Marlin renderer, a JDK-9 Success Story
Vector Graphics on Steroids for Java 2D and JavaFX

Laurent Bourgès - The Developer
Jim Graham - The Computer Graphics Guru
JavaOne 2017-10-04
Outline

Context & History
How Marlin works?
Visual quality Quiz
Marlin usage & benchmarks

MarlinFX
MarlinFX usage & benchmarks
Perspectives and Future Work
Conclusion
Context & History
Role of an Anti-aliasing Renderer

- Geometry is defined by a mathematical description of a path
- **Shapes** may be complex: concave, convex, intersecting ...
- To draw to any kind of raster surface (image, screen), need to map the rendering of ideal path to raster coordinates
- **Points on the path** may rarely map exactly to raster coordinates
- Need to ascertain coverage (anti-aliasing) for each output pixel

<table>
<thead>
<tr>
<th>0x0</th>
<th>2x2</th>
<th>4x4</th>
<th>8x8</th>
<th>16x16</th>
<th>64x64</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="16th.png" alt="16ths" /></td>
<td><img src="16th.png" alt="16ths" /></td>
<td><img src="16th.png" alt="16ths" /></td>
<td><img src="16th.png" alt="16ths" /></td>
<td><img src="16th.png" alt="16ths" /></td>
<td><img src="16th.png" alt="16ths" /></td>
</tr>
</tbody>
</table>
Java2D API in a nutshell

Java2D is a great & powerful Graphics API:

- **Shape** interface (java.awt.geom)
  - Primitives: Rectangle2D, Line2D, Ellipse2D, Arc2D ...
  - General Path2D: moveTo(), lineTo(), quadTo(), curveTo(), closePath()
- **Stroke** interface: BasicStroke (width, dashes fields)
- **Paint** interface: Color, GradientPaint, TexturePaint
- **Composite** interface: AlphaComposite (Porter/Duff blending rules)
- **Graphics** interface: Graphics2D draw(Shape) or fill(Shape) performs shape rendering operations
Anti-aliasing Software renderers available in Java:

- **JDK 1.2** included licensed *closed-source* technology from **Ductus**
  - Ductus provides *high performance, but single threaded*
  - State of the art for its time
  - Ductus still ships with Oracle JDK-9 (no longer default)
  - `sun.dc.DuctusRenderingEngine` (native C code)

- However **OpenJDK 1.6 replaced Ductus with "Pisces"**
  - Pisces developed + owned by Sun, in part for Java ME
  - So could be *open-sourced but performance much poorer*
  - `java2d.pisces.PiscesRenderingEngine` (java code)

Note: **Other renderers for non-AA cases**
Marlin renderer = OpenJDK’s Pisces fork

Status in 2013:

- OpenGL & D3D pipelines provide only few accelerated operations (blending surfaces), except the `glg2d Graphics2D`
- `BufferedImage` blending made by software loops (C macros)

2013.3: My first patches to OpenJDK-8:

- Pisces patches to 2d-dev@openjdk.java.net: too late for JDK-8
- Small interest / few feedbacks

Andréa Aimé (GeoServer team) pushed me to go on:

- Fork OpenJDK’s Pisces as a new open-source project
2014.1: **Marlin renderer** & **MapBench** projects @ github (GPL v2)

- [https://github.com/bourgesl/marlin-renderer](https://github.com/bourgesl/marlin-renderer)
  - branch 'openjdk-dev': dev branch
  - branch 'openjdk': in sync with OpenJDK-9 & 10
  - branch 'use_Unsafe': main for Marlin releases
- 28 **Releases**, Wiki

- [https://github.com/bourgesl/mapbench](https://github.com/bourgesl/mapbench)
  - New **MapBench** tool: serialize & replay rendering commands
Objectives:

- **Faster** alternative with very good scalability
- Improve rendering **quality**
- Compatible with both Oracle & Open JDK 7 / 8 / 9

**Huge personal work** on my spare time:

- **Performance & Test Driven Development:**
  - **Regression** tests: MapDisplay (diff Pisces vs Marlin outputs)
  - **Performance** tests: MapBench benchmarks (+ profiler)
- Major feedback (GeoServer) providing use cases & testing releases
• **FOSDEM 2015**: Great discussion with OpenJDK ’managers’ (Dalibor & Mario) on how to contribute the Marlin renderer back

⇒ I joined the **graphics-rasterizer** project in March 2015 to **contribute** Marlin as a new standalone renderer for OpenJDK-9

• **I worked really hard** (again, single developer) with Jim Graham & Phil Race (reviewers) in 2015 to push 4 big patches ≈ **10,000 LOC**!

