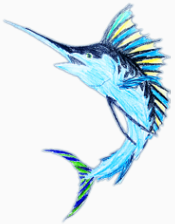
JavaOne™


JavaOne™

Marlin renderer, a JDK-9 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

Context & History

How Marlin works ?

Visual quality Quiz

Marlin usage & benchmarks

MarlinFX

MarlinFX usage & benchmarks

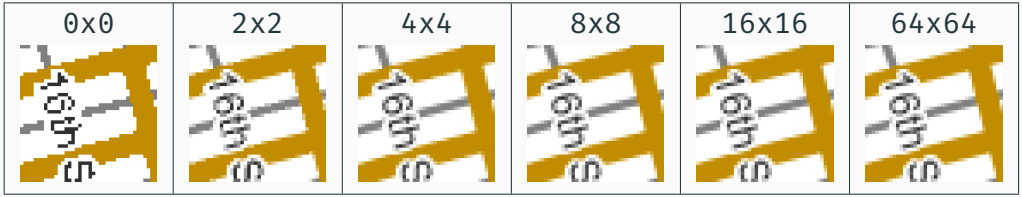
Perspectives and Future Work

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**



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)

Anti-aliasing 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: **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**
- New **MapBench** tool: serialize & replay rendering commands

Marlin renderer @ github

⇒ 2014.1: **Marlin renderer** & MapBench projects @ github (GPL v2)

- 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

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:

- **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

Marlin renderer back into OpenJDK-9

- **FOSDEM 2015**: Great 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 OpenJDK-9**

- **I worked really hard** (again, single developer) with Jim Graham & Phil Race (reviewers) in 2015 to push 4 big patches $\approx$ **10,000 LOC** !
- Reviews improved a lot the code and Computer Graphics algorithms

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

JDK-9	Marlin 0.7.4
JDK-10	Marlin 0.7.5

Marlin Releases @ github

Major Releases:

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

Marlin Releases @ github (continued)

Major Releases:

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

Feedback on contributing to OpenJDK

- **Very interesting experience** & 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 `http://cr.openjdk.java.net/~lbourges/`
  - Long review threads on mailing lists for my patches ($\approx$ 50 mails)
  - Timezone difference: delays + no direct discussion
- **Few Java2D / computer graphics skills**: small field + NO DOC !

Feedback on contributing to OpenJDK (continued)

In General:

- **Missing active contributors in the Graphics stack** (Java2D & JavaFX) !
- CI: 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

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

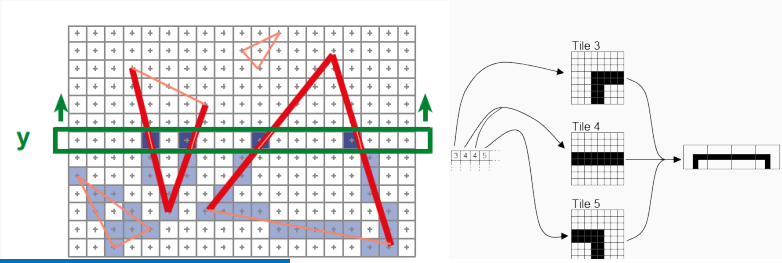
```
1 AATileGenerator:
2    public int getTypicalAlpha();
3    public void nextTile();
4    public void getAlpha(byte tile[], ...);
```

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 of 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 ie 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 edges at each scan-line
- Estimate sub-pixel 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 !



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 !
- `RendererContext` (TL/CLQ) to reuse memory ⇒ almost no GC
  - Kept by weak / soft reference: see `ReentrantContext`
  - Class instances + initial arrays takes 512Kb
  - Weak-referenced **array cache for larger arrays**
- Other considerations:
  - Use **Unsafe**: allocate/free memory + less 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 + oprofile + internal metrics

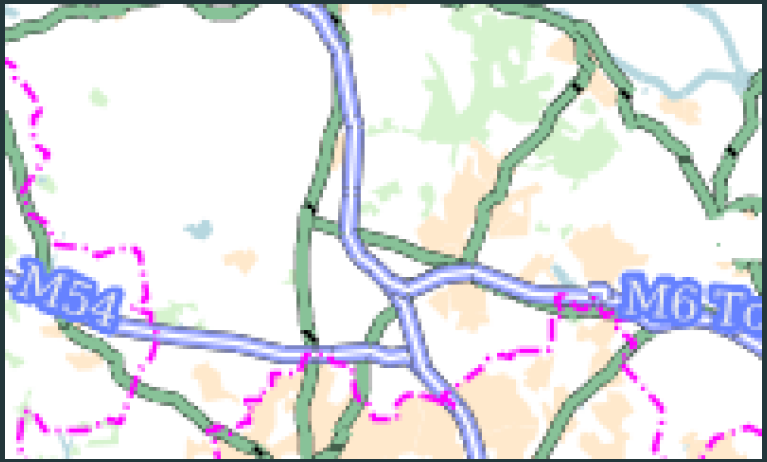
- **Fine tuning** of Pisces algorithms:
  - Optimized **Merge Sort** (in-place)
  - Custom rounding [float to int]
  - **DDA in Renderer** with correct pixel center handling
  - Tile stride approach (32px) instead of all shape mask
  - Pixel alpha transfers (RLE) $\Rightarrow$ adaptive approach
  - **Optimized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

A lot more ...

Visual quality Quiz

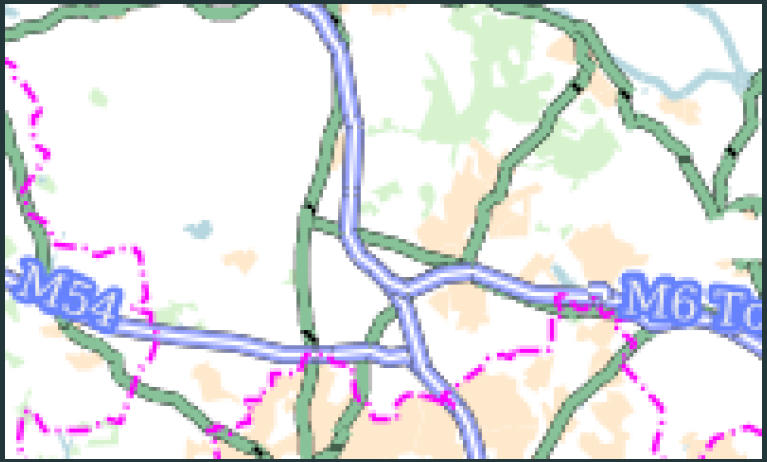
Visual quality - Blind test [Image 1]

Who is [Ductus, Pisces, Marlin] ?



