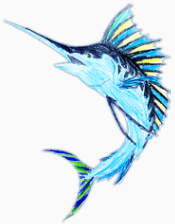
JavaOne™


JavaOne™

Marlin renderer, a JDK9 Success Story

Vector Graphics on Steroids for Java 2D and JavaFX



Laurent Bourgès

JavaOne 2017-10-04

OSUG

Observatoire des Sciences de l'Univers de Grenoble

cnrs

OpenJDK

JavaOne™

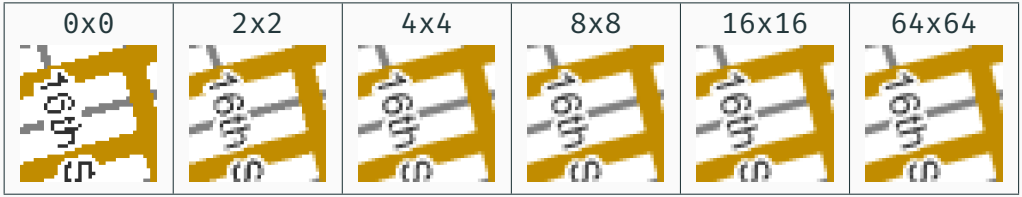
Outline

MarlinFX

Context & History

Role of an Antialiasing 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



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(), closeTo()`
- `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

Background (before Marlin)

Antialiasing 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
  - `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: all of these are AA rendering only. Other renderers for non-AA cases

Marlin renderer = OpenJDK's Pisces fork

Status in 2013:

- Ductus: faster but scalability issue (multi-threading)
- Pisces: slower but better scalability (GPL)
- GPU ? Java2D OpenGL & D3D pipelines provide only few accelerated operations (blending)
- March-Mai 2013: my first patchs to OpenJDK 8:
  - Pisces patchs to `2d-dev@openjdk.java.net`: too late
  - Small interest / few feedbacks
- Andréa Aimé (GeoServer team) pushed me to go on:
  - New MapBench tool: serialize & replay rendering commands
  - Fork OpenJDK's Pisces as a new open-source project

Marlin renderer @ github

⇒ 01/2014: **Marlin renderer** & MapBench projects on github (GPL v2) with only 2 contributors (Me and Andrea Aimé) !

- `https://github.com/bourgesl/marlin-renderer`
  - branch 'use_Unsafe': trunk
  - branch 'openjdk': in synch with OpenJDK9
- `https://github.com/bourgesl/mapbench`

TODO: add branches

Marlin & MapBench projects @ github

Objectives:

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

Very big personal work:

- Many releases in 2014: see releases
- Test Driven Development:
  - Regression: MapDisplay (diff pisces / marlin outputs)
  - Performance: MapBench & GeoServer benchmarks (+ oprofile)
- Important feedback within the GIS community (GeoServer) providing complex use cases & testing releases

Marlin Release history @ github

- Releases: see releases

TODO: table of major features

- Marlin 0.7: improve coordinate rounding arround subpixel center
- Marlin 0.7.2: improve large pixel chunk copies (coverage data)

Marlin renderer back into OpenJDK 9

- Late 2014: several mails to `2d-dev@openjdk.java.net`
- FOSDEM 2015: discussion with OpenJDK managers (Dalibor & Mario) on how to contribute the Marlin renderer back

$\Rightarrow$ I joined the graphics-rasterizer project in march 2015 to contribute Marlin as a new standalone renderer for OpenJDK9.

- **I worked hard** (single coder) with Jim Graham & Phil Race (reviewers) between march 2015 to december 2015 (4 big patches)

JEP 265: Marlin renderer back into OpenJDK 9

We proposed the JEP 265 in July 2015 and make it completed !

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

- Marlin was integrated in OpenJDK9 b96 (dec 2015) and got enhancements in 2016.
- Current integrated releases within OpenJDK:
  - JDK9 = Marlin 0.7.4
  - JDK10 = Marlin 0.7.5

My feedback on contributing to OpenJDK

- Very interesting & many things learnt
- 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 `cr.openjdk.java.net/~<mylogin>/`
  - long discussions on mailing lists for my patches (50 mails)
  - timezone issue: delays + no skype
- Few Java2D / computer graphics skills = small field + NO DOC !

My feedback on contributing to OpenJDK

General:

- Missing contributors into Graphics stack !
- CI: missing ’open’ multi-platform machines to perform tests & benchmarks outside of Oracle
- Funding community-driven effort ? support collaboration with outsiders

How Marlin works ?

How the Java2D pipeline works ?

Java2D uses only 1 RenderingEngine implementation at runtime:

- SunGraphics2D.draw/fill(shape)
- AAShapePipe.renderPath(shape, stroke)
  - aatg = RenderingEngine.getAATileGenerator(shape, at)
    - Coverage mask computation (tiles) as alpha transparency [0-255]
  - aatg.getAlpha(byte[] alpha, ...) to get next tile ...
  - output pipeline.renderPathTile(byte[] alpha):
    - MaskFill operations (software / OpenGL pipeline) on dest surface

```
1 RenderingEngine:
2     public static synchronized RenderingEngine getInstance();
3     public AATileGenerator getAATileGenerator(Shape s,
4                                 AffineTransform at, ...);
5  AATileGenerator:
```

How Marlin works ? Pisces / Marlin pipeline

MarlinRenderingEngine.getAATileGenerator(shape, stroke...):

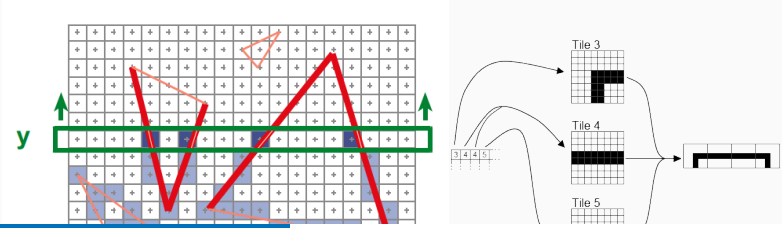
- use shape.getPathIterator() ⇒ apply the pipeline to path elements:
- Dasher (optional):
  - generates path dashes (curved or segments)
- Stroker (optional):
  - generates edges arround of every path element
  - generates edges for decorations (cap & joins)
- Renderer:
  - curve decimation into line segments
  - addLine: basic clipping + convert float to subpixel coordinates
  - determine the shape bounding box
  - perform edge rendering into tile strides ie compute pixel coverages
  - fill the MarlinCache with pixel coverages as byte[] (alpha)

How Marlin works ? the AA algorithm

- Scanline algorithm [8x8 supersampling] to estimate pixel coverages

= Active Edge table (AET) variant with ”java” pointers (integer-based)

- sort edges at each scanline
- estimate subpixel coverage and accumulate in the alpha row
- Once a pixel row is done: copy pixel coverages into cache
- Once 32 (tile height) pixel rows are done: perform blending & repeat !



How Marlin works ? the AA algorithm

- Java2D's internal drawing renders rectangular tiles.
- Marlin starts with the path defining the shape
- Takes the elements of the path and flattens curves
- Uses 8x8 super-sampling to estimate coverage.
- For each sample line calculates intersections.
- Sorts where the path edges cross the scan line
- Output is one byte per pixel representing alpha coverage.
- Process 32 entire pixel rows at a time and cache sample-line intersection info
- Provide 32x32 rectangular tiles needed by the internal Java2D rendering pipelines

Marlin performance optimizations

- Initially GC allocation 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 !
- Marlin RendererContext (TL/CLQ) = reused memory $\Rightarrow$ almost no GC
  - kept by weak / soft reference
  - class instances + initial arrays takes 512Kb
  - weak-referenced array cache for larger arrays
- Other considerations:
  - Unsafe: allocate/free memory + less bound checks
  - zero-fill (recycle arrays) on used parts only !
  - use dirty arrays when possible: C like !

