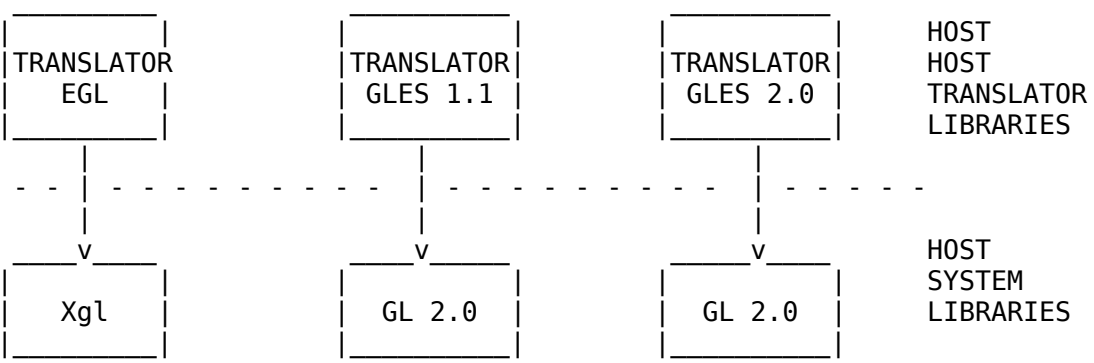
Android Hardware OpenGLES emulation design overview
==

Introduction:

Hardware GLES emulation in the Android platform is implemented with a mix
of components, which are:

  - Several host "translator" libraries. They implement the EGL, GLES 1.1 and
    GLES 2.0 ABIs defined by Khronos, and translate the corresponding function
    calls into calls to the appropriate desktop APIs, i.e.:

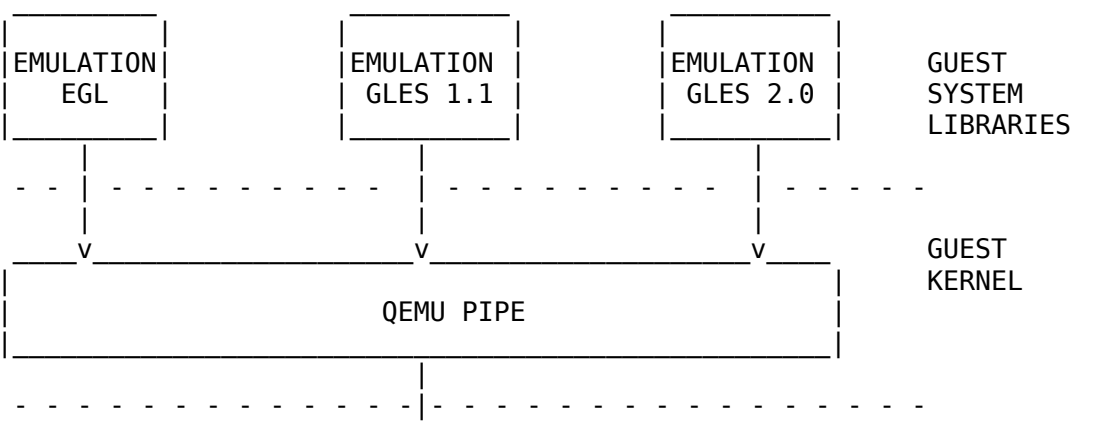
      - Xgl (Linux), AGL (OS X) or WGL (Windows) for EGL
      - desktop GL 2.0 for GLES 1.1 and GLES 2.0

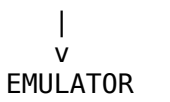


  - Several system libraries inside the emulated guest system that implement
    the same EGL / GLES 1.1 and GLES 2.0 ABIs.

