Photogrammetric Potsherd Profiling

Karl Edwards

2017-10-23

Contents

Abstract

An archaeologist lamented the fact that students having any interest in programming typically go off to study computer science. At the same time, students who are fully committed to archaeology have neither the interest nor the capacity to develop the types of computational tools that would speed up the tedious, repetetive aspects of archaeological practice. Perhaps specialists, in collaboration with generalists, can accomplish more together than either could accomplish alone. For example, imagine how useful it would be to leverage recent advances in photogrammetry and machine learning for the purpose of more quickly and accurately measuring a batch of objects and arranging them into a provisional hierarchy. This article describes a simple method for photographing pottery vessels and for turning those photographs into 3-D models, from which we derive characteristic measurements that will be useful in the evaluation of object similarity, and, ultimately, classification. The example includes fully-annotated **R** code for manipulating the modeled object in space, obtaining cross-sections, and calculating characteristic quantities. Subsequent articles will describe how to quantify similarity between objects, present a principled method for determining feature importance, and suggest an iterative process for generating provisional archetypes and resolving any resulting mis-classifications.

Related Work

Photogrammetry Using Photoscan[1]

Björn K. Nilssen[2] combines photography with minimal manual modeling to create 3D models from of very large objects, such as sculptures and buildings. The Poor Man's Guide To Photogrammetry[3] illustrates in great detail the modeling of small objects.

[1] Photoscan is available at (http://www.agisoft.com)

[2] SketchUp with PhotoScan plugin (http://bknilssen.no/X/Photogrammetry/)

[3] Poor Man's Guide (http://bertrand-benoit.com/blog/the-poor-mans-guide-to-photogrammetry/)

Open Source Photogrammetry

For more control over the modeling process, the artist, Gleb Alexandrov, presents a photo scanning workflow based on open source projects VisualSFM[4], Meshlab[5], and Blender.[6]

[4] Visual Structure From Motion, by Changchang Wu(http://ccwu.me/vsfm/)

[5] Meshlab, the open source system for processing and editing 3D triangular meshes (http://www.meshlab.net)

[6] Blender, an open source 3D creation suite (https://www.blender.org)

Photogrammetry for Archeology

The nonprofit corporation, chi[7], demonstrates photogrammetry in the field of cultural heritage. The Archaeology Data Service at University of York has published a number of Guides to Good Practice, including Close-Range Photogrammetry: A Guide to Good Practice.[8] Potsherd: Atlas of Roman Pottery[9] recommends CCD flatbed scanning for small, flat objects.

[7] Cultural Heritage Imaging

[8] Guides to Good Practice (http://guides.archaeologydataservice.ac.uk/g2gp/Phot 1)

[9] `http://potsherd.net/atlas`, P A Tyers, Sherd Scanning Advice (http://potsherd.net/atlas/topics/scanning)

Machine Learning for Archeology

A group of German researchers[10] describe how, using measurements obtained from a 3D-scanner, they have classified nearly 600 clay vessels from a Bronze-Age site in Saxony. Beginning with a pair-wise analysis of morphological features, they explain how to assess similarity between pairs of artifacts, and then find clusters of similar items. In the second phase, they develop a rational approach to calculating feature importance. Then they alternate between describing archetypes and classifying the artifacts using those preliminary definitions, adjusting the typology until the complete hierarchy emerges.

[10] Hörr, Lindinger, and Brunnett, in **Machine Learning Based Typology Development in Archaeology**

Photography

You will need:

- One or more potsherds to measure
- Location with plenty of natural light. See photography guidelines
- Work table
- Camera
- Tripod
- Turntable
- Dark background

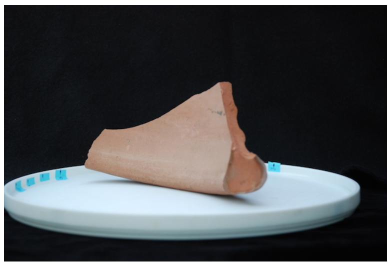


Figure 1: Using a Turntable

Prepare the photography area as follows:

- Put dark paper or cloth under and behind the turntable to serve as a backdrop
- Adjust the height of the camera on the tripod so the center of the lense is at the same height as the center of the artifact to be photographed

Photograph each artifact

- Place the pot (or potsherd) on the turntable.
- Even though the accompanying photo does not illustrate this, prop the artifact so that the rim of the pot is parallel to the turntable (Rim up or rim down is not so important)

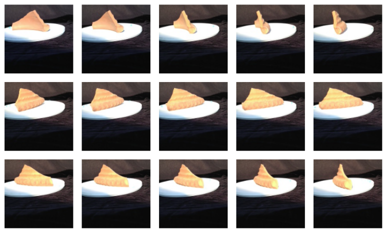


Figure 2: Every Ten Degrees

- To minimize distortion, place the artifact so that the center of the rim (or base) is approximately on the center of the turntable
- Take a series of photographs at roughly 10-degree intervals, resulting in 36 images per artifact

Model Creation

image files –> textured mesh –> stereolithography

The basic workflow is: (1) Take photographs from various perspectives, (2) Convert the photographs into a 3-Dimensional model, and, (3) since the model arrives as a textured mesh, convert it to stereolithography

Image Files –> Textured Mesh

- Upload images to ARC3D[11] web service

[11] If you are not already an ARC3D user, apply for a free account at `https://homes.esat.kuleuven.be/~visit3d/webservice/v2/request_login.php`

- Pour yourself a cup of coffee
- In a few minutes to a few hours, if all goes well, ARC3D will send you a textured mesh object

Textured Mesh –> Stereolithography

Enter the following command into a terminal window:

```
./meshconv textured_mesh.obj -c stl -o stereolithograph
```

This tells the conversion utility[12] three things:

1. The object to use as input: *textured_mesh.obj*,
2. The action(s) to perform: convert to stereolithography format [**-c stl**],
3. Where to put the results: in a file called **-o stereolithograph[.stl]**

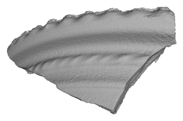


Figure 3: Textured Mesh Model

[12] The *meshconv* utility is available at (http://www.patrickmin.com/meshconv/)

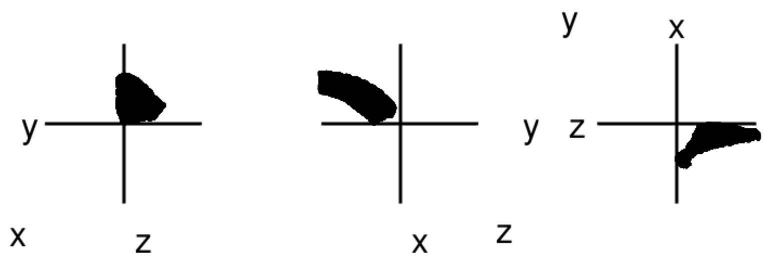
Cross-Sectioning

THE BASIC IDEA is to (1) orient the model squarely in the reference frame, (2) estimate the location of the center of the vessel, (3) extract relevant characteristics and dimensions from the profile.

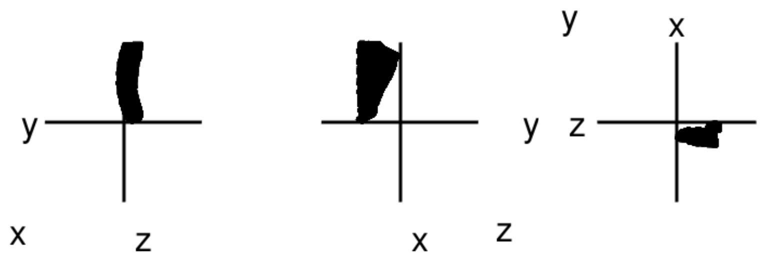
A. Align Model With Reference Frame

To find the best orientation for the model of the sherd, begin by measuring the extent of the model along each axis, and calculating the product of these dimensions. Then tilt the model slightly, recalculate the volume, and repeat until finding the minimum.

Initially, the object requires a box volume of at least 0.3653 units.

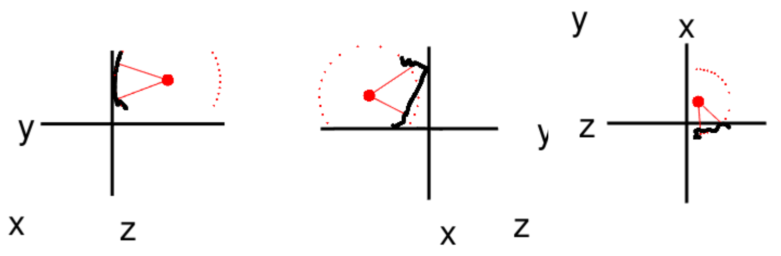


After alignment with the reference frame, the box is 0.1845 units.



B. Locate the Center of the Vessel

1. Extract a thin slice from the model, parallel with each axis
2. One of the resulting cross-sections should be roughly circular.
3. We will use this to estimate the center of the vessel and orientation of the axis of rotation



C. Re-Section

Taking the roundest cross-section as the top view and then re-sectioning the model through the vessel's axis of rotation results in a profile of the pot. In order to get the most information, cut at the tallest (widest, as shown here) portion of the sherd.

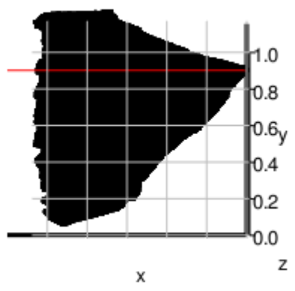
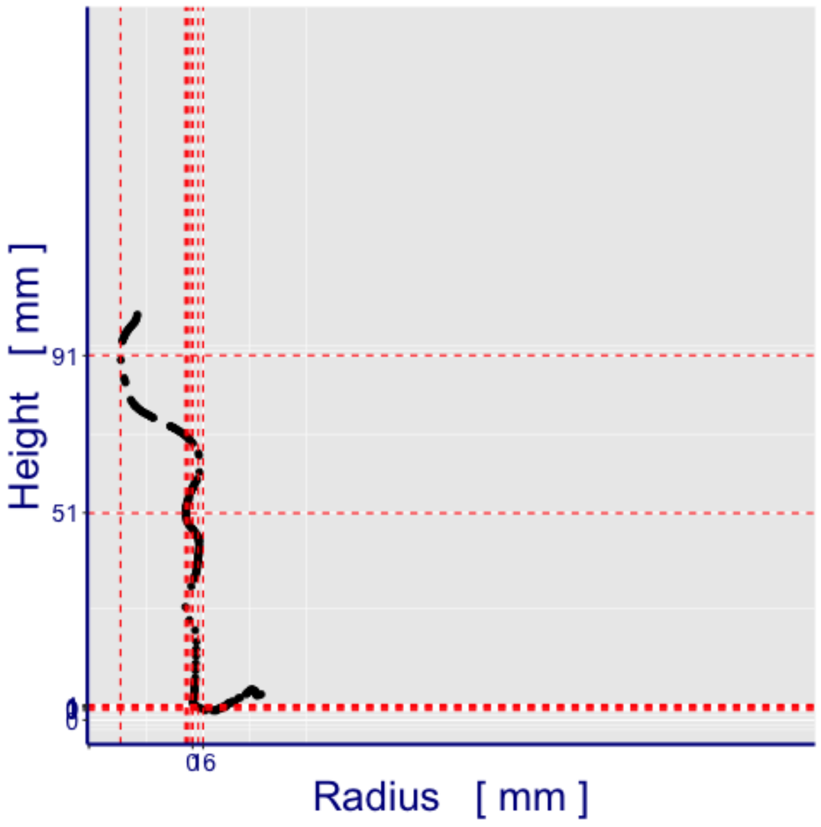


Figure 4: Where to cut?

Measurement

```
## pdf
##   2
```



Conclusion

A. Summary

This document describes in some detail the steps to photograph a potsherd and generate a mathematical model from the resulting images, and, in much less detail, suggests how to automatically measure to object represented by the model. The R source code is available on github and an R package is will be available soon. A number of simplifications, described below, accelerated this initial draft.