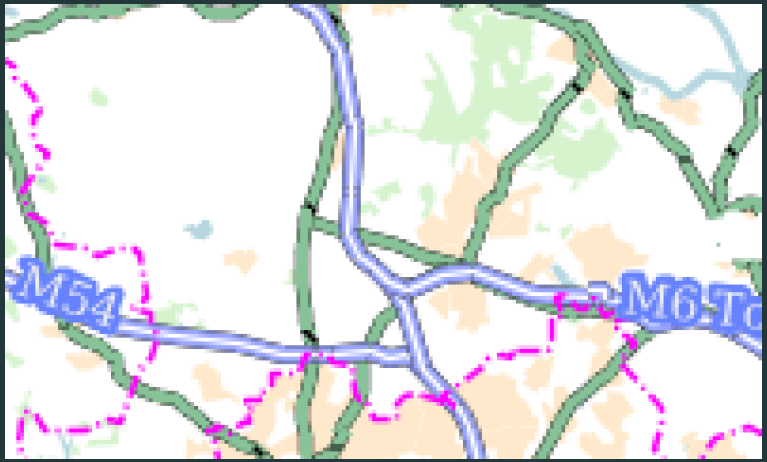
Marlin performance optimizations

- Need good profiler: use oprofile + gather 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) $\Rightarrow$ adaptive approach
  - Optimized Pixel copies (block flags)

A lot more ...

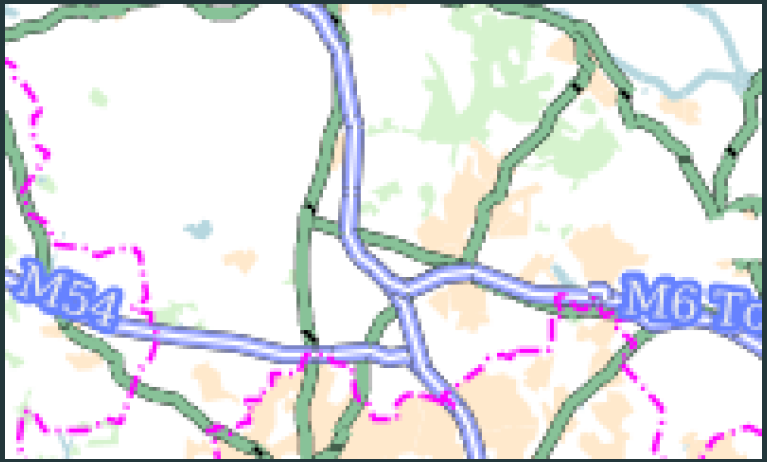
Marlin visual quality (Blind Test)

Who is [Ductus, Pisces, Marlin] ?



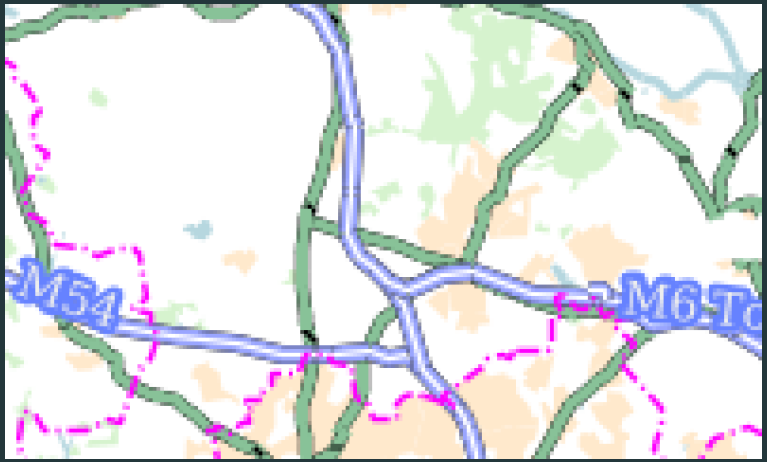
Marlin visual quality (Blind Test)

Who is [Ductus, Pisces, Marlin] ?



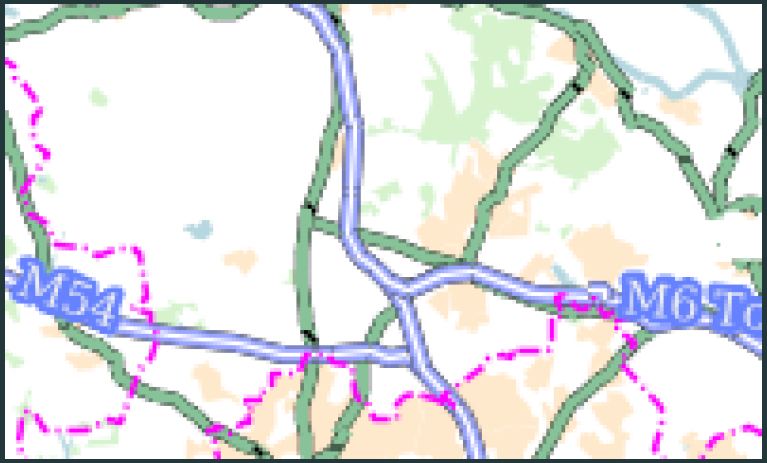
Marlin visual quality (Blind Test)

Who is [Ductus, Pisces, Marlin] ?



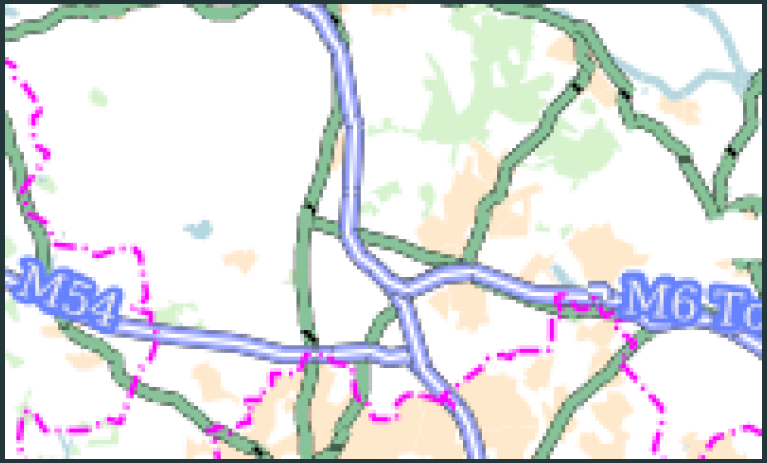
Marlin visual quality (Blind Test)

Answer: Ductus



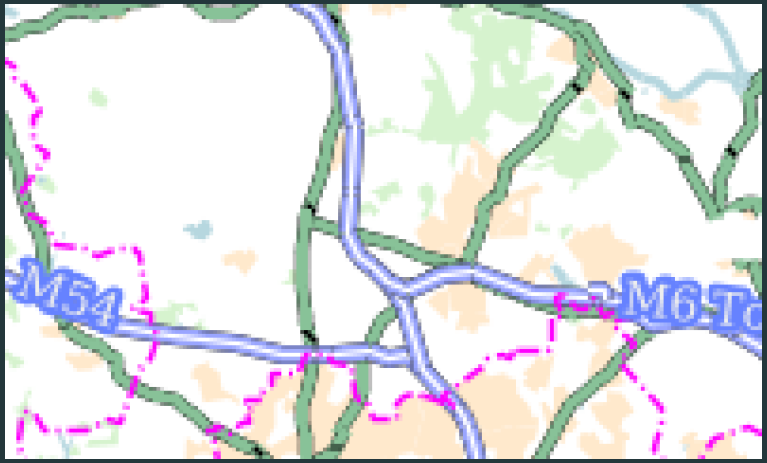
Marlin visual quality (Blind Test)