• Reviews improved a lot of the code and Computer Graphics algorithms
JEP 265: Marlin renderer back into OpenJDK-9

We proposed the JEP 265 in July 2015 and mark it completed:

- **JEP 265: Marlin Graphics Renderer**
- [http://openjdk.java.net/jeps/265](http://openjdk.java.net/jeps/265)
- **Developer:** Laurent Bourgès
- **Reviewer:** Jim Graham

- Marlin integrated in OpenJDK-9 b96 (dec 2015), enhancements in 2016
- **Current integrated releases** within OpenJDK:

<table>
<thead>
<tr>
<th>JDK</th>
<th>Marlin version</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDK-9</td>
<td>Marlin 0.7.4</td>
</tr>
<tr>
<td>JDK-10</td>
<td>Marlin 0.7.5</td>
</tr>
</tbody>
</table>
## Marlin Releases @ github

### Major Releases:

<table>
<thead>
<tr>
<th>Release</th>
<th>Main features</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>0.3</strong> [2014.1]</td>
<td>Initial: renderer context, array cache, dirty array</td>
</tr>
<tr>
<td><strong>0.4</strong></td>
<td>Internal settings made customizable from system properties</td>
</tr>
<tr>
<td><strong>0.5</strong> [2014.3]</td>
<td>Use <code>sun.misc.Unsafe</code> (off-heap memory) (may crash your JVM)</td>
</tr>
<tr>
<td><strong>0.5.6</strong> [2015.3]</td>
<td>Optimized merge sort for newly added edges</td>
</tr>
<tr>
<td><strong>0.7</strong> [2015.8]</td>
<td>Improve coordinate rounding around sub-pixel center</td>
</tr>
<tr>
<td></td>
<td>Perform DDA in scan-line edge processing</td>
</tr>
<tr>
<td></td>
<td>Optimized cubic / quad flattening</td>
</tr>
<tr>
<td><strong>0.7.1</strong></td>
<td>Hybrid approach (raw or RLE) to copy pixel coverages into mask</td>
</tr>
</tbody>
</table>
## Marlin Releases @ github (continued)

### Major Releases:

<table>
<thead>
<tr>
<th>Release</th>
<th>Main features</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.7.2 [2015.11]</td>
<td><strong>Improve large pixel chunk copies</strong> (coverage)</td>
</tr>
<tr>
<td>0.7.3.3</td>
<td><strong>Handle coordinate overflow</strong> : ignore (NaN / Infinity)</td>
</tr>
<tr>
<td>0.7.4 [2016.6]</td>
<td><strong>Array cache refactoring &amp; tuning</strong> [→ JDK-9]</td>
</tr>
<tr>
<td>0.7.5 [2016.12]</td>
<td><strong>Added the Double pipeline for higher accuracy</strong> [→ JDK-10]</td>
</tr>
<tr>
<td></td>
<td>Optimized tile filling (almost empty / full)</td>
</tr>
<tr>
<td></td>
<td><strong>Higher precision of the curve conversion</strong> to line segments</td>
</tr>
<tr>
<td>0.8.0 [2017.8]</td>
<td><strong>Added path clipper for stroked shapes</strong> (cap, joins, tx)</td>
</tr>
<tr>
<td>0.8.1</td>
<td><strong>Added path clipper for filled shapes</strong> (N.Z. winding rule)</td>
</tr>
</tbody>
</table>
Feedback on contributing to OpenJDK