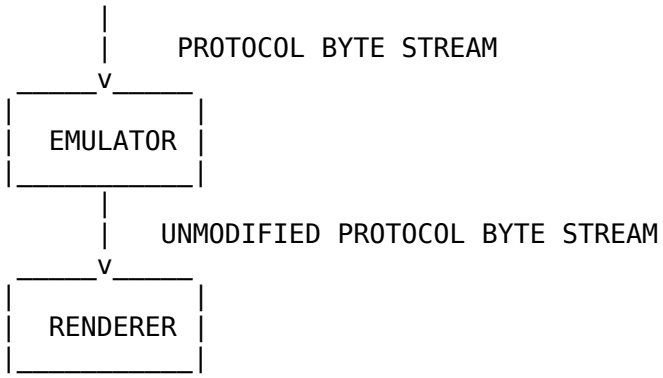
    They collect the sequence of EGL/GLES function calls and translate then
    into a custom wire protocol stream that is sent to the emulator program
    through a high-speed communication channel called a "QEMU Pipe".

    For now, all you need to know is that the pipe is implemented with a
    custom kernel driver, and provides for _very_ fast bandwidth. All read()
    and writes() from/to the pipes are essentially instantaneous from the
    guest's point of view.

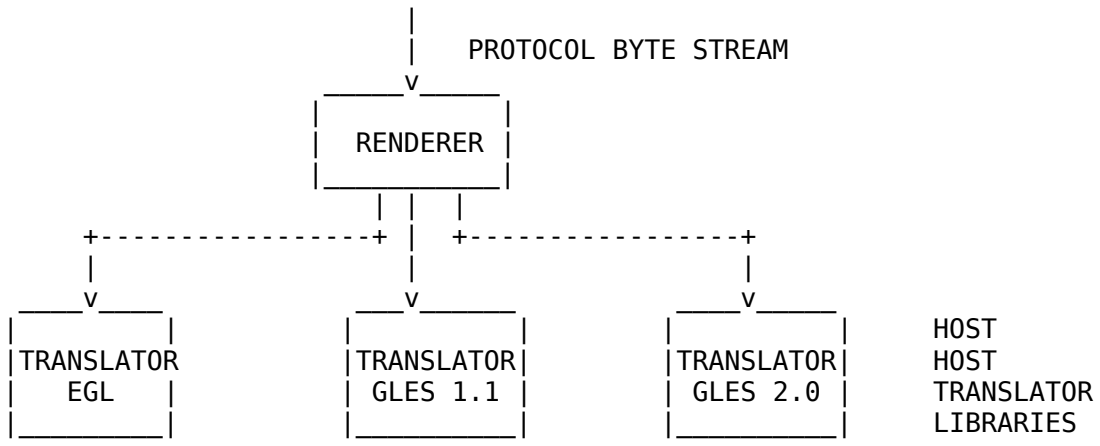




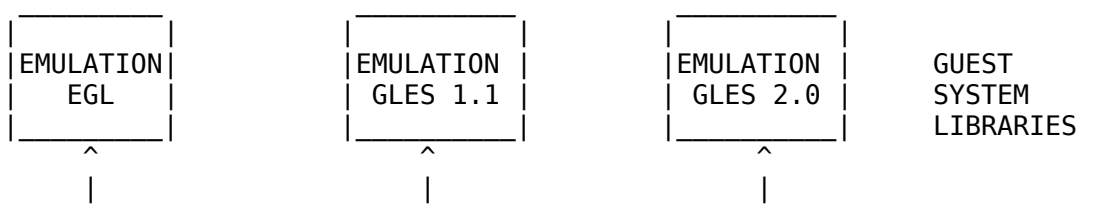
- Specific code inside the emulator program that is capable of transmitting the wire protocol stream to a special rendering library or process (called the "renderer" here), which understands the format.