Answer: Marlin



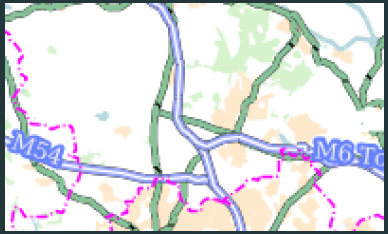
Marlin visual quality (Blind Test)

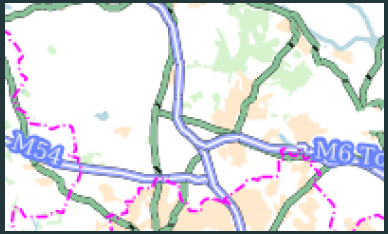
Answer: Pisces

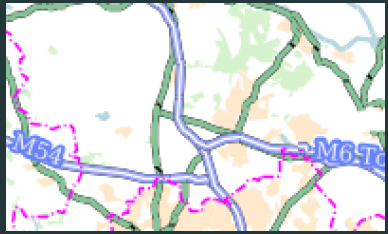


Marlin visual quality (Blind Test)

Who is [Ductus, Pisces, Marlin] ?







Ductus **Marlin** **Pisces**

$\Rightarrow$ **Quite an hard challenge !**

Marlin usage & benchmarks

How to use Marlin ?

- Just download any Oracle or OpenJDK-9 release
  - `https://jdk9.java.net/`
- For JDK 1.7 or JDK-8:
  - Download the latest Marlin release @ github
- Howto: `https://github.com/bourgesl/marlin-renderer/wiki/How-to-use`
- Start your java program with:

```
java -Xbootclasspath/a:../lib/marlin-0.7.5-Unsafe.jar
     -Dsun.java2d.renderer=sun.java2d.marlin.MarlinRenderingEngine
     ...
```

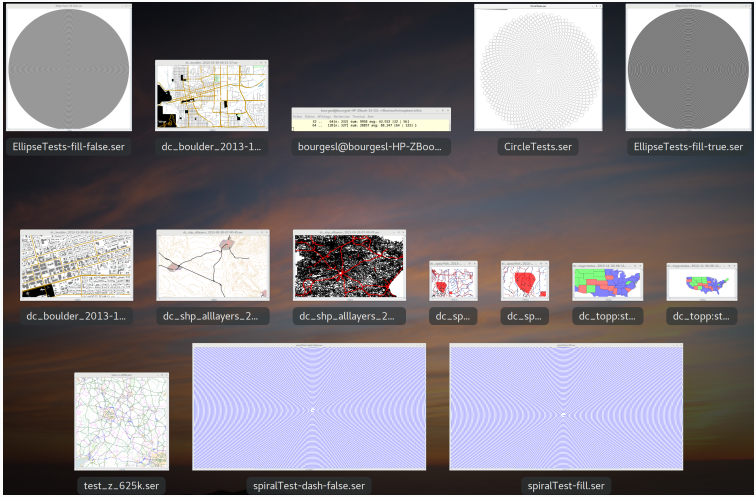
MapBench benchmarks

- MapBench tool:
  - Multi-threaded java2d benchmark that replays serialized graphics commands: see `ShapeDumperGraphics2D`
  - Calibration & warmup 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)
  - Setup JVM: select JDK + basic jvm settings = CMS GC 2Gb Heap
  - Use the profile 'shared images' to reduce GC overhead

$\Rightarrow$ Reduce variability (and cpu affinity issues)

MapBench tests

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

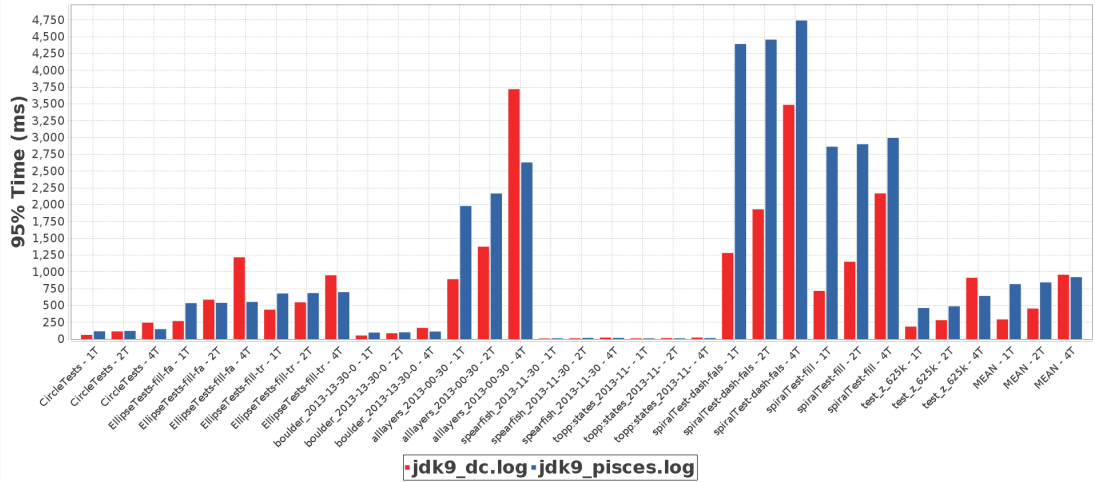


Marlin usage & benchmarks

Comparing AA renderers in JDK-9 (Ductus, Pisces, Marlin)

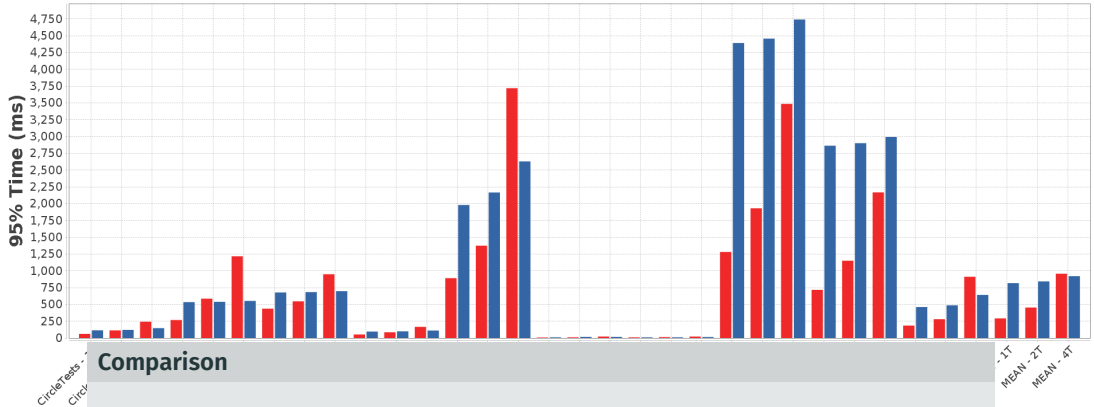
Before Marlin (Oracle JDK-9 EA b181)

- OracleJDK Ductus (red) & Pisces (blue)