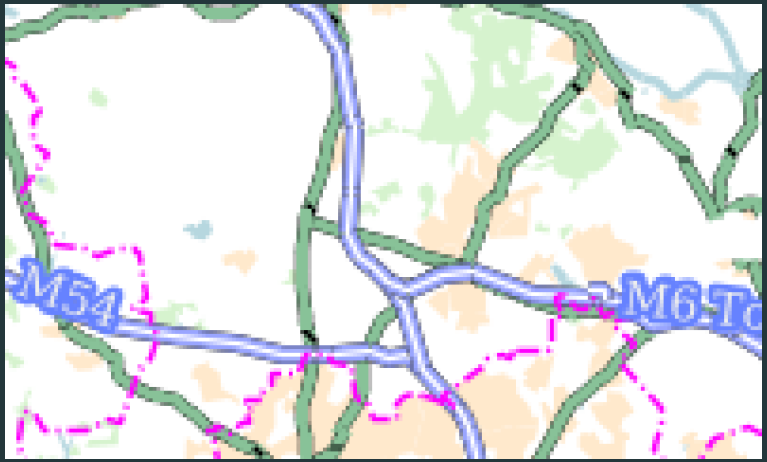
Visual quality - Blind test [Image 2]

Who is [Ductus, Pisces, Marlin] ?



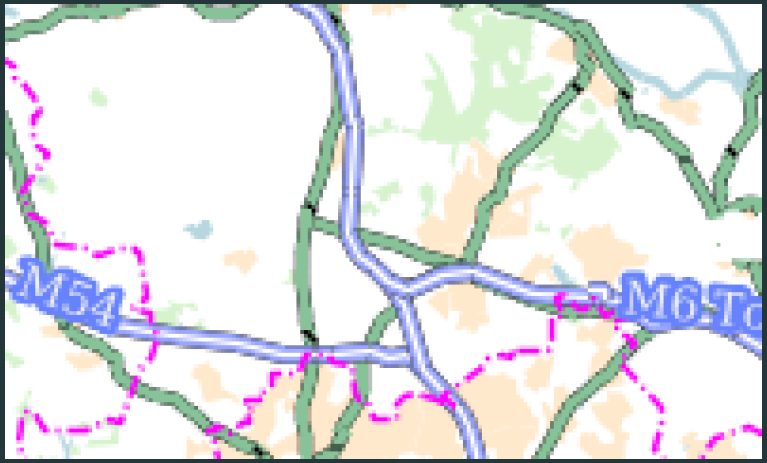
Visual quality - Blind test [Image 3]

Who is [Ductus, Pisces, Marlin] ?



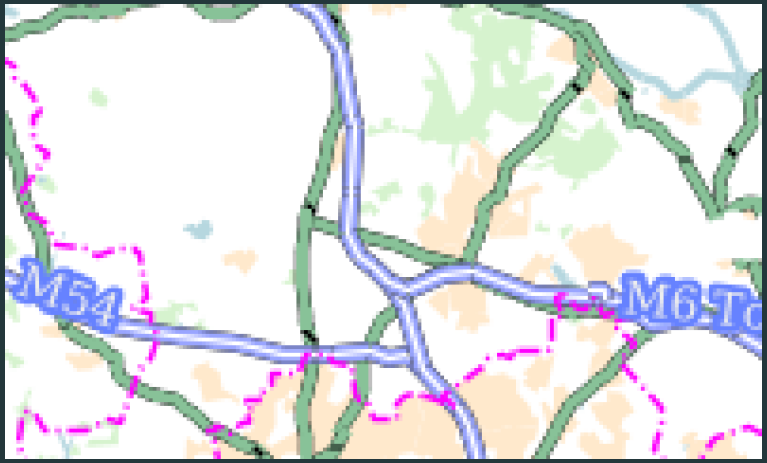
Visual quality - Test result [Image 1]

Answer: Ductus



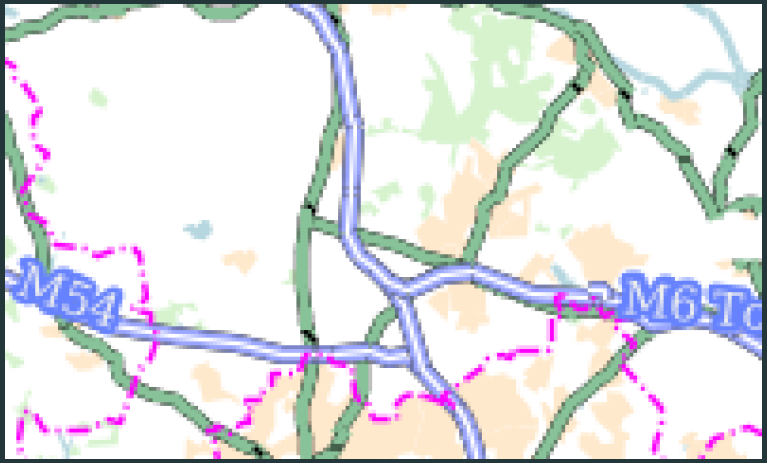
Visual quality - Test result [Image 2]

Answer: Marlin



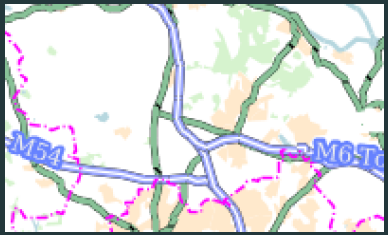
Visual quality - Test result [Image 3]

Answer: Pisces

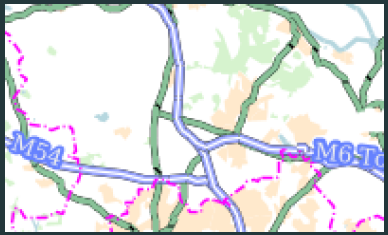


Marlin visual quality

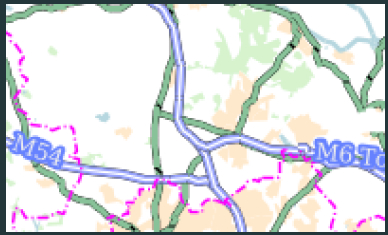
Who is [Ductus, Pisces, Marlin] ?



