Renderman Geometry

Using the Python API

CSG

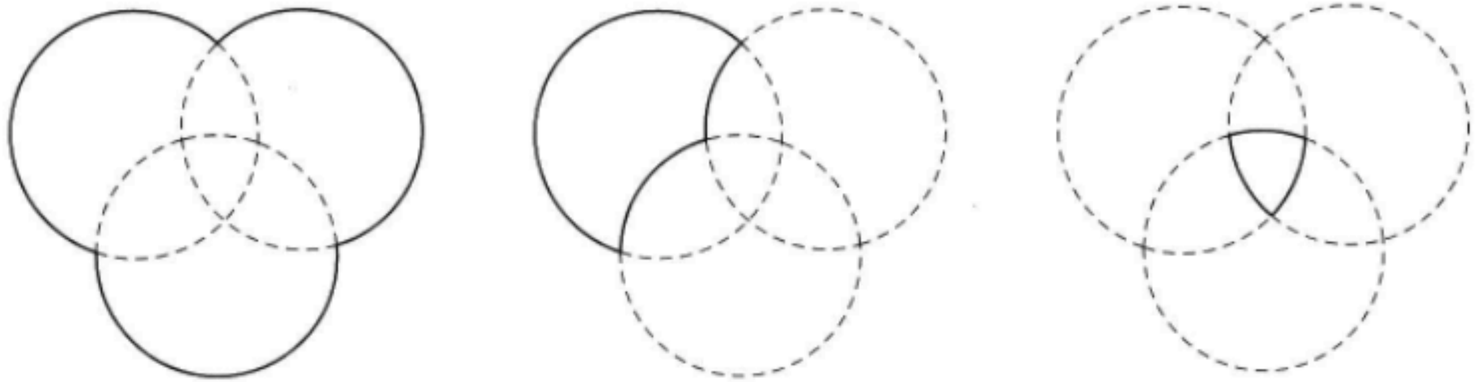


Figure 4.6 The three basic CSG operations: union, difference, and intersection.

- Renderman allows us to use Constructive Solid Geometry (CSG) using different Boolean type operations
- These are union, difference and intersection as shown
- intersection and union operations form the set intersection and union of the specified solids.
- Difference operations require at least 2 parameter solids and subtract the last n-1 solids from the first (where n is the number of parameter solids).

SolidBegin / End

- RiSolidBegin the definition of a solid. operation may be one of the following tokens:
  - "primitive"
  - "intersection"
  - "union"
  - "difference"

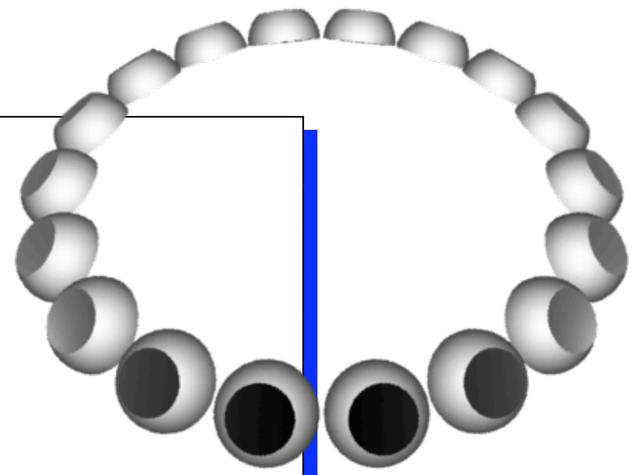
Difference

```
ri.SolidBegin("difference")

ri.SolidBegin("primitive")
ri.TransformBegin()
ri.AttributeBegin()
ri.Color([1,1,1])
ri.Translate( 0,-1.0,0)
ri.Rotate(-90,1,0,0)
ri.Rotate(36,0,0,1)
ri.Scale(0.4,0.4,0.4)
ri.Surface("plastic")
ri.Geometry("teapot")
#ri.Sphere(1.0,-1,1,360)
ri.AttributeEnd()
ri.TransformEnd()
ri.SolidEnd()

for i in range(0,360,20) :
  ri.SolidBegin("primitive")
  ri.Rotate(i,0,1,0)
  ri.Translate(-0.6,-0.3,0)
  ri.Scale(0.3,0.2,0.3)
  ri.Sphere(1,-1,1,360)
  #ri.Geometry("teapot")

  ri.SolidEnd()
ri.SolidEnd()
```

```
SolidBegin "difference"
    SolidBegin "primitive"
      TransformBegin
        AttributeBegin
          Color [1 1 1]
          Translate 0 -1 0
          Rotate -90 1 0 0
          Rotate 36 0 0 1
          Scale 0.4 0.4 0.4
          Surface "plastic"
          Geometry "teapot"
        AttributeEnd
      TransformEnd
    SolidEnd
    SolidBegin "primitive"
      Rotate 0 0 1 0
      Translate -0.6 -0.3 0
      Scale 0.3 0.2 0.3
      Sphere 1 -1 1 360
    SolidEnd
  .........
SolidEnd
```