Before Marlin (Oracle JDK-9 EA b181)

- OracleJDK Ductus (red) & Pisces (blue)

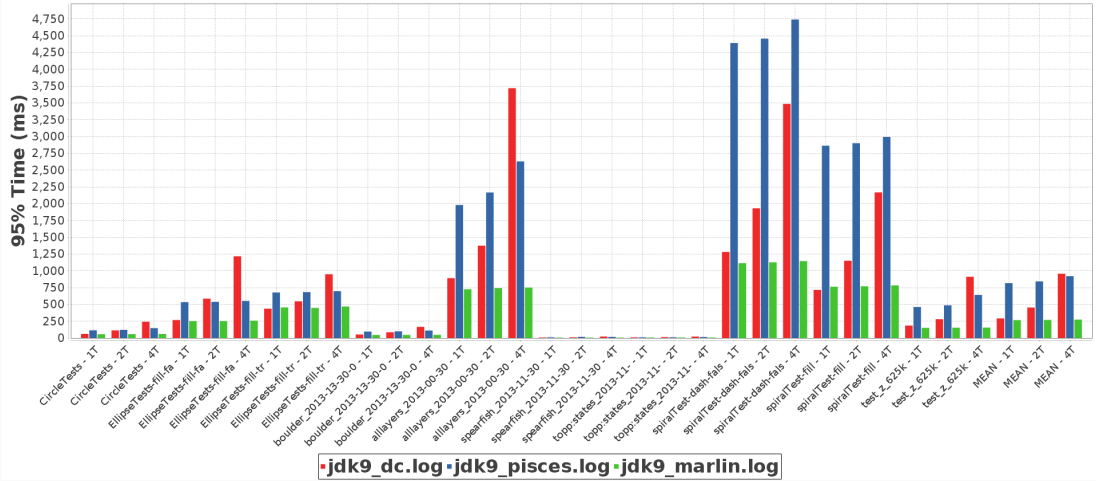


Comparison

- Ductus: faster (1T) but scalability issue !
- Pisces: slower but higher scalability

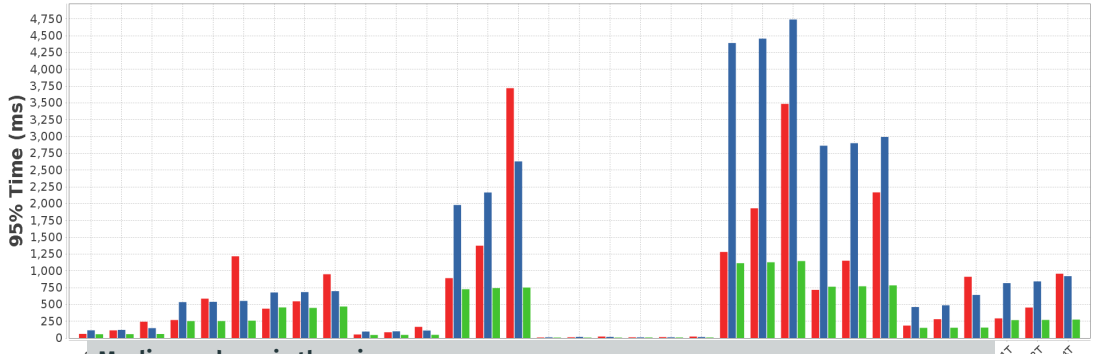
With Marlin (Oracle JDK-9 EA b181)

- OracleJDK Ductus (red) & Pisces (blue) & Marlin (green)



With Marlin (Oracle JDK-9 EA b181)

- OracleJDK Ductus (red) & Pisces (blue) & Marlin (green)

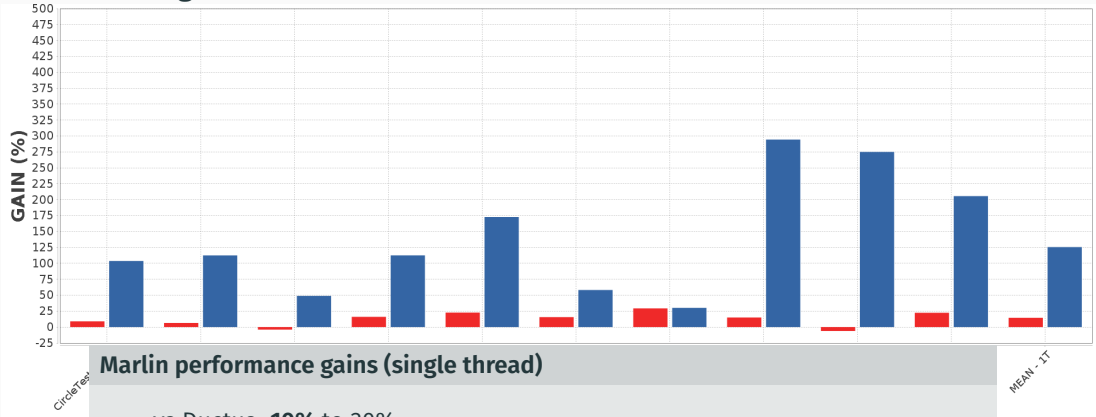


Marlin renderer is the winner

- Better performance compared to Ductus (or equal) or Pisces
- Perfect scalability

Performance gains (1 thread)

- Marlin gains over OracleJDK Ductus (red) & Pisces (blue)

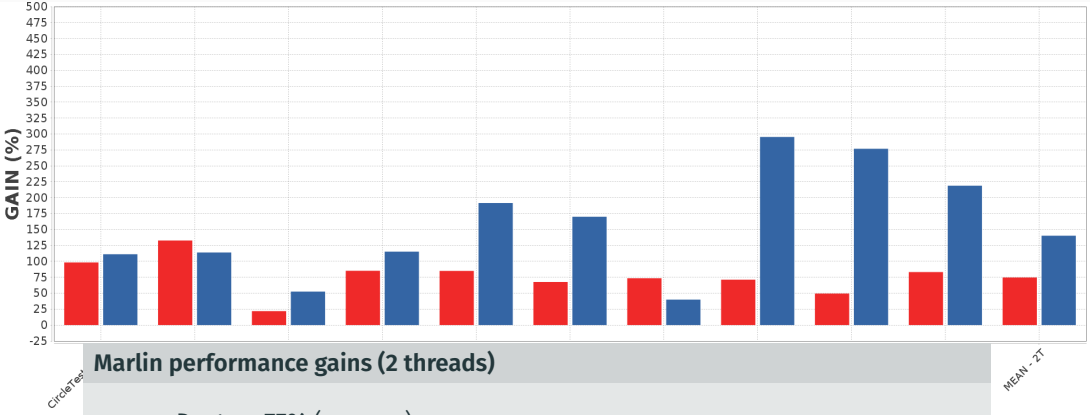


Marlin performance gains (single thread)

- vs Ductus: **10%** to 30%
- vs Pisces: **125%** (average), up to 300% (stroked spiral test)

Performance gains (2 thread)

- Marlin gains over OracleJDK Ductus (red) & Pisces (blue)