1. Ductus



2. Marlin



3. Pisces

$\Rightarrow$ Quite an 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:

```
1 java -Xbootclasspath/a:../lib/marlin-0.7.5-Unsafe.jar
2      -Dsun.java2d.renderer=sun.java2d.marlin.MarlinRenderingEngine
3      ...
```

Marlin System properties

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

Log-2 values for sub-pixel & tile sizes: $3 \rightarrow 8$, $5 \rightarrow 32$

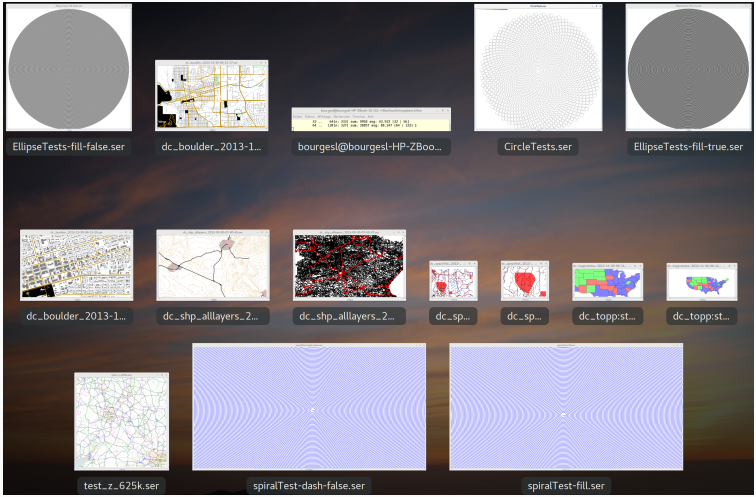
MapBench benchmarks

- **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

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

MapBench tests

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

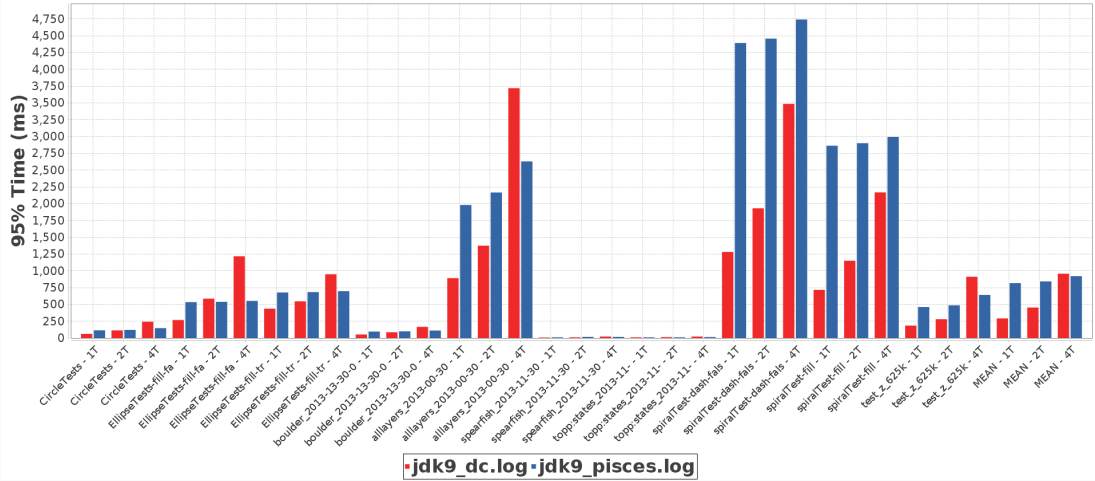


Marlin usage & benchmarks

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

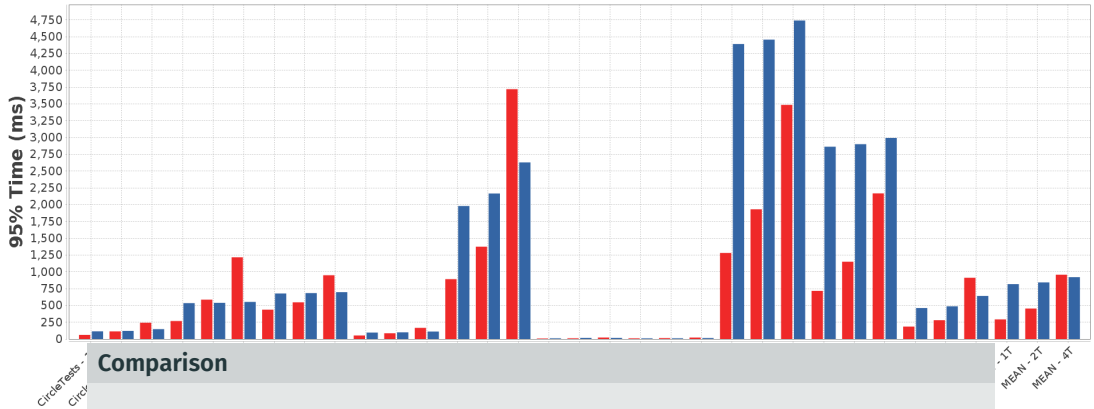
Before Marlin (Oracle JDK-9 EA b181)

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



Before Marlin (Oracle JDK-9 EA b181)

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

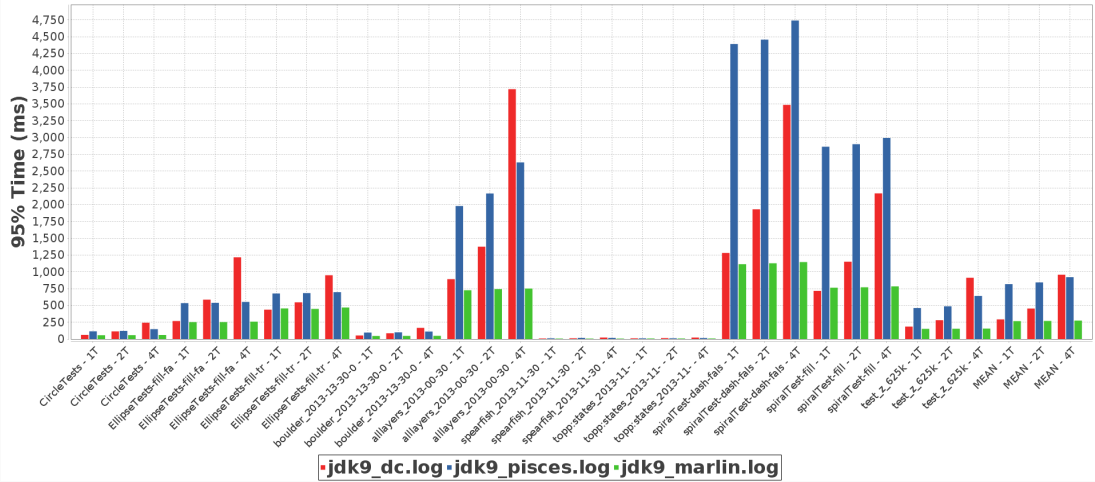


Comparison

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

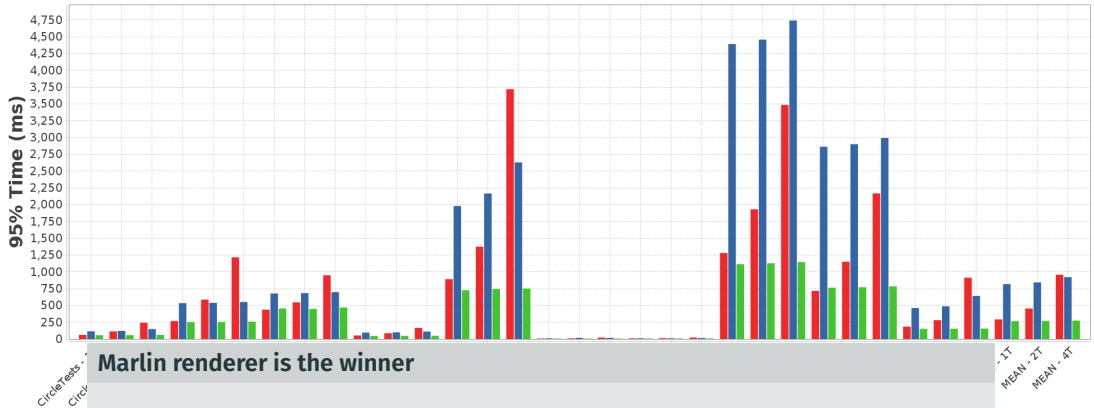
With Marlin (Oracle JDK-9 EA b181)

