

Julia-for-HEP discussion, 9 November 2022

Julia and Geometry: Practical evaluation of the language

Pere Mato / EP-SFT

Evaluation Goals

- ❖ Julia is a priori a good programming language candidate for HEP
 - ❖ It combines **high-level expressibility** for scientific computational problems together with **high-performance** execution
- ❖ My goal has been exactly to evaluate these two aspects using a [very partial] re-write of the VecGeom geometry package
 - ❖ Running the code on CPU as well as on GPU
- ❖ Other aspects to evaluate as well
 - ❖ Development process and environment
 - ❖ Development of tests
 - ❖ Performance optimisation

The Geom4hep.jl Package

- ❖ Repository: <https://github.com/peremato/Geom4hep>
- ❖ Current functionality:
 - ❖ Basic shapes: **Box, Tube, Trd, Trap, Cone, CutTube, Polycone, Boolean**
 - ❖ Materials: **Isotope, Element, Material**
 - ❖ Essential functions to navigate on them (distanceToIn, distanceToOut, ...)
 - ❖ Hierarchical geometry models with **Volume** and **PlacedVolume**
 - ❖ Unit tests
 - ❖ Very basic navigator (linear search among all daughters) and BVH navigator
- ❖ 3D graphics support for displaying the shapes and geometry models using the Makie package
- ❖ GDML reader to input geometries
- ❖ Benchmark functions for shapes
- ❖ Support for single and double precision (all functions and types parametrized with the float type)
- ❖ A number of examples to demonstrate the available functionality
- ❖ 100% Julia code, build on top of:
 - ❖ **StaticArrays, GeometryBasics, LinearAlgebra, Rotations, CUDA, [GL]Makie, LightXML**

Code Examples

```
struct Tube{T<:AbstractFloat} <: AbstractShape{T}
  rmin::T # inner radius
  rmax::T # outer radius
  z::T    # half-length in +z and -z direction
  φ₀::T   # starting φ value (in radians)
  Δφ::T   # delta φ value of tube segment (in radians)

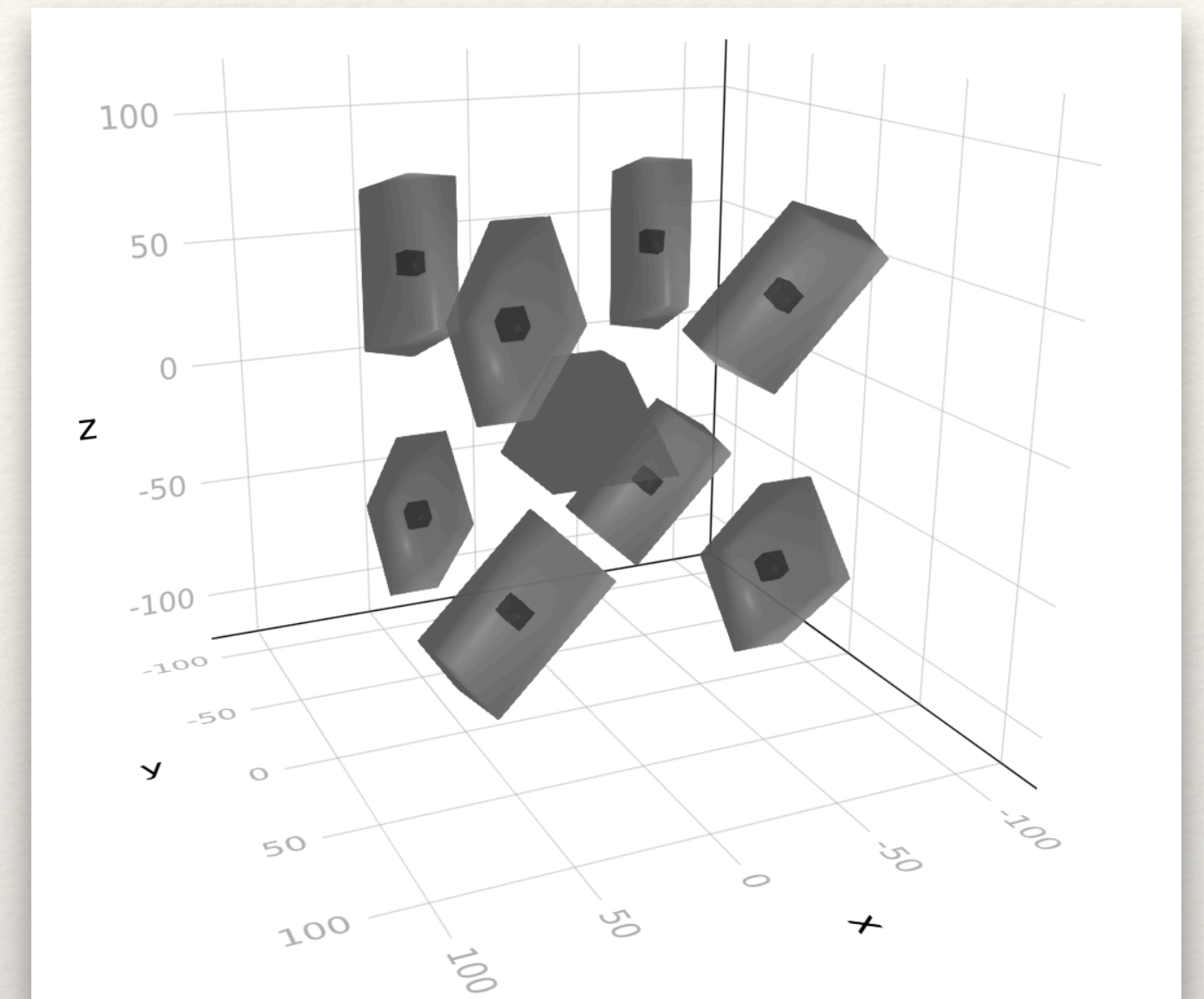
  # cached derived values (to avoid recalculations)
  rmin2::T
  rmax2::T
  ...
end
```