B. Item-Specific Configuration

- Scaling to Real-World Units The process of selecting a scale factor will most likely need to be automated for measurement of multiple objects, since the relationship between model units and real-world units may depend on factors that were not tightly controlled, such as the distance between camera and object.
- Manual Selection of Cross-Section Axis The example suggests that knowing the orientation of the most nearly circular cross-section tells us which reference axis, X, Y, or Z, is the axis of rotation for the vessel. The illustration of measurement uses a manually-selected orientation; This selection will need to be automated.
- Fragility of Dimensioning Process Since the process has been applied to a single example, it is almost certain that object-specific assumptions have been included inadvertently in the calculations. The only way to identify and purge such assumptions is to apply the process to a variety of objects, evaluate the performance, and make refinements.
- Photographic Procedure Similarly, the photographic procedure has been used by only one person on only one object, and would benefit from being exposed to more rigorous evaluation.

C. Next Steps

Solicit feedback on clarity of photographic instructions
Test the photographic procedure on a small number of artifacts
Test and refine the measurement functions
Create an R package and associated vignette

Appendix A: Configuration

```
#+ require_libraries, echo = FALSE, include = FALSE
# Load Libraries
require( rgl )
require( ggplot2 )
require( grid )
require( gridExtra )
require( purrr )
require( tibble )
require( lattice )
require( png )

# Paths and Files
FIGURES_PATH   <- './images/'
STEREOLITHOGRAPHY_FILE <- './R/stereolithograph.stl'
MODEL_FILE     <- 'model.RDS'
PERIMETER_FILE <- 'perimeter.RDS'

# Scaling to Real-World Units
SCALE_FACTOR   <- 8.0 * 25.4

# Names for convenience and readability
X_AXIS         <- 1
Y_AXIS         <- 2
Z_AXIS         <- 3
axes           <- 1:3
axes[ X_AXIS ] <- 'x'
axes[ Y_AXIS ] <- 'y'
axes[ Z_AXIS ] <- 'z'

name_axes <- function( m ){
  colnames( m ) <- axes
  m
}

# Wireframe parameters
STRIPE_WIDTH   <- 0.001
STRIPE_TOL     <- 0.0005
BASE_RADIUS    <- 0.3
WIREFRAME_HEIGHTS <- seq( 0.10, 0.60, by=0.05 )
POINTS_ALONG_X    <- seq( 0.20, 0.80, by=0.05 )
```

```
# Different ways of looking at the model
TOP_VIEW_A     <- viewpoint( list( theta=   0, phi=  90, fov=10, zoom=1 ))
TOP_VIEW_B     <- viewpoint( list( theta=  90, phi=  90, fov=0, zoom=1 ))
TOP_VIEW_C     <- viewpoint( list( theta= 180, phi=  90, fov=0, zoom=1 ))
TOP_VIEW_D     <- viewpoint( list( theta= 270, phi=  90, fov=0, zoom=1 ))
BOTTOM_VIEW_A  <- viewpoint( list( theta=   0, phi= -90, fov=0, zoom=1 ))
BOTTOM_VIEW_B  <- viewpoint( list( theta=  90, phi= -90, fov=0, zoom=1 ))
BOTTOM_VIEW_C  <- viewpoint( list( theta= 180, phi= -90, fov=0, zoom=1 ))
BOTTOM_VIEW_D  <- viewpoint( list( theta= 270, phi= -90, fov=0, zoom=1 ))
BACK_VIEW      <- viewpoint( list( theta=   0, phi=   0, fov=0, zoom=1 ))
FRONT_VIEW     <- viewpoint( list( theta= 170, phi=   0, fov=10, zoom=1 ))
LEFT_VIEW      <- viewpoint( list( theta= 260, phi=   0, fov=10, zoom=1 ))
RIGHT_VIEW     <- viewpoint( list( theta=  90, phi=   0, fov=0, zoom=1 ))

TOP_VIEW       <- TOP_VIEW_A
BOTTOM_VIEW    <- BOTTOM_VIEW_A

STANDARD_VIEWS <- list( LEFT_VIEW, FRONT_VIEW, TOP_VIEW )
```