• **Very interesting experience** & many things learned
• **License issue: OCA** for all contributors, no third-party code!
• **Webrev process: great but heavy task:**
  • Create webrevs (hg status, webrev.ksh with options)
  • Push on [http://cr.openjdk.java.net/~lbourges/](http://cr.openjdk.java.net/~lbourges/)
  • Long review threads on mailing lists for my patches (≈ 50 mails)
  • Timezone difference: delays + no direct discussion
• **Few Java2D / computer graphics skills:**
  • Small & Legacy field
  • NO documentation!
In General:

- **Missing active contributors in the Graphics stack** (Java2D & JavaFX)!
- **Continuous Integration**: missing ’open’ multi-platform machines to perform tests & benchmarks outside of Oracle
- **Funding community-driven effort?**
  - Support collaboration with outsiders (meeting, costs)
  - Promote Code challenges on focused items
How Marlin works?
How the Java2D pipeline works?

Java2D uses one RenderingEngine implementation at runtime:

- **Custom impl.** using the System property 'sun.java2d.renderer'
- Called by SunGraphics2D.draw/fill(shape) in Java2D pipeline

```java
RenderingEngine:
public static synchronized RenderingEngine getInstance();
public AATileGenerator getAATileGenerator(Shape s,
                                          AffineTransform at, ...);
```

- The AATileGenerator interface defines the tile coverage provider
How the Java2D pipeline works?

- `AAShapePipe.renderPath(shape, stroke)`
  - Use `RenderingEngine.getAATileGenerator(shape, at)`
    - **Coverage mask computation** (tiles) as alpha transparency [00-FF]
  - `getAlpha(byte[] alpha, ...)` to get next tile ...
  - Output: `pipeline.renderPathTile(byte[] alpha)`
    - **Pixel blending** (software / OpenGL pipeline) on dest surface

```java
AATileGenerator:
public int getTypicalAlpha();
public void nextTile();
public void getAlpha(byte tile[], ...);
```
How Marlin works? Pisces / Marlin pipeline

- **MarlinRenderingEngine** pipeline:
  - Apply the pipeline to path elements `Shape.getPathIterator()`
  - **Dasher** (optional):
    - Generates path dashes (curved or segments)
  - **Stroker** (optional):
    - Generates edges around every path element, cap & joins
  - **Renderer**:
    - Curve decimation into line segments
    - `AddLine()`: basic clipping + convert float to sub-pixel coordinates
    - Perform edge rendering into tile strides i.e. compute pixel coverages
    - Fill the MarlinCache with pixel coverages as byte[](alpha)
- **MarlinTileGenerator**:
  - Provide tile data (32x32) from MarlinCache (packed byte[])
How Marlin works? the AA algorithm

- **Scan-line algorithm** [8x8 super-sampling] to estimate pixel coverages
  - *Active Edge table* (AET) variant with "java" pointers (integer-based)
- (Merge) Sort crossings at each scan-line
- **Estimate sub-pixel coverages** [spans] and **accumulate** [alpha row]
- Once a pixel row is done: **copy pixel coverages** into the tile cache
- Once 32 (tile height) pixel rows are done: **perform blending & repeat**
Marlin performance optimizations

- Initially **GC High allocation rate issue:**
  - Many *growing arrays + zero-fill*
  - Many arrays involved to store edge data, alpha pixel row ...
  - **Value-Types** may be very helpful: manually coded here!
- **RendererContext** (TL/CLQ): "zero waste, no GC" (recycling)
  - Kept by weak / soft reference: see **ReentrantContext**
  - Class instances + **initial arrays takes \( \approx 512Kb \)**
  - Weak-referenced **array cache for larger arrays**
- Other considerations:
  - Use **Unsafe**: allocate/free memory + fewer bound checks
  - **Optimized Zero-fill** (recycle arrays) on used parts only!
  - Use **dirty arrays** when possible: C like!
Marlin performance optimizations

• **Need good profiler**: netbeans + linux perf + internal metrics

• **Fine tuning** of Pisces algorithms:
  - Optimized **Merge Sort** (in-place)
  - Custom rounding [float to int]
  - **DDA in Renderer** with correct pixel center handling
  - Tile stride approach (32px) instead of all shape mask
  - Pixel alpha transfers (RLE) ⇒ adaptive approach
  - **Optimized Pixel copies** (block flags)

A lot more ...
Visual quality Quiz
Visual quality - Blind test [Image 1]

Which is [Ductus, Pisces, Marlin]?
Visual quality - Blind test [Image 2]

Which is [Ductus, Pisces, Marlin]?
Visual quality - Blind test [Image 3]

Which is [Ductus, Pisces, Marlin]?
Visual quality - Test result [Image 1]

Answer: [Ductus]
Visual quality - Test result [Image 2]

Answer: [Marlin]
Answer: [Pisces]
Marlin visual quality

Which is [Ductus, Pisces, Marlin]?