- ❖ User defined types as **immutable structs**
- ❖ Functions defined outside the struct
 - ❖ Multi-dispatch, extendable, ...
- ❖ Unicode in variable and function names
 - ❖ Closer to math formulations

```
function inside(tub::Tube{T}, point::Point3{T}) where
T<:AbstractFloat
  x, y, z = point
  # Check Z
  outside = abs(z) > tub.z + kTolerance(T)/2
  outside && return k0Outside
  cinside = abs(z) < tub.z - kTolerance(T)/2
  # Check on RMax
  r2 = x * x + y * y
  outside |= r2 > tub.rmax2 + kTolerance(T) * tub.rmax
  outside && return k0Outside
  cinside &= r2 < tub.rmax2 - kTolerance(T) * tub.rmax
  # Check on RMin
  if tub.rmin > 0.
    outside |= r2 <= tub.rmin2 - kTolerance(T) * tub.rmin
    outside && return k0Outside
    cinside &= r2 > tub.rmin2 + kTolerance(T) * tub.rmin
  end
  # Check on Phi
  if tub.Δφ < 2π
    outside |= isOutside(tub.φWedge, x, y)
    cinside &= isInside(tub.φWedge, x, y)
  end
  return outside ? k0Outside : cinside ? kInside : kSurface
end
```

Defining Geometry Models

```
function buildGeom(T::Type)
  world = Volume{T}("World", Box{T}(100,100,100), Material("vacuum"; density=0.0))
  box1 = Volume{T}("box1", Box{T}(10,20,30), Material("iron"; density=7.0))
  box2 = Volume{T}("box2", Box{T}(4,4,4), Material("gold"; density=19.0))
  placeDaughter!(box1, Transformation3D{T}(0,0,0), box2)
  placeDaughter!(world, Transformation3D{T}( 50, 50, 50, RotXYZ{T}(π/4, 0, 0)), box1)
  placeDaughter!(world, Transformation3D{T}( 50, 50,-50, RotXYZ{T}(0, π/4, 0)), box1)
  placeDaughter!(world, Transformation3D{T}( 50,-50, 50, RotXYZ{T}(0, 0, π/4)), box1)
  placeDaughter!(world, Transformation3D{T}( 50,-50,-50, RotXYZ{T}(π/4, 0, 0)), box1)
  placeDaughter!(world, Transformation3D{T}(-50, 50, 50, RotXYZ{T}(0, π/4, 0)), box1)
  placeDaughter!(world, Transformation3D{T}(-50,-50, 50, RotXYZ{T}(0, 0, π/4)), box1)
  placeDaughter!(world, Transformation3D{T}(-50, 50,-50, RotXYZ{T}(π/4, 0, 0)), box1)
  placeDaughter!(world, Transformation3D{T}(-50,-50,-50, RotXYZ{T}(0, π/4, 0)), box1)
  trd1 = Volume{T}("trd1", Trd{T}(25,10,25,10,20), Material("water", density=1.0))
  placeDaughter!(world, Transformation3D{T}(0,0,0), trd1)
  return world
end
```



```
world = buildGeom(Float64)
# or
world = processGDML("examples/boxes.gdml")

fig = Figure()
scene = LScene(fig[1, 1])
draw(scene, world)
display(fig)
```

- ❖ Programmatically
 - ❖ Rather compact writing (minimalistic - only needed info)
- ❖ Importing a GDML file
 - ❖ Currently partial implementation (only the implemented shapes)

3D Drawing

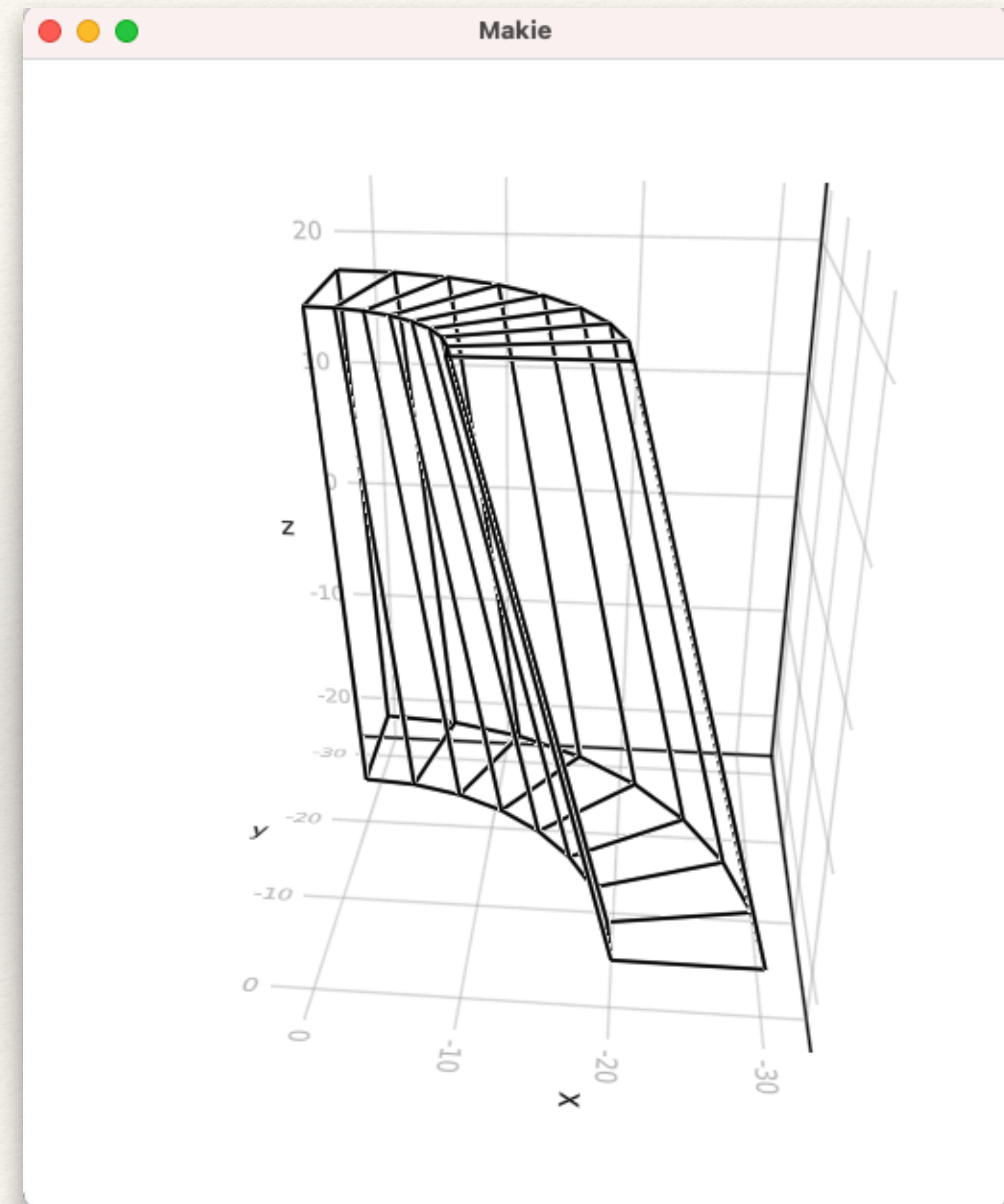
```
julia> cone = Cone{Float64}(20, 30, 10, 20, 20,  $\pi$ ,  $\pi/2$ )
Cone{Float64}(20.0, 30.0, 10.0, 20.0, 20.0,
3.141592653589793, 1.5707963267948966)

julia> draw(cone, wireframe=true)
GLMakie.Screen(...)
```

