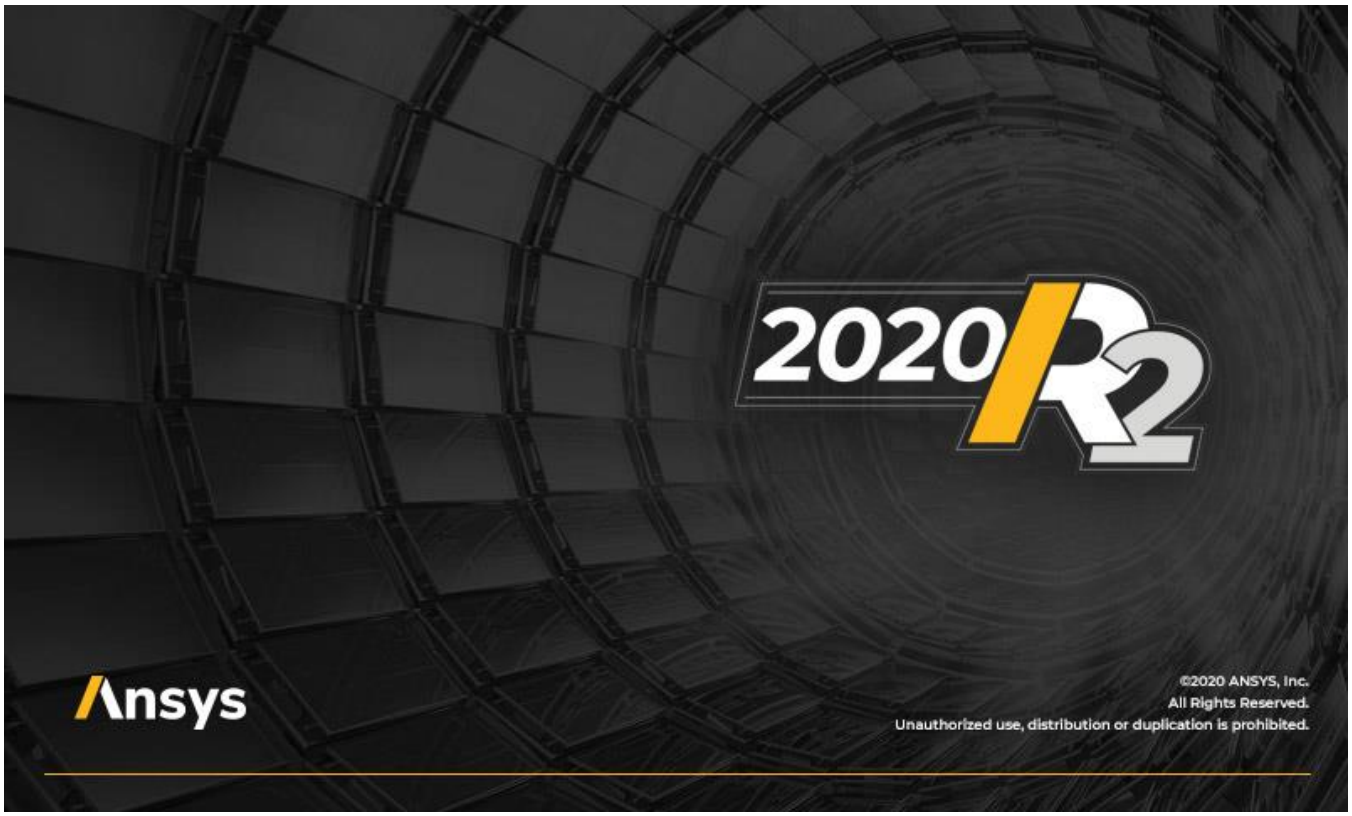


ANSYS ACT Customization Guide for SpaceClaim

ANSYS, Inc.
Southpointe
2600 ANSYS Drive
Canonsburg, PA 15317
ansysinfo@ansys.com
http://www.ansys.com
(T) 724-746-3304
(F) 724-514-9494

Release 2020 R2
July 2020

Copyright and Trademark Information

Disclaimer Notice

U.S. Government Rights

Third-Party Software

Published in the U.S.A.

