"JavaScript side".** It does the graphics rendering (D3, THREE.js).



Graphics model (2)

Advantages of this model:

- **Independent** from **any local graphics backend** (X11 or Cocoa).
- Allows **remote display for free** on all kind of devices (PC, tablet, phone ..)
- For **local display** the JavaScript rendering might be performed in a **local canvas** via libraries like libChromium.

➔ A implementation of a such system (client side) already exists for ROOT 6: “**JSROOT**”
(by Bertrand Bellenot and Sergey Linev).

The initial goal of **JSROOT** was to read/browse objects in ROOT files and display them in a web browser using JavaScript.

Once displayed the objects can be manipulated in the web browser (zoomed, scaled, etc…).

It is **also used by the ROOT jupyter interface**.

Graphics output on files (Batch Output)

Two main kinds of batch 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.

2. **Bitmap output** like png, gif, jpeg tiff etc .. In ROOT 6 the bitmap outputs are implemented natively on top of libAfterImage.

In ROOT 7:

➔ **Goal:** not rely on any native ROOT library.

Still under investigation.

One possibility:

Browsers like Chrome or Firefox provide a **"headless" mode**. In this mode the complete HTML/JavaScript/SVG code works without screen display.

The idea would be to **use this "headless" mode to generate SVG/PNG/JPEG/PDF images.**

Need to find a solution for TeX output

Basic Concepts (1)

Pad:

- Base entities containing the list of graphics objects to be drawn (Drawable).
- Implemented in the **RPad** C++ class.

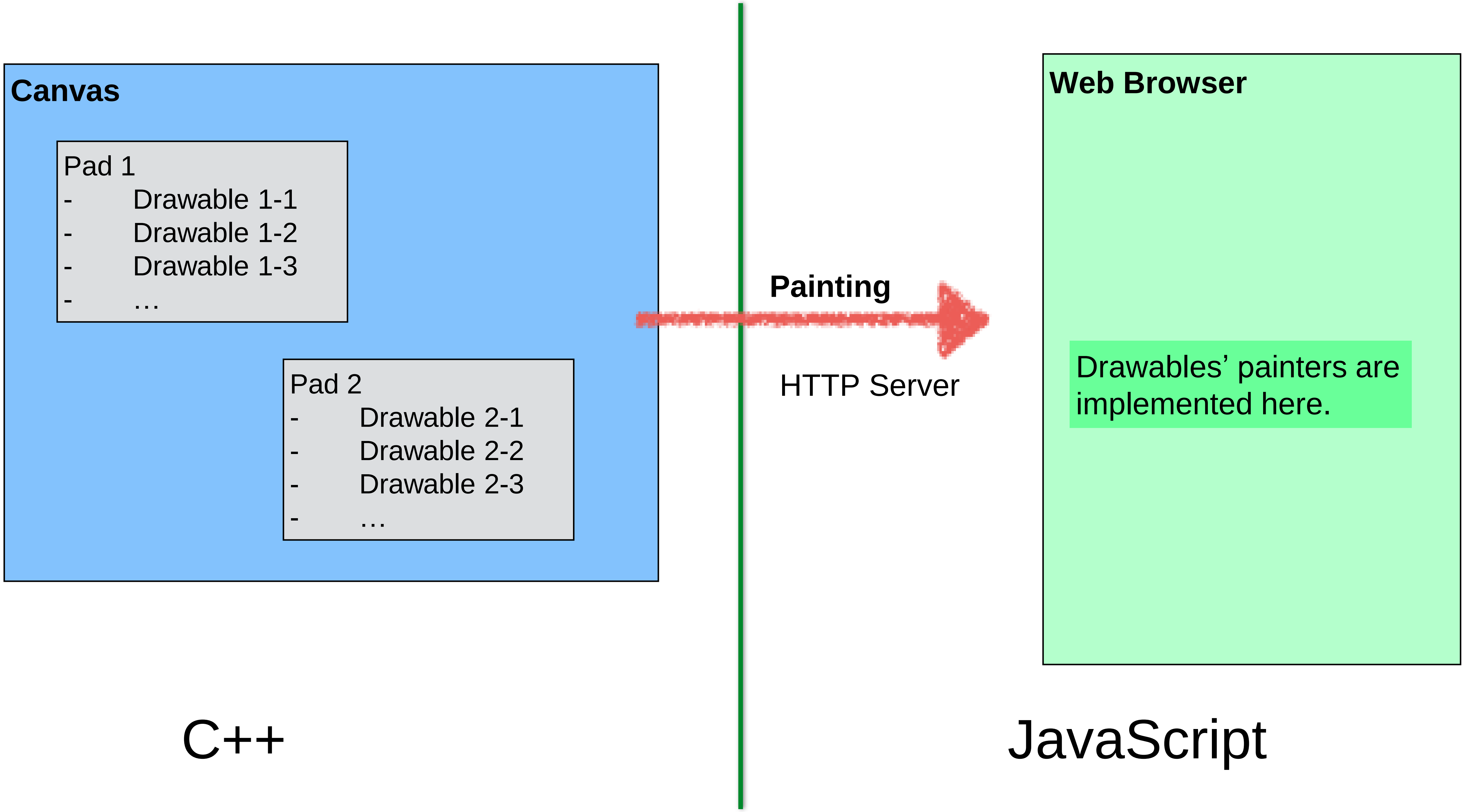
Canvas:

- A window's topmost pad.
- Implemented in the **RCanvas** C++ class.

Drawable:

- Something which can be drawn on a pad.
- Implemented in the **RDrawable** C++ class.
- Each drawable entities has a GetDrawable method

Basic Concepts (2)



GetDrawable()

➔ Decouple data and graphics

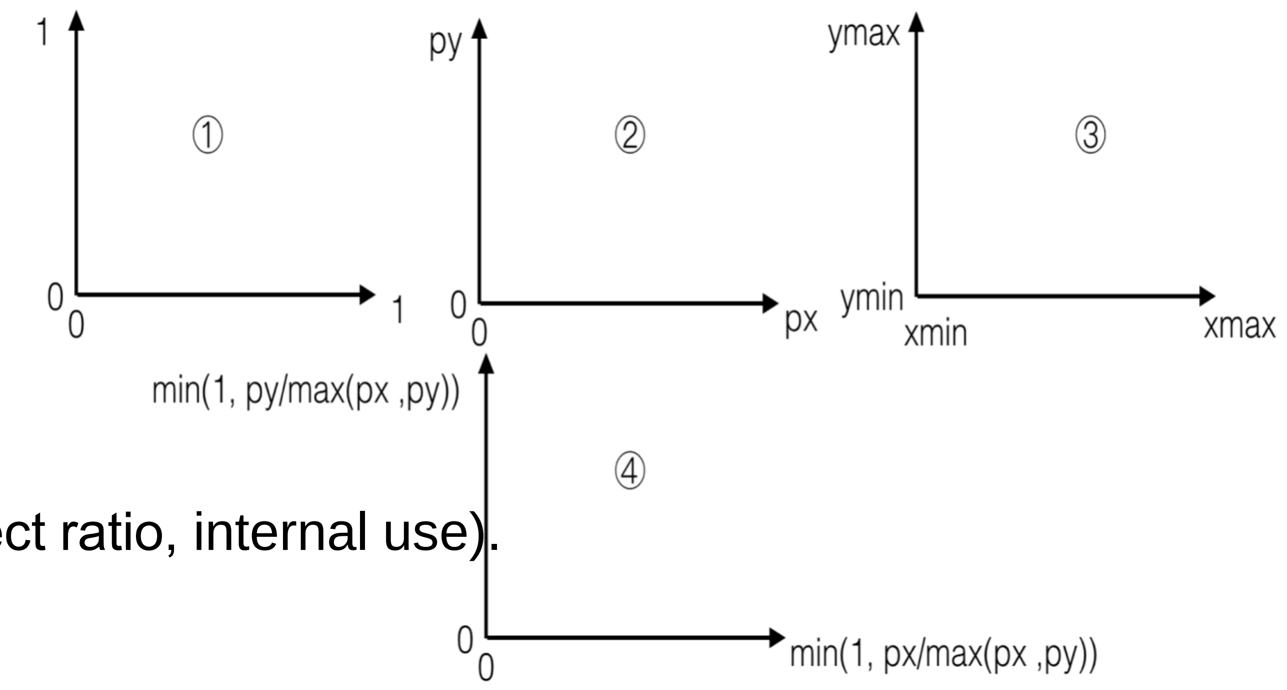
Nor more: **`something->Draw(option)`**
Instead: **`pad->Draw(something, options ...)`**