1. Ductus  
2. Marlin  
3. Pisces

⇒ Quite a hard challenge!
Marlin usage & benchmarks
How to use Marlin?

- Just download any Oracle JDK-9 or OpenJDK-9 release
  - Oracle JDK
  - OpenJDK builds: AdoptOpenJDK.net or zulu.org

- For JDK 1.7 or JDK-8:
  - Use Zulu 8 or JetBrains JDK-8u (marlin integrated)
  - or download the latest Marlin release @ github

- See How to use

- Start your java program with:

```java
java -Xbootclasspath/a:/lib/marlin-0.7.5-Unsafe.jar
   -Dsun.java2d.renderer=sun.java2d.marlin.MarlinRenderingEngine
   ...
```
## Major Marlin System properties

<table>
<thead>
<tr>
<th>System property &lt;KEY&gt;</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Dsun.java2d.renderer.&lt;KEY&gt;=</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>log</strong></td>
<td>true - false</td>
<td>Enable Log (initial settings)</td>
</tr>
<tr>
<td><strong>doStats</strong></td>
<td>true - false</td>
<td>Log rendering statistics</td>
</tr>
<tr>
<td><strong>useThreadLocal</strong></td>
<td><strong>true</strong> - false</td>
<td>RdrCtx in TL or CLQ</td>
</tr>
<tr>
<td><strong>edges</strong></td>
<td>4096 in [64-64K]</td>
<td>Initial capacity for edges</td>
</tr>
<tr>
<td><strong>pixelsize</strong></td>
<td>2048 in [64-32K]</td>
<td>Typical shape W/H in pixels</td>
</tr>
<tr>
<td><strong>subPixel_log2_X, ...Y</strong></td>
<td>3 in [1-8]</td>
<td>Sub-pixel count on X, Y axis</td>
</tr>
<tr>
<td><strong>tileSize_log2</strong></td>
<td>5 in [3-8]</td>
<td>Pixel width/height for tiles</td>
</tr>
</tbody>
</table>

Log-2 values for sub-pixel & tile sizes: 3 → 8, 5 → 32
MapBench benchmarks

- **MapBench tool:**
  - **Multi-threaded java2d benchmark** that replays serialized graphics commands: see `ShapeDumperGraphics2D`
  - Calibration & warm-up phase at startup + correct statistics [95th percentile...]

- **Protocol:**
  - Disable Hyper-Threading (in BIOS)
  - Use fixed CPU frequencies (2.0 GHz) on my laptop [i7-6820 HQ, 32Gb, Nvidia GPU Quadro M1000, Ubuntu 17.4 64bits]
  - Setup JVM: select JDK + basic jvm settings = CMS GC 2Gb Heap
  - Use the profile ‘shared images’ to reduce GC overhead

⇒ Reduce variability (and CPU affinity issues)
MapBench tests

- Test showcase: 9 maps, 5 large ellipse & spiral drawings
Marlin usage & benchmarks

Comparing AA renderers in JDK-9
(Ductus, Pisces, Marlin)
Before Marlin (Oracle JDK-9 EA b181)

- Oracle JDK **Ductus** & **Pisces**
Before Marlin (Oracle JDK-9 EA b181)

- Oracle JDK  **Ductus** &  **Pisces**

**Comparison**

- Ductus: faster (1T) but scalability issue!
- Pisces: slower but higher scalability
With Marlin (Oracle JDK-9 EA b181)

• Oracle JDK  **Ductus** &  **Pisces** &  **Marlin**
With Marlin (Oracle JDK-9 EA b181)

- Oracle JDK  Ductus & Pisces & Marlin

Marlin renderer is the winner

- Better performance compared to Ductus (or equal) or Pisces
- Perfect scalability
Performance gains (1 thread)