Table of Contents

Introduction

This guide assumes that you are familiar with the general ACT usage information in the *ACT Developer's Guide*. This first section supplies ACT usage information specific to SpaceClaim:

ACT Start Page and Tool Access

Extension Installation and Loading

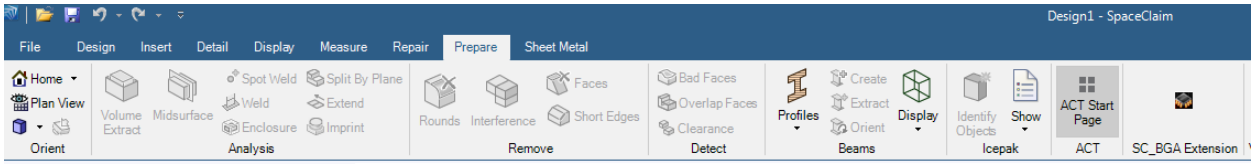
While SpaceClaim does not currently support using ACT to create custom features, it does support target product wizards. The subsequent section (p. 3) describes how you create target product wizards for SpaceClaim.

Note:

For information on all ACT API changes and known issues and limitations that may affect your existing ACT extensions, see Migration Notes and Known Issues and Limitations in the *ANSYS ACT Developer's Guide*.

ACT Start Page and Tool Access

From a stand-alone instance of SpaceClaim, you access the ACT Start Page by clicking **ACT Start Page** in the **Prepare** toolbar.



The **ACT Start Page** for a stand-alone instance of SpaceClaim has an icon for accessing the Extension Manager. However, when SpaceClaim is opened within Workbench, the **ACT Start Page** accessed in this way does not have the icon. This is because you must manage extensions from the **ACT Start Page** for Workbench.

Once accessed, the **ACT Start Page** and ACT tools are all used as described in the *ANSYS ACT Developer's Guide*.

Extension Installation and Loading

For a stand-alone instance of SpaceClaim, the installation location for an extension with a SpaceClaim wizard differs. You must save the extension and associated files to one of the following locations:

- `%ANSYSversion_DIR%\scdm\Addins`

- Any of the additional folders specified by using the gear icon on the graphic-based Extension Manager accessed from the ACT Start Page

From the **ACT Start Page** for the stand-alone instance of SpaceClaim, you then access the Extension Manager to load the extension and the Wizard launcher to start the wizard.

SpaceClaim Wizards

You can use ACT to create target product wizards for SpaceClaim. Two supplied extensions include SpaceClaim wizards: **`WizardDemos`** and **`SC_BGA_Extension`**.

The SpaceClaim wizard in the extension **`WizardDemos`** shows how to build a bridge. The SpaceClaim wizard in the extension **`SC_BGA_Extension`** shows how to generate a ball grid assembly.

SpaceClaim Wizard for Building a Bridge

Space Claim Wizard for Generating a Ball Grid Assembly (BGA)

Note:

- You use the Extension Manager to install and load extensions and the Wizards launcher to start a target product wizard.
- The graphics window in SpaceClaim is not updated until the callbacks for a step have been executed. This change ensures graphics stability and better performance.

Tip:

Included in the package ACT Wizard Templates is a folder named **`Template-SpaceClaim-Wizard`**. It contains an extension with a target product wizard for SpaceClaim. This extension shows all the property capabilities for ACT and how to include reports and charts. For download information, see Extension and Template Examples.

SpaceClaim Wizard for Building a Bridge

The supplied extension **`WizardDemos`** contains a project wizard, multiple target product wizards, and a mixed wizard. This section describes the target project wizard for SpaceClaim. Named **`CreateBridge`**, this two-step wizard is for building a bridge:

Creating the SpaceClaim Wizard for Building a Bridge

Defining Functions for the SpaceClaim Wizard for Building a Bridge

---

Note:

The extension **`WizardDemos`** contains two wizards named **`CreateBridge`**. The first one is for DesignModeler, and the second one is for SpaceClaim. This topic describes the SpaceClaim wizard. The DesignModeler wizard for DesignModeler is described in DesignModeler Wizards in the *ACT Customization Guide for DesignModeler*.