Appendix B: Model Functions

```
#' A caching Function
#'
#' Create a model matrix, which is really a list, containing functions to...
#' * move, rotate, calculate, clip, show, ... the model
#'
#' @param model_data numeric vector contents.
#' @keywords caching
#' @examples model <- make_model( list( theta=-90, phi=-2, fov=25, zoom=1 ) )
#' @export

#+ cache_model, echo = TRUE
make_model <- function( model_data ){
  cache       <- NULL                                  # Begin with an empty model

  cache_it    <- function( model_data ) cache <<- model_data  # Cache the model data

  calculate   <- function( f ) apply( model_data, 2, f )

  # Cut off the top/bottom, front/back, or left/right sides of the model
  clip_at <- function( ax=1, mn=0.3, mx=0.6 ){
    model_data <<- model_data[ model_data[ ,ax ] > mn & model_data[ ,ax ] < mx, ]
  }

  data_length <- function() cache                      # num elements in matrix

  extents     <- function() apply( model_data, 2, function( x ) max( x ) - min( x ) )

  get         <- function() name_axes( model_data )    # Return the model data

  # Clip a thin band out of the middle, and keep that portion that was removed
  get_band        <- function( ax, ctr, thickness ){
    mn <- ctr - 0.5 * thickness
    mx <- ctr + 0.5 * thickness
    model_data <<- model_data[ model_data[ ,ax ] > mn & model_data[ ,ax ] < mx, ]
  }

  move_forward_backward <- function( distance ) model_data[ ,3 ] <<- model_data[ ,3 ] + distance
  move_right_left       <- function( distance ) model_data[ ,1 ] <<- model_data[ ,1 ] + distance
  move_up_down          <- function( distance ) model_data[ ,2 ] <<- model_data[ ,2 ] + distance

  rotate_on   <- function( ax, angle ){
   switch( ax
     , x =  model_data <<- rotate3d( model_data, angle * pi / 180, 1, 0, 0 )
     , y =  model_data <<- rotate3d( model_data, angle * pi / 180, 0, 1, 0 )
     , z =  model_data <<- rotate3d( model_data, angle * pi / 180, 0, 0, 1 )
   )
  }
  scale_it    <- function( scale_factor ){
    model_data[ ,1 ] <<- scale_factor * model_data[ ,1 ]
    model_data[ ,2 ] <<- scale_factor * model_data[ ,2 ]
    model_data[ ,3 ] <<- scale_factor * model_data[ ,3 ]
  }

  show         <- function( config, lims=rep( c( -1, 1 ), 3 ), size = 2.0 ){
    clear3d()
    par3d( cex = size )
    plot3d(
        name_axes( model_data )
      , xlim = lims[ 1:2 ] , ylim = lims[ 3:4 ] , zlim = lims[ 5:6 ]
      , axes = FALSE, box = FALSE
    )
    abclines3d( x = matrix( 0, ncol = 3 ), a = diag( 3 ), col = 'black', lwd = 3 )
    view( config )
    background( 'white' )
  }

  # Return the list of functions
  list(
      cache_it                = cache_it
    , calculate               = calculate
    , clip_at                 = clip_at
    , data_length             = data_length
    , extents                 = extents
    , get                     = get
    , get_band                = get_band
    , move_forward_backward = move_forward_backward
    , move_right_left         = move_right_left
    , move_up_down            = move_up_down
    , rotate_on               = rotate_on
    , show                    = show
    , scale_it                = scale_it
  )
}

#' A caching Function
```

```
#'
#' Create a model, or update it if it exists already.
#'
#' @param model_data numeric vector contents. triple dot.
#' @keywords caching
#' @examples model <- cache_model( model )
#' @export

cache_model <- function( model_data, ... ){
  # Given the data, see if its length has already been determined
  cache <- model_data$data_length()

  # If so...
  if( !is.null( cache )){
    message( "getting cached data" ) # Announce use of cached data
    cache                            # Return the length of the model_data matrix
  }

  # Otherwise...
  data <- model_data$get()           # Get the model_data
  cache <- length( data, ... )       # cache the model_data
  model_data$cache_it( cache )       # Remember the model_data
  cache                              # Return the cached model_data
}
```