- Marlin gains over Oracle JDK Ductus & Pisces

Marlin performance gains (single thread)

- vs Ductus: 10% to 30%
- vs Pisces: 125% (average), up to 300% (stroked spiral test)
Performance gains (2 thread)

- **Marlin** gains over Oracle JDK **Ductus & Pisces**

- vs Ductus: **75%** (average)
- vs Pisces: **140%** (average)
Performance gains (4 thread)

- **Marlin** gains over Oracle JDK **Ductus** & **Pisces**

Marlin performance gains (2 threads)

- vs Ductus: **250%** (average)
- vs Pisces: **160%** (average)
• Compared to **Ductus**:
  • Same performance (1T) but better scalability up to 250% gain (4T)
• Compared to **Pisces**:
  • 2x faster: 100% to 150% gain
  • Perfect scalability 1T to 4T
Marlin usage & benchmarks

Comparing Marlin renderer changes between JDK-9 & OpenJDK-10
Marlin Performance changes (JDK-9 vs OpenJDK-10)

- Marlin 0.7.5 vs 0.7.4 (curve accuracy, large fills, Double variant)

![Graph showing performance comparison between JDK-9 and OpenJDK-10 for Marlin 0.7.5 vs 0.7.4]
Marlin Performance changes (JDK-9 vs OpenJDK-10)

- Marlin 0.7.5 vs 0.7.4 (curve accuracy, large fills, Double variant)

Minor timing changes

- large fills: 10% faster (large ellipse tests)
- curves: 10% slower (lots of curves)
Marlin Performance Summary (JDK-9 vs OpenJDK-10)

- Marlin 0.7.5 vs 0.7.4 (curve accuracy, large fills, Double variant)

Same Marlin (average) performance in OpenJDK-10 as in JDK-9

- Large fills are 10% faster, improved curve accuracy is only 10% slower
- No difference between Float and Double Marlin variants
MarlinFX
MarlinFX project

2016.10: **Port Marlin into JavaFX** Prism engine

- https://github.com/bourgesl/marlin-fx
- Marlin integration into Prism:
  - In action with the [SW] pipeline anyway
  - In action with [D3D / ES2] shader pipelines:
    - Only **Complex or dashed Paths**: not ellipse, rounded rect., nor 3D meshes
  - **Complete shape mask**, no more tiles!
  - GPU back-end is faster but uploading texture may be costly
  - But **JavaFX rendering is single-threaded** (GUI)
  - TBD: use Marlin to implement Shape.createStrokedShape()
MarlinFX Prism integration

- MarlinFX provides **software rasterization** (AA & non AA):
  - System property prefix ’sun.java2d.marlin’ → ’javafx.prism.marlin’

- **Double-precision variant to improve accuracy** on very large shapes (stroke / dashes)

- **BaseShaderGraphics**, **SWContext** ← **SWGraphics** or **CachingShapeRep**:
  - Calls **ShapeUtil.rasterizeShape(shape, stroke...)**
  - to get **MaskData** information (cacheable)

- **Integrated in OpenJFX-9** (dec 2016) but **disabled by default**
My feedback on contributing to OpenJFX

- **Few documentation about Prism** (pipelines, shaders) or rendering internals (Shape, Canvas ...)
- **OpenJFX patches were more quickly integrated** into OpenJFX-9 (Jim Graham, again) as only the integration layer is different
- **Code duplication** between Marlin (Java2D) & MarlinFX (JavaJFX) to deal with different interfaces:
  - More branches to merge / sync the code among github, OpenJDK & OpenJFX (9 & 10)

- Note: JavaFX Shape node uses the Area class (CPU & memory issues)
MarlinFX usage & benchmarks
How to use MarlinFX?