- ❖ Intended for debugging purposes
- ❖ Using Makie.jl package
 - ❖ Any shape can be drawable if there is a conversion to a mesh

```
function GeometryBasics.coordinates(cone::Cone{T}, facets=36) where T
...
end

function GeometryBasics.faces(cone::Cone{T}, facets=36) where T
...
end
```



High-level expressibility

- ❖ For example the type `Transform3D`
 - ❖ easy way to define basic operators between `Transform3D` and `Point3`, `Vector3`, and `Transform3D`
 - ❖ compact definition of functions
 - ❖ C++ equivalent in `VecGeom` (`Transformation3D.h` [`.cpp`]) has 1300 LOC, in Julia 55 LOC

```
#---Transformation3D-----  
struct Transformation3D{T<:AbstractFloat}  
    rotation::RotMatrix3{T}  
    translation::Vector3{T}  
end  
  
#---Transforms-----  
@inline transform(t::Transformation3D{T}, p::Point3{T}) where T<:AbstractFloat = t.rotation * (p - t.translation)  
@inline transform(t::Transformation3D{T}, d::Vector3{T}) where T<:AbstractFloat = t.rotation * d  
@inline invtransform(t::Transformation3D{T}, p::Point3{T}) where T<:AbstractFloat = t.translation + (p' * t.rotation)'  
@inline invtransform(t::Transformation3D{T}, d::Vector3{T}) where T<:AbstractFloat = (d' * t.rotation)'  
@inline Base.*(t::Transformation3D{T}, p::Point3{T}) where T<:AbstractFloat = transform(t,p)  
@inline Base.*(t::Transformation3D{T}, d::Vector3{T}) where T<:AbstractFloat = transform(t,d)  
@inline Base.*(p::Point3{T}, t::Transformation3D{T}) where T<:AbstractFloat = invtransform(t,p)  
@inline Base.*(d::Vector3{T}, t::Transformation3D{T}) where T<:AbstractFloat = invtransform(t,d)  
  
#---Compose-----  
Base.*(t1::Transformation3D{T}, t2::Transformation3D{T}) where T<:AbstractFloat =  
    Transformation3D{T}(t1.rotation * t2.rotation, t1.translation + t1.rotation * t2.translation)  
Base.inv(t::Transformation3D{T}) where T<:AbstractFloat = Transformation3D{T}(t.rotation', - t.rotation * t.translation)  
  
#---Utilities-----  
Base.one(::Type{Transformation3D{T}}) where T<:AbstractFloat = Transformation3D{T}(one(RotMatrix3{T}), Vector3{T}(0,0,0))  
Base.isone(t::Transformation3D{T}) where T<:AbstractFloat = isone(t.rotation) && iszero(t.translation)  
hasrotation(t::Transformation3D{T}) where T<:AbstractFloat = !isone(t.rotation)  
hastranslation(t::Transformation3D{T}) where T<:AbstractFloat = !iszero(t.translation)
```

High-level impacts performance

- ❖ Writing code using high-level constructs may add some execution penalty
- ❖ In some cases additional allocations of temporaries
- ❖ User needs to be aware
- ❖ Optimal code is always closer to C/C++ than high-level Julia

```
julia> @btime distanceToOut($box, $point, $dir)
 9.262 ns (0 allocations: 0 bytes)
10.0
julia> @btime distanceToOut_LL($box, $point, $dir)
 5.053 ns (0 allocations: 0 bytes)
10.0
```

```
!--Using high-level constructs (array broadcast, maximum, minimum functions, etc.)
function distanceToOut(box::Box{T}, point::Point3{T}, direction::Vector3{T}) where T
    safety = maximum(abs.(point) - box.fDimensions)
    distance = minimum((copysign.(box.fDimensions, direction) - point) * (1.0 ./ direction))
    safety > kTolerance(T)/2 ? -1.0 : distance
end

!--Writing explicit loops
function distanceToOut_LL(box::Box{T}, point::Point3{T}, direction::Vector3{T}) where T
    safety = -Inf
    for i in 1:3
        d = abs(point[i]) - box.fDimensions[i]
        if d > safety
            safety = d
        end
    end
    if safety > kTolerance(T)/2
        return -1.0
    end
    dist = Inf
    for i in 1:3
        d = (copysign(box.fDimensions[i], direction[i]) - point[i])/direction[i]
        if d < dist
            dist = d
        end
    end
    return dist
end
```


Unit Tests

- ❖ Julia provide a very easy way to create unit tests
 - ❖ `@testset` and `@test` macros
 - ❖ The *approx* operator \approx very convenient for comparing floating point numbers
 - ❖ Types are first class entities
e.g. `for T in (Float64, Float32)`

```
@testset "Trans3D{$T}" for T in (Float64, Float32)
    point = Point3{T}(-1., 1., 2.)
    t0 = one(Transformation3D{T})
    @test t0 == one(Transformation3D{T})
    @test isone(t0)
    @test !hasrotation(t0)
    @test !hastranslation(t0)
    @test transform(t0, point) == point
    @test t0 * point == point

    t1 = Transformation3D{T}(-2,-2,-2)
    @test !isone(t1)
    @test hastranslation(t1)
    @test !hasrotation(t1)
    @test t1 * point == Point3{T}(1, 3, 4)

    t2 = Transformation3D{T}(2, 2, 2)
    @test t2 * (t1 * point) == point
    @test (t2 * t1) * point == point

    t3 = Transformation3D{T}(0, 0, 0, 0, 0, π)
    @test !isone(t3)
    @test !hastranslation(t3)
    @test hasrotation(t3)
    @test t3 * point ≈ Point3{T}(1,-1, 2)

    t4 = Transformation3D{T}(0, 0, 1., RotXYZ{T}(0,0,π/4))
    @test t4 * Point3{T}(1,0,0) ≈ Point3{T}(√2/2, √2/2, -1)

    @test (t4 * Point3{T}(1,2,3)) * t4 ≈ Point3{T}(1,2,3)

    @test isapprox(t4 * inv(t4), one(Transformation3D{T}); atol=5*eps(T))
    @test isapprox(inv(t4) * t4, one(Transformation3D{T}); atol=5*eps(T))
end
```

