Marlin renderer

a successful fork and join the OpenJDK 9 project

Laurent Bourgès

github.com/bourgesl

OpenJDK

FOSDEM 2016, Jan 30th

Outline

1. Context & History
2. How Marlin works ?
3. Marlin benchmarks
4. How to use Marlin ?
5. Demo
6. Marlin renderer tuning
7. Future work

Context

Java2D is a great API (since 1997) to perform graphics rendering.

Antialiasing renderers = Graphics2D.draw/fill(Shape):

- Ductus (closed-source) in Sun / Oracle JDK (jdk 1.2)
  - sun.dc.DuctusRenderingEngine (native C code)
- Pisces (open-source) integrated in OpenJDK (2007)
  - java2d.pisces.PiscesRenderingEngine (java)

Status in 2013:

- Ductus: faster but does not scale well (multi-threading)
- Pisces: slower but scales better
- GPU ? java2D pipelines (OpenGL, D3D...) provide only few accelerated operations (or switch to `glg2d`)
- JavaFX only for client applications (not server-side)

Marlin renderer = OpenJDK's Pisces fork

- March-Mai 2013: my first patchs to OpenJDK 8:
  - Pisces patchs to `2d-dev@openjdk.java.net`: too late
  - small interest / few feedback
- Andréa Aimé (GeoServer team) pushed me to go on:
  - new MapBench tool: serialize & replay map rendering
  - fork OpenJDK's Pisces as a new open-source project

⇒ 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`

Marlin & MapBench projects at github

Objectives:

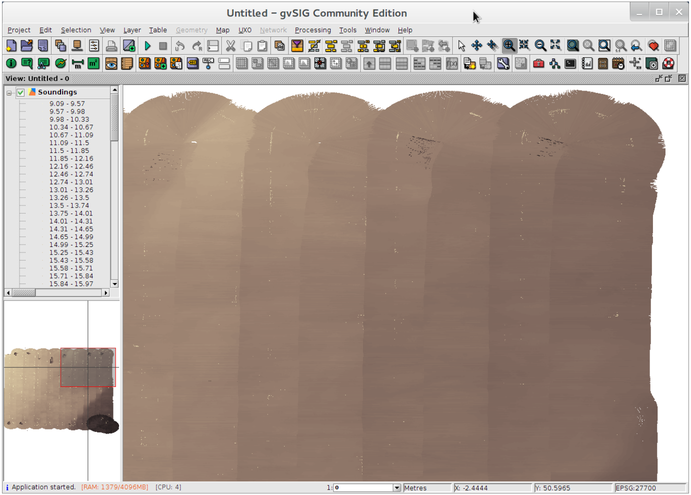
- faster alternative with very good scalability
- improve rendering quality
- Compatible with both Oracle & Open JDK 7 / 8 / 9

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 (web), gvSIG CE (Swing) providing complex use cases & testing releases

Point cloud rendering in gvSIG CE

- Marlin allows parallel rendering of large point clouds (100M):

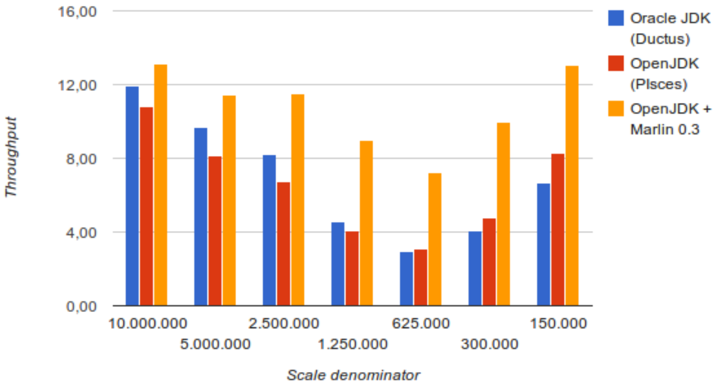


Application started. [RAM: 1379/4096MB] [CPU: 4]

1:0 Metres X: -2.4444 Y: 50.5965 EPSG:27700

Marlin project on the web

- Famous blog post (02.2014): Achieving Extreme GeoServer Scalability with the new Marlin vector rasterizer



- Marlin wiki: Benchmarks page

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

⇒ 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)
- We proposed the 'JEP 265: Marlin Graphics Renderer' in July 2015 and make it completed !

- It is now integrated in OpenJDK9 b96 ⇒ Marlin even faster:
  - Marlin 0.7: improve coordinate rounding arround subpixel center
  - Marlin 0.7.2: improve large pixel chunk copies (coverage data)

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 !

General:

- CI: missing 'open' multi-platform machines to perform tests & benchmarks outside of Oracle
- Funding community-driven effort ? support collaboration with outsiders

How Java2D 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

```
RenderingEngine:
    public static synchronized RenderingEngine getInstance();
    public AATileGenerator getAATileGenerator(Shape s,
                                AffineTransform at, ...);
 AATileGenerator:
    public int getTypicalAlpha();
    public void nextTile();
    public void getAlpha(byte tile[], ...);
```

How Marlin works ? Pisces / Marlin pipeline

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

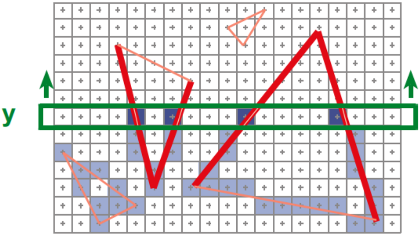
- use shape.getPathIterator() ⇒ apply the pipeline to path elements:
- Dasher (optional):
  - generates path dashes (curved or segments)
- Stroker (optional):
  - generates edges arround of every path element
  - generates edges for decorations (cap & joins)
- Renderer:
  - curve decimation into line 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)
- MarlinTileGenerator:
  - provide tile data (32x32) from MarlinCache (packed byte[])

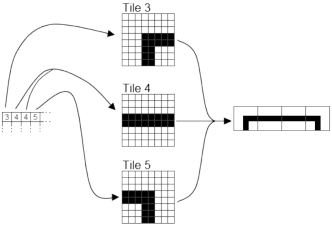
How Marlin works ? the AA algorithm

- 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 !





Marlin performance optimizations

Intially 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) = reused memory ⇒ almost no GC:

- kept by weak / soft reference
- class instances + initial arrays takes 512Kb
- weak-referenced array cache for larger arrays

Use:

- 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:
  - custom rounding [float to int]
  - DDA in Renderer with correct pixel center handling
  - tile stride approach instead of all tiles (32px)
  - pixel alpha transfers (RLE) ⇒ adaptive approach

All lot more ...

MapBench benchmarks

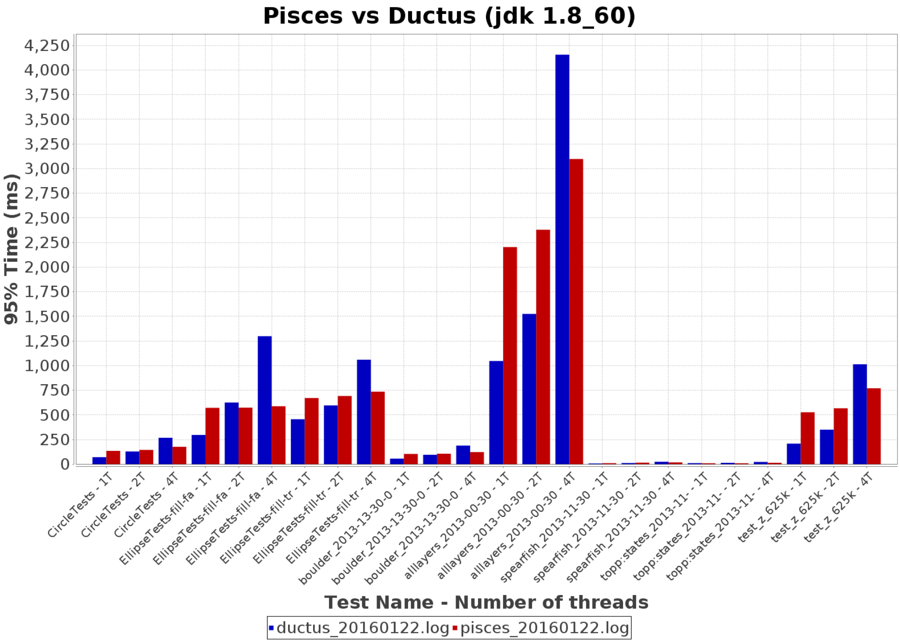
- MapBench tool:
  - a multi-threaded java2d benchmark that replays serialized graphics commands (see ShapeDumperGraphics2D)
  - calibration & warmup phase at startup + correct statistics [min, median, average, 95th percentile, max]

Procedure:

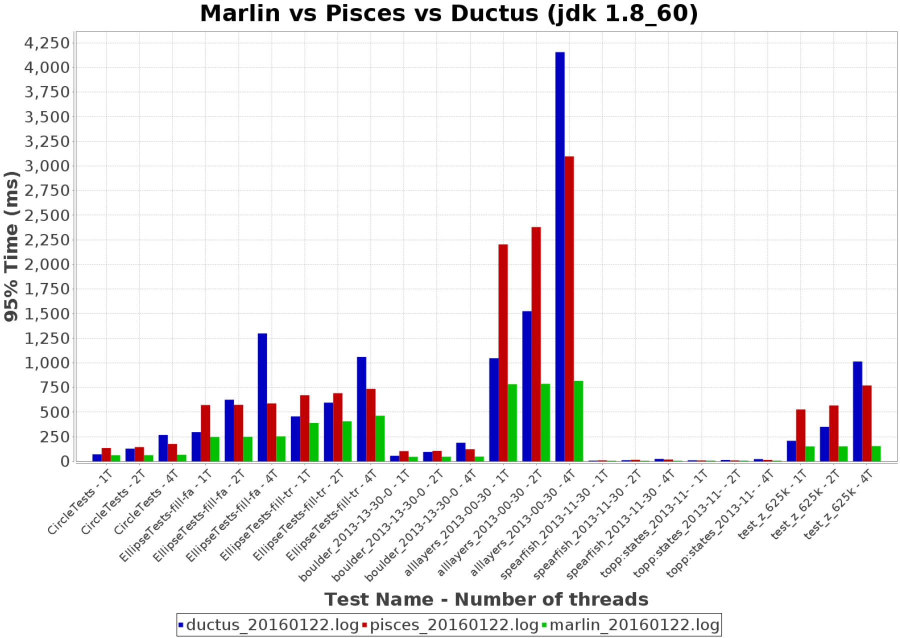
- disable HyperThreading (in BIOS)
- use fixed cpu frequencies (2GHz) on my laptop (i7 4800)
- setup the jvm: jdk to use + basic jvm settings = CMS gc 2Gb Heap
- use a profile (shared images) to reduce GC overhead

⇒ Reduce variability (and cpu affinity issues)

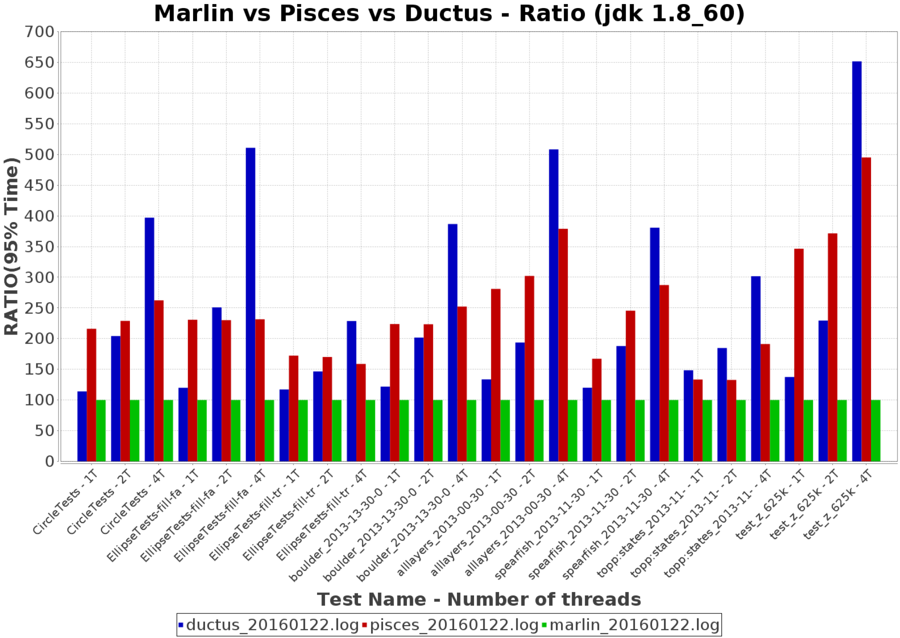
Before Marlin



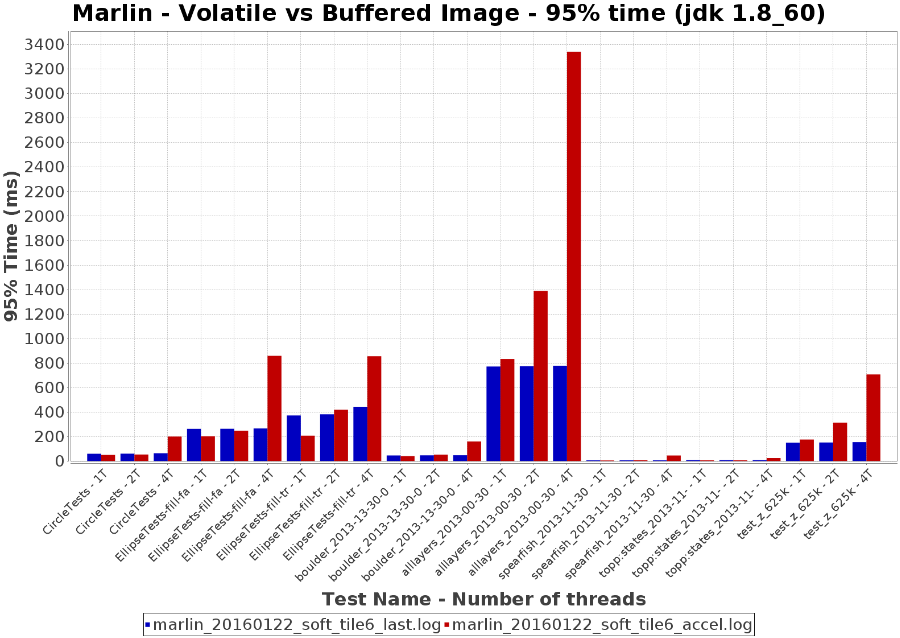
With Marlin


Marlin vs Pisces vs Ductus (jdk 1.8_60)
95% Time (ms)
4,250
4,000
3,750
3,500
3,250
3,000
2,750
2,500
2,250
2,000
1,750
1,500
1,250
1,000
750
500
250
0
CircleTests - 1T
CircleTests - 2T
CircleTests - 4T
EllipseTests-fill-fa - 1T
EllipseTests-fill-fa - 2T
EllipseTests-fill-fa - 4T
EllipseTests-fill-tr - 1T
EllipseTests-fill-tr - 2T
EllipseTests-fill-tr - 4T
boulder_2013-13-30-0 - 1T
boulder_2013-13-30-0 - 2T
boulder_2013-13-30-0 - 4T
alllayers_2013-00-30 - 1T
alllayers_2013-00-30 - 2T
alllayers_2013-00-30 - 4T
spearfish_2013-11-30 - 1T
spearfish_2013-11-30 - 2T
spearfish_2013-11-30 - 4T
topp:states_2013-11- - 1T
topp:states_2013-11- - 2T
topp:states_2013-11- - 4T
test_z_625k - 1T
test_z_625k - 2T
test_z_625k - 4T
Test Name - Number of threads
■ ductus_20160122.log ■ pisces_20160122.log ■ marlin_20160122.log

Performance summary



VolatileImage issue



How to use Marlin ?

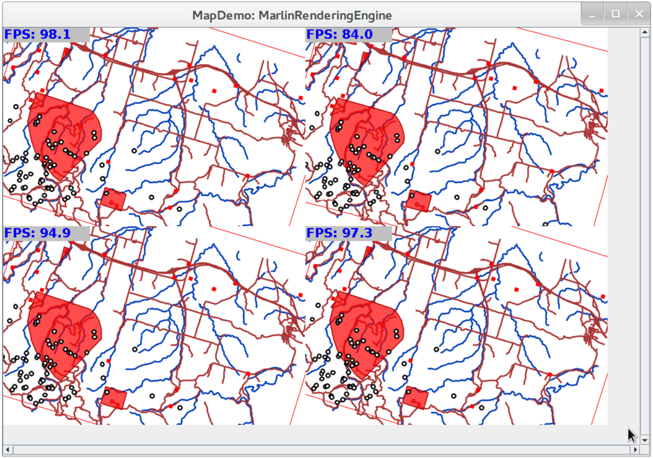
See:

`https://github.com/bourgesl/marlin-renderer/wiki/How-to-use`

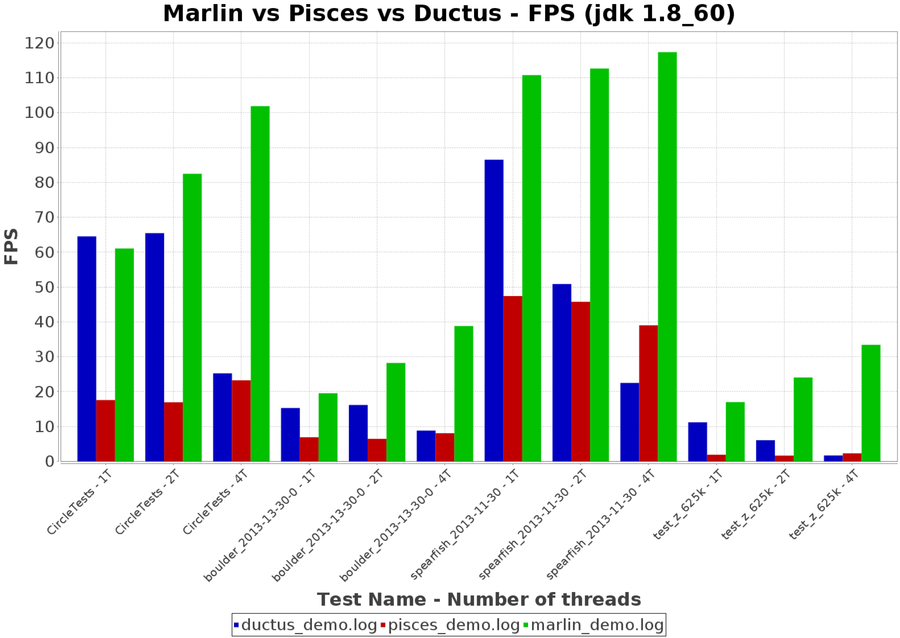
- Just download the latest Marlin release
- Start your java program with:
  - -Dsun.java2d.renderer=sun.java2d.marlin.MarlinRenderingEngine
  - Oracle or Open JDK 1.7 or 1.8 needed
- OR download any Oracle or Open JDK9 EA builds
  - `https://jdk9.java.net/`

Demo

Here is a demo comparing OpenJDK Pisces vs Marlin on intensive rendering tasks (based on MapBench) = MapDemo class !



Demo Performance summary



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

Log2 for subpixel & tile sizes:

- subPixel = 3 means 8x8
- tileSize = 5 means 32x32

Future work

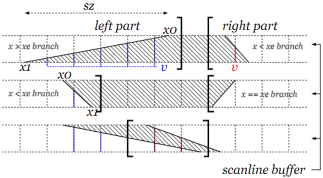
I may have still spare time to improve Marlin...

But your help is needed:

- try your applications & use cases with Marlin
- contribute: let's implement new algorithms (gamma correction, clipping ...)
- provide feedback, please !

Quality Ideas

- NaN / Overflow handling
- Higher precision maths: double vs float in Dasher / Stroker maths and affine transforms
- **Handle properly the gamma correction**: (MaskFill C macros)
  - very important for visual quality
  - note: stroke width must compensate the gamma correction to avoid having thin shapes.
- Analytical pixel coverage: using signed area coverage for a trapezoid $\Rightarrow$ compute the exact pixel area covered by the polygon



Performance ideas

- Clipping:
  - 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 ⇒ 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 !

That's all folks !

- Please ask your questions
- or send them to `marlin-renderer@googlegroups.com`

Special thanks to:

- Andréa Aimé (GeoServer)
- Benjamin Ducke (gvSIG CE)
- OpenJDK teams for their help, reviews & support:
  - Jim Graham & Phil Race (java2d)
  - Mario Torre & Dalibor Topic
  - Mark Reinhold (openjdk 9)
- ALL Marlin users