A Graphical Visualizer for Benchmark Data

Author: Francisco Pinto Supervisor: Andrzej Nowak

CERN openlab, September 2012

Contents

Abstract

The aim of this document is to describe an openlab Summer Student Program project consisting of building a generic graphical visualizer for benchmark data.

Introduction

CERN openlab, through the various collaborations it has with leading industry partners, often encounters the task of analyzing, testing and benchmarking bleeding edge hardware. Some of the benchmarks employed make use of toolkits used in real High Energy Physics applications, such as the Geant[1] suite, used in simulations of the passage of particles through matter. The goal of this project was to investigate Geant's interfaces and structure and, with the information gathered, build a tool capable of interpreting the output of a benchmark built with Geant and visualizing it in a visually attractive way.

The visualizer's job is to parse the output produced by the benchmark and then display visual output on a histogram. It also has the ability of displaying some related statistics (e.g. event count, frames per second, etc.).

The program is implemented in the Python[2] programming language, using pygame[3], a library for Python game development and PyOpenGL[4], OpenGL bindings for Python.

Provisions were made to make the resulting program easily configurable and modifiable:

- Aesthetic configurations (e.g. colors, margins, typefaces) can be made by editing a configuration file.
- Data related configuration (e.g. adding or removing statistical information from the sidebar) can be made by modifying the source code itself.

The program was built taking into account that it should have a minimal amount of dependencies, be cross-platform and be modular enough to be extensible to other benchmarks. Since maintainers are likely to have little or no OpenGL knowledge, the code should be well documented and contain some examples on how to extend it to do common tasks (e.g. render a 3D scene, capture user input, etc.).

[1] https://geant4.web.cern.ch/
[2] http://www.python.org/
[3] http://www.pygame.org/
[4] http://pyopengl.sourceforge.net/

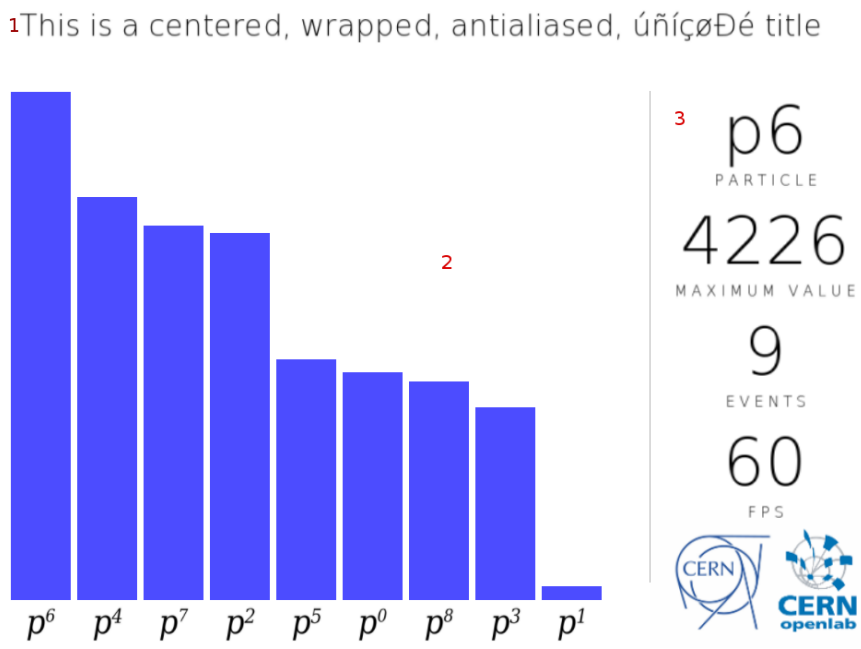

Figure 1: The visualizer running. Title (1), Display area, showing the histogram (2), Sidebar, displaying some stats and the logo (3)

Implementation

Visualizer

A class named `Histogram` (`Histogram.py`) concerns itself with windowing and displaying data. The data collection is made by a single function located in `main.py`.

The data collection and display code are modular and separate. The flow of information is roughly as follows:

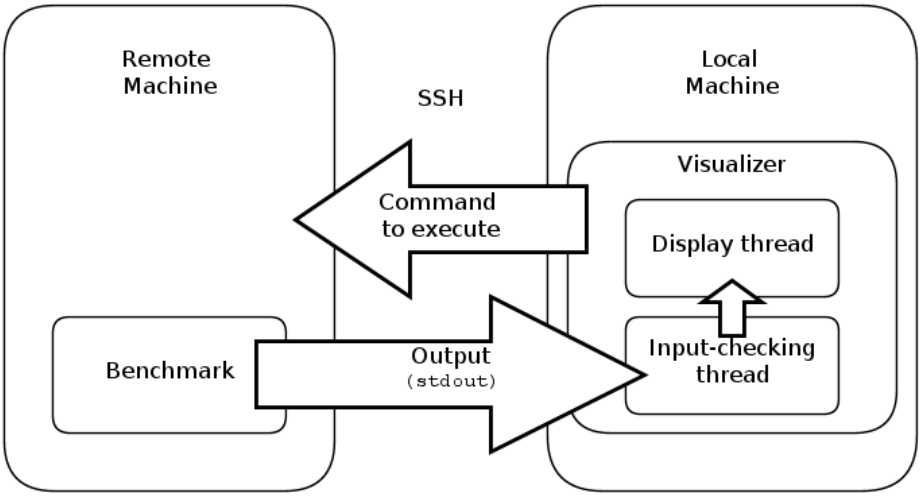

Figure 2: The architecture of the program

1. A connection is established from the local to the remote machine over SSH.
2. A command is sent over the wire for the remote machine to execute. The benchmark starts running.
3. The benchmark outputs some data.
4. A thread checking for input collects, interprets and stores it.
5. After a time-interval elapses, all samples collected in that time interval are sent to the `Histogram` instance.
6. The `Histogram` instance, which checks every frame for updates to the dataset, receives the updated data and displays it accordingly.

Initially, a prototype of the visualizer was implemented in C++, using GLFW[5] for windowing. However, Unicode support proved to be rather quirky and

[5] http://www.glfw.org/

inconsistent across platforms. This and general portability concerns led to a Python port of the existing codebase. This decision made some tasks much easier (e.g. the configuration file parsing was done using a Python standard library module) as well as generally increasing productivity due to the greater expressiveness of Python and faster development cycle as compared to C++. The switch to Python initially caused some performance concerns due to the overhead introduced by the Python runtime. After testing, however, the visualizer's performance was proved to be more than adequate for the task at hand.

Structure

What follows is an enumeration of the most relevant functions and methods, together with a short explanation of their relevance to the overall structure of the program. The functions/methods are organized by the file they are in. Refer to the source code's documentation for additional clarifications.

`main.py`

`update_histogram_real_data(histogram)` This function:

- Connects via `ssh` to a remote machine;
- Executes a command remotely;
- Receives and parses output from that command;
- Sends data collected to the `Histogram` instance passed as an argument.

`update_histogram_fake_data(histogram)` This function is similar to the above, except it generates random data to to update the histogram with. Useful for testing.

`Histogram.py`

`run` This is the display loop, where the graphics context is, at first, set up. The elements composing the visualizer are then periodically rendered and any user input handled.

`setup_*` Methods prefixed by `setup_` generally execute simple operations that are rarely (if ever) repeated. `setup_ortho`, for example, prepares the graphics environment for 2D drawing.

`render_*` Methods prefixed by `render_` do the actual drawing of geometry to the window.

`load_*` Methods prefixed by `load_` concern themselves with filesystem I/O operations. Currently, this only means texture loading.

`calculate_*` These methods calculate the position of a given element inside the viewport. These are generally called once at startup and again when the viewport is resized.

`handle_*` These methods process events, such as user input (`handle_user_actions`) or a window resize (`handle_resize`).

`build_*` Methods prefixed by `build_` generally process data to make it usable in the visualization.

An important case is `build_stats`. This is the method that should be modified in order to change which data appears on the sidebar. `build_stats` collects the values to be shown on the sidebar in a list. These values are then rendered as text to a texture. The textures are only rebuilt if the values change.

Additional features

There were a few features that were added to serve mostly as examples for future development.

Image monitoring The visualizer can monitor an image for changes and render it. It operates by reloading the image on a periodic interval. Both the image's path and the duration of the interval can be set using the configuration file (see the appendix for further info).

3D scene rendering An interactive 3D scene can be rendered. At the moment, the scene is a simple flat-shaded sphere. It can be zoomed, panned and rotated by using the mouse. The mouse input handling itself serves as an example of handling mouse input such as dragging, button presses/releases, etc.

Speedometer A simple speedometer was added. It can represent a single value on a dial (currently, the maximum value on the histogram). Configuration should be done using the configuration file (see the appendix for further info).

Benchmark

The benchmark used was a parallelized version of Geant's `test40`, commonly refered to as `test40p`. This parallelization was achieved thanks to the work Gene Cooperman and Xin Dong (Northeastern University) did in collaboration with CERN openlab [1].

It is important to note that `test40` is a toy benchmark; One of the final objectives of this work is to visualize CMS data.

Modifications

`test40` was slightly modified to print relevant data after Geant's verbosity switches proved insufficient:

- `/run/verbose` produced nothing of interest
- `/event/verbose` produced nothing of interest
- `/tracking/verbose` produced track information. An histogram of the particles observed can be produced by looking at track data. However, the amount of data produced was far too large (around 400MB for 400 events) for this to be a viable solution.

The `PrimaryGeneratorAction::GeneratePrimaries` method was modified to allow operation in 3 different modes, which differ in the method of choice of particle to be fired from the particle gun:

- `RANDOM_FROM_TABLE` - a particle from Geant's `G4ParticleTable` is pseudo-randomly chosen.
- `RANDOM_FROM_LIST` - a particle from a predefined list of particles is pseudo-randomly chosen. The list contains only particles which make for a CPU intensive benchmark.
- `ELECTRON` - only electrons. This is the original, pre-modification, operating mode.

The choice is governed by a `#define` macro located in `PrimaryGeneratorAction.cc`, for example:

```
#define CHOICE RANDOM_FROM_LIST
```

This method is called every time an event occurs. Output printed is the name of the chosen particle in the following format:

```
Name:  e-
```

In this case, a single electron was fired from the particle gun.

Note that other benchmarks can be made compatible with the visualizer by applying the changes described above.

Performance and scalability

Modifying the benchmark to print data to `stdout` raises some performance concerns:

- Printing is a relatively expensive operation. Since printing isn't that frequent, this is unlikely to be a problem.
- In a parallel environment, having multiple writers to `stdout` may cause writers to wait for the lock's release. Since every thread only prints every few tens of milliseconds, printing is unlikely to be a bottleneck unless the number of threads is significantly high.

The maximum number of threads the benchmark was tested with was 16 on a dual Xeon L5520 workstation (total of 8 cores, 2 threads per core). Using this machine, the benchmark was run 10 times on mode `ELECTRON`, with an input file specifying 400 events. The time of experiments was then measured with output enabled and then disabled.

Output enabled	Output disabled
15.647 sec.	15.697 sec.

Table 1: Averaged time of experiments over 10 runs

From these figures, we can conclude that for this specific setup, the overhead introduced by having the program output to `stdout` is negligible.

Running the program

The visualizer's dependencies are:

- Python 2.6
- pygame 1.7.1
- PyOpenGL 3.0.1-1

The version numbers represent the earliest version the program was tested with – it might run on earlier versions or break with newer ones.

The program can be run by running `main.py`. Do note that before running the program, you should configure section `Connection` of the config file. See the appendix for instructions on how to do so.

Conclusion

The result of this work is a cross-platform visualizer for benchmark data. Python proved to be an adequate implementation language for simple scientific visualization when coupled with pygame and PyOpenGL. Even though pygame is meant for videogame development, the library was very useful and easy to develop with, since the two fields heavily rely on computer graphics and share characteristics.

Future development of the visualizer is likely to benefit from a switch to an object oriented paradigm. Even though the current codebase was built with modifiability in mind, to have the visualizer's elements as objects would be useful if future users would like to have, for example, two histograms. Adding new kinds of elements would also be cleaner.

From what was learned about Geant4, it is safe to conclude that it is not ready for this sort of visualization: Geant4's output tends to be utilitarian. Indeed, the manual mentions this limitation [2]. It might have been possible to do this kind of work using only Geant4-provided tools, but it would likely not be straightforwards. On top of this, using parallel benchmarks such as `test40p` would have been much harder, since Geant4 has no support for multithreaded graphics.

Appendix

Configuration

A file named `config.cfg` is included with the visualizer. This file follows the INI informal standard. An INI file contains *properties*, organized into *sections*.

A property looks like this:

```
name = value
```

A section declaration looks like this:

```
[ExampleSection]
```

The following is an overview of the available properties organized by their respective section.

Section `Connection`

```
COMMAND = cd ~/test40/ && ./test40 test40.in.1 2
```

This specifies command that shall be executed upon connection to the remote machine.

```
HOST = example.com
```

The remote host's name.

```
USERNAME = someusername
```

The username that will be used for login on the remote machine.

```
PASSWORD = somepassword
```

The password respective to the username above.

Section `Histogram`

```
WINDOW_WIDTH = 1000
WINDOW_HEIGHT = 750
```

These two properties govern the initial window size in pixels.

```
FRAMES_PER_SECOND = 60
```

The maximum frames per second the visualizer will render. This is an approximation; for a value of 60 the actual number of frames rendered per is around 61 or 62.

```
START_FULLSCREEN = 0
```

Boolean switch to make the visualizer start in fullscreen mode. 1 means "true", 0 means "false".

```
HISTOGRAM_VISIBLE = 1
TARGET_IMAGE_VISIBLE = 0
SPEEDOMETER_VISIBLE = 0
SPHERE_VISIBLE = 0
```

Boolean switches that govern which elements are visible when the visualizer starts.

```
FONTS_DIR = fonts/
LABELS_DIR = labels/
LOGOS_DIR = logos/
```

Paths for the directories containing fonts, labels for the histogram and the logo. Can be absolute or relative.

```
TEXTURE_EXTENSION = .png
```

The extension for all textures (labels, logos).

```
LOGO_PATH = cernplusopenlab
```

The filename for the logo (minus the extension), which should be placed inside the directory specified by `LOGOS_DIR`.

```
LOGO_HEIGHT_PERCENT = 60
```

The ratio, in percentage, of the logo's height to the logo's width.

```
SIDE_MARGIN_PERCENT = 5
BOTTOM_MARGIN_PERCENT = 7
TOP_MARGIN_PERCENT = 2
```

The margins for the histogram expressed as a percentage.

```
DIVIDER_LOCATION_PERCENT = 75
```

The divider's position, expressed as a percentage of the window width, counting from the left.

```
CAPTION_MARGIN = 3
STAT_MARGIN = 120
```

The margins, in pixels, for the captions and values in the sidebar. `STAT_MARGIN` is the distance, in pixels, between a caption/value pair and `CAPTION_MARGIN` is the distance, in pixels, from the caption to the value it refers to.

```
STATS_TOP_MARGIN_PERCENT = 25
```

The distance, expressed as a percentage of the window's height, from the top of the viewport to the first caption/value pair in the sidebar.

```
TITLE_MARGIN_PERCENT = 7
```

The padding around the title expressed as a percentage of the window's height.

```
BAR_WIDTH_PERCENT = 90
```