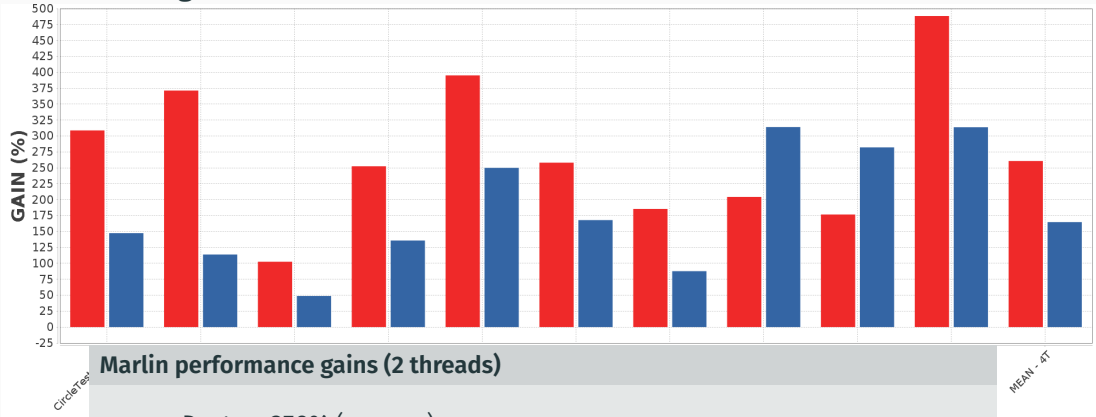


Marlin performance gains (2 threads)

- vs Ductus: **75%** (average)
- vs Pisces: **140%** (average)

Performance gains (4 thread)

- Marlin gains over OracleJDK Ductus (red) & Pisces (blue)

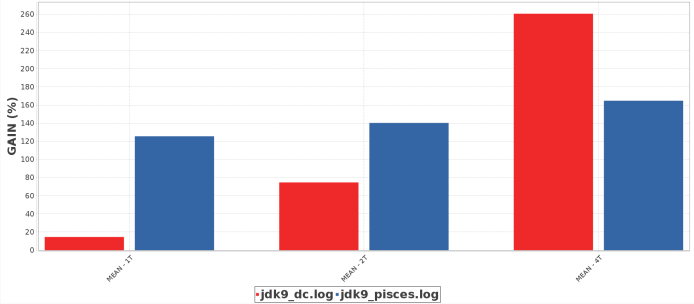


Marlin performance gains (2 threads)

- vs Ductus: **250%** (average)
- vs Pisces: **160%** (average)

Marlin Performance Summary (JDK-9)

- 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

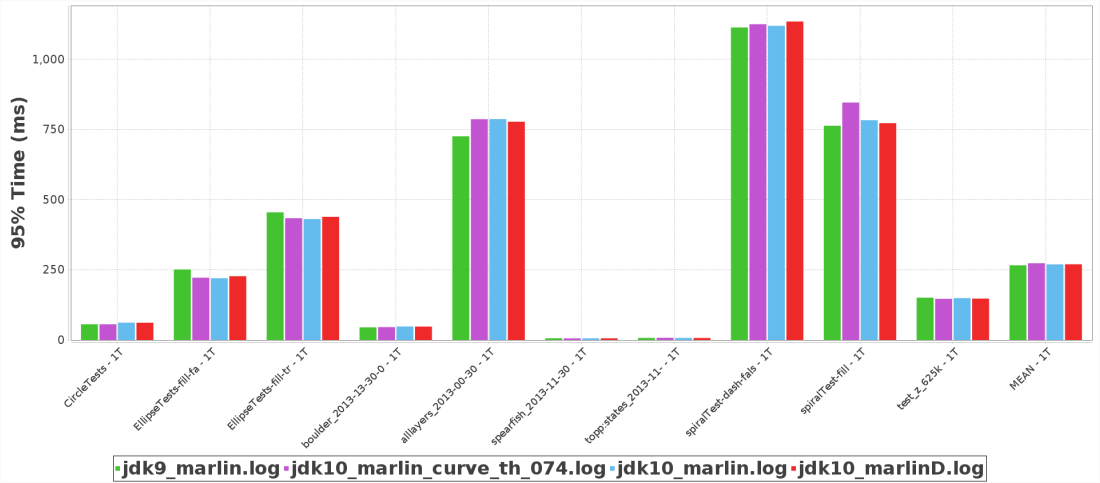


Marlin usage & benchmarks

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)



Marlin Performance changes (JDK-9 vs OpenJDK-10)

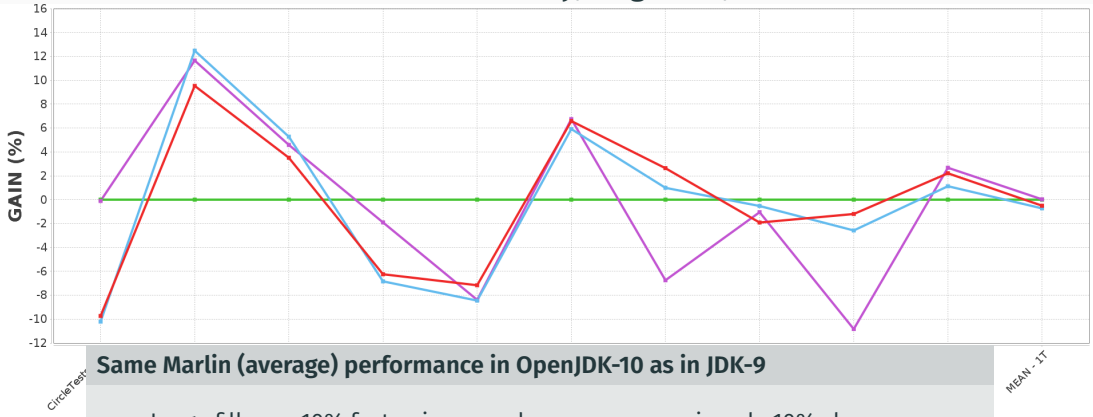
- Marlin 0.7.5 vs 0.7.4 (curve accuracy, large fills, Double variant)

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

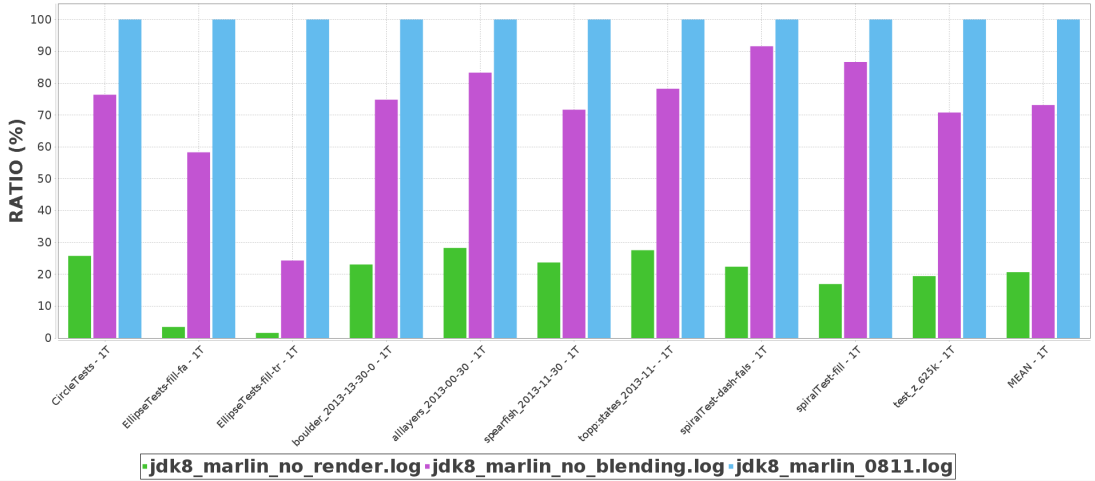
- Large fills are 10% faster, improved curve accuracy is only 10% slower
- **No difference between Float and Double Marlin variants**