Simple Navigator

```
mutable struct NavigatorState{T<:AbstractFloat} <: AbstractNavigatorState
  topvol::Volume{T}           # Typically the unplaced world
  currvol::Volume{T}         # the current volume
  isinworld::Bool            # inside world volume
  volstack::Vector{Int64}     # path to current daughter
  tolocal::Vector{Transformation3D{T}} # Stack of transformations
end
```

```
function computeStep!(state::NavigatorState{T}, gpoint::Point3{T},
                     gdir::Vector3{T}, step_limit::T) where T
  lpoint, ldir = transform(state.tolocal, gpoint, gdir)
  volume = currentVolume(state)
  step, idx = getClosestDaughter(volume, lpoint, ldir, step_limit)
  #---If didn't hit any daughter return distance to out
  if idx == 0
    step = distanceToOut(volume.shape, lpoint, ldir)
    if step >= 0.
      popOut!(state)
    end
  else
    #---We hit a daughter, push it into the stack
    step += kTolerance(T) # avoid to stay at the surface
    pvol = volume.daughters[idx]
    pushIn!(state, pvol)
  end
  return step
end
```

- ❖ Implemented a simple navigator that loops over all daughters
- ❖ The 'state' keeps the path to the current daughter (indexes) and the list of transformations

CUDA Interface

- ❖ The CUDA module facilitates enormously the use of GPUs
 - ❖ @cuda macro to launch kernels, CuArray to move in/out arrays, etc.
- ❖ Kernels are written in Julia and can call unmodified Julia functions (e.g. distanceToOut in this example) **with some limitations**
 - ❖ no recursion, kernel must return nothing, no strings, must have type-inferred code, no garbage collection on device, kernel cannot allocate (very limited), and only isbits types in device arrays, etc.

```
#---CPU implementation-----  
function cpu_d2out!(dist, shape, points, dirs)  
    N = length(points)  
    for i in 1:N  
        @inbounds dist[i] = distanceToOut(shape, points[i], dirs[i])  
    end  
    return  
end  
  
#---GPU implementation-----  
function gpu_d2out!(dist, shape, points, dirs)  
    N = length(points)  
    index = (blockIdx().x - 1) * blockDim().x + threadIdx().x  
    stride = blockDim().x * gridDim().x  
    for i = index:stride:N  
        @inbounds dist[i] = distanceToOut(shape, points[i], dirs[i])  
    end  
    return  
end  
  
function bench_gpu!(dist, shape, points, dirs)  
    numblocks = ceil(Int, length(points)/256)  
    CUDA.@sync begin  
        @cuda threads=256 blocks=numblocks gpu_d2out!(dist, shape, points, dirs)  
    end  
end
```

CUDA Interface (2)