Creating the SpaceClaim Wizard for Building a Bridge

An excerpt from the file `WizardDemos.xml` follows. Code is omitted for the element **`<uidefinition>`** and all wizards other than the SpaceClaim wizard **`CreateBridge`**.

```
<extension version="2" minorversion="1" name="WizardDemos">
 <guid shortid="WizardDemos">7fdb141e-3383-433a-a5af-32cb19971771</guid>
 <author>ANSYS Inc.</author>
 <description>Simple extension to test wizards in different contexts.</description>

 <script src="main.py" />
 <script src="ds.py" />
 <script src="dm.py" />
 <script src="sc.py" />

 <interface context="Project|Mechanical|SpaceClaim">
  <images>images</images>
 </interface>

 <interface context="DesignModeler">
  <images>images</images>

  <toolbar name="Deck" caption="Deck">
   <entry name="Deck" icon="deck">
    <callbacks>
     <onclick>CreateDeck</onclick>
    </callbacks>
   </entry>
   <entry name="Support" icon="Support">
    <callbacks>
     <onclick>CreateSupport</onclick>
    </callbacks>
   </entry>
  </toolbar>

 </interface>

 <simdata context="DesignModeler">
  <geometry name="Deck" caption="Deck" icon="deck" version="1">
   <callbacks>
    <ongenerate>GenerateDeck</ongenerate>
   </callbacks>
   <property name="Length" caption="Length" control="float" unit="Length" default="300 [m]" />
   <property name="Width" caption="Width" control="float" unit="Length" default="20 [m]" />
   <property name="Beams" caption="Beams" control="integer" default="31" />
  </geometry>
 </simdata>

 <simdata context="DesignModeler">
  <geometry name="Support" caption="Support" icon="support" version="1">
   <callbacks>
    <ongenerate>GenerateSupport</ongenerate>
   </callbacks>
   <property name="Length" caption="Length" control="float" unit="Length" default="300 [m]" />
   <property name="Height" caption="Height" control="float" unit="Length" default="100 [m]" />
   <property name="Width" caption="Width" control="float" unit="Length" default="20 [m]" />
   <property name="Number" caption="Number" control="integer" default="3" />
  </geometry>
 </simdata>

      ...

 <wizard name="CreateBridge" version="1" context="SpaceClaim" icon="wizard_icon">
  <description>Simple wizard for demonstration in SpaceClaim.</description>

  <step name="DeckSC" caption="DeckSC" version="1" context="SpaceClaim">
   <description>Create the deck.</description>

   <callbacks>
```

```
    <onupdate>UpdateDeckSC</onupdate>
   </callbacks>

   <propertygroup display="caption" name="Deck" caption="Deck Definition" >
    <property name="Length" caption="Length" control="float" unit="Length" default="300 [m]" />
    <property name="Width" caption="Width" control="float" unit="Length" default="20 [m]" />
    <property name="Beams" caption="Beams" control="integer" default="31" />
   </propertygroup>
  </step>

  <step name="SupportsSC" caption="SupportsSC" context="SpaceClaim" enabled="true" version="1">
   <description>Create supports.</description>

   <callbacks>
    <onupdate>UpdateSupportsSC</onupdate>
   </callbacks>

   <propertygroup display="caption" name="Supports" caption="Supports Definition" >
    <property name="Height" caption="Height" control="float" unit="Length" default="100 [m]" />
    <property name="Number" caption="Number" control="integer" default="3" />
   </propertygroup>
  </step>
 </wizard>

 ...

</extension>
```

Understanding the elements **<interface>** and **<simdata>** is necessary to understanding the SpaceClaim wizard **CreateBridge**.

Wizard Interface Definition

The element **<interface>** defines two user interfaces for the extension **WizardDemos**. The first element **<interface>** is used by the SpaceClaim wizard **CreateBridge**.

Simdata Definition

The element **<simdata>** provides data. This extension has two such elements to provide data for creating the geometries **Deck** and **Support**. The first element **<simdata>** is used by this wizard as it has the attribute **context** set to **SpaceClaim**.

Wizard Definition

The element **<wizard>** named **CreateBridge** in the XML code excerpt has the attribute **context** set to **SpaceClaim** to indicate that this is the product in which the wizard executes.

Step Definition

The element **<step>** defines a step in the wizard. This wizard has two steps: **DeckSC** and **SupportsSC**.

- For the step **DeckSC**, the callback **<onupdate>** executes the function **UpdateDeckSC**, creating the deck using the geometry **Deck**.
- For the step **SupportsSC**, the callback **<onupdate>** executes the function **UpdateSupportsSC**, creating the bridge supports using the geometry **Support**.