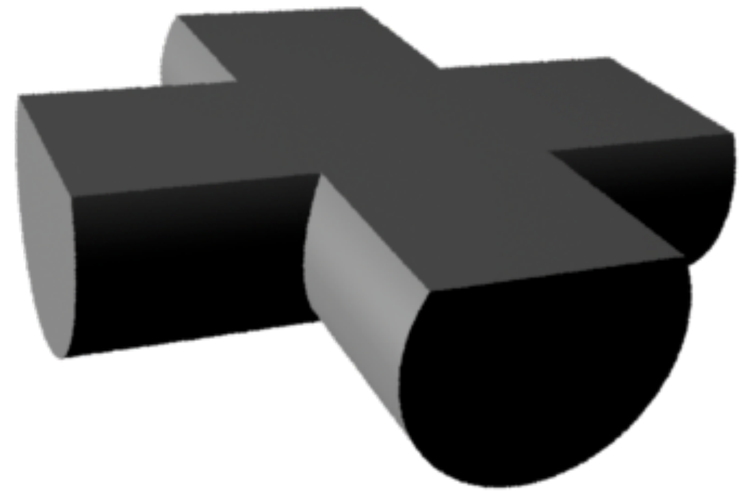
intersection

```
ri.SolidBegin("intersection")

ri.SolidBegin("primitive")
ri.TransformBegin()
ri.AttributeBegin()
ri.Color([1,1,1])
ri.Translate( 0,-1.0,0)
ri.Surface("plastic")
ri.Sphere(1.0,-1,1,360)

ri.Translate(0,-0.1,0)
ri.Sphere(1.0,-1,1,360)

ri.AttributeEnd()
ri.TransformEnd()
ri.SolidEnd()


ri.SolidBegin("primitive")

for i in range(0,360,20) :
  ri.TransformBegin()
  ri.Rotate(i,0,1,0)
  ri.Translate(-0.6,-0.3,0)
  ri.Scale(0.1,0.1,0.1)
  ri.Sphere(1,-1,1,360)
  ri.TransformEnd()
ri.SolidEnd()

ri.SolidEnd()
```



```
SolidBegin "intersection"
  SolidBegin "primitive"
    TransformBegin
      AttributeBegin
        Color [1 1 1]
        Translate 0 -1 0
        Surface "plastic"
        Sphere 1 -1 1 360
        Translate 0 -0.1 0
        Sphere 1 -1 1 360
      AttributeEnd
    TransformEnd
  SolidEnd
  SolidBegin "primitive"
    TransformBegin
      Rotate 0 0 1 0
      Translate -0.6 -0.3 0
      Scale 0.1 0.1 0.1
      Sphere 1 -1 1 360
    TransformEnd
    TransformBegin
      Rotate 20 0 1 0
      Translate -0.6 -0.3 0
      Scale 0.1 0.1 0.1
      Sphere 1 -1 1 360
    TransformEnd
  .......
  SolidEnd
SolidEnd
```