Marlin usage & benchmarks

Studying Marlin & Java2D internal stages

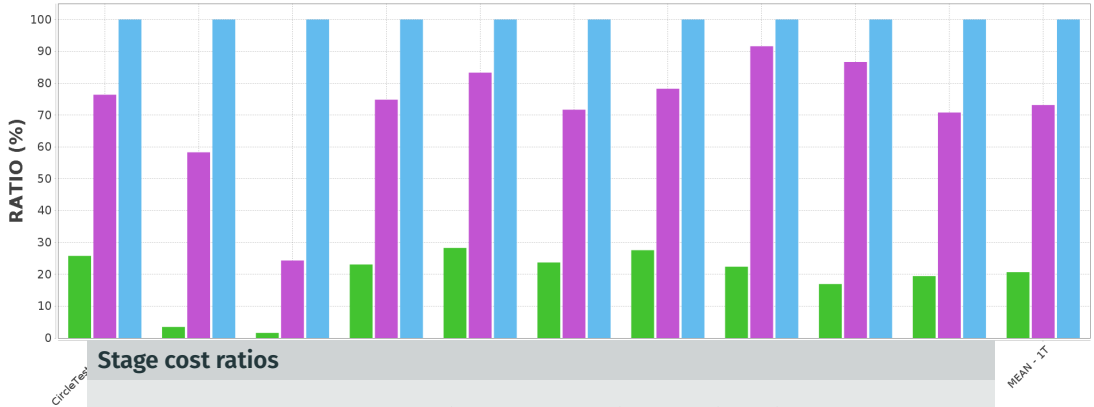
Marlin internal timings

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



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 ?*

Marlin renderer tuning

Marlin renderer tuning

Marlin can be customized by using system properties:

- adjust subpixel sampling:
  - X/Y=3: [8x8] (by default)
  - smaller values are faster but less accurate
  - higher values are slower but more accurate
- pixel sizing: typical largest shape width / height (2048 by default)
- adjust tile size: 6 [64x64] seems better than 5 [32x32]

Debugging:

- log statistics to know what happens
- enable checks if segfault or artefacts !

Marlin System properties

System property -Dsun.java2d.renderer.<KEY>=	Values	Description
useThreadLocal	**true** - false	RdrCtx in TL or CLQ (false)
useRef	**soft** - weak - hard	Reference type to RdrCtx
pixelsize	**2048** in [64-32K]	Typical shape W/H in pixels
subPixel_log2_X	**3** in [1-8]	Subpixel count on X axis
subPixel_log2_Y	**3** in [1-8]	Subpixel count in Y axis
tileSize_log2	**5** in [3-8]	Pixel width/height for tiles
doStats	true - **false**	Log rendering statistics
doChecks	true - **false**	Perform array checks
useLogger	true - **false**	Use j.u.l.Logger

Log-2 values for subpixel & tile sizes:

Other Java 9 improvements

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

- JDK-6488522 **PNG writer** should permit setting compression level and iDAT chunk maximum size → **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 linear

MarlinFX

MarlinFX project

2016: port Marlin into JavaFX Prism

- Prism integration
  - Complete mask no more tiles
  - GPU backend is faster (upload texture may be costly)
  - but single-threaded (GUI)
- Double-precision variant to improve accuracy on very large shapes (stroke / dashes)
- Integrated in OpenJFX9 (dec 2016) but disabled by default

MarlinFX System properties

’sun.java2d.marlin’ prefix $\rightarrow$ ’javafx.prism.marlin’ prefix

TODO: complete

My feedback on contributing to OpenJFX

Contrary to Marlin, OpenJFX patches were more quickly integrated into OpenJFX9 (Jim Graham, again) !

TODO: complete

MarlinFX usage & benchmarks

How to use MarlinFX ?

MarlinFX is only available in JDK-9:

- Enable MarlinFX using prism settings:

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

- Select Marlin Double or Float variant:

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

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

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

DemoFX benchmarks

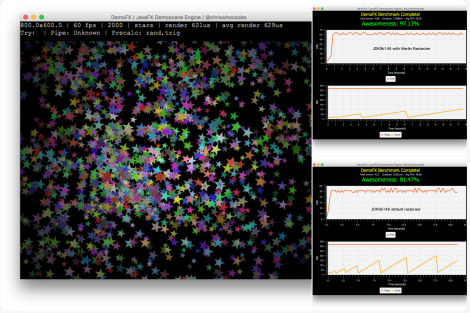
- OpenJFX9 b146 first results with DemoFX Triangle test (VSYNC)

Vous avez retweeté

Chris Newland @chriswhocodes · 28 nov. 2016

The new MarlinFX rasterizer for #JavaFX in JDK9 EA b146 is fast and uses less memory! Awesome work by @laurent_bourges :)

À l'origine en anglais

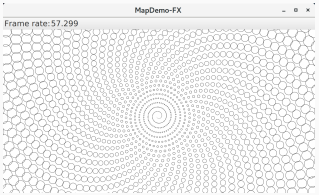


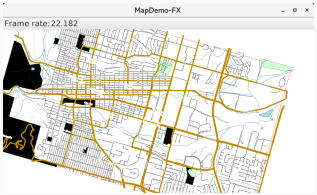
4 28 35

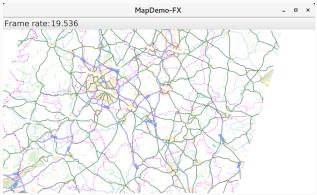
- **58 fps vs 48 fps = 20% gain**

MapBenchFX benchmarks

- MapBenchFX: JavaFX application to render maps
- Port of the Java2D benchmark using FXGraphics2D