Defining Functions for the SpaceClaim Wizard for Building a Bridge

The IronPython script `sc.py` follows. This script defines all functions executed by the callbacks in the steps for the SpaceClaim wizard **CreateBridge**.

```
import units

def createBox(xa, ya, za, xb, yb, zb):
    win = Window.ActiveWindow
    context = win.ActiveContext
    part = context.ActivePart

    lengthX = xb - xa
    lengthY = yb - ya
    lengthZ = zb - za
    xa = xa + lengthX * 0.5
    ya = ya + lengthY * 0.5

    p = Geometry.PointUV.Create(0, 0)
    body = Modeler.Body.ExtrudeProfile(Geometry.RectangleProfile(Geometry.Plane.PlaneXY, lengthX,
lengthY, p, 0), lengthZ)
    designBody = DesignBody.Create(part, "body", body)

    translation = Geometry.Matrix.CreateTranslation(Geometry.Vector.Create(xa, ya, za))
    designBody.Transform(translation)


def UpdateDeckSC(step):
    length = step.Properties["Deck/Length"].Value
    width = step.Properties["Deck/Width"].Value
    num = step.Properties["Deck/Beams"].Value

    createBox(0., -width/2., -0.3, length,width/2., 0.)

    w = (length-0.1*num)/(num-1.)+0.1
    for i in range(num-1):
        createBox(i*w,-width/2.,-0.6, i*w+0.1,width/2.,-0.3)

    createBox(length-0.1, -width/2., -0.6, length, width/2., -0.3)

    createBox(0., -width/2., -1., length, -width/2.+0.2, -0.6)
    createBox(0., width/2.-0.2, -1., length,width/2., -0.6)

    return True


def UpdateSupportsSC(step):
    length = step.PreviousStep.Properties["Deck/Length"].Value
    width = step.PreviousStep.Properties["Deck/Width"].Value
    height = step.Properties["Supports/Height"].Value
    num = step.Properties["Supports/Number"].Value

    w = (length-2.*num)/(num+1.)+2.
    for i in range(num):
        createBox((i+1)*w, -width/2., -1.-height, (i+1)*w+2., width/2.,-1.)

    beamGen = createBox(0., -width/2., -5., 2., width/2., -1.)
    beamGen = createBox(length-2., -width/2.,-5., length,width/2., -1.)

    return True
```

Space Claim Wizard for Generating a Ball Grid Assembly (BGA)

The supplied extension **SC_BGA_Extension** contains a product wizard named **BGAWizard**. This wizard shows how to generate a ball grid assembly (BGA), which is a surfaced on which to mount for integrated circuits. First, the wizard first generates a die, such a microprocessor. Then, it generates the substrate and finally the solder balls for mounting the die.

The following topics describe the wizard **BGAWizard**:

Creating the SpaceClaim Wizard for Generating a BGA

Defining Functions for the SpaceClaim Wizard for Generating a BGA

Creating the SpaceClaim Wizard for Generating a BGA

The file SC_BGA_Extension.xml follows.

```
extension version="1" name="SC_BGA_Extension">
  <script src="main.py" />
  <guid shortid="SC_BGA_Extension">5107C33A-E123-4F55-8166-2ED2AA59B3B2</guid>
  <interface context="SpaceClaim">
    <images>images</images>

    <callbacks>
      <oninit>oninit</oninit>
    </callbacks>

    <toolbar name="SC_BGA_Extension" caption="SC_BGA Extension">
      <entry name="SC_BGA_Package"  icon="icepak_package">
        <callbacks>
          <onclick>createMyFeature</onclick>
        </callbacks>
      </entry>
    </toolbar>
  </interface>

 <wizard name="BGAWizard" version="1" context="SpaceClaim">
  <description>BGA Wizard</description>

  <step name="Die" caption="Die" version="1">
   <callbacks><onupdate>GenerateDie</onupdate></callbacks>
   <property name="Thickness" caption="Height" unit="Length" control="float" default="0.3[mm]"/>
   <property name="Width"     caption="Width"  unit="Length" control="float" default="5 [mm]"/>
  </step>

  <step name="SubstrateAndSolderMask" caption="Substrate and SolderMask" version="1">
   <callbacks><onupdate>GenerateSubstrateAndSolderMask</onupdate></callbacks>
   <propertygroup name="SubstrateDetails" caption="SubstrateDetails" display="caption">
    <property name="Thickness" caption="Thickness" unit="Length" control="float" default="0.4 [mm]" ></property>
    <property name="Length" caption="Length" unit="Length" control="float" default="13 [mm]" ></property>
   </propertygroup>
   <propertygroup name="SolderMaskDetails" caption="SolderMaskDetails" display="caption">
    <property name="Height" caption="Solder Mask Height"  unit="Length" control="float" default="0.05 [mm]"/>
   </propertygroup>
  </step>

  <step name="SolderBall" caption="Solder ball" version="1">
   <callbacks><onupdate>GenerateBalls</onupdate></callbacks>
   <property name="Face" caption="Face" control="scoping">
    <attributes selection_filter="face"/>
   </property>

   <propertygroup name="SolderBallDetails" caption="Solder Ball Details" display="caption">
    <propertygroup display="property" name="BallsPrimitive" caption="Balls primitive" control="select" default="
     <attributes options="sphere,cylinder,cone,cube,gear"/>
    </propertygroup>
    <property name="Pitch" caption="Pitch" unit="Length" control="float" default="0.8 [mm]"/>
    <property name="Radius" caption="Radius"  unit="Length" control="float" default="0.35 [mm]"/>
    <property name="Number of Solder Ball Columns" caption="No of Solder Ball Columns" control="integer" default
    <property name="Number of Solder Ball Rows" caption="No of Solder Ball Rows" control="integer" default="16"/
   </propertygroup>

     <propertygroup name="Central Balls" caption="Central Thermal Balls">
    <propertygroup display="property" name="Central Thermal Balls" caption="Want to Supress Central Balls" contr
     <attributes options="Yes,No"/>
    <property name="Number of Solder Ball Columns" caption="No of Solder Ball Columns" control="integer" default
    <property name="Number of Solder Ball Rows" caption="No of Solder Ball Rows" control="integer" default="4" v
    </propertygroup>
   </propertygroup>
```

```
  </step>

 </wizard>

</extension>
```