Invokes: **`GetDrawable(something, options...)`**

`options` specify the drawing options (and attributes) to be use to render **`something`**.

Possibility to share attributes between Drawables.

Coordinates systems



1. **Normalized Coordinates** (% of canvas size).
2. **Pixels Coordinates** (on the client).
3. **User Coordinates** (ie: histogram range)
4. **Normalized Device Coordinates** (keep objects' aspect ratio, internal use).

Using "user defined literal" one can specify coordinates like:

```
Px = 0.5_normal - 20_pixel + 3.14_user;
```

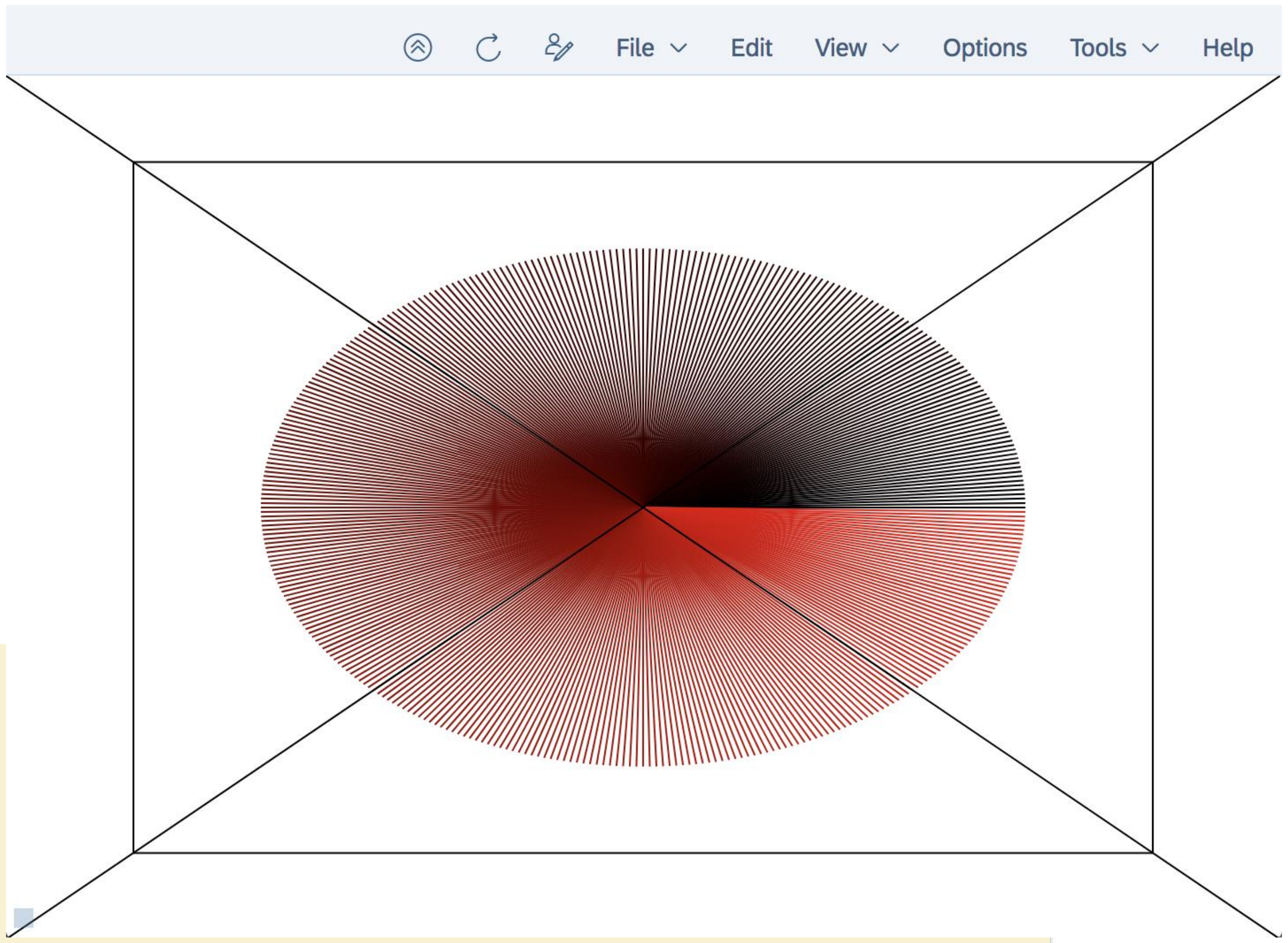
Code examples

The next slides show some **working ROOT 7 code examples**.

They illustrate :

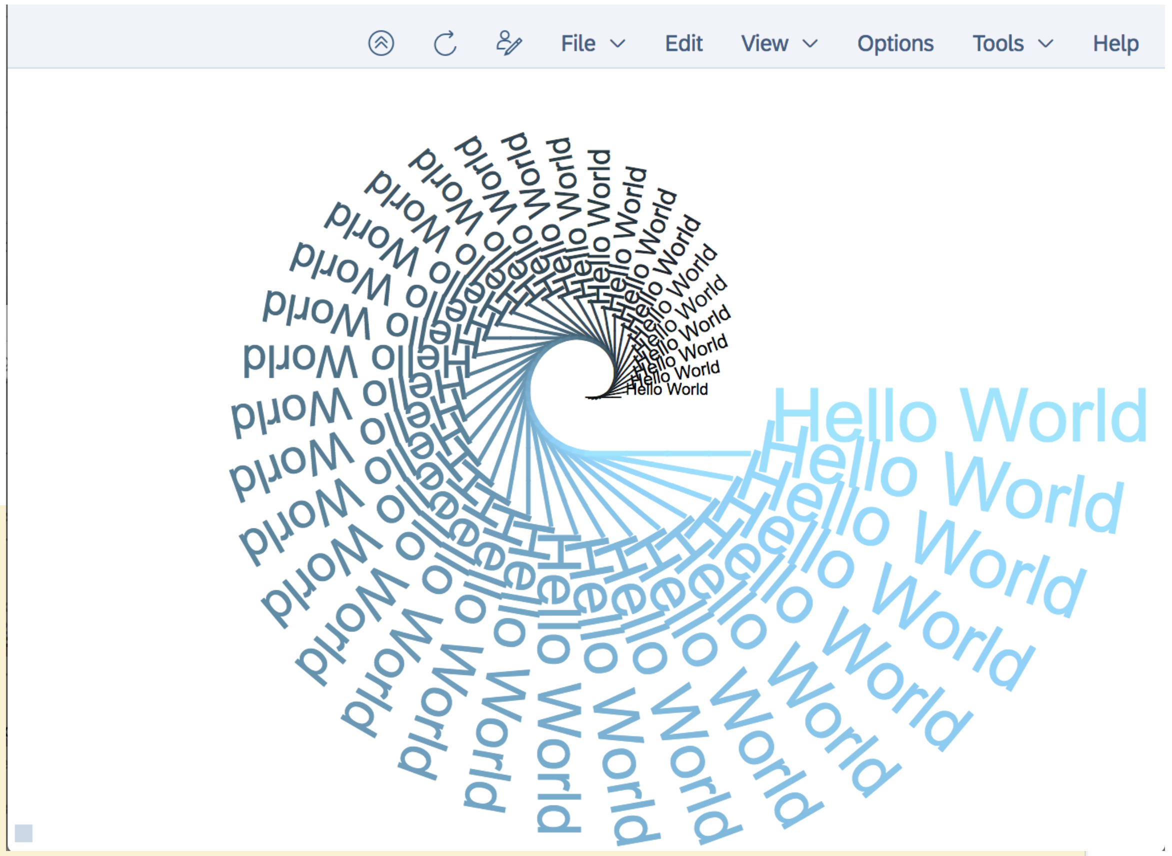
- Some of the new graphics classes implemented.
- The graphics coordinates.
- The graphics attributes.
- The canvas and pad usage.

