jAER Rendering Pipeline Review

Sven Göthel, Göthel Software e.K. (jausoft.com), 2024-04-02 rev 0.1

Table of Contents

Initial Assessment....1
OpenGL ES/Core Programmable Pipeline....2
Compatibility....2
Performance....2
Texture Handling....3
Order of Texture Commands....3
Texture ID Bug....3
Programmable Shader (GLSL)....4
Conclusion....5
Dropping AWT Dependencies....5
jAER's Rendering Pipeline....6
Java Platform....7
Java Graphics....7
Back to the Browser....7

Initial Assessment

Application uses the following features

- OpenGL fixed function pipeline (FFP) including its
  - PMV matrix
  - Fixed states
  - Graphic Primitives
- AWT and Swing/GLJPanel utilization
  - Reliance on AWT libraries including JOGL's AWT toolkits like TextRenderer

The followup discusses potential changes for the application addressing different goals, such as

- Future compatibility
- Correctness
- Performance

Most of the changes would deeply impact the current codebase and may only be reasonable if performed together, e.g. dropping FFP functionality with AWT-only JOGL code etc. The result would be a more modern OpenGL ES/core programmable rendering pipeline (PP) capable to run without AWT, which might also perform better (responsiveness, resources and processing).

OpenGL ES/Core Programmable Pipeline

Moving from the fixed function pipeline (FFP) to the programmable pipeline (PP) would not be difficult but certainly time consuming.

Compatibility

Using the PP would further enable more potential devices like embedded, handhelds, etc. This is true even if considering moving away from Java at some point, as the rendering data structures would not need to change and the shader design could be maintained (if any).

Today, the FFP is mostly implemented in software either by the OpenGL driver itself or external intermediate layers. Hence using such intermediate libraries could remove the necessity of rewriting the rendering code for the PP. However, less layers and libraries are usually desired and exposes the application to less complexity and bugs (see below) - this may in itself be a strong motivation.

Performance

Using the core pipeline and hence self-handling (primitive) GPU data buffers including textures is usually more efficient than compiling them via the fixed function pipeline for each frame. Such enhancement would naturally only be visible if using a high polygon count in the rendered content, not for mostly texture based material.

Texture Handling

Order of Texture Commands

Taken from JOGL's Texture documentation:

Due to many confusions w/ texture usage, following list described the order and semantics of texture unit selection, binding and enabling.

- *Optional:* Set active textureUnit via gl.glActiveTexture(GL.GL_TEXTURE0 + textureUnit), 0 is default.
- Bind textureId -> active textureUnit's textureTarget via gl.glBindTexture(textureTarget, textureId)
- *Compatible Context Only:* Enable active textureUnit's textureTarget via glEnable(textureTarget).
- *Optional:* Fiddle with the texture parameters and/or environment settings.
- GLSL: Use textureUnit in your shader program, enable shader program.
- Issue draw commands

From a shader standpoint, there are only texture units visible, no texture IDs. A shader can utilize multiple texture units at once.
Texture IDs reflect host (CPU) side texture objects of a given activated texture unit and can be used to program different texture configurations and upload different data sets within one texture unit.
Bound texture objects survive after unbinding them, the last bound texture ID object of a given activated texture unit is being used in a shader.
Texture objects are more a concept of host (CPU) managed resources and switching the bound texture ID object may cause GPU transfers, depending on the OpenGL implementation.
To survive texture data across frames and to minimize GPU data transfers, it is recommended to use multiple texture units (not texture IDs).

Texture ID Bug

Commit a956833640d8f25d69fd12bed34c1cfb15139870 mitigated the texture ID issue on MacOS, originally lead to my engagement.

On MacOS a software implemented FFP is being used and it might be the case that switching the texture ID is erroneous within this implementation, despite operating on the same texture unit.

ChipRendererDisplayMethodRGBA's displayQuad(..) operates on the default texture unit and then renders different texture objects in the following order

1. switching the texture unit
2. setup of pixel store data transfer
3. texture parametrization
4. uploads the texture data
5. enables texture rendering
6. renders the texture object (using FFP)
7. disables texture rendering

There is nothing wrong with this approach, even though switching the texture ID and reconfigure them plus uploading the data for each frame might not be efficient.

Indeed, using a texture ID object and have it configured once would be more optimal – however, as the issue on MacOS showed – there were issues. It would need to be investigated, whether the state flow order shown was missing a texture ID switch or – more likely – the asynchronous nature of the FFP execution simply overwrote the previous bound texture by performing the actual rendering after texture state programming. *The latter could be potentially remedied by issuing a glFlush() command after each draw-command-block.*

Programmable Shader (GLSL)

JAER is already utilizing GLSL in SpaceTimeRollingEventDisplayMethod.

Similar to other used hardware accelerated technologies like compute shader or processing (neuronal) network data via SIMD/MIMD on a GPU or ASIC, programmable shader (GLSL) offer the same properties to a rendering pipeline.

Not only are they capable to render the usual primitives like points and polygons, apply texture data for each pixel (fragment) – but one could also implement custom shape, effects and also reinterpret and render data cached via textures. The latter is demonstrated with JogAmp's FFMpeg video frame shader, which convert the given video pixel format to RGBA.