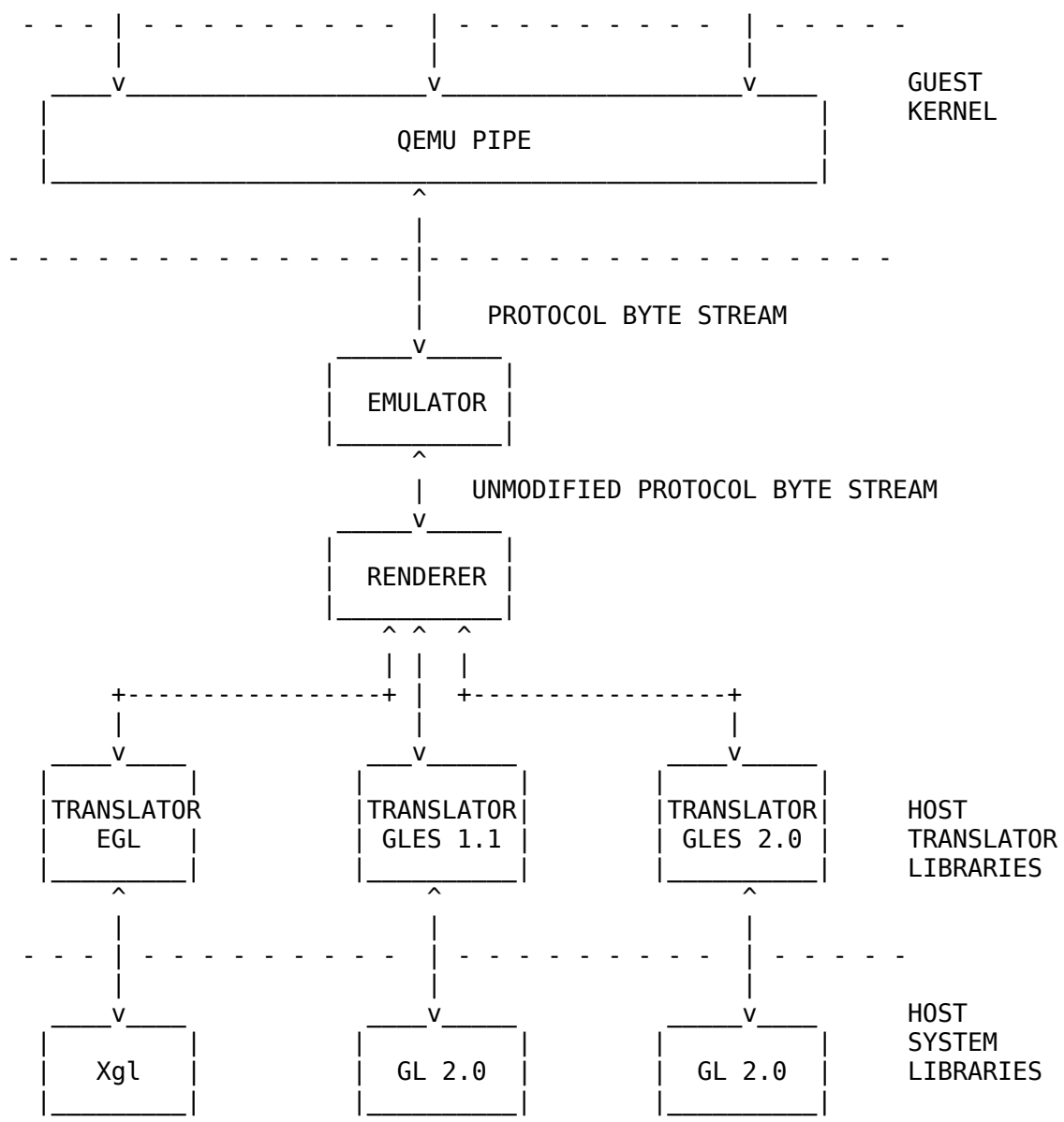


- The renderer decodes the EGL/GLES commands from the wire protocol stream, and dispatches them to the translator libraries appropriately.



- In reality, the protocol stream flows in both directions, even though most of the commands result in data going from the guest to the host. A complete picture of the emulation would thus be:





(NOTE: 'Xgl' is for Linux only, replace 'AGL' on OS X, and 'WGL' on Windows).

Note that, in the above graphics, only the host system libraries at the bottom
are _not_ provided by Android.

Design Requirements:

The above design comes from several important requirements that were decided
early in the project:

1 - The ability to run the renderer in a separate process from the emulator
    itself is important.

