Supplementary Materials for

**FabSquare: Fabricating Photopolymer Objects by Mold 3D Printing and UV Curing**

Vahid Babaei, Javier Ramos, Yongquan Lu, Guillermo Webster and Wojciech Matusik, Computational Fabrication Group, Massachusetts Institute of Technology

- **Pseudocode for FabSquare editor**

  Capitalized functions, `Like_This()`, are generic primitives for which we do not provide pseudocode - their names should be self-explanatory, and their implementation would be highly dependent on the data representation of choice. They include, for example, `Intersect(face, ray)`, `Distance(contour)`, `Find_Closest_Pt(contour, pt)`, `Translate(mesh, vector)`.

  Functions labeled with a `lib` namespace, `lib::Like_This()`, are wrappers that call external geometry libraries. In particular,
  - `lib::boolean_subtract` and `lib::boolean_intersection` are wrappers around the corresponding mesh boolean operations in `libIGL`.
  - `lib::constrained_delaunay_triangulation` is a wrapper for `CDT` generation in `CGAL`.
  - `lib::offset` is a wrapper for calculating contour offsets in `Clipper`.

  // Main entry point
  generate_mold(mesh, mold_shield_thickness, mold_wall_thickness, screw_spacing):
  // Exit early if overhangs are unavoidable in chosen orientation of mesh
  for face in target_mesh.faces:
    ray := cast_ray_from_center_of_face_downwards(face)
    num_intersections := find_intersections(ray, mesh)
    if num_intersections > 2:
      throw Exception("overhang detected, aborting.")

    cutting_surface, mold_outline := generate_cutting_surface_and_mold_outline(
      mold,
      mold_wall_thickness
    )

  // Run CAD extrusion / Boolean operations to yield the two mold halves with
  // the mesh subtracted away in the middle
  mold_height := mesh.zheight + mold_shield_thickness * 2
  mold_top_half := extrude_z(mold_outline, -mold_height / 2, mold_height / 2)
  mold_bottom_half := cut(mold_solid, cutting_surface)

  // mold_top_half, mold_bottom_half as is are already of the right shape to form
  // the desired object. But we still need to subtract away space above and below
  // the mesh to leave only a thin shell for optimal UV curing.
  negative_space_in_mold_top := extrude_z(faces_above_cut(mold_solid, cutting_surface), 0, mold_height / 2)
  negative_space_in_mold_bottom := extrude_z(faces_below_cut(mold_solid, cutting_surface), -mold_height / 2, 0)

  // Subtract negative_space_in_mold_top from mold_top_half, with xyz offset (0,0, mold_shield_thickness)
  mold_top_half = subtract_with_offset_from(
    mold_top_half,
    negative_space_in_mold_top,
    (0,0, mold_shield_thickness)
  )

  mold_bottom_half = subtract_with_offset_from(
    mold_bottom_half,
    negative_space_in_mold_bottom,
    (0,0, -mold_shield_thickness)
  )
// Subtract away equally spaced cylinders along the perimeter of the mold  
// to yield screw holes for bolting the two halves together.  
contour_of_screws = expand_3d_contour(mold_outline, -mold_wall_thickness / 2)  
add_screw_holes(mold_top_half, contour_of_screws, screw_spacing, mold_height)  
add_screw_holes(mold_bottom_half, contour_of_screws, screw_spacing, mold_height)  
return mold_top_half, mold_bottom_half

// Utility function for generating mold  
generate_cutting_surface_and_mold_outline(mesh, wall_thickness):  
edges_on_cutting_surface := new Set()  
// An edge lies on the cutting contour if its two incident faces  
// have normals point in opposite directions with respect to the  
// z axis.  
for e in mesh.edges:  
  face1, face2 = Faces_Edge_Separates(e)  
  if Z_Direction(face1) != Z_Direction(face2):  
    edges_on_cutting_surface.Add(edge)  

// We found these edges as an unordered set, so we need to trace them  
// into a ordered closed loop.  
cutting_contour := closed_loop_from(edges_on_cutting_surface)  

// Expand the cutting contour out  
mold_outline := expand_3d_contour(cutting_contour, wall_thickness)  

// Project the cutting contour and mold outline into 2D  
// Run constrained delaunay triangulation there and lift back into 3D  
constraints := z_project_to_2d(cutting_contour) + z_project_to_2d(mold_outline)  
cutting_surface_2d = lib::constrained_delaunay_triangulation(constraints)  
contour3d = cutting_contour.concat(mold_outline)  
cutting_surface_3d = lift_back_to_3d(cutting_surface_2d, contour3d);  
return cutting_surface_3d, mold_outline

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//functions  
cast_ray_from_center_of_face_downwards(face):  
  faceCenter = 1/3 * (face.v1 + face.v2 + face.v3)  
  // a ray has two components, its origin and its direction  
  return new Ray(faceCenter, new Vector(0, 0, -1))

find_intersections(ray, mesh):  
  num_intersections = 0  
  for face in ray.faces:  
    if Intersects(face, ray)  
      num_intersections += 1  
  return num_intersections

extrude_z(contour_2d, height):  
  // fill in the 2d contour with triangles  
  faces_2d := lib::constrained_delaunay_triangulation(contour_2d)  
  points := []  
  faces := []  
  for point2d, idx in contour2d  
    pointBot := new Vector(point2d.x, point2d.y, 0)  
    pointTop := new Vector(point2d.x, point2d.y, height)  
    points.push(pointBot)  
    points.push(pointTop)  
  // build top and bottom surface  
  for face2d in faces2d:  
    idx0 := FindIdx(contour2d, face2d.points[0])
idx1 := FindIdx(contour2d, face2d.points[1])
idx2 := FindIdx(contour2d, face2d.points[2])
faces.push(points[2 * idx0], points[2 * idx1], points[2 * idx2])
faces.push(points[2 * idx0 + 1], points[2 * idx1 + 1], points[2 * idx2 + 1])
// build the side walls
for i in range(len(points))
    if i == 0
        pointBot := points[0]
        pointTop := points[1]
        prevPointBot := points[-2]
        prevPointTop := points[-1]
    else
        pointBot := points[2 * i]
        pointTop := points[2 * i + 1]
        prevPointBot := points[2 * i - 2]
        prevPointTop := points[2 * i - 1]
    faces.push([pointBot, pointTop, prevPointTop])
    faces.push([prevPointTop, prevPointBot, pointBot])
return new Solid(points, faces)

add_screw_holes(mold_half, contour_of_screws, screw_spacing)
radius := 2.55 // const
d := Distance(contour_of_screws)
cumulativeD := 0
screws := []
while cumulativeD < d
    screwPos := PointAlongContourAtDist(contour_of_screw, cumulativeD)
    screw := new Cylinder(screwPos, radius, mold_height)
    mold_half := lib::boolean_subtract(mold_half, screw)
    cumulativeD += screw_spacing
return mold_half

z_project_to_2d(contour3d):
    contour2d := []
    for point3d in contour3d:
        contour2d.push(new Point2d(point3d.x, point3d.y))
    return contour2d

lift_back_to_3d(surface2d, contour3d):
surface3d := []
for face in surface2d
    point0 := find_corresponding_3d_point(face.points[0], contour3d)
    point1 := find_corresponding_3d_point(face.points[1], contour3d)
    point2 := find_corresponding_3d_point(face.points[2], contour3d)
    surface3d.push(new Face(point0, point1, point2))

find_corresponding_3d_point: (point2d, contour3d):
    for point3d in contour3d
        if point3d.x == point2d.x & point3d.y == point2d.y
            return point3d
    throw Exception("2D point cannot be lifted back into 3D")

closed_loop_from(unordered_edges):
    // start by finding point with largest x-coord when projected
    running_max := -10e10;
    for edge in unordered_edges:
        p1, p2 := edge.end_points
        if p1.x > running_max
            running_max := p1.x
            rightmost_pt := p1
        if p2.x > running_max
            running_max := p2.x
            rightmost_pt := p2
    cur_pt := rightmost_pt
    contour := [rightmost_pt]
do {
next_pt := cur_pt.otherEnd
found_next_edge := false
// loop through edges to find an edge that shares an endpt at next_pt
for edge in unordered_edges:
    p1, p2 := edge.end_points
    if p1 == next_pt && p2 != cur_pt
        found_next_edge := true
        cur_pt := p1
        break
    else if p2 == next_pt && p1 != cur_pt
        found_next_edge := true
        cur_pt := p2
        break
if !found_next_edge
    throw new Exception('Unable to link edges up into closed loop')
contour.push(cur_pt)
) while (cur_pt != rightmost_pt)
return contour

faces_above_cut(mold_solid, cutting_surface):
faced_above := []
for face in mold_solid.faces:
    p1, p2, p3 := face.points
    if is_pt_above_surface(p1, cutting_surface) &&
    is_pt_above_surface(p1, cutting_surface) &&
    is_pt_above_surface(p1, cutting_surface)
        faces_above.push(face)
return faces_above

faces_below_cut(mold_solid, cutting_surface):
faced_below := []
for face in mold_solid.faces:
    p1, p2, p3 := face.points
    if !is_pt_above_surface(p1, cutting_surface) &&
    !is_pt_above_surface(p1, cutting_surface) &&
    !is_pt_above_surface(p1, cutting_surface)
        faces_below.push(face)
return faces_below

is_pt_above_surface(pt, surface):
    point_2d := new Point2d(pt.x, pt.y)
    for face3d in surface:
        face2d := Project_To_2D(face3d)
        if Contains(face2d, point_2d) && pt.z > face.points.getMaxZ()
            return true
    return false

subtract_with_offset_from(mold, negative_space, translation):
negative_space := Translate(negative_space, translation)
return lib::boolean_subtract(mold, negative_space)

cut(mold, cutting_surface):
top_half := extrude_z(cutting_surface, 0, mold_height / 2)
bot_half := extrude_z(cutting_surface, -mold_height / 2, 0)
return lib::boolean_intersection(mold, top_half), lib::boolean_intersection(mold, bot_half)

expand_3d_contour(contour3d, d):
// Expand contour in 2D, and lift back to 3D
// by using the height of the closest point in 2D
contour2d := z_project_to_2d(contour3d)
expanded_contour2d := lib::offsetcontour2d, d)
expanded_contour3d := []
for pt in expanded_contour2d:
    closestPt = Find_Closest_Pt(contour2d, pt)
    expanded_contour_3d.push(new Point3d(pt.x, pt.y, closestPt.z))
return expanded_contour_3d