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



With Marlin (Oracle JDK-9 EA b181)

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

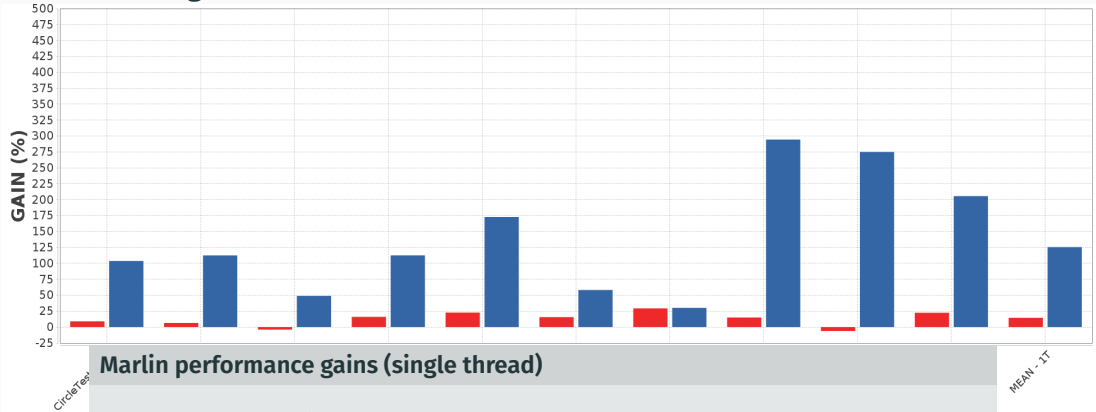


Marlin renderer is the winner

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

Performance gains (1 thread)

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

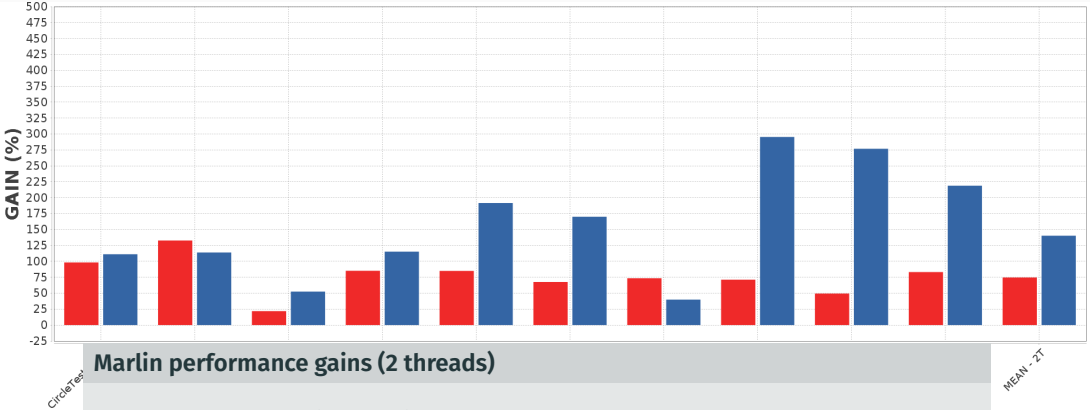


Marlin performance gains (single thread)

- 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**

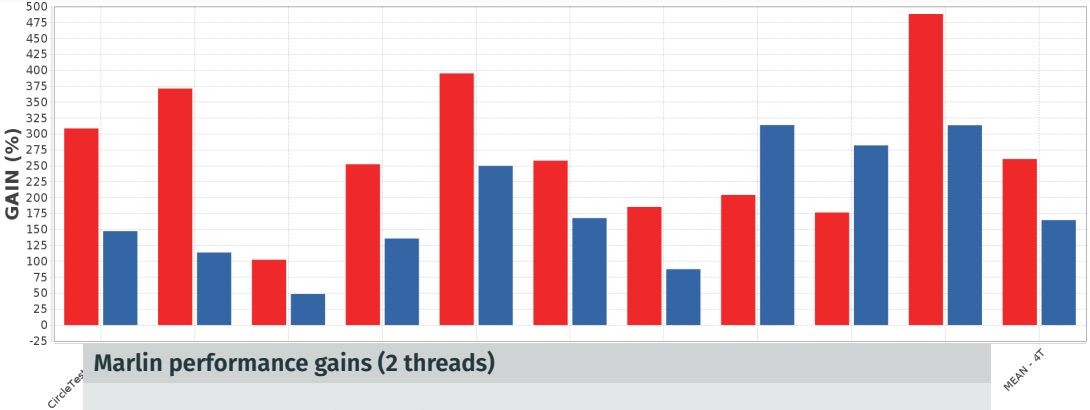


Marlin performance gains (2 threads)

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

Performance gains (4 thread)

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

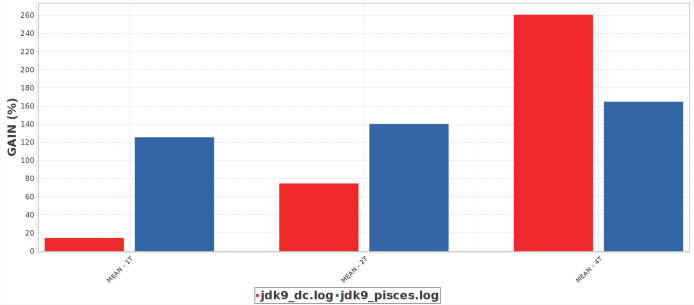


Marlin performance gains (2 threads)