- ❖ Basic Shapes fulfills the limitations (at least the ones currently implemented), but not the volume hierarchical structure
 - ❖ it has variable length vectors (e.g. vector of daughters)
- ❖ We need to re-shape the geometry model within the limitations
 - ❖ i.e. using indexes to arrays of structs
- ❖ Adapt the navigator code to this new set of structs

```
struct CuVolume{T<:AbstractFloat}
  shapeIdx::UInt32
  materialIdx::UInt32
  daughterOff::UInt32
  daughterLen::UInt32
end

struct CuPlacedVolume{T<:AbstractFloat}
  idx::UInt32
  transformation::Transformation3D{T}
  volume::UInt32
end

...

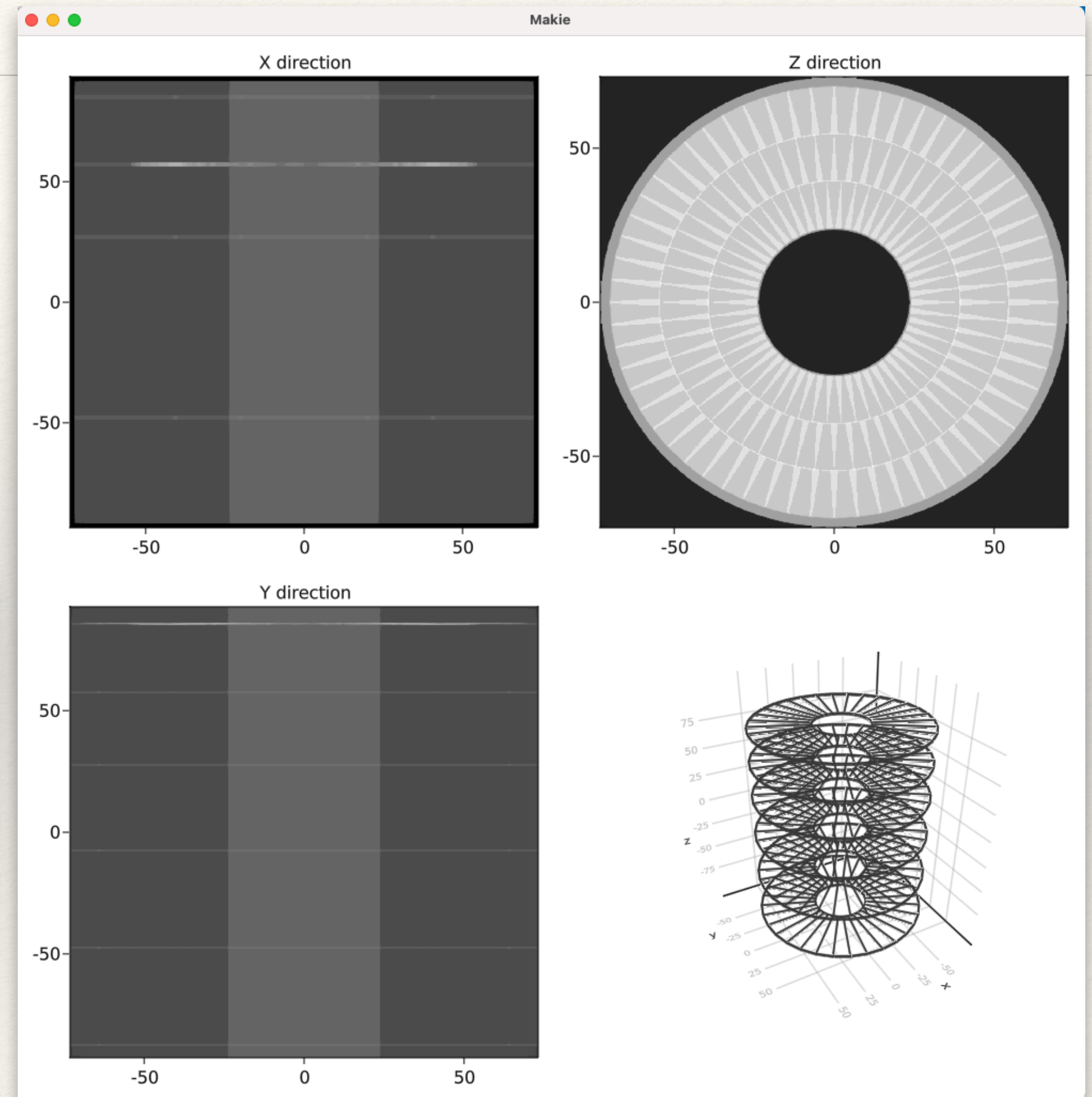
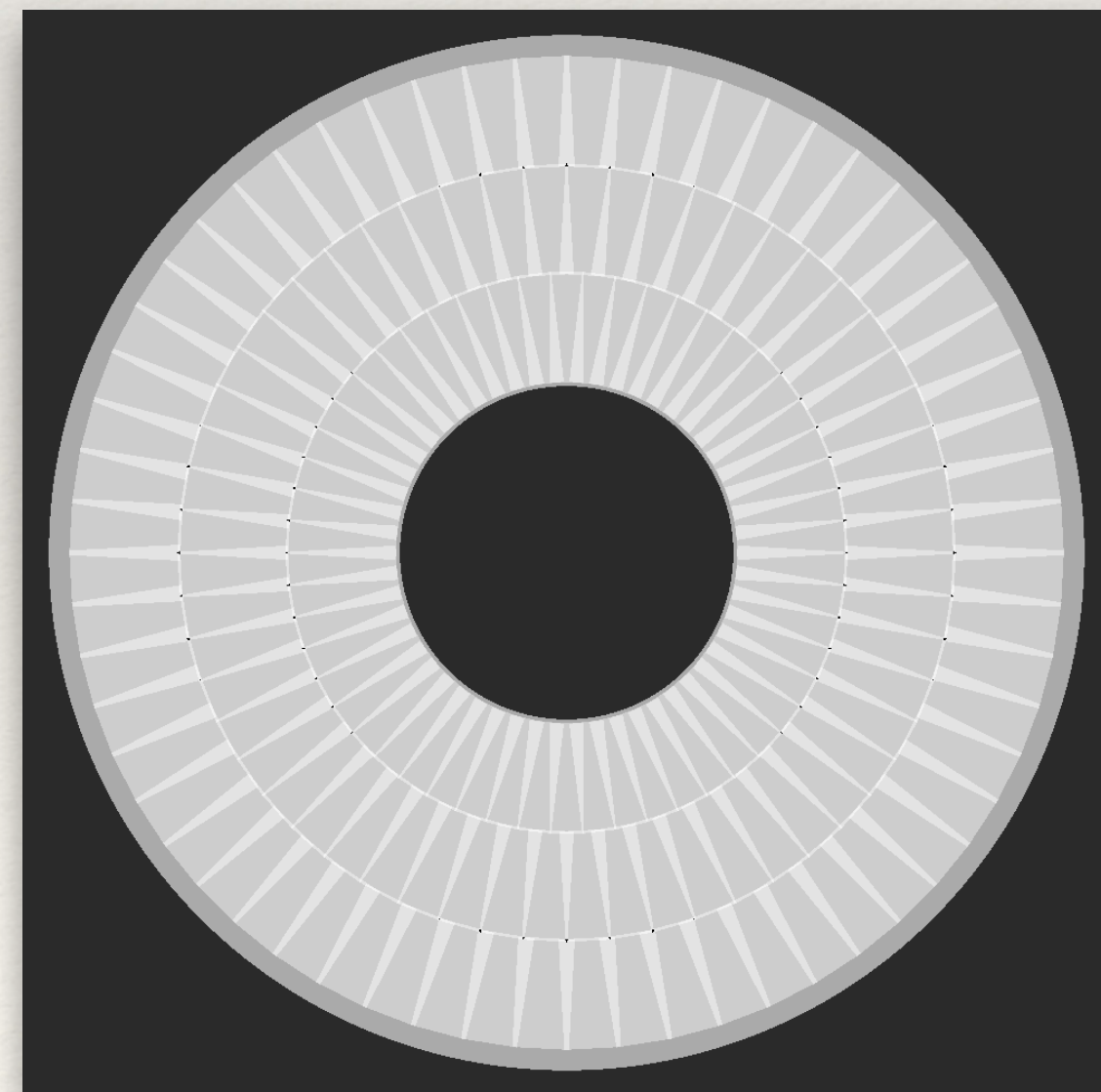
struct CuGeoModel{T<:AbstractFloat}
  volumes::Vector{CuVolume{T}}
  materials::Vector{CuMaterial}
  placedvolumes::Vector{CuPlacedVolume{T}}
  shapes::Vector{Union{Box{T},Tube{T},Trd{T}}}
end

#---Convert the geometry model to Cuda model

function fillCuGeometry(vol::Volume{T}) where T
  indexes = Dict{UInt64, UInt32}()
  model = CuGeoModel{T}()
  pushVolume!(indexes, model, vol)
  return model
end
```

X-Ray Benchmark

- ❖ Using a part of TrackML geometry (Strips1_12) in Z direction (1000x1000)
- ❖ Comparing with VecGeom
`XRayBenchmarkFromROOTFile trackML.root Strips1_12 z 1000`
- ❖ Running with CUDA adapted geometry model on GPU with the module `CUDA.jl`
 - ❖ parallelising on pixels of the image
 - ❖ same results!!



