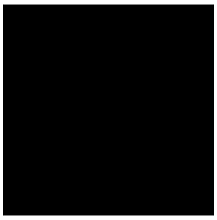
```julia
# using Images,ImageCore,ImageView,ImageDraw
```

```julia
using Colors,ColorVectorSpace,Images,ImageDraw
```

```
BoundaryFill
```

```julia
begin
abstract type AbstractPolyFillAlgorithm end
img = zeros(RGB, 7, 7)
expected = copy(img)
expected[2:6, 2] .= RGB{N0f8}(1)
expected[2:6, 6] .= RGB{N0f8}(1)
expected[2, 2:6] .= RGB{N0f8}(1)
expected[6, 2:6] .= RGB{N0f8}(1)
verts = [CartesianIndex(2, 2), CartesianIndex(2, 6),CartesianIndex(6, 6),
CartesianIndex(6, 2),CartesianIndex(2,2)]

struct BoundaryFill{T<:Colorant} <: AbstractPolyFillAlgorithm
    x::Int
    y::Int
    fill_color::T
    boundary_color::T
    function BoundaryFill(x::Int,y::Int,fill_color::T,boundary_color::T) where
{T<:Colorant}
        println("Test 1")
        new{T}(x,y,fill_color,boundary_color)
    end
end
function
BoundaryFill(;x::Int=0,y::Int=0,fill_color::Colorant=RGB(1),boundary_color::Colorant
=RGB(1))
        BoundaryFill(x,y,fill_color,boundary_color)
end
end
```

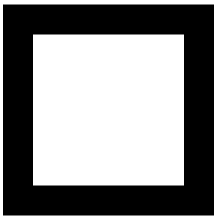


```julia
img
```

```
Main.workspace244.draw
```

```julia
begin

"""
    Main boundary fill

"""
```

```

function (f::BoundaryFill)(res::AbstractArray{T,2},verts::Vector{CartesianIndex{2}},x::Int,y::Int,fill_color::T,boundary_color::T) where {T<:Colorant}
    println(x, y, f.fill_color, f.boundary_color,res[y, x],"\n",f )
    if (res[y, x] != f.boundary_color && res[y, x] != f.fill_color)
        if checkbounds(Bool, res, y, x) res[y, x] = f.fill_color end
         f(res,verts,  x + 1, y, fill_color, boundary_color)
         f(res,verts,  x, y + 1, fill_color, boundary_color)
         f(res,verts,  x - 1, y, fill_color, boundary_color)
         f(res,verts,  x, y - 1, fill_color, boundary_color)
end
   res
end


# function draw!(img::AbstractArray{T,2}, verts::Array{CartesianIndex{2},1}, f::AbstractPolyFillAlgorithm;connectverts::Bool = true) where T <: Colorant
#     res = copy(img)
#     f(res, verts, f.x, f.y, f.fill_color, f.boundarycolor)
# end

# function draw(img::AbstractArray{T,2}, verts::Array{CartesianIndex{2},1}, f::AbstractPolyFillAlgorithm) where T <: Colorant
#     res = copy(img)
#     f(res, verts, f.x, f.y, f.fill_color, f.boundarycolor)
# end
# function connectvertices(res::AbstractArray{T,2},verts::Array{CartesianIndex{2},1})where {T<:Colorant}
#   fill_color=RGB(1)
#   for i in 1:length(verts)-1
#       draw!(res, LineSegment(verts[i], verts[i+1]), fill_color)
#     end
# end

"""
    Main API
    draw(img::AbstractArray{T,2}, verts::Vector{CartesianIndex{2}}, f::AbstractPolyFillAlgorithm)

"""
function draw(img::AbstractArray{T,2}, verts::Vector{CartesianIndex{2}}, f::AbstractPolyFillAlgorithm;connectverts::Bool = true) where {T<:Colorant}
   res = copy(img)
   if(connectverts==true)
        for i in 1:length(verts)-1
            draw!(res, LineSegment(verts[i], verts[i+1]), f.fill_color)
        end
   end
   f(res, verts,f.x,f.y,f.fill_color,f.boundary_color)
end
end
```



```
draw(img,verts,BoundaryFill(x=4,y=4,fill_color=RGB(1),boundary_color=RGB(1));connectverts=true)# connectvertices(res,verts))
```

```
# begin
# struct floodfill{T<:Colorant} <: AbstractPolyFillAlgorithm
```

```julia
#   x::Int
#   y::Int
#   fill_color::T
#     boundarycolor::T
#     # function boundaryfill(boundarycolor::T) where {T<:Colorant}
#     #     boundarycolor < 0 && throw(ArgumentError("window_size should be non-negative."))
#     #     println("Test 1")
#     #     new{T}(boundarycolor)
#     # end
# end

# function floodfill(;boundarycolor::Colorant=RGB(1))
#     println("Object Creation")
#     floodfill(x,y,fill_color,boundarycolor)
# end
# # function floodfill(x::Int, y::Int, fill_color::T, boundary_color::T) where T<:Colorant
# #     println("Enter Boundary Fill")
# #     if (res[y, x] != boundary_color && res[y, x] !=fill_color)
# #             if checkbounds(Bool, res, y, x) res[y, x] = fill_color end
# #              boundaryfill(x + 1, y, fill_color, boundary_color)
# #              boundaryfill(x, y + 1, fill_color, boundary_color)
# #              boundaryfill(x - 1, y, fill_color, boundary_color)
# #              boundaryfill(x, y - 1, fill_color, boundary_color)
# #     end
# #     img
# # end
# function floodfillcolor(res, x, y, current_color, fill_color)
#            if (res[y, x] != current_color || res[y, x] == fill_color) return end
#            if checkbounds(Bool, res, y, x) res[y, x] = fill_color end
#            floodfillcolor(res , x + 1, y, current_color, fill_color)
#            floodfillcolor(res , x - 1, y, current_color, fill_color)
#            floodfillcolor(res , x, y + 1, current_color, fill_color)
#            floodfillcolor(res , x, y - 1, current_color, fill_color)
# end

# function floodfill(res, x, y, fill_color)
#            current_color = res[y,x]
#          current_color= RGB(0)
#            floodfillcolor(res, x, y, current_color, fill_color)
# end

# function (f::floodfill)(res::AbstractArray{T,2},verts::Vector) where {T<:Colorant}
#     println("Enter boundary fill API")
#   y=4
#   x=4
#   # println(verts)
#   # println(img)
#   imshow(res)
#   fill_color = RGB(1)
#   boundary_color = RGB(1)
#   for i in 1:length(verts)-1
#       draw!(res, LineSegment(verts[i], verts[i+1]), fill_color)
#       imshow(res)
#     end
#   floodfill(x::Int, y::Int, fill_color::T, boundary_color::T)
#    res
# end
# end
```

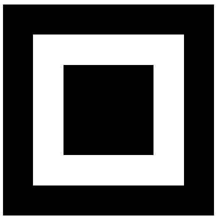
```
UndefVarError: boundaryfill not defined

  1. top-level scope @ Local: 1
```

```julia
draw(img,verts,boundaryfill(x=4,y=4,fill_color=RGB(1),boundarycolor=RGB(1)))
```

```
Array{CartesianIndex{2},1}
```

```
typeof(verts)
```

```
# begin
# """
#     AbstractPolyFillAlgorithm

# The root of polygon filling algorithms type system
# """

# abstract type AbstractPolyFillAlgorithm end