    For various practical reasons, we plan to completely separate the core QEMU
    emulation from the UI window by using two distinct processes. As such, the
    renderer will be implemented as a library inside the UI program, but will

need to receive protocol bytes from the QEMU process.

The communication channel will be either a fast Unix socket or a Win32 named pipe between these two. A shared memory segment with appropriate synchronization primitives might also be used if performance becomes an issue.

This explains why the emulator doesn't alter or even try to parse the protocol byte stream. It only acts as a dumb proxy between the guest system and the renderer. This also avoids adding lots of GLES-specific code inside the QEMU code base which is terribly complex.

2 - The ability to use vendor-specific desktop EGL/GLES libraries is important.

GPU vendors like NVidia, AMD or ARM all provide host versions of the EGL/GLES libraries that emulate their respectivie embedded graphics chipset.

The renderer library can be configured to use these instead of the translator libraries provided with this project. This can be useful to more accurately emulate the behaviour of specific devices.

Moreover, these vendor libraries typically expose vendor-specific extensions that are not provided by the translator libraries. We cannot expose them without modifying our code, but it's important to be able to do so without too much pain.

Code organization:

All source code for the components above is spread over multiple directories in the Android source trees:

- The emulator sources are under $ANDROID/external/qemu, which we'll call $QEMU in the rest of this document.

- The guest libraries are under $ANDROID/development/tools/emulator/opengl, which we'll call $EMUGL_GUEST

- The host renderer and translator libraries are under $ANDROID/sdk/emulator/opengl, which we'll call $EMUGL_HOST

- The QEMU Pipe kernel driver is under $KERNEL/drivers/misc/qemupipe

Where $ANDROID is the top of the open-source Android source tree, and $KERNEL is the top of the qemu-specific kernel source tree (using one of the android-goldfish-xxxx branches here).

The emulator sources related to this projects are:

```
   $QEMU/hw/goldfish_pipe.c  -> implement QEMU pipe virtual hardware
   $QEMU/hw/opengles.c       -> implement GLES initialization
   $QEMU/hw/hw-pipe-net.c    -> implements the communication channel
                                between the QEMU Pipe and the renderer library
```

The other sources are:

```
   $EMUGL_GUEST/system   -> system libraries
   $EMUGL_GUEST/shared   -> guest copy of shared libraries
   $EMUGL_GUEST/tests    -> various test programs
   $EMUGL_HOST/host      -> host libraries (translator + renderer)
   $EMUGL_HOST/shared    -> host copy of shared libraries
   $EMUGL_HOST/tests     -> various test programs
```

The reason the shared libraries aren't actually shared is historical: at one point both guest and host code lived in the same place. That turned out to be impractical with the way the Android SDK is branched, and didn't support the requirement that a single emulator binary be able to run several generations of Android.

Translator libraries:

There are three translator host libraries provided by this project:

```
   libEGL_translator       -> EGL 1.2 translation
   libGLES_CM_translator   -> GLES 1.1 translation
   libGLES_V2_translator   -> GLES 2.0 translation
```

The full name of the library will depend on the host system.
For simplicity, only the library name suffix will change (i.e. the 'lib' prefix is not dropped on Windows), i.e.:

```
   libEGL_translator.so    -> for Linux
   libEGL_translator.dylib -> for OS X
   libEGL_translator.dll   -> for Windows
```

The source code for these libraries is located under the following path in the Android source tree:

```
   $EMUGL_HOST/host/libs/Translator/EGL
   $EMUGL_HOST/host/libs/Translator/GLES_CM
   $EMUGL_HOST/host/libs/Translator/GLES_V2
```

The translator libraries also use a common routines defined under:

```
   $EMUGL_HOST/host/libs/Translator/GLcommon
```

Wire Protocol Overiew:

The "wire protocol" is implemented as follows:

- EGL/GLES function calls are described through several "specification" files, which describes the types, function signatures and various attributes for each one of them.

- These files are read by a tool called "emugen" which generates C source files and headers based on the specification. These correspond to both encoding, decoding and "wrappers" (more on this later).

- System "encoder" static libraries are built using some of these generated

files. They contain code that can serialize EGL/GLES calls into simple
byte messages and send it through a generic "IOStream" object.