The margin around each bar on the histogram. Taking 90% as an example value, the bar occupies 90% of the total width possible for a bar (i.e. it has 5% margin on each side). Values greater than 100% are possible and would make the bars overlap.

```
ANIMATION_DURATION = 0.5
```

The duration, in seconds, for the animation.

```
BAR_COLOR_RED = 0.3
BAR_COLOR_GREEN = 0.3
BAR_COLOR_BLUE = 1
BAR_COLOR_ALPHA = 1

BACKGROUND_COLOR_RED = 1
BACKGROUND_COLOR_GREEN = 1
BACKGROUND_COLOR_BLUE = 1
BACKGROUND_COLOR_ALPHA = 1

DIVIDER_COLOR_RED = 0
DIVIDER_COLOR_GREEN = 0
DIVIDER_COLOR_BLUE = 0
DIVIDER_COLOR_ALPHA = 0.3

CAPTION_FONT_COLOR_RED = 0
CAPTION_FONT_COLOR_GREEN = 0
CAPTION_FONT_COLOR_BLUE = 0
CAPTION_FONT_COLOR_ALPHA = 1

VALUE_FONT_COLOR_RED = 0
VALUE_FONT_COLOR_GREEN = 0
VALUE_FONT_COLOR_BLUE = 0
VALUE_FONT_COLOR_ALPHA = 1

TITLE_COLOR_RED = 0
```

```
TITLE_COLOR_GREEN = 0
TITLE_COLOR_BLUE = 0
TITLE_COLOR_ALPHA = 1
```

Colors for, respectively:

- Bars in the histogram;
- Background;
- Divider (i.e. the bar between the histogram and sidebar);
- Captions (the text under each value in the sidebar);
- Values (the values in the sidebar);
- Title.

Every value should be in the [0, 1] interval.

```
WINDOW_TITLE = Histogramming!
```

The window's title.

```
TITLE_TEXT = This is a centered, wrapped, antialiased, úñíçøÐé title
```

The title that should appear at the top of the viewport.

```
TITLE_FONT_PATH = DejaVuSans-ExtraLight.ttf
TITLE_FONT_SIZE = 32
VALUE_FONT_PATH = DejaVuSans-ExtraLight.ttf
VALUE_FONT_SIZE = 72
CAPTION_FONT_PATH = DejaVuSans-ExtraLight.ttf
CAPTION_FONT_SIZE = 16
```

Paths for the typefaces for the title, values and captions, respectively. These should be placed in the directory specified as `FONTS_DIR`.

```
TARGET_IMAGE_PATH = test_images/target.png
TARGET_IMAGE_CHECKING_INTERVAL = 3 ; Seconds
```

These two properties are related to the monitored, or "target", image. You must specify which image you want to monitor (`PATH`) and the maximum interval between checks (`CHECKING_INTERVAL`).

```
SPEEDOMETER_DIAL_PATH = speedometer/speedometer.png
SPEEDOMETER_NEEDLE_PATH = speedometer/needle.png
```

These two properties specify the path to the images that shall be used to represent the speedometer's two components: the dial and the needle.

```
SPEEDOMETER_OFFSET = 20 ; px
```

The vertical offset to the speedometer's axis.

```
NEEDLE_OFFSET = -15 ; px
```

The vertical offset to the needle's axis.

```
SPEEDOMETER_START_ANGLE = 270
SPEEDOMETER_END_ANGLE = 90
SPEEDOMETER_MAX_VALUE = 5000
```

These three properties are related to the way the needle moves. The origin is 12 o'clock and the angles are measured clockwise. In this case, the needle starts at 270 degrees, ends at 90 degrees (moving clockwise, as previously noted) and spans a measurement interval of 0 to 5000 units.

References

[1] X. Dong and G. Cooperman, "Thread Parallelism for Geant4," *13th Geant4 Collaboration Kobe Workshop*, 2008.

[2] Geant4Collaboration, "Geant4 User's Guide for Application Developers, Appendix 2: Histogramming," 2012.