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

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

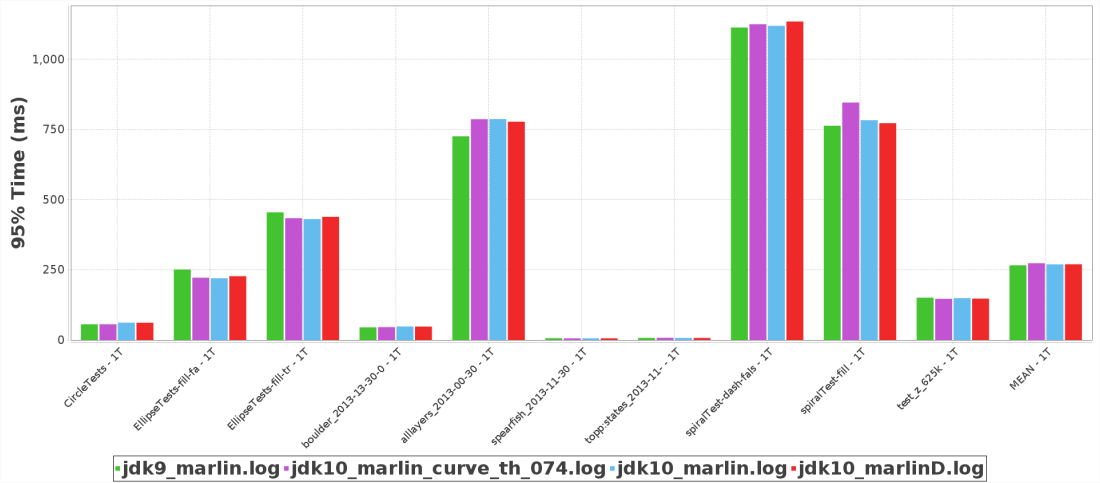


Marlin usage & benchmarks

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)

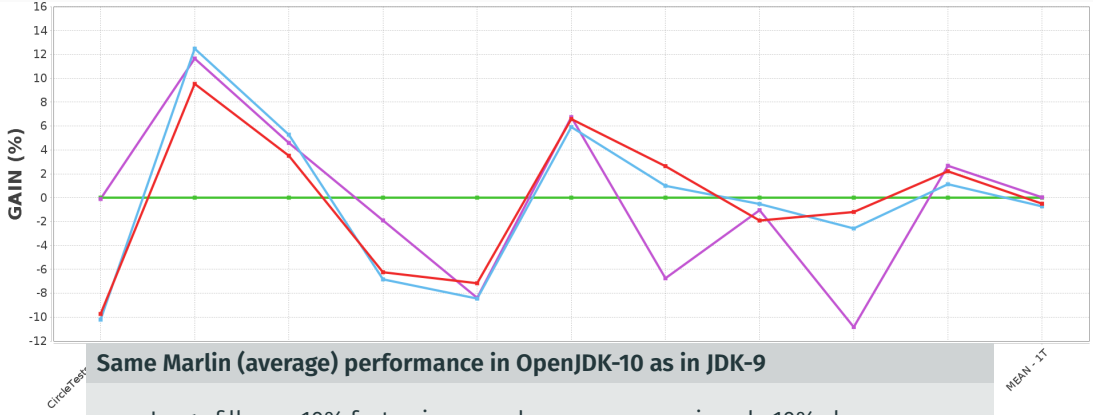
- Marlin 0.7.5 vs 0.7.4 (curve accuracy, large fills, Double variant)

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**

MarlinFX

MarlinFX project

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

- https://github.com/bourgesl/marlin-fx
- **Prism integration**:
  - Only concerns complex paths (not ellipse or rounded rect., 3D meshes)
  - Complete mask, no more tiles
  - GPU back-end is faster but uploading texture may be costly
  - but single-threaded (GUI)
  - TBD: use Marlin to implement Shape.createStrokedShape()
- **Double-precision variant to improve accuracy** on very large shapes (stroke / dashes)
- **Integrated in OpenJFX-9** (dec 2016) but disabled by default

MarlinFX Prism integration

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

TODO: complete

- Accelerated pipelines (shader) vs software pipeline
- Area class issues (cpu / memory consuming) (inner / outer strokes)
- Uploading texture masks can be very costly (bottleneck ?)

My feedback on contributing to OpenJFX

Contrary to Marlin, OpenJFX patches were more quickly integrated into OpenJFX-9 (Jim Graham, again) !

Few documentation about Prism (pipelines, shader graphics) or rendering internals (Shape, Canvas...)

TODO: complete

MarlinFX usage & benchmarks

How to use MarlinFX ?

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

- Enable MarlinFX using prism settings:

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

- Select Marlin Double or Float variant:

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

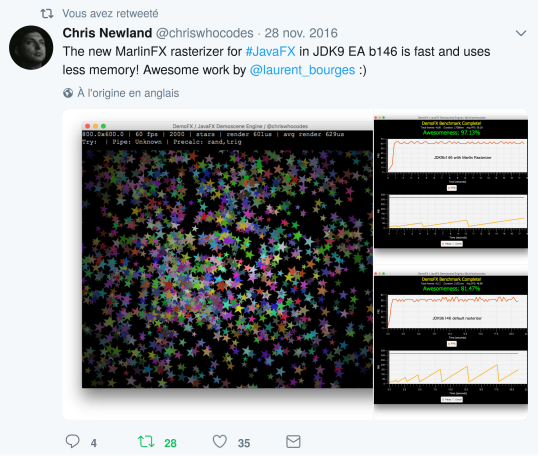
- To enable the Prism log, add **`-Dprism.verbose=true`** :

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

DemoFX benchmarks

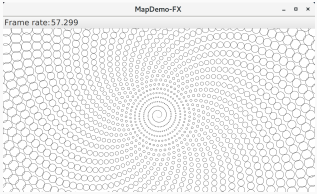
- OpenJFX-9 b146 **first results with DemoFX** Triangle test [VSYNC]

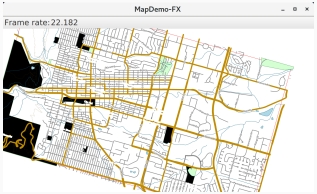
58 fps vs 48 fps = 20% gain

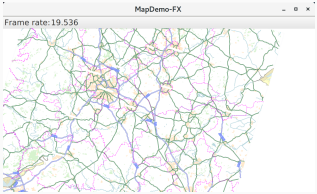


MapBenchFX benchmarks

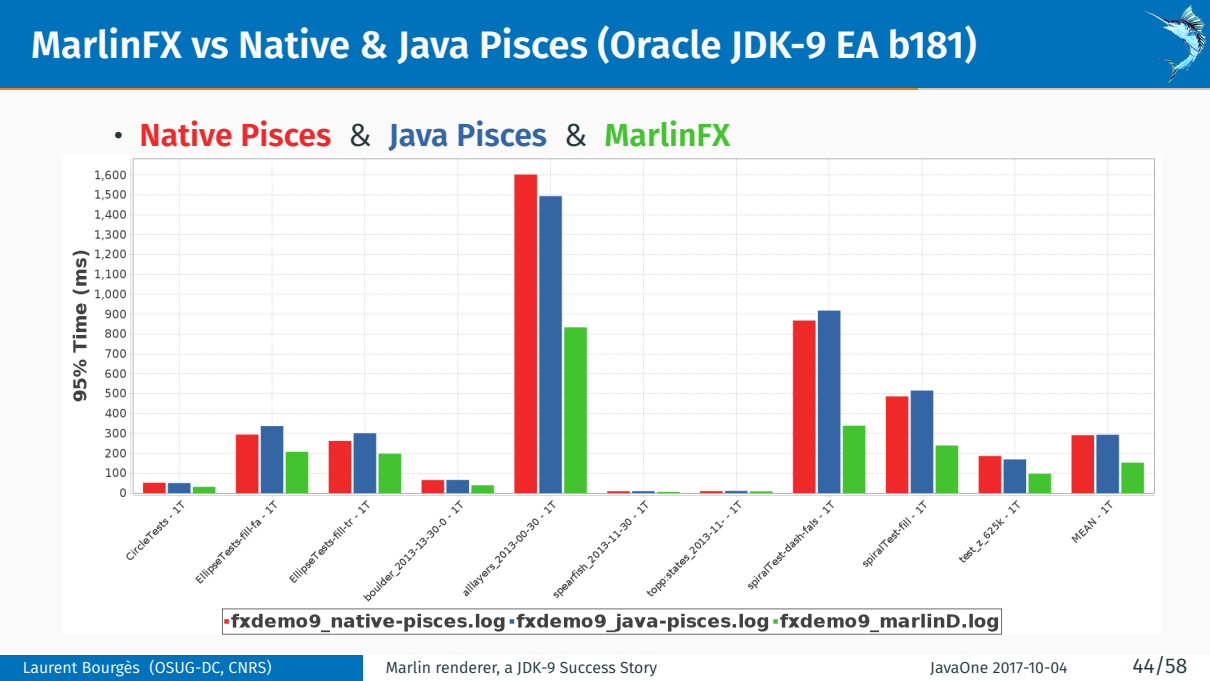
- **MapBenchFX**: JavaFX application to render maps
  - Port of the **Java2D MapBench benchmark** using FXGraphics2D
  - https://github.com/bourgesl/mapbench-fx







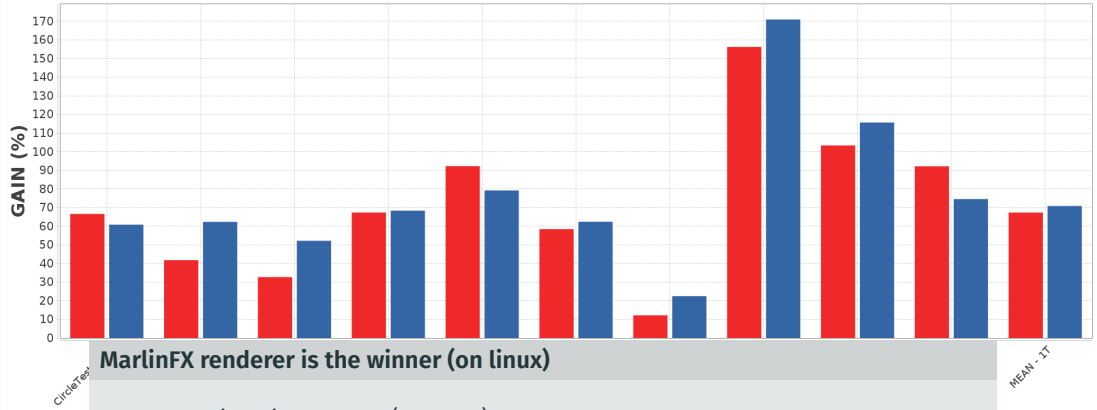
MarlinFX vs Native & Java Pisces (Oracle JDK-9 EA b181)



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

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...

Perspectives and Future Work

We want You

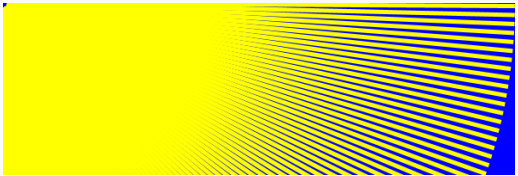
I may have still spare time to improve Marlin… **but no more alone** !

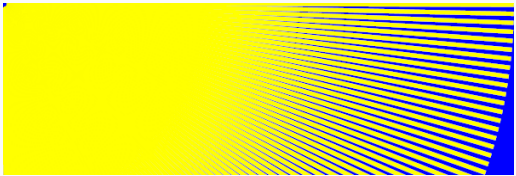
We want You = your help is needed:

- **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 !

Quality Ideas

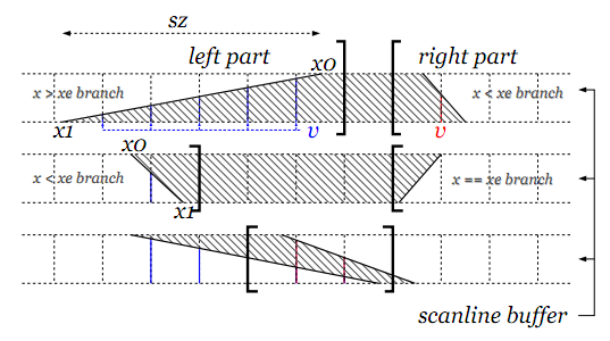
- **Handle properly the gamma correction**:
  - in MaskFill for images (C macros) & GPU back-ends
  - **Very important for visual quality**
  - note: Stroke's width must compensate the gamma correction to avoid having thiner / fatter shapes.





Quality Ideas (continued)

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



Performance ideas

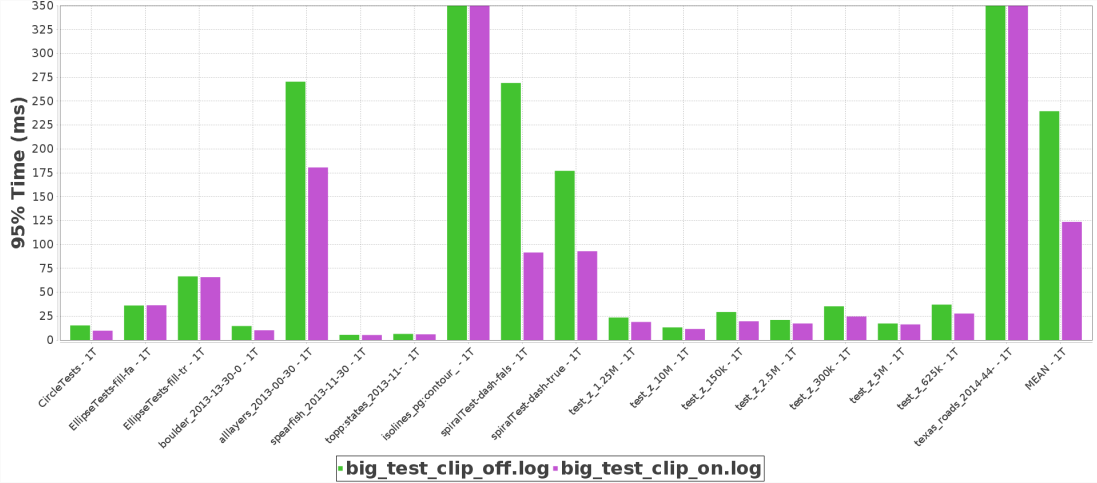
- **Clipping** (**coming** in release 0.8):
  - Implement early efficient path clipping (major impact on dashes)
  - Take care of affine transforms (margin), stroke's cap & joins
- **Scan-line processing** (8x8 sub-pixels):
  - 8 scan-lines per pixel row $\Rightarrow$ compute exact area covered in 1 row
  - see `http://nothings.org/gamedev/rasterize/`
  - 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

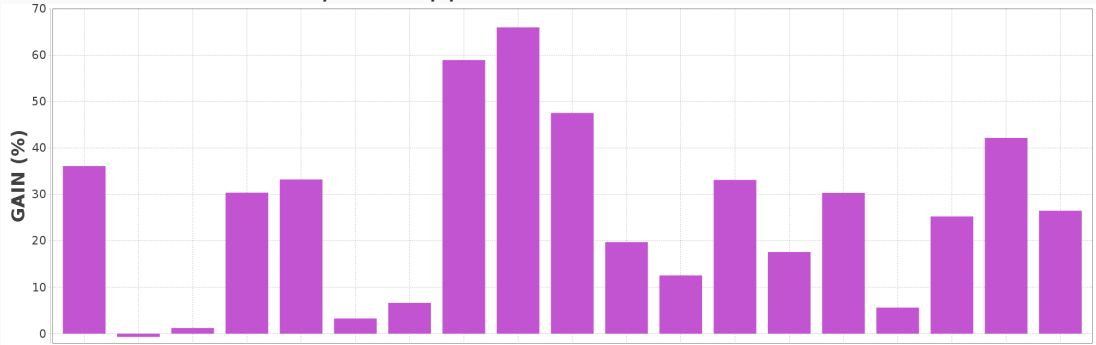
Performance impact of the path clipping (Marlin 0.8.1)

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



Performance impact of the path clipping (Marlin 0.8.1)

- Gain with Marlin path clipper enabled:



Promising Marlin's path clipper

- **25%** gain (average)
- up to **65%** gain (complex shapes)

Conclusion

Conclusion

- **Marlin renderer rocks in Java 9 !**
  - **Performance goals** met
    - **Scales for multiple threads** whereas Ductus does not
    - **Single threaded performance** generally **as good or better**
  - **Quality goals** met
- **Marlin is now the default AA renderer** for Oracle JDK (and OpenJDK).
- **MarlinFX** will be the **default renderer in OpenJFX-10**
- **JavaFX** needs a **proper Path** node ...
- **Coming patches** are promising for **OpenJDK-10**...
- **What future for Java Client & Desktop ?**

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

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.

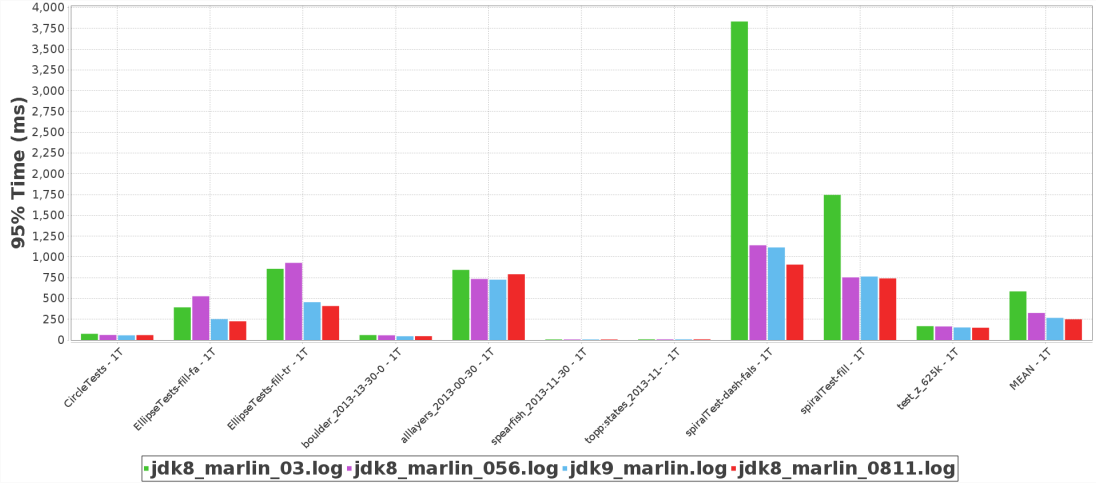
Extra

Extra

Comparing Marlin renderer releases using JDK-8

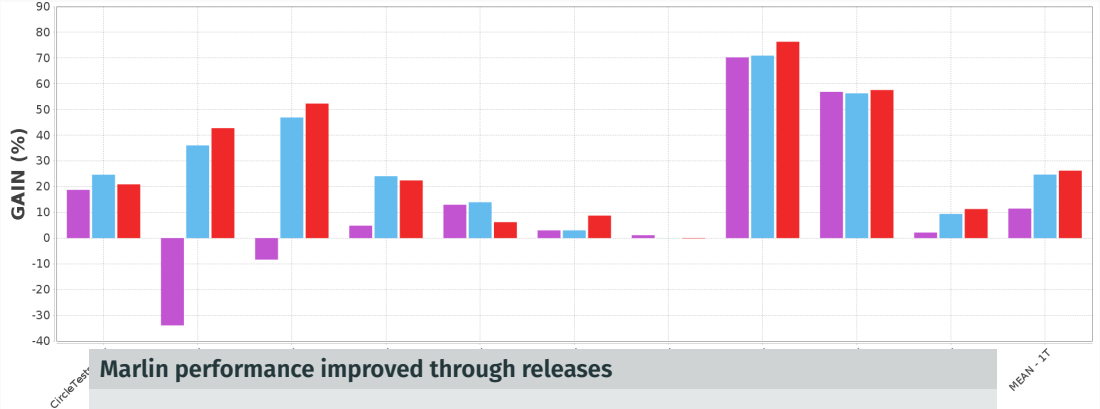
Performance evolution between Marlin 0.3 & 0.8.1 (JDK-8)