- Host "decoder" static libraries are also built using some of these
generated files. Their code retrieves byte messages from an "IOStream"
object, and translates them into function callbacks.

IOStream abstraction:
- - - - - - - - - - -

The "IOStream" is a very simple abstract class used to send byte messages
both in the guest and host. It is defined through a shared header under
$EMUGL/host/include/libOpenglRender/IOStream.h

Note that despite the path, this header is included by *both* host and guest
source code. The main idea around IOStream's design is that to send a message,
one does the following:

1/ call stream->allocBuffer(size), which returns the address of a
memory buffer of at least 'size' bytes.

2/ write the content of the serialized command (usually a header + some
payload) directly into the buffer

3/ call stream->commitBuffer() to send it.

Alternatively, one can also pack several commands into a single buffer with
stream->alloc() and stream->flush(), as in:

1/ buf1 = stream->alloc(size1)
2/ write first command bytes into buf1
3/ buf2 = stream->alloc(size2)
4/ write second command bytes into buf2
5/ stream->flush()

Finally, there are also explict read/write methods like stream->readFully()
or stream->writeFully() which can be used when you don't want an intermediate
buffer. This is used in certain cases by the implementation, e.g. to avoid
an intermediate memory copy when sending texture data from the guest to the
host.

The host IOStream implementations are under $EMUGL/shared/OpenglCodecCommon/,
see in particular:

$EMUGL_HOST/shared/OpenglCodecCommon/TcpStream.cpp
-> using local TCP sockets
$EMUGL_HOST/shared/OpenglCodecCommon/UnixStream.cpp
-> using Unix sockets
$EMUGL_HOST/shared/OpenglCodecCommon/Win32PipeStream.cpp
-> using Win32 named pipes

The guest IOStream implementation uses the TcpStream.cpp above, as well as
an alternative QEMU-specific source:

$EMUGL_GUEST/system/OpenglSystemCommon/QemuPipeStream.cpp
-> uses QEMU pipe from the guest

The QEMU Pipe implementation is _significantly_ faster (about 20x) due to

several reasons:

  - all succesful read() and write() operations through it are instantaneous
    from the guest's point of view.

  - all buffer/memory copies are performed directly by the emulator, and thus
    much faster than performing the same thing inside the kernel with emulated
    ARM instructions.

  - it doesn't need to go through a kernel TCP/IP stack that will wrap the
    data into TCP/IP/MAC packets, send them to an emulated ethernet device,
    which is itself connected to an internal firewall implementation that
    will unwrap the packets, re-assemble them, then send them through BSD
    sockets to the host kernel.

However, would it be necessary, you could write a guest IOStream implementation
that uses a different transport. If you do, please look at
$EMUGL_GUEST/system/OpenglCodecCommon/HostConnection.cpp which contains the
code used to connect the guest to the host, on a per-thread basis.

Source code auto-generation:
- - - - - - - - - - - - - -

The 'emugen' tool is located under $EMUGL_HOST/host/tools/emugen. There is a
README file that explains how it works.

You can also look at the following specifications files:

For GLES 1.1:
    $EMUGL_HOST/host/GLESv1_dec/gl.types
    $EMUGL_HOST/host/GLESv1_dec/gl.in
    $EMUGL_HOST/host/GLESv1_dec/gl.attrib

For GLES 2.0:
    $EMUGL_HOST/host/GLESv2_dec/gl2.types
    $EMUGL_HOST/host/GLESv2_dec/gl2.in
    $EMUGL_HOST/host/GLESv2_dec/gl2.attrib

For EGL:
    $EMUGL_HOST/host/renderControl_dec/renderControl.types
    $EMUGL_HOST/host/renderControl_dec/renderControl.in
    $EMUGL_HOST/host/renderControl_dec/renderControl.attrib

Note that the EGL specification files are under a directory named
"renderControl_dec" and have filenames that begin with "renderControl"

This is mainly for historic reasons now, but is also related to the fact that
this part of the wire protocol contains support functions/calls/specifications
that are not part of the EGL specification itself, but add a few features
required to make everything works. For example, they have calls related to
the "gralloc" system library module used to manage graphics surfaces at a
lower level than EGL.