Code example: line.cxx



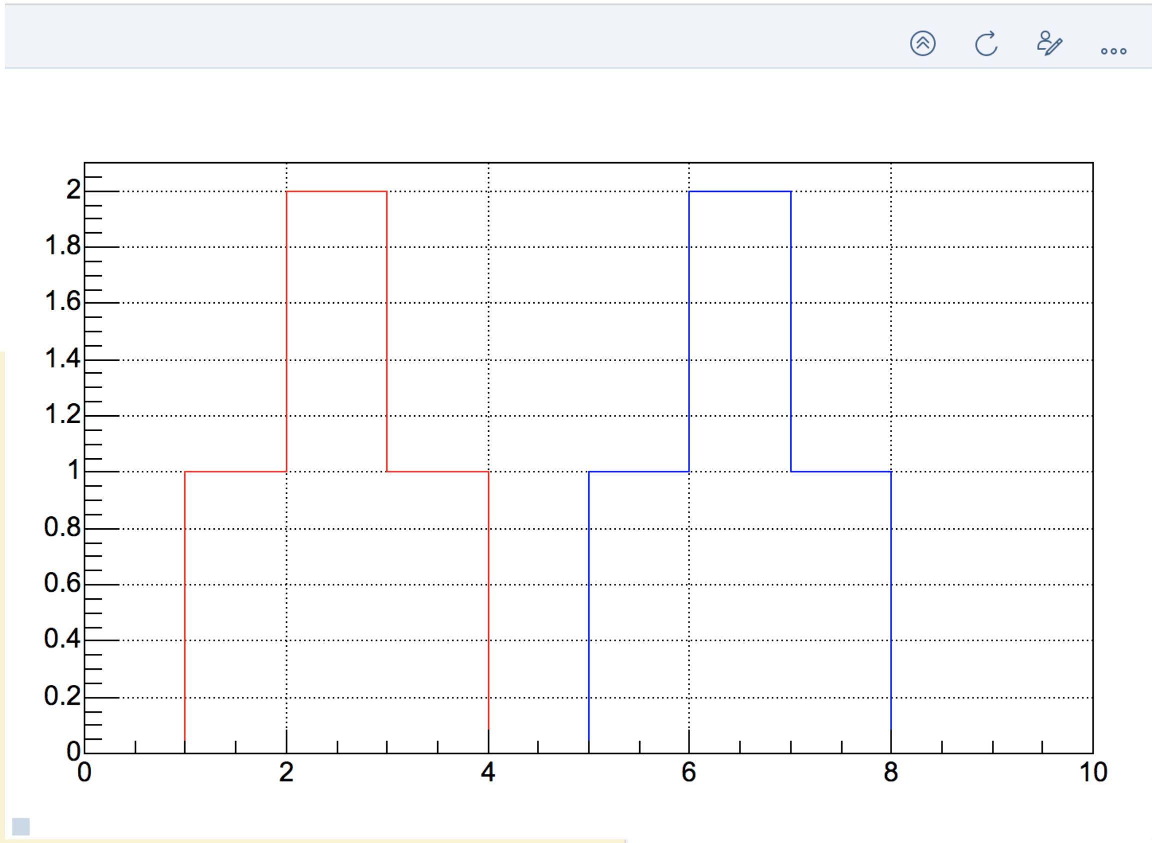
```
void line()
{
   // Create a canvas to be displayed.
   auto canvas = RCanvas::Create("Canvas Title");

   for (double i = 0; i < 360; i+=1) {
      double angle = i * TMath::Pi() / 180;
      RPadPos p(0.3_normal*TMath::Cos(angle) + 0.5_normal,
                0.3_normal*TMath::Sin(angle) + 0.5_normal);
      auto opts = canvas->Draw(RLine({0.5_normal, 0.5_normal} , p));
      RColor col(0.0025*i, 0, 0);
      opts->SetLineColor(col);
      opts->SetLineWidth(1);
    }

   canvas->Show();
}
```