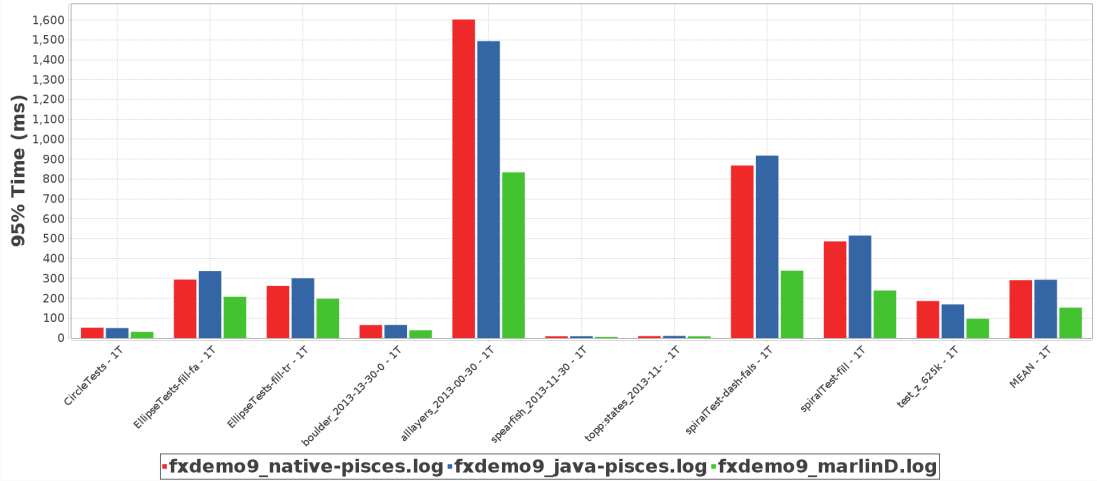






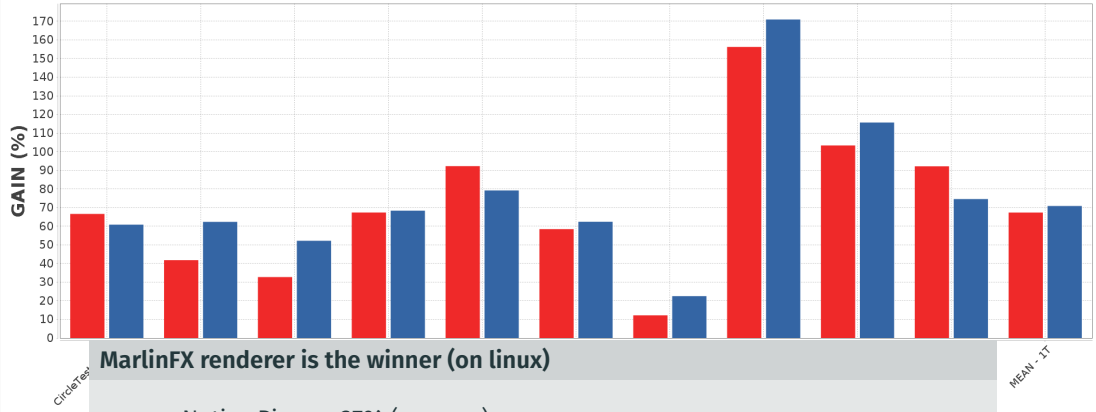
MarlinFX vs Native & Java Pisces (Oracle JDK-9 EA b181)

- Native Pisces (red) & Java Pisces (blue) & MarlinFX (green)



MarlinFX vs Native & Java Pisces (Oracle JDK-9 EA b181)

- Native Pisces (red) & Java Pisces (blue) & MarlinFX (green)



MarlinFX renderer is the winner (on linux)

- vs Native Pisces: **65%** (average)
- vs Java Pisces: **70%** (average)

Future work

Future work

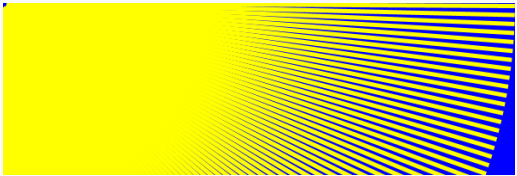
I may have still spare time to improve Marlin...

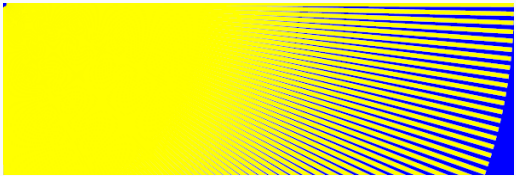
But your help is needed:

- Try your applications & use cases with the Marlin renderer
- Contribute to the Marlin project: let's implement new algorithms (gamma correction, clipping ...)
- Provide feedback, please !

Quality Ideas

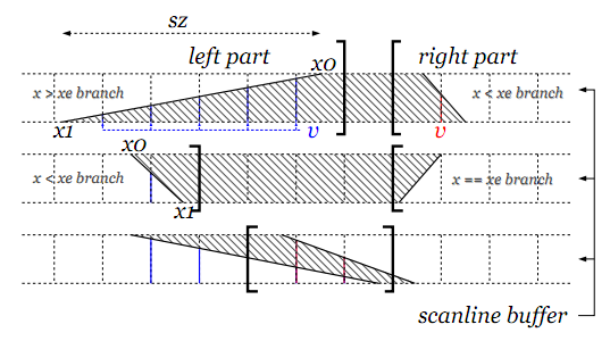
- **Handle properly the gamma correction**:
  - in MaskFill for images (C macros) & GPU backends
  - **Very important for visual quality**
  - note: Stroke's width must compensate the gamma correction to avoid having thiner / fatter shapes.





Quality Ideas

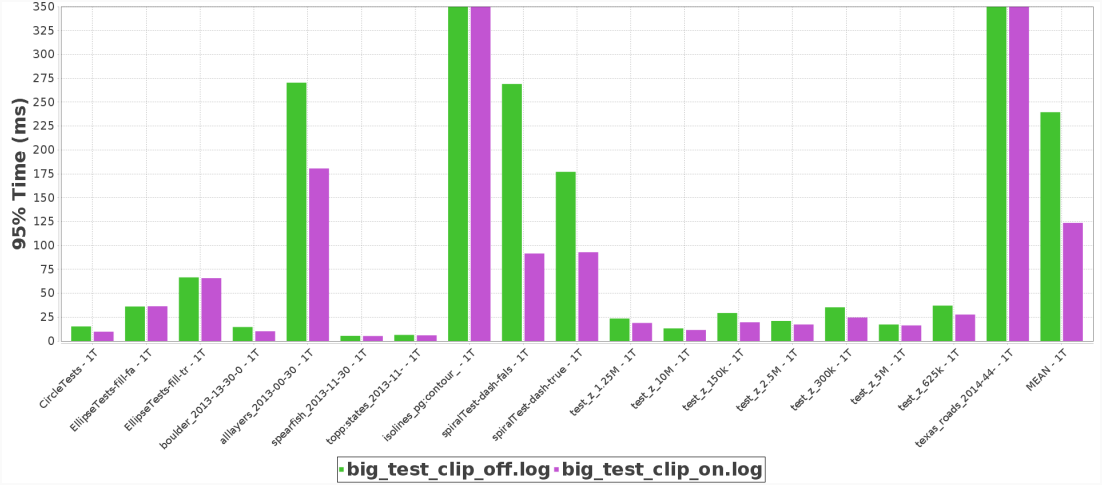
- Analytical pixel coverage: using signed area coverage for a trapezoid
  $\Rightarrow$ compute the exact pixel area covered by the polygon



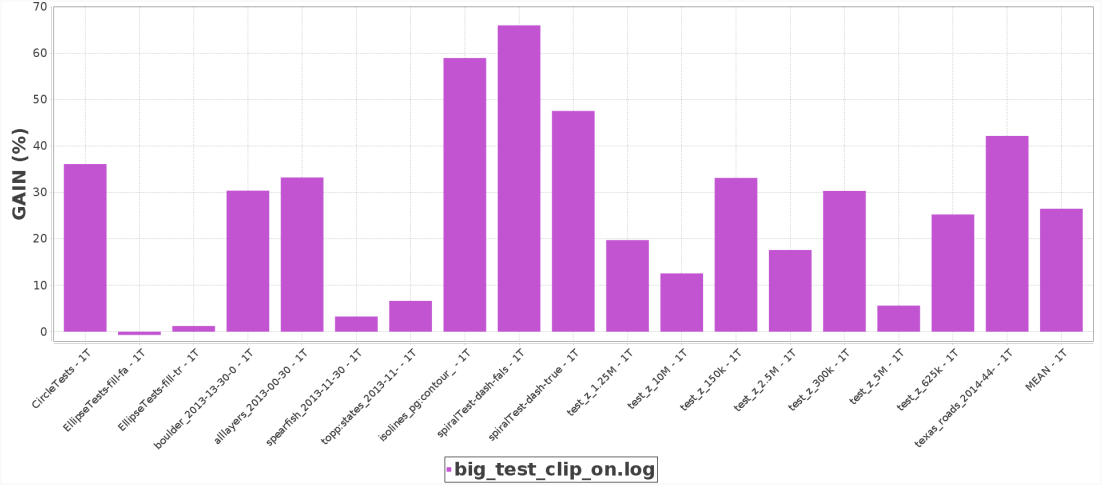
Performance ideas

- Clipping (coming in release 0.8):
  - implement early efficient path clipping (major impact on dashes)
  - take care of affine transforms (margin, not always rectangle)
- Cap & join processing (Stroker):
  - do not emit extra collinear points for squared cap & miter joins
  - improve Polygon Simplifier ?
- Scanline processing (8x8 subpixels):
  - 8 scanlines per pixel row $\Rightarrow$ compute exact area covered in 1 row
  - see algorithmic approach (AGG like): `http://nothings.org/gamedev/rasterize/`
  - may be almost as fast but a lot more precise !