Generally speaking, guest encoder sources are located under directories
named $EMUGL_GUEST/system/<name>_enc/, while the corresponding host decoder
sources will be under $EMUGL_HOST/host/libs/<name>_dec/

However, all these sources use the same spec files located under the
decoding directories.

The encoder files are built from emugen and spec files located in $EMUGL_HOST
and copied to the encoder directories in $EMUGL_GUEST by the gen-encoder.sh
script. They are checked in, so that a given version of Android supports a
specific version of the protocol, even if newer versions of the renderer (and
future Android versions) support a newer protocol version. This step needs to
be done manually when the protocol changes; these changes also need to be
accompanied by changes in the renderer to handle the old version of the
protocol.

System libraries:

Meta EGL/GLES system libraries, and egl.cfg:
- -

It is important to understand that the emulation-specific EGL/GLES libraries
are not directly linked by applications at runtime. Instead, the system
provides a set of "meta" EGL/GLES libraries that will load the appropriate
hardware-specific libraries on first use.

More specifically, the system libEGL.so contains a "loader" which will try
to load:

  - hardware-specific EGL/GLES libraries
  - the software-based rendering libraries (called "libagl")

The system libEGL.so is also capable of merging the EGL configs of both the
hardware and software libraries transparently to the application. The system
libGLESv1_CM.so and libGLESv2.so, work with it to ensure that the thread's
current context will be linked to either the hardware or software libraries
depending on the config selected.

For the record, the loader's source code in under
frameworks/base/opengl/libs/EGL/Loader.cpp. It depends on a file named
/system/lib/egl/egl.cfg which must contain two lines that look like:

    0 1 <name>
    0 0 android

The first number in each line is a display number, and must be 0 since the
system's EGL/GLES libraries don't support anything else.

The second number must be 1 to indicate hardware libraries, and 0 to indicate
a software one. The line corresponding to the hardware library, if any, must
always appear before the one for the software library.

The third field is a name corresponding to a shared library suffix. It really
means that the corresponding libraries will be named libEGL_<name>.so,
libGLESv1_CM_<name>.so and libGLESv2_<name>.so. Moreover these libraries must
be placed under /system/lib/egl/

The name "android" is reserved for the system software renderer.

The egl.cfg that comes with this project uses the name "emulation" for the

hardware libraries. This means that it provides an egl.cfg file that contains the following lines:

```
   0 1 emulation
   0 0 android
```

See $EMUGL_GUEST/system/egl/egl.cfg and more generally the following build files:

```
   $EMUGL_GUEST/system/egl/Android.mk
   $EMUGL_GUEST/system/GLESv1/Android.mk
   $EMUGL_GUEST/system/GLESv2/Android.mk
```

to see how the libraries are named and placed under /system/lib/egl/ by the build system.

Emulation libraries:
- - - - - - - - - - -

The emulator-specific libraries are under the following:

```
  $EMUGL_GUEST/system/egl/
  $EMUGL_GUEST/system/GLESv1/
  $EMUGL_GUEST/system/GLESv2/
```

The code for GLESv1 and GLESv2 is pretty small, since it mostly link against the static encoding libraries.

The code for EGL is a bit more complex, because it needs to deal with extensions dynamically. I.e. if an extension is not available on the host it shouldn't be exposed by the library at runtime. So the EGL code queries the host for the list of available extensions in order to return them to clients. Similarly, it must query the list of valid EGLConfigs for the current host system.

"gralloc" module implementation:
- - - - - - - - - - - - - - - - - -

In addition to EGL/GLES libraries, the Android system requires a hardware-specific library to manage graphics surfaces at a level lower than EGL. This library must be what is called in Android land as a "HAL module".