Appendix C: Cross-Sectioning Functions

```
get_midpoint <- function( f ) min( f ) + 0.5 * ( max( f ) - min( f ) )

curve_axes <- function( mdls, idx ){
  # Since the model is a thin slice, one axis will have much smaller
  # extents than the other two, which we will use
  # for our 2D curve coordinates, X and Y.
  # For now, assume that the largest extent is X
  # and the second-largest extent is Y
  extents <- mdls[[ idx ]]$extents()

  # Treat the slice as a two-dimensional curve
  # with coordinates along curve_x_axis and curve_y_axis
  # and curve_z_axis being razor thin.

  curve_x_axis <- which.max( extents ) %>% names() # 'x', 'y', or 'z'
  curve_z_axis <- which.min( extents ) %>% names() # 'x', 'y', or 'z'
  curve_y_axis <- setdiff( axes, c( curve_x_axis, curve_z_axis ))

  list( cx = curve_x_axis, cy = curve_y_axis, cz = curve_z_axis )
}

get_curve <- function( x, idx, fun='min', ax = axs ){
  # The wall of the vessel might be thick, and the inner
  # profile could differ from the outer profile
  # fun='min' gets one side; fun='max' gets the other side
  get_curve_points(
      model_matrix = models[[ idx ]]$get()
    , as_width  = ax$cx
    , as_height = ax$cz
    , as_curve  = ax$cy
    , tol = 5 * STRIPE_WIDTH
    , height = models[[ idx ]]$calculate( median )[ ax$cz ]
    , width  = x
    , fun = fun
  )
}

# Calculate the slope of each segment defined by
# each consecutive pair of perimeter points
local_slope <- function( i, data ){
  names( data ) <- c( 'x', 'y' )
  delta_y <- data[ i, 'y' ] - data[ i - 1, 'y' ]
  delta_x <- data[ i, 'x' ] - data[ i - 1, 'x' ]
  round( delta_y / delta_x, 3 )
}

find_center <- function( data, slopes, a = 20, b = 80 ){
  # Given two points, A and B, on an arc,
  # find the intersection of rays AO and BO

  N <- nrow( data ) # Number of points available on arc
  index_A <- round( a/100 * N, 0 ) # offset of point A
  index_B <- round( b/100 * N, 0 ) # offset of point B
  ray_AO  <- get_perpendicular2D( m = slopes[ index_A ], P = data[ index_A, ] )
  ray_BO  <- get_perpendicular2D( m = slopes[ index_B ], P = data[ index_B, ] )
  get_intersection2D( ray_AO, ray_BO )
}

sketch_radius <- function( xyz, fixed_coordinate, ctr, data, i ){
  NUM_POINTS <- 2

  # Decide the X, Y, and Z coordinates to be plotted
  items <- list()
  items[[ 1 ]] <- rep( fixed_coordinate, NUM_POINTS )
  items[[ 2 ]] <- seq( ctr[ 'x' ], data[ i, 'x' ], length.out = NUM_POINTS )
  items[[ 3 ]] <- seq( ctr[ 'y' ], data[ i, 'y' ], length.out = NUM_POINTS )

  # Decide which axis is X, Y, and Z for this object
  segx <- unlist( items[ xyz[[ 1 ]] ] )
  segy <- unlist( items[ xyz[[ 2 ]] ] )
  segz <- unlist( items[ xyz[[ 3 ]] ] )
  segments3d( x = segx, y = segy, z = segz )
}

# receive a series of X coordinates as data[ i ]
# convert each X coordinate into a Y coordinate on
# the perimeter of a circle having center ctr and
# radius r

sketch_arc <- function( xyz, fixed_coordinate, ctr, arc_data, i, r ){
  NUM_POINTS <- 10                # Number of arc points

  # Decide the X, Y, and Z coordinates to be plotted
  items <- list()
```

```r
  # The first item is the Z coordinate
  # The first items is easy -- it doesn't change, since
  # we are making a 2D plot

  # Here, we need to calculate Y, given X, r, a, and b
  a <- ctr[ 'x' ]
  b <- ctr[ 'y' ]
  x <- arc_data[ i, 'x' ]
  yplus  <- b + sqrt( (r + ( x - a ) ) * ( r - ( x - a ) ))
  yminus <- b - sqrt( (r + ( x - a ) ) * ( r - ( x - a ) ))
  y <- c( yplus, yminus )
  x <- c( x    , x      )
  items[[ 1 ]] <- rep( fixed_coordinate, NUM_POINTS )
  items[[ 2 ]] <- x               # The second item is the X coordinate
  items[[ 3 ]] <- y               # The third item is the Y coordinate

  # Decide which axis is X, Y, and Z for this object
  segx <- unlist( items[ xyz[[ 1 ]] ] )
  segy <- unlist( items[ xyz[[ 2 ]] ] )
  segz <- unlist( items[ xyz[[ 3 ]] ] )

  # As a side-effect, plot the point
  points3d( x = segx, y = segy, z = segz, color = 'red', size = 0.5 )

  # Return the coordinates X, Y
  list( x = x, y = y )
}

illustrate_slice_centers <- function( models, model_index, file_name ){
  N <- 20 # Number of arc points to use

  smooth_curve <- get_xy( models = models, model_index = model_index, N = N )

  # Calculate the slope of each segment defined by each consecutive pair of perimeter points
  slopes <- map_dbl( 2:N, ~local_slope( .x, smooth_curve ))

  # Estimate the center, based on several perimeter points
  centers <- map2_df( 15:25, 65:75, ~find_center( data = smooth_curve, slopes=slopes, .x, .y ))
  est_ctr <- apply( centers, 2, mean)
  est_radius <- mean( map_dbl( 1:nrow( smooth_curve ), ~euclidean_distance( est_ctr, smooth_curve[ .x,

  sliced_at <- models[[ model_index ]]$calculate( range )[ , model_index ] %>% mean()
  items <- list()
  items[[ 1 ]] <- sliced_at
  items[[ 2 ]] <- est_ctr[ 'x' ]
  items[[ 3 ]] <- est_ctr[ 'y' ]

  xyz <- orient( model_index )
  origin_x <- items[ xyz[[ 1 ]] ]
  origin_y <- items[ xyz[[ 2 ]] ]
  origin_z <- items[ xyz[[ 3 ]] ]

  show_slice( mdls = models, idx = model_index )
  show_center( origin_x, origin_y, origin_z, est_radius, flavor='point' )
  show_radii(
      idx = model_index
    , ctr = est_ctr
    , crv = smooth_curve
    , fixed_coordinate = sliced_at
  )
  arc_points <- show_arc(
      idx = model_index
    , ctr = est_ctr
    , crv = smooth_curve
    , fixed_coordinate = sliced_at
    , r   = est_radius
  )
  rgl.snapshot( filename = paste0( FIGURES_PATH, file_name, '.png' ))
}

orient <- function( model_index ){
  switch( model_index
    , { x <- 1; y <- 2; z <- 3 }  # 1
    , { x <- 2; y <- 1; z <- 3 } # 2
    , { x <- 3; y <- 2; z <- 1 } # 3
  )
  list( x = x, y = y, z = z )
}

show_radii <- function( idx, ctr, crv, fixed_coordinate ){
  xyz <- orient( idx )

  c( 20, 80 ) %>%
    walk(
      ~sketch_radius(
          xyz  = xyz
        , fixed_coordinate = fixed_coordinate
        , ctr  = ctr
        , data = crv
        , i    = .x * 0.01 * nrow( crv )
      )
    )
}

show_arc <- function( idx, ctr, crv, fixed_coordinate, r ){
  xyz <- orient( idx )
  seq( 0, 100, 10 ) %>% # send sketch_arc() a series of X coordinates as data[ i ]
    map(
      ~sketch_arc(
          xyz  = xyz
        , fixed_coordinate = fixed_coordinate
        , ctr  = ctr
        , arc_data = crv
        , i    = .x * 0.01 * nrow( crv )
        , r    = r
      )
    ) -> arc_points
