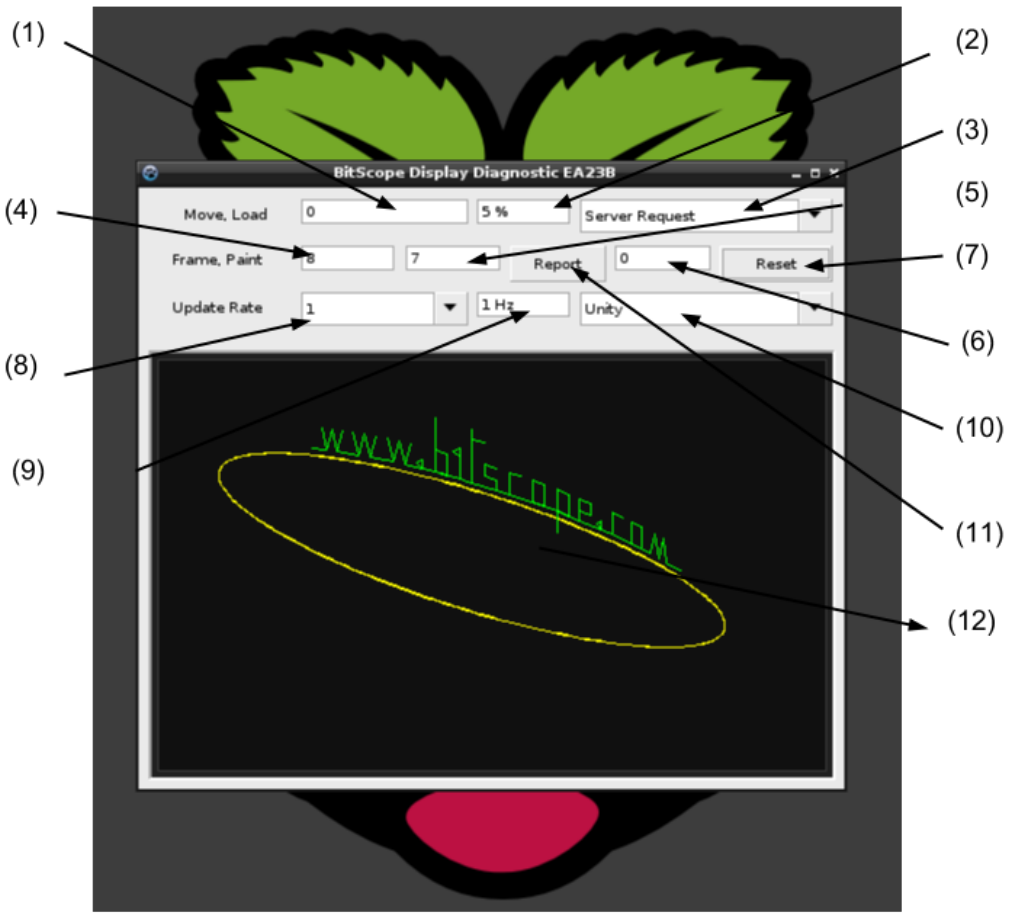
BitScope Display Diagnostic User Guide

BitScope Display is a cross-platform tool to allow the display performance of different systems to be analyzed and compared for real-time interactive graphical user interface applications like BitScope DSO.



User Interface and Waveform Display (12)

1. **Mouse Events**. Counts and reports mouse move events as they are received.
2. **Application Load**. Reports application load as a percentage of available CPU.
3. **Update Mode**. Selects the application display update mode (server or client).
4. **Frame Count**. Counts display frame requests as they occur.
5. **Paint Events**. Counts paint events that result from frame requests.
6. **RePaint Events**. Counts paint events that drive full window redraws.
7. **Counters Reset**. Resets all the event counters.
8. **Refresh Rate** (Requested). The frame rate requested.
9. **Refresh Rate** (Realized). The frame rate actually realized.
10. **Display Mode**. Chooses the type of test waveform for calculate.
11. **Fast Report**. Update information widgets at the frame rate (not 1Hz).
12. **Display Mode**. Selects which waveform processing is enabled and used.

Operation Summary

Simply execute the program binary from the command line in an X terminal or run the app from its icon.

When first started it synthesizes and displays a simple lissajous waveform updating at 1 frame per second.

The application load should be <5% on almost any system including Raspberry Pi when idle. When moving the mouse, resizing the application window or otherwise interacting with the application, the load will vary and mouse and repaint events reported. This allows the CPU cost of these user operations to be observed.

If the frame rate is increased a point will be reached beyond which the actual refresh rate can no longer keep up. The load may reach 100%. This gives an idea of the maximum display refresh rate the host system can support for this type of graphical display application at the selected window size.