- 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

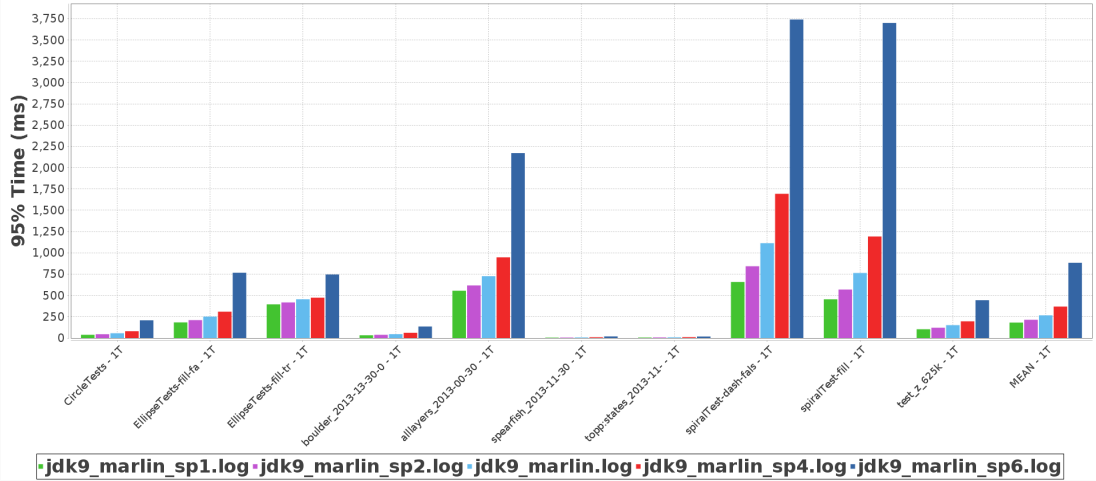
- Loss in 0.5.6 (large ellipse fills) fixed in 0.7
- Important improvements on complex stroked shapes (spirals)

Extra

Performance impact of the sub-pixel tuning

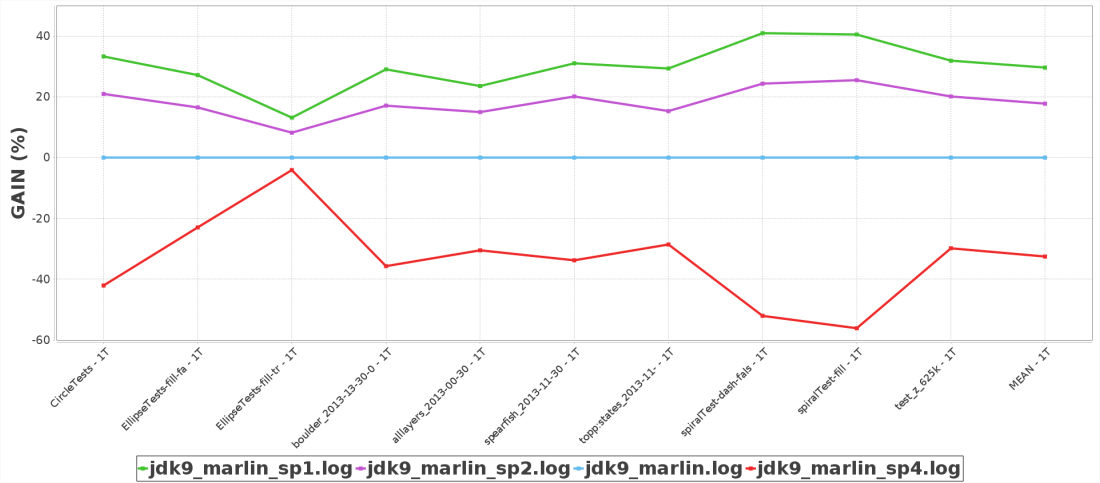
Performance impact of the sub-pixel tuning

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



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]:

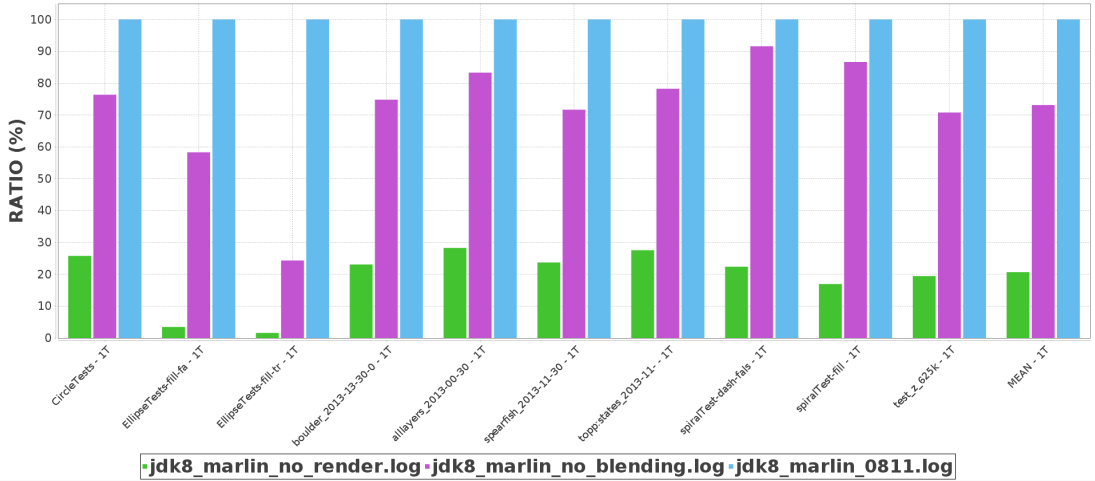


Extra

Studying Marlin & Java2D internal stages

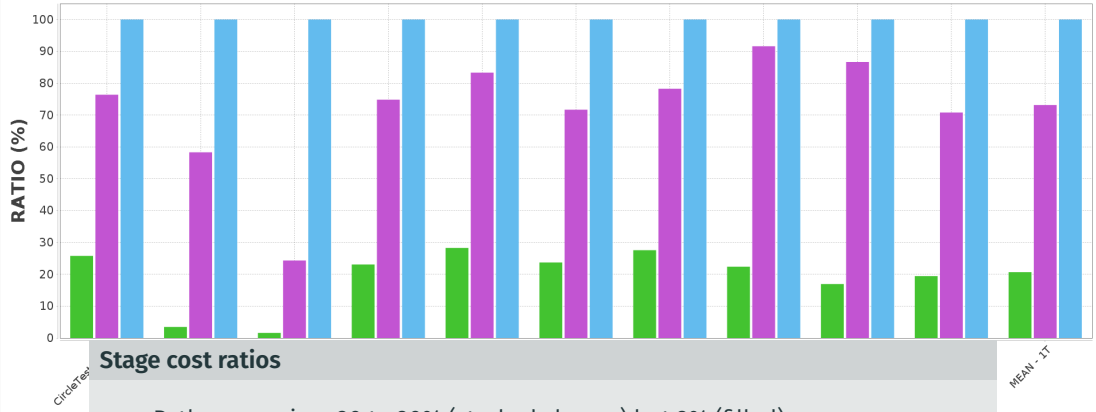
Marlin internal timings

- 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 ?*