  arc_points
}

show_center <- function( x, y, z, est_radius, flavor='point' ){
  if( flavor %in% 'point' ){
    # Show center on the slice
    rgl.points(
        x     = x
      , y     = y
      , z     = z
      , r     = est_radius
      , color = 'red'
      , size  = 10
    )
  } else {  # 'sphere'
    # Show center on the slice
    rgl.spheres(
        x     = x
      , y     = y
      , z     = z
      , r     = est_radius
      , color = 'red'
      , alpha = 0.1               # 0: transparent ... 1: opaque
    )
  }
```

```r
}

get_xy <- function( models=models, model_index=model_index, N=N ){

    # Draw a ray, perpendicular to the perimeter at points A and B
    # This should be two radii ( if they are directed inward )

    # ??? What if this is a complete circle, instead of just an arc?
    # Pick two points, A, and B, along the perimeter

    # For the current model, determine which axis is razor-thin,
    # and which axis has the largest extent.
    axs       <- curve_axes( mdls = models, idx = model_index )
    max_x     <- models[[ model_index ]]$calculate( f = max )[ axs$cx ]
    min_x     <- models[[ model_index ]]$calculate( f = min )[ axs$cx ]

    # Create data for a 2D curve representing this model's slice
    curve_x   <- seq( min_x, max_x, length.out=N )
    curve_y   <- map_dbl( curve_x, ~get_curve( x = .x, idx = model_index, fun='min', ax = axs ))
    curve_xy  <- data.frame( x=curve_x, y=curve_y )
    curve_xy[ is.finite( curve_xy$y ), ]
    data.frame( x=curve_xy$x, y=predict( loess( y ~ x, curve_xy, span=0.50 )) )
  }

show_slice <- function( mdls, idx ) switch(
    idx
  ,  mdls[[ idx ]]$show( LEFT_VIEW  ) # 1
  ,  mdls[[ idx ]]$show( TOP_VIEW   ) # 2
  ,  mdls[[ idx ]]$show( FRONT_VIEW ) # 3
)

plot_limits <- function( f ) sort(
  c(
      apply( f, 2, max ) %>% max()
    , apply( f, 2, min ) %>% min()
  )
)

slice_advice <- function( mdls, ax=X_AXIS ){
  # This function has side effects:
  #   It modifies the model specified by its input arguments
  best  <- best_slice( mdls[[ ax ]]$get(), ax )
  cat(
    sprintf(
   '\nThe tallest cross-section is at %s = %f'
   , toupper( axes[ ax ] )
   , round( best, 1 )
  )
 )
 mdls[[ ax ]]$get_band( ax = ax, ctr = best, thickness =  2.0 * STRIPE_WIDTH )
}

slice_the_slice <- function( mdls, ax=X_AXIS ){
 model_data        <- mdls[[ ax ]]$get()
 histogram_buckets <- model_data[ , ax ] %>% hist( plot = FALSE )
 best_mid          <- as.list( histogram_buckets )[[ 'mids' ]][ which.max( histogram_buckets$counts )
 as.data.frame( get_band( model_data, ax, best_mid ))
}

micro_slice <- function( base_name, mdls, ax ){
 slice_advice( mdls, ax )
 show_multi_view( paste0( base_name, axes[ ax ] ), mdls[[ ax ]] )
 thin_slice <- slice_the_slice( mdls, ax )
 lims <- plot_limits( thin_slice)
 # Show the resulting profile
 fn <- paste0( FIGURES_PATH, base_name, axes[ ax ], 'thin.png' )
 cat( sprintf( '\nPlotting %s\n', fn ))
 keep_axes <- axes[ -ax ]
 # _____ Begin Plotting _____
 png( fn )
  print({
   ggplot( thin_slice, aes_string( x = keep_axes[ 1 ], y = keep_axes[ 2 ] )) +
    geom_point()  +
    xlim( lims ) +
    ylim( lims )
  })
 dev.off()
 # _____ End Plotting _____
 thin_slice
}
```