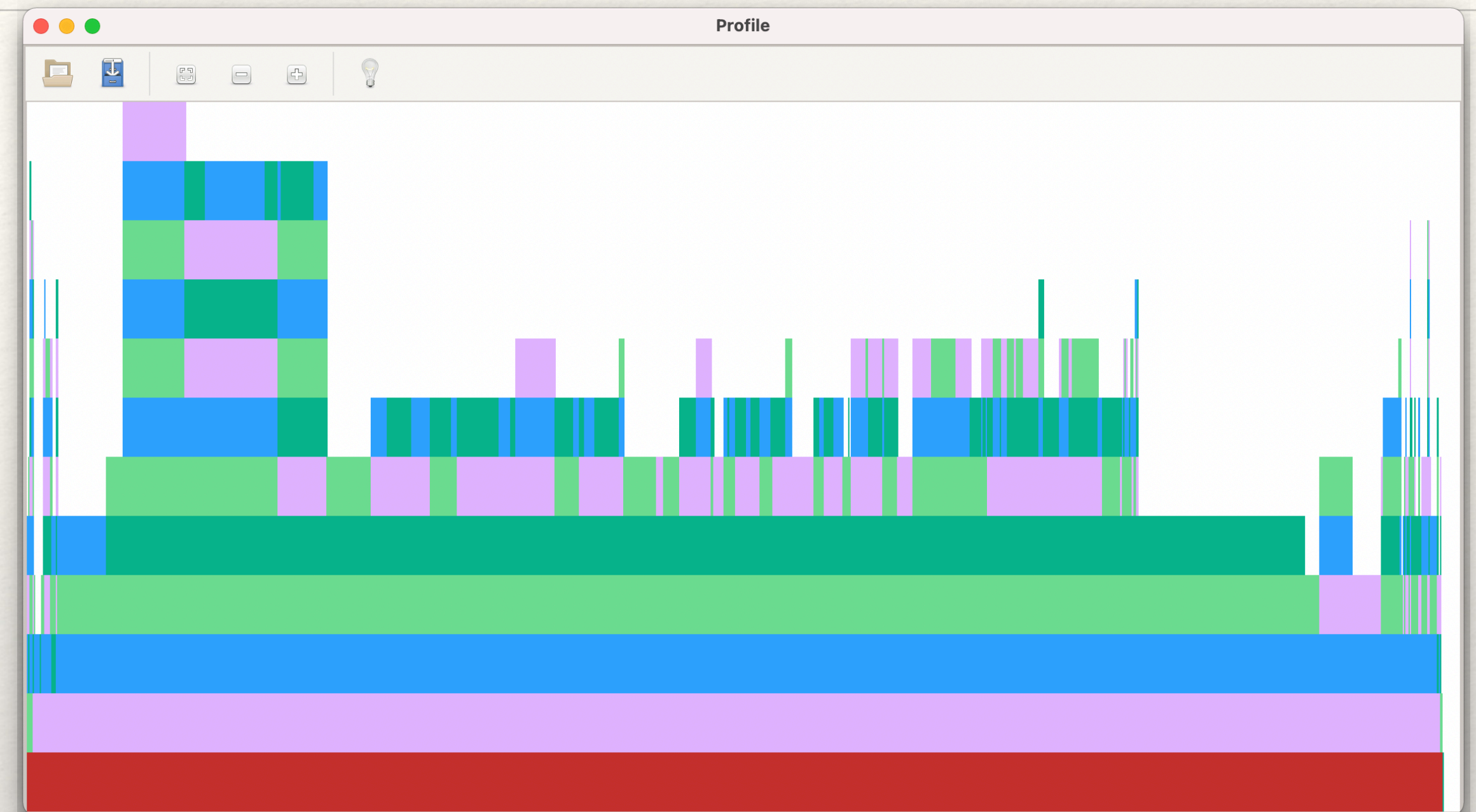
X-Ray Benchmark Results

- ❖ Between the Cuda 'friendly' geometry model and original we loose ~16%
- ❖ (*) Boolean solids not supported

	time(s)	speedup	node
VecGeom C++	39.4		MacBook Core i7
Geom4hep Julia	38.2	3%	MacBook Core i7
CuGeom (*) CPU	44.5	-12%	MacBook Core i7
CuGeom (*) CPU	69.0		<u>lcgapp-centos7-x86-64-gpu-02.cern.ch</u> (T4)
CuGeom (*) GPU	5.0	13.5 x	<u>lcgapp-centos7-x86-64-gpu-02.cern.ch</u> (T4)

Performance Optimisation

- ❖ Performance measurement and optimization is very present in Julia
- ❖ From the built-in `@time`, `@code_llvm`, ... macros to many packages facilitating benchmarking and profiling
 - ❖ BenchmarkTools, Profile, ProfileView, etc.
 - ❖ In general extremely easy to use. E.g. `@profile myFunc(...)`
- ❖ Hunting undesired memory allocations requires some practice 😞



