Graphviz plugins

Plugins provide a means of flexibly extending the graphviz tools: **dot**, **neato**, etc.

The objectives for graphviz plugins are to:
- Support independent compilation so that features can be provided by third-parties.
- Support independent packaging to that features can be installed at the user's discretion, at a later time than the base package.
- Support optional platform-specific functionality without adding complexity or dependencies to the base package (e.g. renderers for a specific windowing system).
- Unbundle some lesser-used functionality from dot into plugins **s**o that the core of dot can be smaller and faster.
- Unbundle some older functionality from dot into plugins so that the facilities are still available, but optionally and not in the core. For example the use of libgd.

Some constraints considered in there design were:
- To minimize any performance penalty from the use of plugins.
- To make the use of plugins transparent to the user and sys admin.
- To permit an option to link plugins statically at build time.
- To permit an option to always load plugins at run-time, not waiting for demand.
- To continue to support the old-style codegens, at least until completely replaced by plugins.

Terminology

A **plugin** (or **plugin_type**) provides a facility like a PNG renderer or an Xlib window driver, Each **plugin** implements a **plugin-api**.

A **plugin-api** is the exposed interface between the graphviz core and the **plugin**.
The set of apis is described in *lib/gvc/gvplugin.h* and each api is individually described in, e.g. *lib/gvc/gvplugin_render.h*. The set of apis currently consists of:

device:	output devices and file formats, e.g png, svg, gtk
render:	drawing engines, e.g. cairo
layout:	layout engines such as: dot, neato, circo, ...
textlayout:	code that estimates the size of a block of text. Uses pango if available.
loadimage:	code that loads an image in one format and makes it available in another format.

A **plugin-library** is an arbitrary collection of **plugins**, for one or more **plugin-apis**, that are loaded all together when any one of the contained **plugins** are demanded. The contents of a **plugin-library** are typically determined by other resources that are common to the set of **plugins**. For example, all the code that uses *libgd.so* is in the “gd” plugin.

A **demand-loaded plugin** is one that is not loaded until its need for use on the has been determined from the current usage.

A **builtin-plugin** is one that is available at program start, whether or not it will be needed for the current usage. **Builtin-plugins** can still be in the form of shared libraries, or they can be statically linked

A **statically-linked builtin-plugin** is one that is linked to the binary image at build time, rather than from a shared library at run-time.

Mechanism

The contents of a **plugin-library** are self-describing. The **plugin-libraries** are named like: *libgvplugin_core.so* where the "*core*" of the library name matches the name of the single exposed symbol: *gvplugin_core_LTX_library*. This symbol is a pointer to a table of **plugin-apis** supported in this **plugin-library**, and for each **plugin-api** the list of **plugins** for that api. Each **plugin** provides a "quality" value which is used to control the default selection when there are multiple **plugins** of the same type loaded.

At installation time (with the file write privileges of the installer) "**dot -c**" is run to search for and verify loadability of all available **plugin-libraries**, and then to generate */usr/lib/graphviz/config* which records their capabilities. This is a human-readable text file. When **dot** is run by a regular user to process a graph, only this */usr/lib/graphviz/config* file is loaded at first so that the set of available **plugins** and their capabilities is known and the **plugin** can be loaded if needed.

When a **plugin** is demanded, the **plugin-library** containing that **plugin** is loaded in its entirety and all of the **plugins** it provides are recorded as loaded. One plugin may be dependent on another, if so the dependency is loaded as well whenever the dependent is loaded. In particular, device plugins are always dependent on a renderer.

Running dot with plugins

Dot attempts to be transparent to the user about its use of **plugins**. By default a plugin selection like "`dot -Tpng`" will always choose the highest "quality" **plugin** listed in */usr/lib/graphviz/config*.

Sometimes it may be necessary to override the default. This can be done for devices by indicating the specific renderer required: e.g.

```
-Tpng:cairo
-Tpng:gd
```

or further qualified by which plugin-library provides thedevice plugin: e.g.

```
-Tpng:cairo:cairo
-Tpng:cairo:gd
```

To allow the user to determine what renderers are available, the user can ask for any invalid plugin and dot will return the set available **plugins**:

```
$ dot -T?
Format: "?" not recognized. Use one of: bmp canon cmap cmapx cmapx_np dia dot
fig gd gd2 gif gtk hpgl ico imap imap_np ismap jpe jpeg jpg mif mp pcl pdf
pic plain plain-ext png ps ps2 svg svgz swf tga tif tiff vml vmlz vrml vtx
wbmp xdot xlib
```

Or, if the **plugin-library** is not recognized then dot will return the set of alternatives:

```
$ dot -Tpng:?
Format: "png:?" not recognized. Use one of: png:cairo:cairo png:cairo:gd
```

```
png:cairo:gdk_pixbuf png:gd:gd png:cairo:devil
```

The order of the alternatives corresponds to their quality ranking with the first listed as the default, so `-Tpng` is equivalent to `-Tpng:cairo:cairo`

For additional information, **dot -v** will identify all available **plugins** for all **plugin-apis**:

```
$ dot -v
The plugin configuration file:
        /home/ellson/FIX/Linux.x86_64/lib/graphviz/config
                was successfully loaded.
    render      :  cairo dot fig gd map ming ps svg vml vrml xdot
    layout      :  circo dot fdp neato nop nop1 nop2 twopi
    textlayout  :  textlayout
    device      :  bmp canon cmap cmapx cmapx_np dia dot fig gd gd2 gif gtk
hpgl ico imap imap_np ismap jpe jpeg jpg mif mp pcl pdf pic plain plain-ext
png ps ps2 svg svgz swf tga tif tiff vml vmlz vrml vtx wbmp xdot xlib
    loadimage   :  (lib) gd gd2 gif jpe jpeg jpg png ps xbm
```

and, **dot -v2** will identify all available plugin *type:dependency:library* for all **plugin-apis**:

```
$ dot -v2
The plugin configuration file:
        /home/ellson/FIX/Linux.x86_64/lib/graphviz/config
                was successfully loaded.
    render      :  cairo:cairo dot:core fig:core gd:gd map:core ming:ming
ps:core svg:core vml:core
vrml:gd xdot:core
    layout      :  circo:neato_layout dot:dot_layout fdp:neato_layout
neato:neato_layout
nop:neato_layout nop1:neato_layout nop2:neato_layout twopi:neato_layout
    textlayout  :  textlayout:cairo textlayout:gd
    device      :  bmp:cairo:gdk_pixbuf bmp:cairo:devil canon:dot:core
cmap:map:core cmapx:map:core cmapx_np:map:core dia:cg dot:dot:core
fig:fig:core gd:cairo:gd gd:gd:gd gd2:cairo:gd gd2:gd:gd gif:cairo:gd
gif:gd:gd gtk:cairo:gtk hpgl:cg ico:cairo:gdk_pixbuf imap:map:core
imap_np:map:core ismap:map:core jpe:cairo:gdk_pixbuf jpe:cairo:gd jpe:gd:gd
jpe:cairo:devil jpeg:cairo:gdk_pixbuf jpeg:cairo:gd jpeg:gd:gd
jpeg:cairo:devil jpg:cairo:gdk_pixbuf jpg:cairo:gd jpg:gd:gd jpg:cairo:devil
mif:cg mp:cg pcl:cg pdf:cairo:cairo pic:cg plain:dot:core plain-ext:dot:core
png:cairo:cairo png:cairo:gd png:cairo:gdk_pixbuf png:gd:gd png:cairo:devil
ps:ps:core ps:cairo:cairo ps2:ps:core svg:svg:core svg:cairo:cairo
svgz:svg:core swf:ming:ming tga:cairo:devil tif:cairo:gdk_pixbuf
tif:cairo:devil tiff:cairo:gdk_pixbuf tiff:cairo:devil vml:vml:core
vmlz:vml:core vrml:vrml:gd vtx:cg wbmp:cairo:gd wbmp:gd:gd xdot:xdot:core
xlib:cairo:xlib
    loadimage   :  (lib):ps:core gd:ps:gd gd:gd:gd gd2:ps:gd gd2:gd:gd
gif:ps:gd gif:gd:gd gif:xdot:core gif:vrml:core gif:fig:core gif:svg:core
jpe:ps:gd jpe:gd:gd jpe:xdot:core jpe:vrml:core jpe:fig:core jpe:svg:core
jpeg:ps:gd jpeg:gd:gd jpeg:xdot:core jpeg:vrml:core jpeg:fig:core
jpeg:svg:core jpg:ps:gd jpg:gd:gd jpg:xdot:core jpg:vrml:core jpg:fig:core
jpg:svg:core png:ps:cairo png:ps:gd png:gd:gd png:xdot:core png:vrml:core
png:fig:core png:svg:core png:cairo:cairo ps:ps:core xbm:ps:gd xbm:gd:gd
```

Installing plugins

On package managed systems using .rpm or .deb or similar, the details of the installation process are handled automatically.

The key point about plugin-library installation or removal is that the */usr/lib/graphviz/config* file must be kept up to date by using "**dot -c**".

Builtins and static linking

A **plugin** is **builtin** if it is linked to the executable at startup.

Dot can be configured and built with an arbitrary combination of **builtin** and **demand loaded plugins,** however p**lugins** that are to be **builtin** must be explicitly listed and linked at build time.
This is done by providing a source file like *lib/gvc/dot_builtins.c* that is linked to the executable, and adding the matching -lgvplugin_xxx to satisfy the symbol reference.

To control whether demand loading is available at all, link with *lib/gvc/demand_loading.c* or *lib/gvc/no_demand_loading.c*

For the common case of maximal use of **demand-loaded plugins** the *libgvc.so* library is already linked with *lib/gvc/demand_loading.c* and *lib/gvc/dot_builtins.c*. To provide you own choice of **builtins**, and **demand-loading** use *libgvc_builtins.so* and your own versions of *builtins.c* and *demand_loading.c*.

Static linking will link all graphviz libraries, and any **plugins-libraries** that are **builtin**, into a single binary. Note that system libraries are not statically linked to the image.

See *cmd/dot/Makefile.am* for the technique used to build **dot_static.**

Writing a plugin

A **plugin_library** is an implementation of one or more **plugin_types** for one or more **plugin_apis**.

Each library has a single public symbol of type: *gvplugin_library_t* provided in a file named like: *devil/gvplugin_devil.c*.

```
gvplugin_library_t gvplugin_devil_LTX_library = { "devil", apis };
```

The symbol must begin with "gvplugin_" foolowed by the name of the library, in this case "devil" followed by "_LTX_library". This form is required by libtool for ease of generating builtins, and by the code for "dot -c" which searches for installed plugins_libraries.

This single public symbol is the address of a struct which provides the name of the library and a pointer to the set of plugin_apis implemented in the library. The apis are defined in a: static *gvplugin_api_t apis[] = {...}* each element of this array provides an api identifier and a pointer to the set of types of that api implemented in this plugin_library.

```
static gvplugin_api_t apis[] = {
        {API_device, &gvdevice_devil_types},
        {(api_t)0, 0},
    };
```

Each **plugin_api** in the **plugin_library** is typically provided by a separate file like: *devil/gvdevice_devil.c*. This file provides the set of types in an array of *gvplugin_installed_t*. Each element of the array provides: an id for the type, a string "name:dependency", a quality value, a pointer to an engine implementing the api, and a pointer to a feature table.

```
    gvplugin_installed_t gvdevice_devil_types[] = {
        {IL_BMP, "bmp:cairo", -1, &devil_engine, &device_features_devil},
        {IL_JPG, "jpg:cairo", -1, &devil_engine, &device_features_devil},
        {IL_JPG, "jpe:cairo", -1, &devil_engine, &device_features_devil},
        {IL_JPG, "jpeg:cairo", -1, &devil_engine, &device_features_devil},
        {IL_PNG, "png:cairo", -1, &devil_engine, &device_features_devil},
        {IL_TIF, "tif:cairo", -1, &devil_engine, &device_features_devil},
        {IL_TIF, "tiff:cairo", -1, &devil_engine, &device_features_devil},
        {IL_TGA, "tga:cairo", -1, &devil_engine, &device_features_devil},
        {0, NULL, 0, NULL, NULL}
    };
```

The engine is a jump table into the functions for the api. Any entry can be NULL if not implemented by the plugin:

```
    static gvdevice_engine_t devil_engine = {
        NULL,
        NULL,
        devil_format,
        NULL,
    };
```

Notes on render api

The render api provides an encapsulation of the drawing operations of the various renderer plugins. The api provides the limited set of primitives required to draw graphs: polygon, line, spline, etc.

At the plugin api, coordinates are presented in one of two corms, depending on the feature: GVRENDER_DOES_TRANSFORMS.

- If the plugin does its own transforms, then coordinates are presented in graph units (1/72 in) with values for: scale, transform, rotation, available in GVJ_t *job, and GVRENDER_Y_GOES_DOWN from features.
- If the plugin does not provide its own transforms, the: scale, transform, rotation, and GVRENDER_Y_GOES_DOWN values are pre-applied to the coordinates.

The inverse-transform for mouse events to graph-coordinates is always done by the common code in gvevent.c; mouse event callbacks use window coordinates.

Notes on device api

The device api provides the event callbacks defined in: gvdevice_callbacks_t in gvcjob.h
The jump-table is defined in gvevent.c and a pointer to this table is copied at run-time into

jobs->callbacks. This is done to avoid linking the plugins with the core code directly.

ToDo

- “file” and “compressed file” output should be implemented as device plugins.
- Implement a mechanism for overriding **plugins** for other **plugin-apis** from the command line.
- Provide a new **plugin-api** and **plugins** for input-parsing – add support for e.g. GXL input.

Author

John Ellson **-** August 10, 2007