Code example: text.cxx



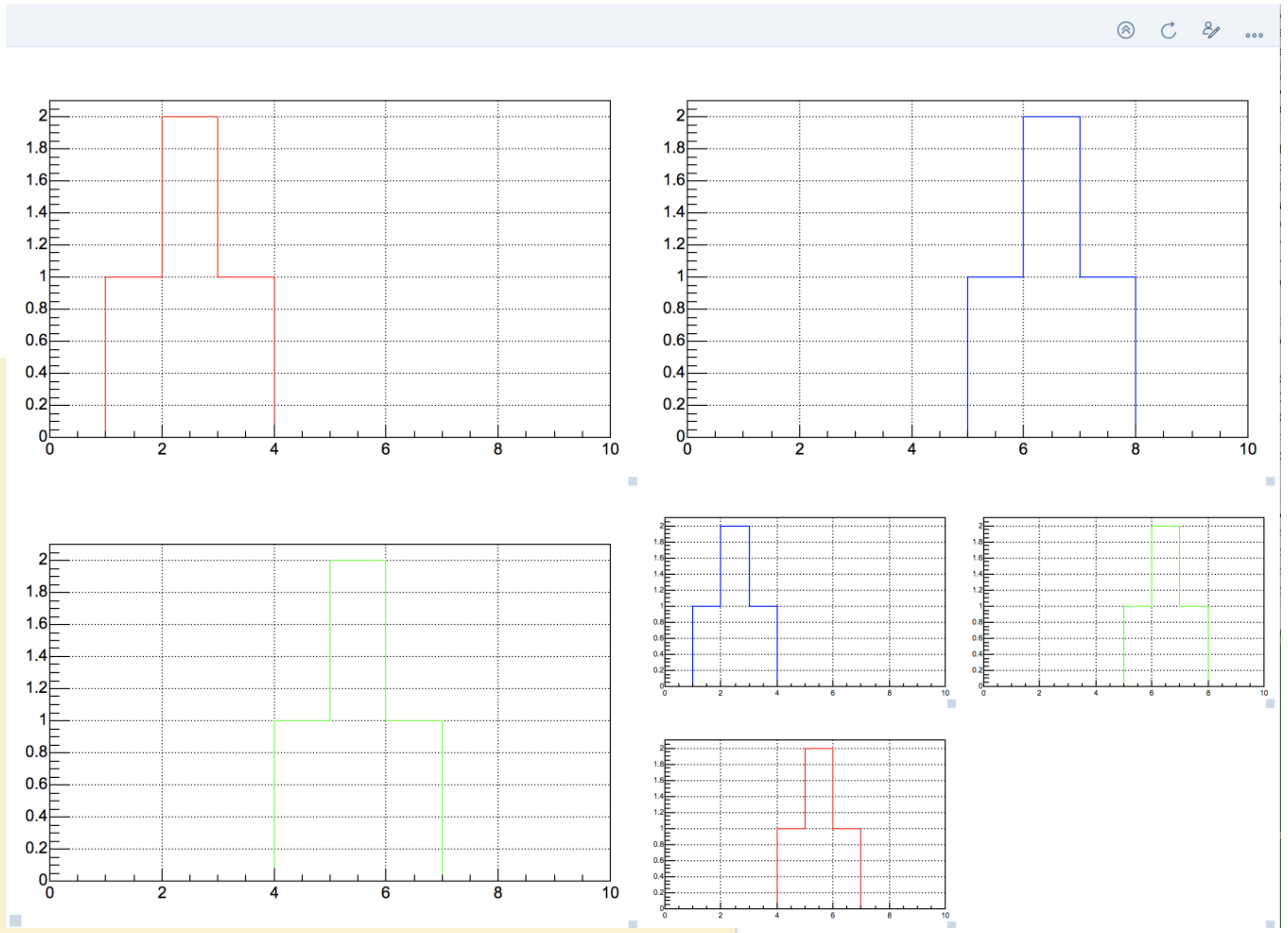
```
void text()
{
   // Create a canvas to be displayed.
   auto canvas = RCanvas::Create("Canvas Title");


   for (int i=0; i<=360; i+=10) {
      auto opts = canvas->Draw(RText({0.5_normal, 0.6_normal}, "____  Hello World"));


      RColor col(0.0015*i, 0.0025*i ,0.003*i);
      opts->SetTextColor(col);
      opts->SetTextSize(10+i/10);
      opts->SetTextAngle(i);
  }

   canvas->Show();
}
```