# struct boundaryfill{T<:Colorant} <: AbstractPolyFillAlgorithm
#     boundarycolor::T
#     # function boundaryfill(boundarycolor::T) where {T<:Colorant}
#     #     boundarycolor < 0 && throw(ArgumentError("window_size should be non-negative."))
#     #     println("Test 1")
#     #     new{T}(boundarycolor)
#     # end
# end

# function boundaryfill(;boundarycolor::Colorant=RGB(1))
#     println("Object Creation")
#     boundaryfill(boundarycolor)
# end
# function boundaryfill(x::Int, y::Int, fill_color::T, boundary_color::T) where T<:Colorant
#   println("Enter Boundary Fill")
#     if (res[y, x] != boundary_color && res[y, x] !=fill_color)
#             if checkbounds(Bool, res, y, x) res[y, x] = fill_color end
#              boundaryfill(x + 1, y, fill_color, boundary_color)
#              boundaryfill(x, y + 1, fill_color, boundary_color)
#              boundaryfill(x - 1, y, fill_color, boundary_color)
#              boundaryfill(x, y - 1, fill_color, boundary_color)
#     end
#     img
# end

# function (f::boundaryfill)(img::AbstractArray{T,2},verts::Vector) where {T<:Colorant}
#     println("Enter boundary fill API")
#   y=4
#   x=4
#   # println(verts)
#   # println(img)
#   fill_color = RGB(1)
#   boundary_color = RGB(1)
#   for i in 1:length(verts)-1
#       draw!(res, LineSegment(verts[i], verts[i+1]), fill_color)
#       imshow(res)
#     end


#   boundaryfill(x::Int, y::Int, fill_color::T, boundary_color::T)
#    res
# end
# function drawnewtype(img::AbstractArray{T,2},verts::Vector{CartesianIndex{2}},f::AbstractPolyFillAlgorithm) where T <: Colorant
#     println("Test 2")
#     f(img,verts);
# end
# function drawnewtype(img::AbstractArray{T,2},
#     verts::Vector{CartesianIndex{2}},
#     f::AbstractPolyFillAlgorithm) where {T<:Colorant}
#    println("Start")
#    f(res,verts)
#    # println(verts)
#    res
# end
# end
```



```
expected
```

```
# begin

# vert=CartesianIndex{2}[]
#   push!(vert, CartesianIndex(2,2))
#   push!(vert, CartesianIndex(3,2))
#   push!(vert, CartesianIndex(4,2))
#   push!(vert, CartesianIndex(5,2))
#   push!(vert, CartesianIndex(6,3))
#   push!(vert, CartesianIndex(7,4))
#   push!(vert, CartesianIndex(8,4))
#   push!(vert, CartesianIndex(9,3))
#   push!(vert, CartesianIndex(10,2))
#   push!(vert, CartesianIndex(11,3))
#   push!(vert, CartesianIndex(12,4))
#   push!(vert, CartesianIndex(13,5))
#   push!(vert, CartesianIndex(12,6))
#   push!(vert, CartesianIndex(11,7))
#   push!(vert, CartesianIndex(10,6))#u
#   push!(vert, CartesianIndex(9,7))
#   push!(vert, CartesianIndex(8,8))
#   push!(vert, CartesianIndex(8,9))
#   push!(vert, CartesianIndex(8,10))
#   push!(vert, CartesianIndex(8,11))
#   push!(vert, CartesianIndex(9,11))
#   push!(vert, CartesianIndex(10,11))
#   push!(vert, CartesianIndex(11,11))
#   push!(vert, CartesianIndex(12,11))
#   push!(vert, CartesianIndex(13,11))
#   push!(vert, CartesianIndex(13,12))
#   push!(vert, CartesianIndex(13,13))
#   push!(vert, CartesianIndex(12,13))
#   push!(vert, CartesianIndex(11,13))
#   push!(vert, CartesianIndex(10,13))
#   push!(vert, CartesianIndex(9,13))
#   push!(vert, CartesianIndex(8,13))
#   push!(vert, CartesianIndex(7,13))
#   push!(vert, CartesianIndex(6,13))
#   push!(vert, CartesianIndex(6,12))
#   push!(vert, CartesianIndex(6,11))#u
#   push!(vert, CartesianIndex(5,11))
#   push!(vert, CartesianIndex(4,11))
#   push!(vert, CartesianIndex(3,11))
#   push!(vert, CartesianIndex(2,11))#u
#   push!(vert, CartesianIndex(3,10))
#   push!(vert, CartesianIndex(4,9))
#   push!(vert, CartesianIndex(5,8))
#   push!(vert, CartesianIndex(4,7))
#   push!(vert, CartesianIndex(4,6))
#   push!(vert, CartesianIndex(5,5))
#   push!(vert, CartesianIndex(4,4))
#   push!(vert, CartesianIndex(3,3))
#   push!(vert, CartesianIndex(2,2))

# end
```