Conclusion

Utilizing the programmable pipeline (PP) with programmable shader (PS) gives the application full control of managing the resources as well as computing and rendering the data in a most flexible way. A further step into the atomic programmable world would be the Vulkan API, however .. this might be considered too much of a hassle at this point and could be targeted after shaping the code to use modern OpenGL.

Dropping the fixed function pipeline (FFP) would also require to replace at least some of the used AWT dependent features, e.g. TextRenderer. The latter could be replaced by using our Graph/GraphUI GPU curve renderer, which also provides a complete True Type Font (TTF) text rendering feature.

Dropping the FFP also allows dropping all AWT dependencies, see below.

Dropping AWT Dependencies

Dropping AWT dependencies would mostly require to replace some of the used functionality, e.g. TextRenderer with Graph/GraphUI (see above).

This task would allow the GLEventListener and renderer code to be more versatile and usable with AWT, but also with an AWT-less environment (NEWT, Android, ..).

Besides mentioned text renderer, keyboard and mouse events would need to be recoded using a platform agnostic interface, e.g. NEWT's KeyEvent and MouseEvent. Both can also be used to receive AWT events besides other subsystems like SWT or native console, X11, Cocoa, etc.

If considering using GraalVM to produce a WebAssembly (wasm) target and hence browser (or the web), dropping AWT probably is desired as well as validating JogAmp against this target platform. For this goal I also recommend using the

OpenGL core or ES2/ES3 profile as it is supported by the browser environment (see below).

jAER's Rendering Pipeline

Application implements GLEventListener, most notable ChipCanvas as an intermediate to its DisplayMethod. Only DisplayMethod's JMenuItem field seems to break its platform independence. Overall the OO layout is as follows

- Chip2DRenderer
  - processing data into a platform agnostic pixmap via *getPixmap()* etc
- Chip2D has_a Chip2DRenderer
  - main node to the actual Chip2D and its data
- ChipCanvas has_a Chip2D, DisplayMethod
  - graphic toolkit dependent visualization canvas node
- DisplayMethod uses ChipCanvas's Chip2D's Chip2DRenderer
  - graphic toolkit dependent visualization rendering node

Notable no *init(..)* and *dispose(..)* interface methods are looped through from the GLEventListener class ChipCanvas to DisplayMethod, which would allow efficient OpenGL resources management for buffers, shader etc. It is recommended to add at least these two methods to DisplayMethod and passing through its events.

This would also align DisplayMethod with the GLEventListener lifecycle, which may call init(..) and dispose(..) multiple times depending on the underlying toolkit.

Currently the lack of exposing init(..) is overcome by a one-shot initialization, e.g. in *SpaceTimeRollingEventDisplayMethod::installShaders(..).* However, this will not work in case the GLContext is taken down and being re-initialized since the *dispose(..)* event is ignored.

Chip2DRenderer's pixmap is implemented as an array backed non-native (NIO) FloatArray. While it is indeed more efficient to use plain Java arrays for small data sets, this is not true for bigger quantities. For the latter the data is not only written to the Java array during data acquisition but also copied into a native array

when writing to the graphic texture object. It might be worthwhile to evaluate a copy-once approach, at least up until the GPU memory barrier.

Java Platform

Java Graphics

Java support for AWT and desktop in general is heavily reduced by Oracle, Red Hat (OpenJDK) and other players in the last years – if not abandoned.

OpenJFX / JavaFX is still maintained by certain companies and used for desktop and embedded targets.

JogAmp's NEWT provides a rudimentary interface to different desktop and embedded windowing systems while JogAmp's Graph/GraphUI targets the UI area in a wholistic manner.

See GraalVM/wasm notes above.

Back to the Browser

Due to the desire to run application not only on specific target devices like Linux, MacOS etc, but also within web browsers – many JogAmp or Java Desktop user desired to drop Java completely in favor for bringing back the web runtime capability since Oracle dropped the Java browser Applet plug-in. Alternatives would be to bite the bullet using JavaScript or better any LLVM supported language like C++ using the WebAssembly (wasm) target.

Emscripten supports the latter combining C/C++ source with LLVM's wasm target and ported libraries such as SDL, OpenGL etc.

Our own late wasm/emscripten evaluation is visible in gfxbox2.

For once, the wasm target is comparable to Java bytecode but the virtual machine might not be as performant, efficient or flexible. The lack of hassle free native threads and performance may remind one of the first JVM steps. The wasm code runs within a virtual machine like the JVM and also lacks of native binding capabilities, besides aforementioned limitations.

Some limitations may be overcome, others are likely by design as it usually runs on the same browser virtual machine as the JavaScript companion.

So far the only question arises *why reinventing the wheel once again*? Dropping utilization of a well working JVM for the web had not technical reasoning.

GraalVM seems to support the wasm target (see notes above), hence one might want to evaluate this Java target platform as well before recoding the whole application.

Opening the door for any language to be compiled to a platform independent intermediate representation (IR) like Java's bytecode suitable for a versatile runtime environment including its virtual machine is surely still desirable. Hopefully this technology will mature and be maintained for the years to come.