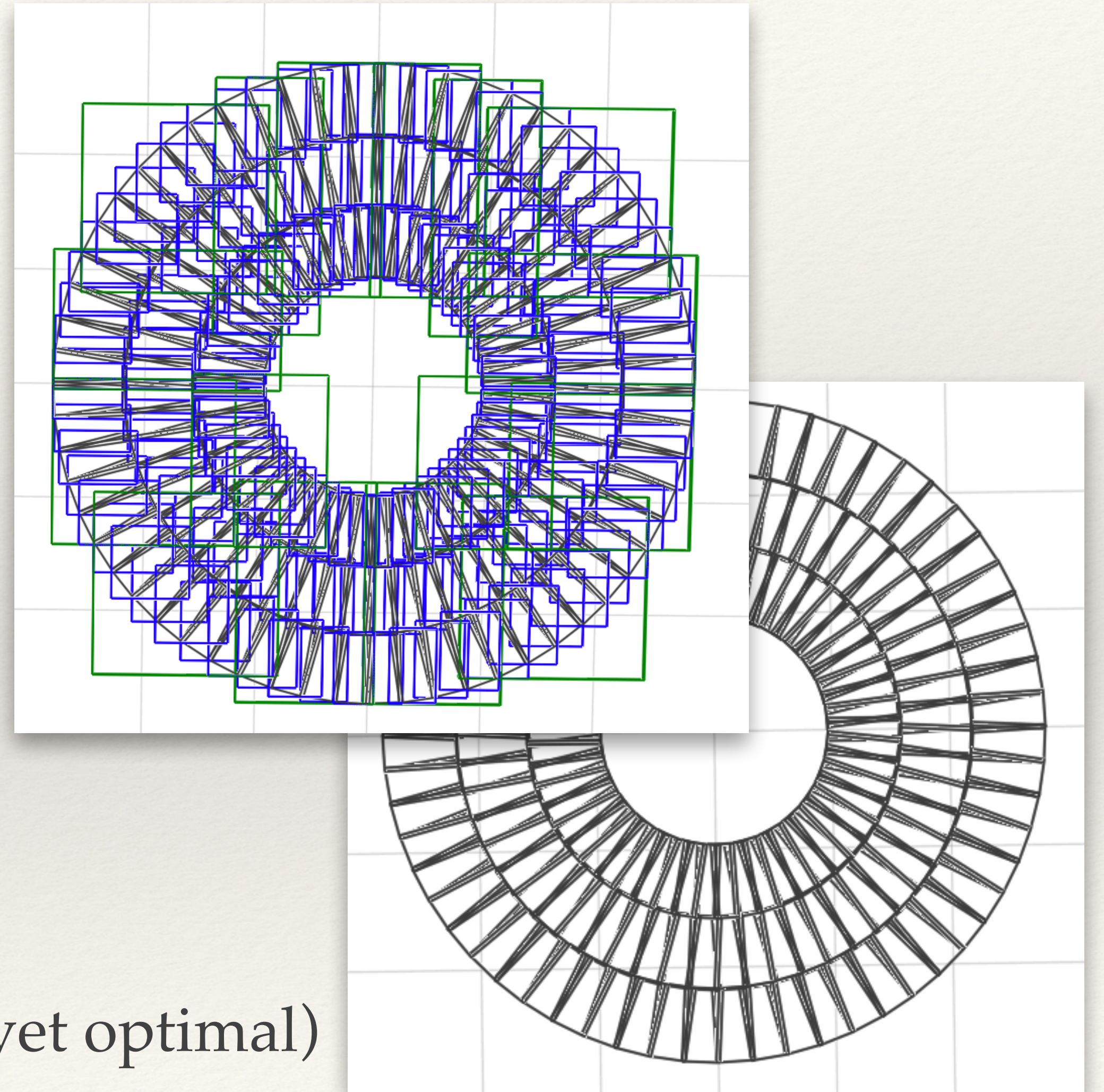
```
julia> @benchmark generateXRay(volume, 1e4, 3) seconds=200
BenchmarkTools.Trial: 517 samples with 1 evaluation.
Range (min ... max): 360.223 ms ... 430.592 ms | GC (min ... max): 0.00% ...
0.00%
Time (median): 383.652 ms | GC (median): 0.00%
Time (mean ± σ): 387.049 ms ± 13.212 ms | GC (mean ± σ): 0.00% ±
0.00%

Histogram: frequency by time
360 ms 423 ms <

Memory estimate: 81.78 KiB, allocs estimate: 17.
```

Accelerating Structures

- ❖ Implemented BVH to speed-up when volumes has many daughters
 - ❖ Algorithm adapted from a raytracing on triangular mesh exercise in Julia
- ❖ Example:
 - ❖ Strips1_12_L2 has 170 daughters
 - ❖ Generating XRay (z-direction)
 - ❖ TrivialNavigator: 31.8 s
 - ❖ BVHNavigator: 10.7 s (most probably not yet optimal)



Slow Startup

- ❖ Julia suffers from long startup times (so called time-to-first-plot)
 - ❖ This is due to the fact that despite pre-compilation (when installing packages and dependencies) the JIT (llvm) needs to produce concrete executable code for concrete types at run-time
- ❖ PackageCompiler.jl mitigates the problem by creating a *sysimage*
 - ❖ Compilation time: 8m
 - ❖ Image size: 480 MB

```
julia> PackageCompiler.create_sysimage(["Geom4hep"];
sysimage_path="Geom4hep.so", precompile_execution_file="examples/
XRay.jl")

$ time julia examples/DrawDetector.jl

real    1m12.293s
user    1m10.338s
sys     0m1.318s

$ time julia --sysimage Geom4hep.so examples/DrawDetector.jl

real    0m6.438s
user    0m5.862s
sys     0m0.555s
```

Preliminary Conclusions

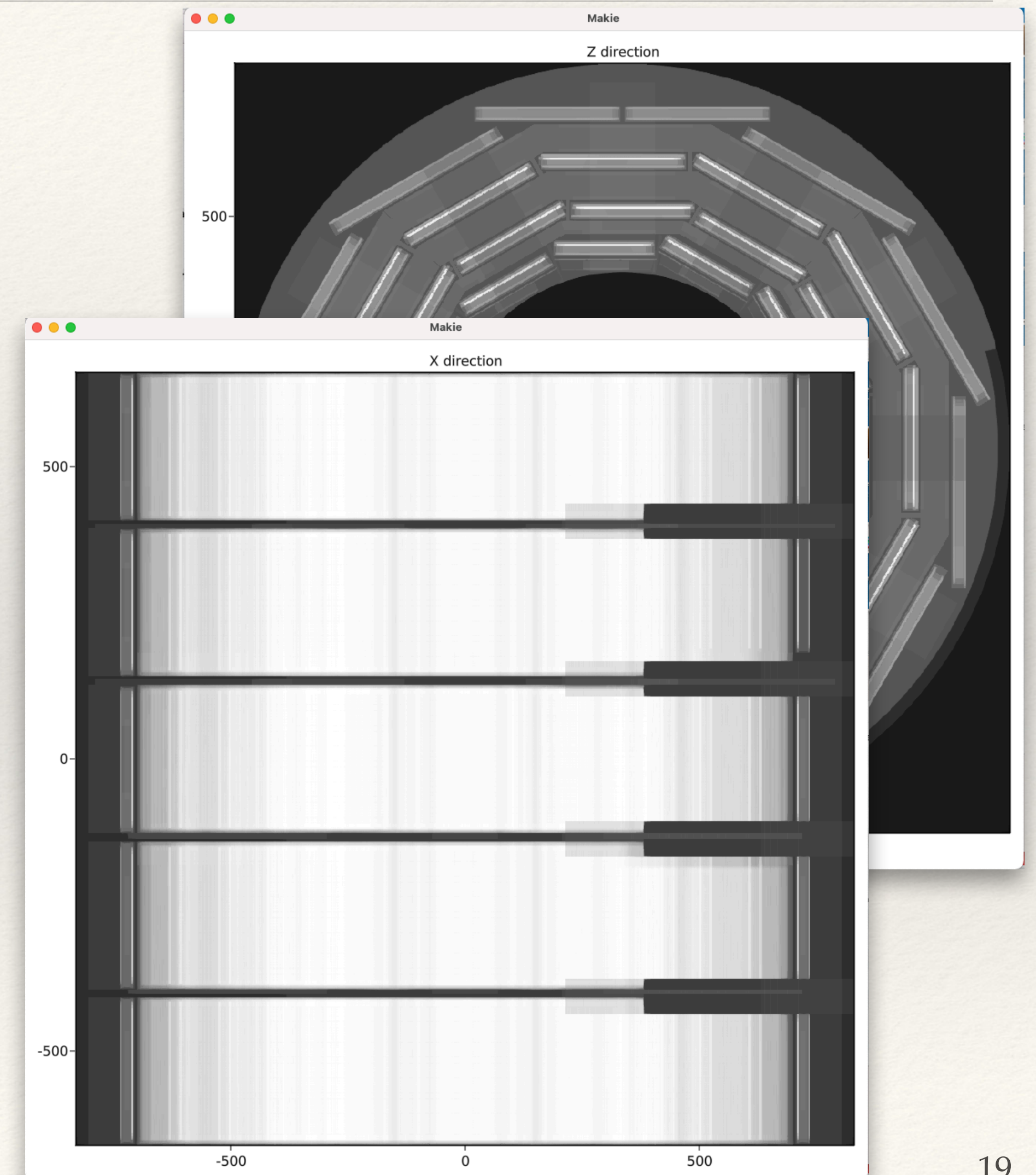
- ❖ Very nice language, ideal for scientific programming, compact
 - ❖ 6.5k LOC in Geom4hep vs. 144k LOC in VecGeom
- ❖ Very productive
 - ❖ Interactive development (REPL - read-eval-print loop together with Revise.jl)
 - ❖ Excellent integration with Visual Studio Code
- ❖ Excellent reuse paradigm
 - ❖ Just-in-time compilation, multiple-dispatch, etc.
 - ❖ Many modules ready to be used
- ❖ But, some of the common C++ data structures not optimal at all (see next)