Code example: draw_rh1.cxx



```
void draw_rh1() {

   // Create the histogram.
   RAxisConfig xaxis(10, 0., 10.);
   auto pHist = std::make_shared<RH1D>(xaxis);
   auto pHist2 = std::make_shared<RH1D>(xaxis);

   . . .

   // Create a canvas to be displayed.
   auto canvas = RCanvas::Create("Canvas Title");
   canvas->Draw(pHist)->SetLineColor(RColor::kRed);
   canvas->Draw(pHist2)->SetLineColor(RColor::kBlue);

   canvas->Show();
}
```

Code example: draw_subpads.cxx



```
void draw_subpads() {


  // Create a canvas to be displayed.
  auto canvas = RCanvas::Create("Canvas Title");

  // Divide canvas on sub-pads
  auto subpads = canvas->Divide(2,2);

  subpads[0][0]->Draw(pHist1);
  subpads[1][0]->Draw(pHist2);
  subpads[0][1]->Draw(pHist3);

  // Divide pad on sub-sub-pads
  auto subsubpads = subpads[1][1]->Divide(2,2);

  subsubpads[0][0]->Draw(pHist1)->SetLineColor(RColor::kBlue);
  subsubpads[1][0]->Draw(pHist2)->SetLineColor(RColor::kGreen);
  subsubpads[0][1]->Draw(pHist3)->SetLineColor(RColor::kRed);

  canvas->Show();
}
```

Future work

- Complete the basic graphics «R» classes. For instance the text like `TLatex` (mathjax)
- Implement `RPolyLine`, `RPolyMarker`, `RFillArea`
- Implement higher level objects like `RPie`, `RGaxis`, `RArrow`, `RBox`, `REllipse`, `RLegend`, Feynman d iagrams primitives etc …
- Data containers like `RGraph`, `RMultiGraph`
- Painters on the JavaScript side for all the high level objects.
- Generation of batch graphics images. See if there is a solution for TeX
- Testing

Conclusion

- Graphics based on web technologies (SVG, HTML, WebGL).
- Client (C++) / Server (JavaScript) model (HTTP server).
- Batch output do not rely on any native ROOT library (headless mode).
- New graphics concepts (`canvas->Draw(something);`).
- Basic graphics classes have been implemented (`RCanvas`, `RPad`, `RDrawable`).