EA Marlin 0.8.1 = Path clipping



EA Marlin 0.8.1 = Path clipping



Conclusion

Conclusion

- Marlin rocks in Java 9!
- Performance goals were met
  - Scales for multiple threads whereas Ductus does not
  - Single threaded performance generally as good or better for complex shapes.
- Quality goals were met.
- Marlin is now the default AA renderer for Oracle JDK (and OpenJDK).
- Implementation, not API. Benefit accrues automatically to apps.

That's all folks !

- Please ask your questions
- 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

JavaOne Sponsors (gofundme)

Special Thanks to all my donators:

- **GeoServer Project Steering Comittee**
- **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.

JavaOne Sponsors (gofundme)

- **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

JavaOne Sponsors (gofundme)

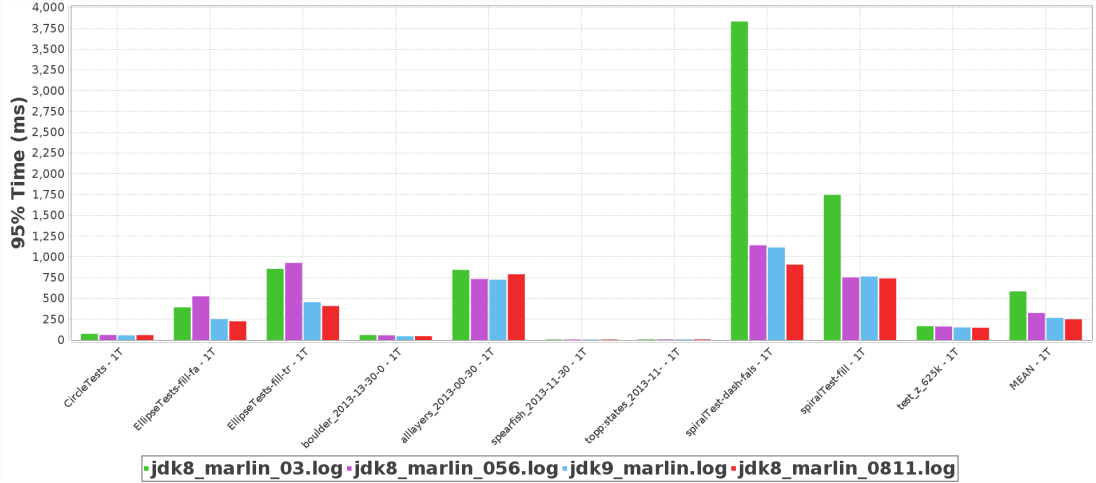
- Carl Dea
  - I'm not going to JavaOne, but hearing about Laurent Bourgès making Java 2D faster caught my attention. Graphics and performance engineers are true heroes. They make users of the APIs look good, while their work under the covers really do the heavy lifting. Go Laurent!
- Mike Duigou
  - Would love to see JavaOne sponsor opensource committers in future years
- M. H.
- J. M.

Extra

Comparing Marlin renderer releases using JDK-8

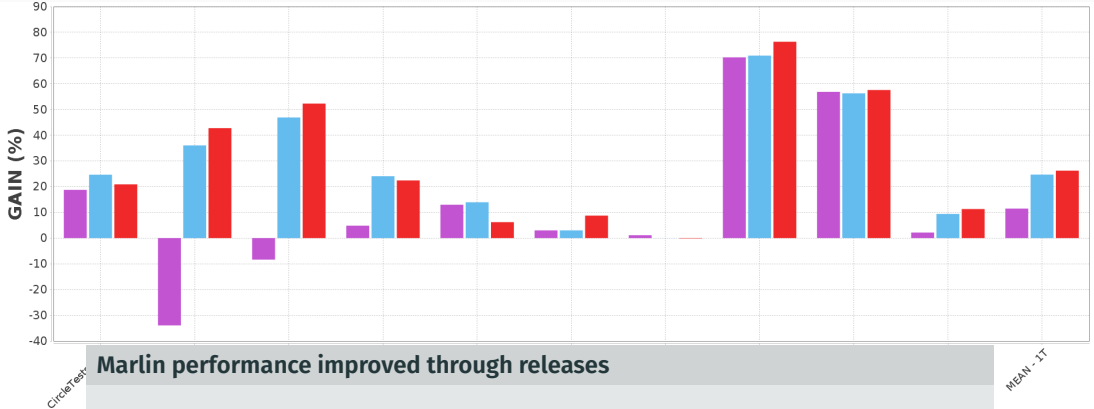
Marlin Performance evolution between 0.3 & 0.8.1 (JDK-8)

- Marlin 0.3 (2014.1) & 0.5.6 (2015.2) & 0.7.4 (JDK-9) & 0.8.1 (2017)



Marlin Performance evolution between 0.3 & 0.8.1 (JDK-8)

- Marlin 0.5.6 (2015.2) & 0.7.4 (JDK-9) & 0.8.1 (2017) vs 0.3 (2014.1)



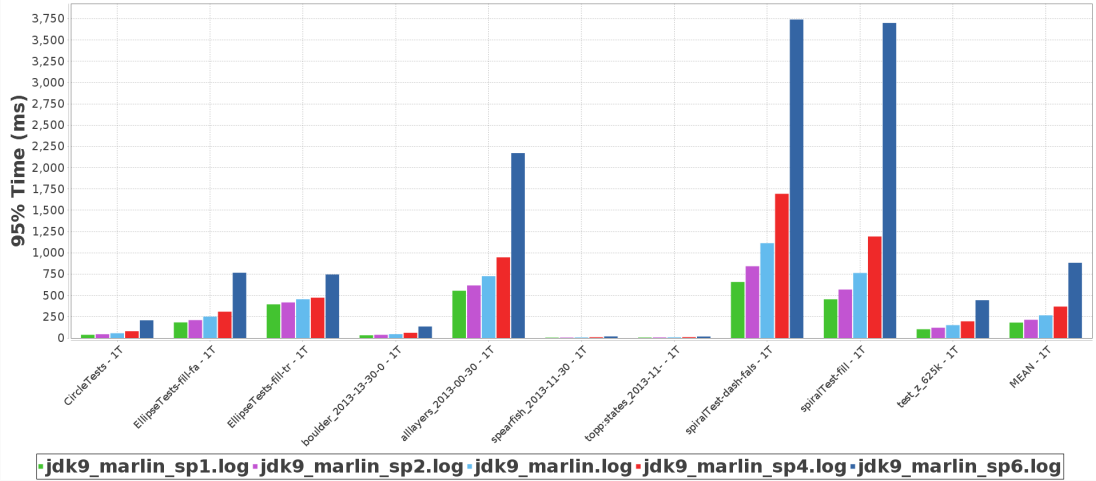
Marlin performance improved through releases

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

Performance impact of the subpixel tuning

Performance impact of the subpixel tuning

- Subpixel settings (1x1 to 6x6):



Performance impact of the subpixel tuning

- Gain & Loss of Subpixel settings (1x1 to 6x6) VS 3x3: