# An Interpretation of The Olive Trees

# By Lexie Hassien

# References for generating flow fields:

# https://cran.r-project.org/web/packages/aRtsy/readme/README.html#flow-fields

# https://tylerxhobbs.com/essays/2020/flow-fields

# https://observablehq.com/@esperanc/flow-fields

library(aRtsy)

library(dplyr)

library(ggplot2)

s1 <- Sys.time()

##################################################################

# internal functions in aRtsy R package

# https://rdrr.io/cran/aRtsy/src/R/internal.R

# This function generates noise

.noise <- function(dims, n = 100, type = c("artsy-knn", "knn", "svm", "rf", "perlin", "cubic", "simplex", "worley"),

k = 20, limits = c(0, 1)) {

type <- match.arg(type)

if (type == "artsy-knn") {

if (length(dims) == 1) {

vec <- expand.grid(limits[1], seq(limits[1], limits[2], length.out = dims))

} else if (length(dims) == 2) {

vec <- expand.grid(seq(limits[1], limits[2], length.out = dims[1]), seq(limits[1], limits[2], length.out = dims[2]))

}

z <- cpp\_knn(stats::runif(n), stats::runif(n), stats::runif(n), vec[, 1], vec[, 2], k)

} else if (type == "svm") {

train <- data.frame(

x = stats::runif(n, limits[1], limits[2]),

y = stats::runif(n, limits[1], limits[2]),

z = stats::runif(n, limits[1], limits[2])

)

fit <- e1071::svm(formula = z ~ x + y, data = train)

xsequence <- seq(limits[1], limits[2], length = dims[1])

ysequence <- seq(limits[1], limits[2], length = dims[2])

canvas <- expand.grid(xsequence, ysequence)

colnames(canvas) <- c("x", "y")

z <- predict(fit, newdata = canvas)

} else if (type == "knn") {

train <- data.frame(

x = stats::runif(n, limits[1], limits[2]),

y = stats::runif(n, limits[1], limits[2]),

z = stats::runif(n, limits[1], limits[2])

)

fit <- kknn::train.kknn(formula = z ~ x + y, data = train, kmax = k)

xsequence <- seq(limits[1], limits[2], length = dims[1])

ysequence <- seq(limits[1], limits[2], length = dims[2])

canvas <- expand.grid(xsequence, ysequence)

colnames(canvas) <- c("x", "y")

z <- predict(fit, newdata = canvas)

} else if (type == "rf") {

train <- data.frame(

x = stats::runif(n, limits[1], limits[2]),

y = stats::runif(n, limits[1], limits[2]),

z = stats::runif(n, limits[1], limits[2])

)

fit <- randomForest::randomForest(formula = z ~ x + y, data = train)

xsequence <- seq(limits[1], limits[2], length = dims[1])

ysequence <- seq(limits[1], limits[2], length = dims[2])

canvas <- expand.grid(xsequence, ysequence)

colnames(canvas) <- c("x", "y")

z <- predict(fit, newdata = canvas)

} else if (type == "perlin") {

z <- ambient::noise\_perlin(dims, frequency = stats::runif(1, 0.001, 0.04))

z <- (z - range(z)[1]) / diff(range(z)) \* diff(limits) + limits[1]

} else if (type == "cubic") {

z <- ambient::noise\_cubic(dims, frequency = stats::runif(1, 0.001, 0.04))

z <- (z - range(z)[1]) / diff(range(z)) \* diff(limits) + limits[1]

} else if (type == "simplex") {

z <- ambient::noise\_simplex(dims, frequency = stats::runif(1, 0.001, 0.04))

z <- (z - range(z)[1]) / diff(range(z)) \* diff(limits) + limits[1]

} else if (type == "worley") {

z <- ambient::noise\_worley(dims)

z <- (z - range(z)[1]) / diff(range(z)) \* diff(limits) + limits[1]

}

return(matrix(z, nrow = dims[1], ncol = dims[2]))

}

# This function performs validation checks on the standardized input arguments of a function

.checkUserInput <- function(background = NULL, resolution = NULL, iterations = NULL) {

if (!is.null(background) && length(background) != 1) {

stop("'background' must be a single character")

}

if (!is.null(resolution) && (resolution < 1 || resolution %% 1 != 0)) {

stop("'resolution' must be a single value > 0")

}

if (!is.null(iterations) && (iterations < 1 || iterations %% 1 != 0)) {

stop("'iterations' must be a single integer > 0")

}

}

# This function turns a matrix into a data frame with columns x, y, and z

.unraster <- function(x, names) {

newx <- data.frame(x = rep(seq\_len(ncol(x)), times = ncol(x)), y = rep(seq\_len(nrow(x)), each = nrow(x)), z = c(x))

colnames(newx) <- names

return(newx)

}

# This function takes a point (x, y) and returns a warped point (x, y)

.warp <- function(p, warpDistance, resolution, angles = NULL, distances = NULL) {

if (is.null(angles)) {

angles <- .noise(c(resolution, resolution), type = sample(c("svm", "perlin", "cubic", "simplex"), size = 1), limits = c(-pi, pi))

} else if (is.character(angles)) {

angles <- .noise(c(resolution, resolution), type = angles, limits = c(-pi, pi))

} else if (is.matrix(angles)) {

if (nrow(angles) != resolution || ncol(angles) != resolution) {

stop(paste0("'angles' should be a ", resolution, " x ", resolution, " matrix"))

}

}

if (is.null(distances)) {

distances <- .noise(c(resolution, resolution), type = sample(c("knn", "perlin", "cubic", "simplex"), size = 1), limits = c(0, warpDistance))

} else if (is.character(distances)) {

distances <- .noise(c(resolution, resolution), type = distances, limits = c(0, warpDistance))

} else if (is.matrix(distances)) {

if (nrow(distances) != resolution || ncol(distances) != resolution) {

stop(paste0("'distances' should be a ", resolution, " x ", resolution, " matrix"))

}

}

return(matrix(c(p[, 1] + c(cos(angles)) \* c(distances), p[, 2] + c(sin(angles)) \* c(distances)), ncol = 2))

}

# This function returns a brownian motion line

.bmline <- function(n, lwd) {

x <- cumsum(stats::rnorm(n = n, sd = sqrt(1)))

x <- abs(x / stats::sd(x) \* lwd)

return(x)

}

cpp\_flow <- function(canvas, angles, lines, iters, ncolors, left, right, top, bottom, stepmax) {

.Call('\_aRtsy\_cpp\_flow', PACKAGE = 'aRtsy', canvas, angles, lines, iters, ncolors, left, right, top, bottom, stepmax)

}

#################

# code based on the source code for the canvas\_flow() function from the aRtsy package

set.seed(985) # 995, 992, 989, 985

sz <- 100

sequence <- seq(0, sz, length = sz)

grid <- expand.grid(sequence, sequence)

grid <- data.frame(x = grid[, 1], y = grid[, 2], z = 0)

left <- -sz

right <- sz

bottom <- -sz

top <- sz

ncols <- right - left

nrows <- top - bottom

background = "#fafafa"

outline <- "square"

lines <- 80000

lwd <- 0.5 # 0.5

stepmax <- 0.05

polar <- FALSE

angles <- NULL

iterations <- 10

colors1 <- c("#7297b2","#779ebb", "#78B1CF","#89A7C1", "#7b9fb7" )

colors2 <- c("#2F4F69","#2e456d", "#57668c","#2b4672", "#5f729d" )

colors3 <- c("#739977", "#476969", "#7a9480", "#7a9480", "#3c5741")

colors4 <- c("#8da29d", "#57865c", "#babf91", "#D0C685", "#E2E2A9")

colors12 <- c("#2F4F69","#2e456d", "#57668c","#2b4672", "#5f729d" )

colors23 <- c("#739977", "#476969", "#7a9480","#57865c", "#3c5741")

colors34 <- c("#2a3d48", "#50534a", "#26353d", "#4a5e67","#5b5949")

# # initialize angles

# option 1: pick a random noise initialization method

# angles <- .noise(

# dims = c(nrows, ncols), n = sample(100:300, 1),

# type = sample(c("knn", "svm", "perlin", "cubic", "simplex", "worley"), 1),

# limits = c(-pi, pi)

# )

# option 2: specify angle initialization method

angles <- .noise(

dims = c(nrows, ncols), n = sample(100:300, 1),

type = "cubic",

limits = c(-pi, pi)

)

# set up canvas

canvas <- matrix(NA, nrow = iterations \* lines, ncol = 5)

ncolors <- length(colors1)

canvas <- cpp\_flow(canvas, angles, lines, iterations, ncolors, left, right, top, bottom, stepmax)

canvas <- canvas[!is.na(canvas[, 1]), ]

for (j in seq\_len(lines)) {

index <- which(canvas[, 3] == j)

canvas[index, 5] <- .bmline(n = length(index), lwd)

}

# control the outline of the artwork

if (outline == "circle") {

canvas[which(sqrt(canvas[["x"]]^2 + canvas[["y"]]^2) > 175 / 2), "color"] <- background

} else if (outline == "square") {

canvas[which(canvas[["x"]] < -75 & canvas[["x"]] > 75 & canvas[["y"]] < -75 & canvas[["y"]] > 75), "color"] <- background

}

canvas <- as.data.frame(canvas)

colnames(canvas) <- c("x", "y", "z", "color", "size")

# set the different color schemes based on y value

sky <- which(canvas$y > 40)

canvas$color[sky] <- sample(colors1, length(sky), replace=T)

mountains <- which(canvas$y < 40 & canvas$y > 10)

canvas$color[mountains] <- sample(colors2, length(mountains), replace=T)

trees <- which(canvas$y < 10 & canvas$y > -40)

canvas$color[trees] <- sample(colors3, length(trees), replace=T)

grass <- which(canvas$y < -40)

canvas$color[grass] <- sample(colors4, length(grass), replace=T)

##### create sine waves to smooth transitions between colors

#skyjitter <- rnorm(length(canvas$x), 0, 3)

sine <- sin(0.1\*canvas$x)\*5 + 50

sine2 <- sin(0.1\*canvas$x)\*10 + 25

yjitter <- canvas$y

skytest <- which(yjitter < sine & yjitter > sine2 )

canvas$color[skytest] <- sample(colors12, length(skytest), replace=T)

treejitter <- rnorm(length(canvas$x), 0, 9)

sine <- sin(0.3\*canvas$x)\*5 + 20

sine2 <- sin(0.3\*canvas$x)\*10 + 0

yjitter <- canvas$y + treejitter

treetest <- which(yjitter < sine & yjitter > sine2 )

canvas$color[treetest] <-sample(colors23, length(treetest), replace=T)

grassjitter <- rnorm(length(canvas$x), 0, 7)

sine <- 0.05\*canvas$x + (sin(0.01\*canvas$x)\*5 - 50 )

sine2 <- 0.05\*canvas$x + (sin(0.01\*canvas$x)\*10 - 30)

yjitter <- canvas$y + grassjitter

grasstest <- which(yjitter > sine & yjitter < sine2 )

canvas$color[grasstest] <- sample(colors34, length(grasstest), replace=T)

#########

# make the color of each line match from point to point

for (i in 1:length(unique(canvas$z))){

thisz <- which(canvas$z == i)

canvas$color[thisz] <- canvas$color[thisz[1]]

canvas$size[thisz] <- sample(seq(0.3,0.8, 0.1),1)

}

canvas2 <- canvas

colnames(canvas2) <- c('y','x','z','color','size')

canvas2$z <- canvas2$z + 20000

canvas12 <- rbind(canvas, canvas2)

usecanvas <- canvas

# plot xy coordinates

artwork <- ggplot2::ggplot(data = usecanvas, mapping = ggplot2::aes(x = x, y = y, group = factor(z))) +

ggplot2::geom\_path(linewidth = usecanvas[["size"]], color = usecanvas[["color"]], lineend = "round")

if (polar) {

artwork <- artwork + ggplot2::coord\_polar()

} else {

artwork <- artwork + ggplot2::coord\_cartesian(xlim = c(left, right), ylim = c(bottom, top))

}

# plot artwork

margin <- 0

artwork <- artwork +

ggplot2::theme(

axis.title = ggplot2::element\_blank(),

axis.text = ggplot2::element\_blank(),

axis.ticks = ggplot2::element\_blank(),

axis.line = ggplot2::element\_blank(),

legend.position = "none",

plot.background = ggplot2::element\_rect(fill = background, colour = background),

panel.border = ggplot2::element\_blank(),

panel.grid = ggplot2::element\_blank(),

plot.margin = ggplot2::unit(rep(margin, 4), "lines"),

strip.background = ggplot2::element\_blank(),

strip.text = ggplot2::element\_blank()

)

artwork

# Print to file using graphics device

tiff(filename = "OliveTree.tif",

width = 8.5, height = 11, units = "in",

compression = "none",

bg = "white", restoreConsole = TRUE,

type = "cairo", res = 300,

symbolfamily="default")

artwork

dev.off()

s2 <- Sys.time()

print(round(s2 - s1, 5))