MarlinFX is available in all JDK-9 builds (Oracle, Zulu, Gluon):

- Enable MarlinFX using prism settings:

```java
java -Dprism.marlinrasterizer=true ...
```

- Select Marlin Double or Float variant:

```java
java -Dprism.marlinrasterizer=true -Dprism.marlin.double=true ...
```

- To enable the Prism log, add `-Dprism.verbose=true`:

```plaintext
Prism pipeline init order: es2
Using Marlin rasterizer (double)
```
DemoFX benchmarks

- OpenJFX-9 b146 **first results with DemoFX Triangle test [VSYNC]**

  58 fps vs 48 fps = 20% gain
MapBenchFX benchmarks

- **MapBenchFX**: JavaFX application to render maps
  - Port of the Java2D MapBench benchmark using FXGraphics2D
  - [https://github.com/bourgesl/mapbench-fx](https://github.com/bourgesl/mapbench-fx)
MarlinFX vs Native & Java Pisces (Oracle JDK-9 EA b181)

- **Native Pisces** & **Java Pisces** & **MarlinFX**

![Bar Chart]

- `fxdemo9_native-pisces.log` • `fxdemo9_java-pisces.log` • `fxdemo9_marlinD.log`
MarlinFX vs Native & Java Pisces (Oracle JDK-9 EA b181)

- MarlinFX gains over Native Pisces & Java Pisces

MarlinFX renderer is the winner (on linux)

- vs Native Pisces: **65%** (average), up to 155%
- vs Java Pisces: **70%** (average), up to 170%
Other Java 9 improvements

JDK-9 bug fixes from http://bugs.openjdk.java.net

- **JDK-6488522 PNG writer** should permit setting compression level
  → *set compression level to medium* [4 vs 9]

- **JDK-8078464 Path2D storage growth algorithms should be less linear**

- **JDK-8084662** Path2D copy constructors and clone method propagate size of arrays from source path

- **JDK-8074587** Path2D copy constructors do not trim arrays

- **JDK-8078192** Path2D storage trimming

- **JDK-8169294** JFX Path2D storage growth algorithms should be less...

- **JDK-8178521** Severe performance drop for path rendering (HW)
Perspectives and Future Work
We want You

I may still have spare time to improve Marlin... but not alone!

We want You = your help is needed:

• Try your applications & use cases with the Marlin renderer
• Contribute to the Marlin project: let’s implement new algorithms (gamma correction, clipping ...)
• to OpenJFX to improve the rendering pipeline (Path, shaders...)
• Adopt the Java Client group?
• Provide feedback, please!
• Properly Handle the gamma correction:
  • Very Important for visual quality
  • In MaskFill for images (C macros) & GPU back-ends
  • Port SW pixel loops to use new Java Vector instructions (Panama)?
  • Note: Stroke’s width must compensate for gamma correction to avoid having thinner / fatter shapes.
• **Analytical pixel coverage**: using signed area coverage for a trapezoid
⇒ compute the exact pixel area covered by the polygon
Performance ideas

- **Clipping** *(coming in release 0.8):*
  - Implement early efficient path clipping (major impact on dashes)
  - Take care of affine transforms (margin), stroke’s cap & joins

- **Scan-line processing** *(8x8 sub-pixels):*
  - 8 scan-lines per pixel row $\Rightarrow$ compute exact area covered in 1 row
  - see [http://nothings.org/gamedev/rasterize/](http://nothings.org/gamedev/rasterize/)
  - may be almost as fast but a lot more precise!

- Cap & join processing (Stroker):
  - Do not emit extra collinear points for squared cap & miter joins

- Optimize the **Area** class (allocation + CPU)?
Perspectives and Future Work

First results of the path clipping in Marlin 0.8.1
Performance impact of the path clipping (Marlin 0.8.1)

- Marlin path clipper Off vs On [clip 400 x 400]:

![Graph showing performance impact of path clipping]

- big_test_clip_off.log
- big_test_clip_on.log
Performance impact of the path clipping (Marlin 0.8.1)

- Gain with Marlin path clipper enabled:
  - 25% gain (average)
  - up to 65% gain (complex shapes)
Conclusion
Conclusion

• Marlin renderer rocks in Java 9!
  • Performance & Quality goals met
    • Single threaded performance generally as good or better
    • Scales for multiple threads whereas Ductus does not

• Marlin is now the default AA renderer for OpenJDK-9
• MarlinFX is the default renderer in OpenJFX-10
• Coming patches are promising for OpenJDK-10...
• Optimizing JavaFX Graphics performance needs you (GPU)!
• What future for Java Client & Desktop?
  ⇒ Contribute to Client or OpenJFX groups
That’s all folks!