Interface Definition

In the element **<interface>**, the child element **<toolbar>** defines a toolbar and toolbar button to display in SpaceClaim.

Step Definition

This wizard has three steps: **Die**, **SubstrateAndSolderMask**, and **SolderBall**.

- For the step **Die**, the callback **<onupdate>** executes the function **GenerateDie**.
- For the step **SubstrateAndSolderMask**, the callback **<onupdate>** executes the function **GenerateSubstrateAndSolderMask**.
- For the step **SolderBall**, the callback **<onupdate>** executes the function **GenerateBalls**.

Defining Functions for the SpaceClaim Wizard for Generating a BGA

The IronPython script `main.py` follows. This script defines all functions executed by the callbacks in the steps for the SpaceClaim wizard **BGAWizard**.

```
import System
import clr
import sys
import os
import math

part = None

def oninit(context):
    return

def createMyFeature(ag):
    ExtAPI.CreateFeature("MyFeature1")

def createSphere(x, y, z, radius):
    global part
    from System.Collections.Generic import List

    # get selected part
    if part==None:
        win = Window.ActiveWindow
        context = win.ActiveContext
        part = context.ActivePart.Master

    center = Geometry.Point.Create(x, y, z)
    profileFrame = Geometry.Frame.Create(center, Geometry.Direction.DirX, Geometry.Direction.DirY)
    sphereCircle = Geometry.Circle.Create(profileFrame, radius)
    sphereRevolveLine = Geometry.Line.Create(center, Geometry.Direction.DirX)
    profile = List[Geometry.ITrimmedCurve]()
    profile.Add(Geometry.CurveSegment.Create(sphereCircle, Geometry.Interval.Create(0, math.pi)));
    profile.Add(Geometry.CurveSegment.Create(sphereRevolveLine, Geometry.Interval.Create(-radius, radius)))
    path = List[Geometry.ITrimmedCurve]()
    sweepCircle = Geometry.Circle.Create(Geometry.Frame.Create(center, Geometry.Direction.DirY, Geometry.Directi
    path.Add(Geometry.CurveSegment.Create(sweepCircle))
    body = Modeler.Body.SweepProfile(Geometry.Profile(Geometry.Plane.Create(profileFrame), profile), path)
    DesignBody.Create(part, "sphere", body)

def createCylinder(x, y, z, radius, h):
```

```
    global part
    from System.Collections.Generic import List

    # get selected part
    if part==None:
        win = Window.ActiveWindow
        context = win.ActiveContext
        part = context.ActivePart.Master

    defaultPointUV = Geometry.PointUV.Create(0, 0)
    profile = Geometry.CircleProfile(Geometry.Plane.PlaneXY, radius, defaultPointUV, 0)

    points = List[Geometry.Point]()
    points.Add(Geometry.Point.Create(0, 0, 0))
    points.Add(Geometry.Point.Create(0, 0, h))
    path = Geometry.PolygonProfile(Geometry.Plane.PlaneXY, points)

    body = Modeler.Body.SweepProfile(profile, path.Boundary)
    designBody = DesignBody.Create(part, "Cylinder", body)

    translation = Geometry.Matrix.CreateTranslation(Geometry.Vector.Create(x, y, z))
    designBody.Transform(translation)

def createCone(x, y, z, radius, h):
    global part
    from System.Collections.Generic import List

    # get selected part
    if part==None:
        win = Window.ActiveWindow
        context = win.ActiveContext
        part = context.ActivePart.Master

    defaultPointUV = Geometry.PointUV.Create(0, 0)
    path = Geometry.CircleProfile(Geometry.Plane.PlaneXY, radius, defaultPointUV, 0)

    points = List[Geometry.Point]()
    points.Add(Geometry.Point.Create(0, 0, 0))
    points.Add(Geometry.Point.Create(radius, 0, h))
    points.Add(Geometry.Point.Create(0, 0, h))
    triangle = Geometry.PolygonProfile(Geometry.Plane.PlaneZX, points)

    body = Modeler.Body.SweepProfile(triangle, path.Boundary)
    designBody = DesignBody.Create(part, "Cone", body)

    translation = Geometry.Matrix.CreateTranslation(Geometry.Vector.Create(x, y, z))
    designBody.Transform(translation)

def createBox(xa, ya, za, xb, yb, zb):
    global part
    # get selected part
    if part==None:
        win = Window.ActiveWindow
        context = win.ActiveContext
        part = context.ActivePart.Master

    lengthX = xb - xa
    lengthY = yb - ya
    lengthZ = zb - za
    xa = xa + lengthX * 0.5
    ya = ya + lengthY * 0.5

    p = Geometry.PointUV.Create(0, 0)
    body = Modeler.Body.ExtrudeProfile(Geometry.RectangleProfile(Geometry.Plane.PlaneXY, lengthX, lengthY, p, 0)
    designBody = DesignBody.Create(part, "body", body)

    translation = Geometry.Matrix.CreateTranslation(Geometry.Vector.Create(xa, ya, za))
    designBody.Transform(translation)

def createGear(x, y, z, innerRadius, outerRadius, width, count, holeRadius):
    global part
```

```
    from System.Collections.Generic import List

    # get selected part
    if part==None:
        win = Window.ActiveWindow
        context = win.ActiveContext
        part = context.ActivePart.Master
    frame = Geometry.Frame.World

    # create gear
    outsideCircle = Geometry.Circle.Create(frame, outerRadius);
    insideCircle  = Geometry.Circle.Create(frame, innerRadius);

    boundary = List[Geometry.ITrimmedCurve]()
    inwardLine = Geometry.Line.Create(frame.Origin, -frame.DirX);
    outwardLine = Geometry.Line.Create(frame.Origin, frame.DirX);
    axis = outsideCircle.Axis;

    nTeeth = count;
    repeatAngle = 2 * math.pi / nTeeth;
    toothAngle = 0.6 * repeatAngle;
    gapAngle = repeatAngle - toothAngle;

    for i in range(0, nTeeth):
        # an arc is just a parameter interval of a circle
        startTooth = i * repeatAngle;
        endTooth = startTooth + toothAngle;
        boundary.Add(Geometry.CurveSegment.Create(outsideCircle, Geometry.Interval.Create(startTooth, endTooth))

        # rotate 'inwardLine' about the circle axis
        rotatedInwardLine = Geometry.Matrix.CreateRotation(axis, endTooth) * inwardLine;
        # a line segment is just a parameter interval of an unbounded line
        boundary.Add(Geometry.CurveSegment.Create(rotatedInwardLine, Geometry.Interval.Create(-outerRadius, -inn

        startGap = endTooth;
        endGap = startGap + gapAngle;
        boundary.Add(Geometry.CurveSegment.Create(insideCircle, Geometry.Interval.Create(startGap, endGap)));

        rotatedOutwardLine = Geometry.Matrix.CreateRotation(axis, endGap) * outwardLine;
        boundary.Add(Geometry.CurveSegment.Create(rotatedOutwardLine, Geometry.Interval.Create(innerRadius, oute

    hole = Geometry.Circle.Create(frame.Create(frame.Origin, frame.DirX, frame.DirY), holeRadius);
    boundary.Add(Geometry.CurveSegment.Create(hole));

    body = Modeler.Body.ExtrudeProfile(Geometry.Profile(Geometry.Plane.Create(frame), boundary), width);
    pieces = body.SeparatePieces().GetEnumerator()
    while pieces.MoveNext():
        designBody = DesignBody.Create(part, "GearBody", pieces.Current);

        translation = Geometry.Matrix.CreateTranslation(Geometry.Vector.Create(x, y, z))
        designBody.Transform(translation)

class Vector:
    def __init__(self, x = 0, y = 0, z = 0):
        self.x = x
        self.y = y
        self.z = z

    def Clone(self):
        return Vector(self.x, self.y, self.z)

    def NormSQ(self):
        return self.x*self.x + self.y*self.y + self.z*self.z

    def Norm(self):
        return math.sqrt(self.x*self.x + self.y*self.y + self.z*self.z)

    def Normalize(self):
        norm = self.Norm()
        self.x = self.x / norm
        self.y = self.y / norm
```

```
        self.z = self.z / norm

    def GetNormalize(self):
        norm = self.Norm(self)
        return Vector(self.x / norm, self.y / norm, self.z / norm)

    def __add__(va, vb):
        return Vector(va.x + vb.x, va.y + vb.y, va.z + vb.z)

    def __sub__(va, vb):
        return Vector(va.x - vb.x, va.y - vb.y, va.z - vb.z)

    def __mul__(v, x):
        return Vector(v.x*x, v.y*x, v.z*x)

    def Cross(va, vb):
        return Vector(va.y*vb.z - va.z*vb.z, -va.z*vb.x + va.x*vb.z, va.x*vb.y - va.y*vb.x)

    def Dot(va, vb):
        return va.x*vb.x + va.y*vb.y + va.z*vb.z

    def ToString(self):
        return "( " + str(self.x) + ", " + str(self.y) + ", " + str(self.z) + " )"

def CreateBalls(primitive, pitch, radius, column, row, supr, columnSupr, rowSupr, center, dirColumn, dirRow):
    dirColumn.Normalize()
    dirRow.Normalize()
    startVector = center - dirColumn*column*pitch*0.5 - dirRow*row*pitch*0.5
    startVector = startVector + dirColumn*radius + dirRow*radius
    startVector = startVector + dirRow.Cross(dirColumn)*radius
    stepVectorColumn = dirColumn * pitch
    stepVectorRow    = dirRow * pitch

    if(supr == "Yes"):
        column_index_to_start_supress = int( column * 0.5 - columnSupr * 0.5 )
        row_index_to_start_supress    = int( row    * 0.5 - rowSupr    * 0.5 )

    v = startVector.Clone()
    for i in range(column):
        for j in range(row):
            createBall = False
            if (supr == "Yes" and  (i <  column_index_to_start_supress or
                                    i >= column_index_to_start_supress + columnSupr or
                                    j <  row_index_to_start_supress or
                                    j >= row_index_to_start_supress+ rowSupr)
            or supr == "No"):
                if primitive == "sphere":
                    createSphere(v.x, v.y, v.z, radius)
                elif primitive == "cylinder":
                    createCylinder(v.x, v.y, v.z, radius, radius * 2.)
                elif primitive == "cone":
                    createCone(v.x, v.y, v.z, radius, radius * 2.)
                elif primitive == "cube":
                    createBox(v.x - radius, v.y - radius, v.z - radius,
                              v.x + radius, v.y + radius, v.z + radius)
                elif primitive == "gear":
                    createGear(v.x, v.y, v.z,
                               radius*0.5, radius, radius*2, 10, radius*0.2)
            v = v + stepVectorRow

        v = startVector.Clone()
        startVector = startVector + stepVectorColumn
        v = v + stepVectorColumn

def CreateDie(width, thickness, zStart):
    createBox(-0.5 * width, -0.5 * width, zStart,
              0.5 * width,  0.5 * width, zStart + thickness)

def CreateSubstrate(width, thickness, zStart):
    createBox(-0.5 * width, -0.5 * width, zStart,
               0.5 * width,  0.5 * width, zStart + thickness)
```

```
def CreateSolderMask(width, thickness, zStart):
    createBox(-0.5 * width, -0.5 * width, zStart,
               0.5 * width,  0.5 * width, zStart + thickness)

def generateBGAGeometry(feature,fct):
    ps = feature.Properties

    Pitch                           = ps["Solder Ball Details/Pitch"].Value
    Solder_Ball_Radius              = ps["Solder Ball Details/Solder Ball Radius"].Value
    No_Of_Solder_Ball_Column        = ps["Solder Ball Details/Number of Solder Ball Columns"].Value
    No_Of_Solder_Ball_Row           = ps["Solder Ball Details/Number of Solder Ball Rows"].Value
    No_Of_Solder_Ball_Column_Supress = ps["Central Balls/Central Thermal Balls/Number of Solder Ball Columns"].V
    No_Of_Solder_Ball_Row_Supress    = ps["Central Balls/Central Thermal Balls/Number of Solder Ball Rows"].Valu
    Substrate_Thickness             = ps["Substrate Details/Substrate Thickness"].Value
    Substrate_Width                 = ps["Substrate Details/Substrate Length"].Value
    Die_Thickness                   = ps["Die Details/Die Thickness"].Value
    Die_Width                       = ps["Die Details/Die Width"].Value
    Solder_Mask_Height              = ps["Solder Ball Details/Solder Mask Height"].Value
    supress_balls                   = ps["Central Balls/Central Thermal Balls"].Value
    ballsPrimitive                  = ps["BallsPrimitive"].Value

    bodies = []