```
# begin
# struct edgetabletuple
#   initial::CartesianIndex
#   final::CartesianIndex
# end

# img = draw(zeros(Gray{Bool},14,14), Polygon(vert));
# function createedgetable()
# edgetable= []
# push!(edgetable, edgetabletuple(CartesianIndex(2,2),CartesianIndex(5,2)))
# push!(edgetable, edgetabletuple(CartesianIndex(5,2),CartesianIndex(7,4)))
# push!(edgetable, edgetabletuple(CartesianIndex(7,4),CartesianIndex(8,4)))
# push!(edgetable, edgetabletuple(CartesianIndex(8,4),CartesianIndex(10,2)))
# push!(edgetable, edgetabletuple(CartesianIndex(10,2),CartesianIndex(13,5)))
# push!(edgetable, edgetabletuple(CartesianIndex(13,5),CartesianIndex(11,7)))
# push!(edgetable, edgetabletuple(CartesianIndex(11,7),CartesianIndex(10,6)))
# push!(edgetable, edgetabletuple(CartesianIndex(10,6),CartesianIndex(8,8)))
# push!(edgetable, edgetabletuple(CartesianIndex(8,8),CartesianIndex(8,11)))
# push!(edgetable, edgetabletuple(CartesianIndex(8,11),CartesianIndex(13,11)))
# push!(edgetable, edgetabletuple(CartesianIndex(13,11),CartesianIndex(13,13)))
# push!(edgetable, edgetabletuple(CartesianIndex(13,13),CartesianIndex(6,13)))
# push!(edgetable, edgetabletuple(CartesianIndex(6,13),CartesianIndex(6,11)))
# push!(edgetable, edgetabletuple(CartesianIndex(6,11),CartesianIndex(2,11)))
# push!(edgetable, edgetabletuple(CartesianIndex(2,11),CartesianIndex(5,8)))
# push!(edgetable, edgetabletuple(CartesianIndex(5,8),CartesianIndex(4,8)))
# push!(edgetable, edgetabletuple(CartesianIndex(4,8),CartesianIndex(4,7)))
# push!(edgetable, edgetabletuple(CartesianIndex(4,7),CartesianIndex(4,6)))
# push!(edgetable, edgetabletuple(CartesianIndex(4,6),CartesianIndex(5,5)))
# push!(edgetable, edgetabletuple(CartesianIndex(5,5),CartesianIndex(2,2)))
# return edgetable
# end
# #find ymin and ymax
# function yminmax(vert)
#   ymax = vert[1][1];
#   ymin = vert[1][1];
#   for i in 1:length(vert)
#      if(vert[i][2] > ymax)
#        ymax = vert[i][2]
#      elseif(vert[i][2] < ymin)
#        ymax = vert[i][2]
#      end
#   end
#   return ymin, ymax
# end

# function findintersections(edgetable, yvalue)
#   points =[]
#     for i in 1:length(edgetable)
#       if(edgetable[i].final[2] > yvalue && edgetable[i].initial[2] > yvalue)
#           continue
#       end
#       if(edgetable[i].final[2] <= yvalue && edgetable[i].initial[2] <= yvalue)
#           continue
#       end
#       x=1
#       deltay = edgetable[i].final[1]-edgetable[i].initial[1]
#       deltax = edgetable[i].final[2]-edgetable[i].initial[2]
#       alpha = deltay/deltax
#       constant = edgetable[i].initial[1]
#       x = alpha * (yvalue - edgetable[i].initial[2]) + constant
#       if (x == -Inf || isnan(x) || x == Inf) continue end
#       push!(points,CartesianIndex(ceil(Int,x),yvalue))
#   end
#   return points
# end
# function timetocolor(points, fill_color::T)where T<:Colorant
#    for i in 1:2:length(points)-1
#     draw!(img, LineSegment(points[i], points[i+1]),fill_color)
#    end
# end
# function scanline(vert, fill_color::T) where T<:Colorant
#   ymin, ymax = yminmax(vert)
#   edgetable = createedgetable()
#   pyramid = []
#   for i in ymin:ymax
#       points = findintersections(edgetable, i)
#       points = sort!(points, by = x -> x[1])
#       timetocolor(points, fill_color)
#       push!(pyramid,img[:,:])
#   end
#   pyramid
# end
# fill_color = Gray{Bool}(1.0)

# end
```

```
# pyramid = scanline(vert, fill_color)
```

```
# connectvertices(res,verts)
```