- Please ask your questions or talk with me
- or send them to marlin-renderer@googlegroups.com

Special thanks to:

- Andréa Aimé & GeoServer team
- OpenJDK teams for their help, reviews & support:
  - Jim Graham & Phil Race (Java2D)
  - Mario Torre & Dalibor Topic
  - Mark Reinhold
- ALL Marlin users
Special Thanks to all these people & organizations:

- GeoServer Project Steering Committee
- Gluon Software
  - Thanks for all that you do to make JavaFX an awesome UI library
- Dirk Lemmermann
  - What would I do without people like you? So here you go!
- AZUL SYSTEMS
- OpenJDK members: Mark Reinhold, Jim Graham, Phil Race, Kevin Rushforth
- Manuel Timita
  - Your work means a lot to us!
- H. K.
• **Andréa Aimé**
  - Laurent helped solving a significant performance/scalability problem in server side Java applications using java2d. Let’s help him reach JavaOne and talk about his work!

• **Chris Newland**
  - Laurent has made a big contribution to OpenJDK with his Marlin and Marlin-FX high performance renderers. Let’s help him get to JavaOne 2017!

• **Simone Giannecchini**
  - Z. B.

• **JUGS e.V.**
  - Brad Hards
  - Mario Ivankovits
  - Peter-Paul Koonings (GeoNovation)
JavaOne Sponsors (gofundme)

- Michael Paus
- Bart van den Eijnden (osgis.nl)
- Mario Zechner
- Jody Garnett
- Tom Schindl
- Thea Aldrich
- Cédric Champeau
- Ondrej Mihályi
- Thomas Enebo
- Hiroshi Nakamura

- Reinhard Pointner
- Adler Fleurant
- Paul Bramy
- Sten Nordström
- Chris Holmes
- Michael Simons
- P. S.
- A. B.
- Igor Kolomiets
- Vincent Privat
• Carl Dea
  • I’m not going to JavaOne, but hearing about Laurent Bourgès making Java 2D faster caught my attention. Graphics and performance engineers are true heroes. They make users of the APIs look good, while their work under the covers really do the heavy lifting. Go Laurent!
• Mike Duigou
  • Would love to see JavaOne sponsor opensource committers in future years
• M. H.
• J. M.
Extra
Extra

Comparing Marlin renderer releases using JDK-8
Performance evolution between Marlin 0.3 & 0.8.1 (JDK-8)

- Marlin 0.3 (2014.1) & 0.5.6 (2015) & 0.7.4 (JDK-9) & 0.8.1 (2017)
Performance evolution between Marlin 0.3 & 0.8.1 (JDK-8)

• Marlin 0.5.6 & 0.7.4 & 0.8.1 vs 0.3

Marlin performance improved through releases

• Loss in 0.5.6 (large ellipse fills) fixed in 0.7
• Important improvements on complex stroked shapes (spirals)
Extra

Performance impact of the sub-pixel tuning
Performance impact of the sub-pixel tuning

- Sub-pixel settings [1x1 to 6x6]:

![Graph showing performance impact of sub-pixel tuning]
Performance gain & loss of the sub-pixel tuning

- Gain & Loss of Sub-pixel settings [1 x 1 to 6 x 6] VS [3 x 3]:

![Graph showing performance gain & loss for different test cases with sub-pixel settings]

- jdk9_marlin_sp1.log - jdk9_marlin_sp2.log - jdk9_marlin.log - jdk9_marlin_sp4.log
Extra

Studying Marlin & Java2D internal stages
Marlin internal timings

- 3 stages in Java2D pipeline: Path Processing → Rendering → Blending

![Graph showing Marlin internal timings with stages: Path Processing, Rendering, Blending.](image-url)
Marlin internal timings

- 3 stages in Java2D pipeline: Path Processing → Rendering → Blending

Stage cost ratios

- Path processing: 20 to 30% (stroked shapes) but 3% (filled)
- Blending: 25% (average) but up to 75% (large ellipse fills): Future work?