    CreateBalls(ballsPrimitive, Pitch, Solder_Ball_Radius, No_Of_Solder_Ball_Column, No_Of_Solder_Ball_Row, supr
                No_Of_Solder_Ball_Column_Supress, No_Of_Solder_Ball_Row_Supress,
                Vector(0, 0, 0), Vector(1, 0, 0), Vector(0, 1, 0))

    #Creating Substrate and soldermask
    CreateSubstrate(Substrate_Width, Substrate_Thickness, 0)
    CreateSolderMask(Substrate_Width, Solder_Mask_Height, 0)

    #Creating Die
    Die_Start = Substrate_Thickness
    CreateDie(Die_Width, Die_Thickness, Die_Start)

    return True

def GenerateDie(step):
    global part
    win = Window.ActiveWindow
    context = win.ActiveContext
    part = context.ActivePart

    ps = step.Properties
    Die_Thickness = ps["Thickness"].Value
    Die_Width     = ps["Width"].Value
    CreateDie(Die_Width, Die_Thickness, 0)

    part = None

def GenerateSubstrateAndSolderMask(step):
    global part
    win = Window.ActiveWindow
    context = win.ActiveContext
    part = context.ActivePart

    Die_Thickness = step.PreviousStep.Properties["Thickness"].Value

    ps = step.Properties
    Substrate_Thickness = ps["SubstrateDetails/Thickness"].Value
    Substrate_Width     = ps["SubstrateDetails/Length"].Value
    Solder_Mask_Height  = ps["SolderMaskDetails/Height"].Value

    CreateSubstrate(Substrate_Width, Substrate_Thickness, Die_Thickness)
    CreateSolderMask(Substrate_Width, Solder_Mask_Height, Die_Thickness + Substrate_Thickness)

    part = None

def GenerateBalls(step):
    global part
```

```
win = Window.ActiveWindow
context = win.ActiveContext
part = context.ActivePart

zStart = 0
zStart += step.PreviousStep.PreviousStep.Properties["Thickness"].Value
zStart += step.PreviousStep.Properties["SubstrateDetails/Thickness"].Value
zStart += step.PreviousStep.Properties["SolderMaskDetails/Height"].Value

ps = step.Properties
faces      = ps["Face"].Value.Faces
pitch      = ps["SolderBallDetails/Pitch"].Value
radius     = ps["SolderBallDetails/Radius"].Value
column     = ps["SolderBallDetails/Number of Solder Ball Columns"].Value
row        = ps["SolderBallDetails/Number of Solder Ball Rows"].Value
primitive  = ps["SolderBallDetails/BallsPrimitive"].Value
columnSupr = ps["Central Balls/Central Thermal Balls/Number of Solder Ball Columns"].Value
rowSupr    = ps["Central Balls/Central Thermal Balls/Number of Solder Ball Rows"].Value
supr       = ps["Central Balls/Central Thermal Balls"].Value

for i in range(0, faces.Count):
    face = faces[i]
    edges = face.Edges
    if edges.Count == 0:
        continue

    # find two edges with a comon point
    edgeA = edges[0]
    startPointA = edgeA.Shape.StartPoint
    endPointA   = edgeA.Shape.EndPoint
    for j in range(1, edges.Count):
        edgeB = edges[j]
        startPointB = edgeB.Shape.StartPoint
        endPointB   = edgeB.Shape.EndPoint

        if startPointB == startPointA:
            basePoint   = startPointB
            pointRow    = endPointA
            pointColumn = endPointB
        elif endPointB == startPointA:
            basePoint   = endPointB
            pointRow    = endPointA
            pointColumn = startPointB
        elif startPointB == endPointA:
            basePoint   = startPointB
            pointRow    = startPointA
            pointColumn = endPointB
        elif endPointB == endPointA:
            basePoint   = endPointB
            pointRow    = startPointA
            pointColumn = startPointB

        if not basePoint is None:
            dirColumn = Vector(pointRow.X - basePoint.X, pointRow.Y - basePoint.Y, pointRow.Z - basePoint.Z)
            dirRow    = Vector(pointColumn.X - basePoint.X, pointColumn.Y - basePoint.Y, pointColumn.Z - bas
            center    = Vector(basePoint.X, basePoint.Y, basePoint.Z) + (dirRow + dirColumn)*0.5
            CreateBalls(primitive, pitch, radius, column, row, supr, columnSupr, rowSupr, center, dirColumn,
            break

part = None
```