Used with tools such as top or task manager it provides insight into display performance bottlenecks.

BitScope Display is a cross-platform application built with FPC/Lazarus. It is derived from a single code base which means the performance of different systems can be compared subject to FPC compiler code generation and underlying graphic framework differences. It's not a formal benchmark tool but it does provide some practical insight into overall display performance from a graphical application perspective.

Update Mode

The Update Mode selects how the waveform display is updated:

1. **Server Request**. The client invalidates the waveform display region upon the frame clock and waits for the server to issue an event to repaint it. This mechanism is the required only supported way to update on some systems (e.g. OSX) but it's optional on others (Windows and Linux). The server is at liberty to choose when to issue the repaint which it may delay if it has other things to do first.

2. **Server Update**: The client requests the server update the waveform display region now i.e. without any discretion by the server as to when the update occurs other than system (over)load.

3. **Client Request**. The client issues a repaint directly upon the frame clock as soon as the client becomes idle. If the system is overloaded (i.e. never idle) the client may never get around to repainting the display in which case the waveform update may pause until the system load drops but client handling of user events such as mouse moves and keyboard events will never be interrupted.

4. **Client Forced**. The same as client requested but repaints are issued regardless of client or server load. In this case waveform display refresh may be limited but it will not stop even if the client or server become overloaded. However, the response to new mouse and keyboard events may be interrupted. If the overload does not cease the user may lose control of the application in this mode. On the other hand this mode should produce the highest possible frame rate for a given system.

5. **No Refresh**. Requests to update the waveform display are disabled. All other processing continues. This mechanism allows the load of the application from all non-display operations to be measured.

Different platforms behave differently in these modes. For example, Windows systems do not allow frame refresh to occur at a rate higher than the monitor frame rate. Mac OSX may not work at all unless Server Requested refresh is used. Some (presumably buggy) X server implementations may issue multiple refresh events upon receiving an Invalidate request (depending on the client framework) which can result in redundant repaints and unnecessary CPU load. Some others (notably Raspberry Pi) issue repaint requests at an exceptionally high rate (relative to graphics capabilities) when the mouse cursor moves across the window. Try different modes to see which work best to ameliorate these effects on a given platform.

Display Mode

The Display Mode selects which lissajous waveform to generate.

Different lissajous exhibit characteristics of different types of waveform display.

The last three modes allow the selection of processing, drawing or repainting only. This helps evaluate which of these operations are the most costly on a given platform. To diagnose these problems use the application modes:

1. **Unity** to **Rollercoaster**. Lissajous waveform choices.
2. **Process Only**. Process waveforms but do not draw or paint them.
3. **Redraw Only**. Draw waveforms but do not reprocess or repaint them.
4. **Repaint Only**. Paint the display but do not reprocess or redraw the waveforms.

Setting a frame rate of zero stops frame updates altogether in which case the application should impose negligible load on the system. However, on some systems moving the mouse can produce a large system load and on Raspberry Pi when the frame clock is enabled it also locks out display updates completely unless the Update Mode is Client Forced.

Fast Report

This button controls how the report widgets are updated. By default (when off) they update at 1Hz (i.e. a slow rate). When selected they update at the frame rate.

We have found that widget updates in most GUI frameworks are “intelligent” in the sense that they update only their region of the window so the bandwidth required to transfer new data is very low. However, these updates are very expensive if performed at a high rate because in most X servers the cost of the transaction itself is very high (we presume in terms of the context switching and server CPU/GPU locking required).

This button allows you to see just how high this cost is on a given platform and frame rate. The modern paradigm in newer server designs is to optimize for data throughput and not worry too much about transaction overheads. The original X designs which focused on minimizing network traffic for remote connections had a different design objective; minimize throughput with smaller frequent transactions.

Downloads

This application may be downloaded for all supported platforms from **here**.

On Linux platforms the framework dependency is GTK. On Windows it uses the Windows' GDI and on Mac OSX, Carbon/Cocoa. On a modern Linux PC frame rates up to 1kHz for a small window can be achieved. The rate drops if the window size is increased and depends on acceleration, the X server driver and X server or graphics implementation.

Conclusion

Running this application on any supported platform should provide an idea of how similar interactive graphical application that uses the standard X server via GTK (in Linux), the GDI (Windows) or Cocoa framework (Mac OS X) can be expected to perform. In particular it is hoped this tool might assist in debugging and tuning the performance of the X server used in the Raspberry Pi Raspbian distribution and other lower powered embedded ARM or older x86 systems.

EA23B EC17A EK7A9 eca4200a-c8c6-4a91-90bb-d2ed9800f11c **JV6RP**