A "HAL module" must provide interfaces defined by Android's HAL (Hardware Abstraction Library). These interface definitions can be found under $ANDROID/hardware/libhardware/include/

Of all possible HAL modules, the "gralloc" one is used by the system's SurfaceFlinger to allocate framebuffers and other graphics memory regions, as well as eventually lock/unlock/swap them when needed.

The code under $EMUGL/system/gralloc/ implements the module required by the GLES emulation project. It's not very long, but there are a few things to notice here:

- first, it will probe the guest system to determine if the emulator that
  is running the virtual device really supports GPU emulation. In certain

  circumstances this may not be possible.

  If this is the case, then the module will redirect all calls to the
  "default" gralloc module that is normally used by the system when
  software-only rendering is enabled.

  The probing happens in the function "fallback_init" which gets called
  when the module is first opened. This initializes the 'sFallback' variable
  to a pointer to the default gralloc module when required.

- second, this module is used by SurfaceFlinger to display "software surfaces",
  i.e. those that are backed by system memory pixel buffers, and written to
  directly through the Skia graphics library (i.e. the non-accelerated ones).

  the default module simply copies the pixel data from the surface to the
  virtual framebuffer i/o memory, but this project's gralloc module sends it
  to the renderer through the QEMU Pipe instead.

  It turns out that this results in _faster_ rendering/frame-rates overall,
  because memory copies inside the guest are slow, while QEMU pipe transfers
  are done directly in the emulator.

Host Renderer:

The host renderer library is located under
$EMUGL_HOST/host/libs/libOpenglRender, and it provides an interface described
by the headers under $EMUGL_HOST/host/include/libOpenglRender/render_api.h
(e.g. for use by the emulator).

In a nutshell, the rendering library is responsible for the following:

  - Providing a virtual off-screen video surface where everything will get
    rendered at runtime. Its dimensions are fixed by the call to
    initOpenglRender() that must happen just after the library is
    initialized.

  - Provide a way to display the virtual video surface on a host application's
    UI. This is done by calling createOpenGLSubWindow() which takes as argument
    the window ID or handle of a parent window, some display dimensions and
    a rotation angle. This allows the surface to be scaled/rotated when it is
    displayed, even if the dimensions of the video surface do not change.

  - Provide a way to listen to incoming EGL/GLES commands from the guest.
    This is done by providing a so-called "port number" to initOpenglRender().

    By default, the port number corresponds to a local TCP port number that the
    renderer will bind to and listen. Every new connection to this port will
    correspond to the creation of a new guest host connection, each such
    connection corresponding to a distinct thread in the guest system.

    For performance reasons, it is possible to listen to either Unix sockets
    (on Linux and OS X), or to a Win32 named pipe (on Windows). To do so, one
    had to call setStreamType() between library initialization
    (i.e. initLibrary()) and construction (i.e. initOpenglRender()).

    Note that in these modes, the port number is still used to differentiate

between several emulator instances. These details are normally handled by
the emulator code so you shouldn't care too much.

Note that an earlier version of the interface allowed a client of the renderer library to provide its own IOStream implementation. However, this wasn't very convenient for a number of reasons. This maybe something that could be done again if it makes sense, but for now the performance numbers are pretty good.

Host emulator:

The code under $QEMU/android/opengles.c is in charge of dynamically loading the rendering library and initializing / constructing it properly.

QEMU pipe connections to the 'opengles' service are piped through the code in $QEMU/android/hw-pipe-net.c. Look for the openglesPipe_init() function, which is in charge of creating a connection to the renderer library (either through a TCP socket, or a Unix pipe depending on configuration. support for Win32 named pipes hasn't been implemented yet in the emulator) whenever a guest process opens the "opengles" service through /dev/qemu_pipe.

There is also some support code for the display of the GLES framebuffer (through the renderer library's subwindow) under $QEMU/skin/window.

Note that at the moment, scaling and rotation are supported. However, brightness emulation (which used to modify the pixel values from the hardware framebuffer before displaying them) doesn't work.

Another issue is that it is not possible to display anything on top of the GL subwindow at the moment. E.g. this will obscure the emulated trackball image (that is normally toggled with Ctrl-T during emulation, or enabled by pressing the Delete key).