Moving up a gear

- ❖ CMS geometry
 - ❖ cms2018.gdml, Muon Barrel detector (MB) - missing *polyhedra* and *torus*
 - ❖ X-ray in x-direction 1000x1000 pixels
- ❖ Numbers are not as nice as before
 - ❖ The BVH implementation perhaps needs more tuning

	trivial navigator	BVH navigator
VecGeom - C++	414 s	60 s
Geom4hep - Julia	1024 s	368 s

- ❖ Where is the performance problem?
 - ❖ 2.9 G memory allocations (~2900 allocations / ray)
 - ❖ Dynamic dispatch



Looking in more detail

- ❖ The memory allocations are triggered by the **Boolean shapes**
- ❖ CMS geometry has booleans (unions and subtractions), not in TrackML
- ❖ Union **Shape** has abstract Boolean elements and this triggers dynamic dispatch and memory allocations at runtime
- ❖ Ongoing discussion at [issue #1](#)

```
struct BooleanUnion{T, SL<:AbstractShape, SR<:AbstractShape} <: AbstractShape{T}
  left::SL          # the mother (or left) volume A
  right::SR         # (or right) volume B
  transformation::Transformation3D{T} # placement of "right" wrt "left"
end
```

```
#---Shape-----
const Shape{T} = Union{NoShape{T},
                      Box{T},
                      Trd{T},
                      Trap{T},
                      Tube{T},
                      Cone{T},
                      Polycone{T},
                      CutTube{T},
                      BooleanUnion{T},
                      BooleanIntersection{T},
                      BooleanSubtraction{T}} where T<:AbstractFloat
```

```
#---Volume-----
struct VolumeP{T<:AbstractFloat, PV}
  label::String
  shape::Shape{T}          # Reference to the actual shape
  material::Material{T}   # Reference to material
  daughters::Vector{PV}   # List of daughter volumes
end
```

```
#---PlacedVolume-----
struct PlacedVolume{T<:AbstractFloat}
  idx::Int64              # Daughter index
  transformation::Transformation3D{T} # Transformation wrt mother
  volume::VolumeP{T, PlacedVolume{T}} # Reference to volume
end
```

After a long series of debugging and trials

	trivial	BVH	
VecGeom - C++	414 s	60 s	
Geom4hep - Julia	1024 s (2.9 Ga)	368 s (2.9 Ga)	
change #1	768 s (10 a)	223 s (10 a)	Non-parametric BooleanShapes
change #2	688 s (10 a)	218 s (10 a)	Transformation3D noop
change #3	631 s (10 a)	211 s (10 a)	Inlines

Not directly comparable since not the same BVH algorithm / implementation

- ❖ With non-parametric BooleanShapes (see [thread](#)) the allocations disappear, but performance is not fully recovered
 - ❖ Large overhead accounted by the 'dynamic dispatch'. Ideally a C++ virtual call would be more efficient
- ❖ Many other promising changes have been tried without much success
 - ❖ manual type splitting (if XXX isa Box; elseif XXX is a Tube; ...)
 - ❖ make use of Virtual.jl
 - ❖ make use FunctionWrappers.jl to create a sort of virtual table
 - ❖

```
struct BooleanShape{T, OP} <: AbstractShape{T}
  left::Union{BaseShape{T}, BooleanShape{T}} # left volume A
  right::Union{BaseShape{T}, BooleanShape{T}} # right volume B
  transformation::Transformation3D{T} # placement of right wrt left
end
```

Performance is not yet C++ equivalent but getting acceptable !

Conclusions

- ❖ I personally like
 - ❖ The **high-level expressivity** and the adequacy for scientific programming
 - ❖ The **huge re-use potential** due to the multiple-dispatch capability, which is completely missing in C++
 - ❖ As a consequence, there exists a **large number of fairly complex packages** developed by the community that can easily be re-used to produce large applications
 - ❖ The fast and very **convenient development cycle** (using REPL and the help of Revise.jl)
- ❖ I personally dislike:
 - ❖ The **quick lost of performance**. Infinitesimal changes in the code can spoil the performance by many factors (mainly memory allocations triggered by type instabilities) but still having a working program
 - ❖ **Common HEP data structures are not obvious**. They can be implemented but they are complicated (tricky) and probably with degradation of performance (no virtual tables)
 - ❖ Very **slow startup** of applications and the impossibility to have a self-contained executable