Union



```
ri.SolidBegin("intersection")

ri.SolidBegin("primitive")
ri.TransformBegin()
#ri.Rotate(-45,1,0,0)
ri.Translate(0,-0.2,0)
Cube(.5,.5,.5)
ri.TransformEnd()
ri.SolidEnd()


ri.SolidBegin("union") #start union

ri.SolidBegin("primitive")
ri.Cylinder(.1,-0.5,0.5,360)
ri.SolidEnd()

ri.SolidBegin("primitive")
ri.TransformBegin()
ri.Rotate(90,0,1,0)
ri.Cylinder(.1,-0.5,0.5,360)
ri.TransformEnd()
ri.SolidEnd()

ri.SolidEnd()

ri.SolidEnd() # end intersection
```

```
SolidBegin "intersection"
  SolidBegin "primitive"
    TransformBegin
      Translate 0 -0.2 0
      #Cube Generated by Cube Function
      Patch "bilinear" "P" [-0.25 -0.25 0.25 -0.25 0.25 0.25 0.25 -0.25 0.25 0.25 0.25 0.25]
      Patch "bilinear" "P" [-0.25 -0.25 -0.25 -0.25 0.25 -0.25 0.25 -0.25 -0.25 0.25 0.25 -0.25]
      Patch "bilinear" "P" [-0.25 -0.25 -0.25 -0.25 0.25 -0.25 -0.25 -0.25 0.25 -0.25 0.25 0.25]
      Patch "bilinear" "P" [0.25 -0.25 -0.25 0.25 0.25 -0.25 0.25 -0.25 0.25 0.25 0.25 0.25]
      Patch "bilinear" "P" [0.25 -0.25 0.25 0.25 -0.25 -0.25 -0.25 -0.25 0.25 -0.25 -0.25 -0.25]
      Patch "bilinear" "P" [0.25 0.25 0.25 0.25 0.25 -0.25 -0.25 0.25 0.25 -0.25 0.25 -0.25]
      #--End of Cube Function--
    TransformEnd
  SolidEnd
  SolidBegin "union"
    SolidBegin "primitive"
      Cylinder 0.1 -0.5 0.5 360
    SolidEnd
    SolidBegin "primitive"
      TransformBegin
        Rotate 90 0 1 0
        Cylinder 0.1 -0.5 0.5 360
      TransformEnd
    SolidEnd
  SolidEnd
SolidEnd
```

The Obj File Format

- Alias Wavefront obj files define the geometry and other properties for objects which can be easily used within animation packages.
- Object files can be in ASCII format (.obj) or binary format (.mod).
- For simplicity the ASCII format will be discussed here as it is easier to parse the data and is a good exercise for file and string handling.
- In its current release, the .obj file format supports both polygonal objects and free-form objects such as curves, nurbs etc.
- For simplicity on polygonal models will be discussed.

File Format

- The following types of data may be included in an .obj file. In this list, the keyword (in parentheses) follows the data type.
    - geometric vertices (v)
    - texture vertices (vt)
    - vertex normals (vn)
    - face (f)
    - group name (g)
    - smoothing group (s)
    - material name (usemtl)
    - material library (mtllib)
- all values are stored on a single line terminated with a \n (new line character)

File Format

```
#begin 8 vertices
v -0.500000 -0.500000 -0.500000
v 0.500000 -0.500000 -0.500000
v -0.500000 0.500000 -0.500000
v 0.500000 0.500000 -0.500000
v -0.500000 -0.500000 0.500000
v 0.500000 -0.500000 0.500000
v -0.500000 0.500000 0.500000
v 0.500000 0.500000 0.500000
#end 8 vertices

```

```
#begin 6 faces
usemtl scene_material1
f 1/1/1 3/2/2 4/3/3 2/4/4
usemtl scene_material1
f 1/5/5 2/6/6 6/7/7 5/8/8
usemtl scene_material1
f 1/9/9 5/10/10 7/11/11 3/12/12
usemtl scene_material1
f 2/13/13 4/14/14 8/15/15 6/16/16
usemtl scene_material1
f 3/17/17 7/18/18 8/19/19 4/20/20
usemtl scene_material1
f 5/21/21 6/22/22 8/23/23 7/24/24
#end 6 faces
```

```
#begin 24 normals
vn 0.000000 0.000000 -1.000000
vn 0.000000 0.000000 -1.000000
vn 0.000000 0.000000 -1.000000
vn 0.000000 0.000000 -1.000000
vn 0.000000 -1.000000 0.000000
vn 0.000000 -1.000000 0.000000
vn 0.000000 -1.000000 0.000000
vn 0.000000 -1.000000 0.000000
vn -1.000000 0.000000 0.000000
vn -1.000000 0.000000 0.000000
vn -1.000000 0.000000 0.000000
vn -1.000000 0.000000 0.000000
vn 1.000000 0.000000 0.000000
vn 1.000000 0.000000 0.000000
vn 1.000000 0.000000 0.000000
vn 1.000000 0.000000 0.000000
vn 0.000000 1.000000 0.000000
vn 0.000000 1.000000 0.000000
vn 0.000000 1.000000 0.000000
vn 0.000000 1.000000 0.000000
vn 0.000000 0.000000 1.000000
vn 0.000000 0.000000 1.000000
vn 0.000000 0.000000 1.000000
vn 0.000000 0.000000 1.000000
#end 24 vertex normals
```

```
#begin 24 texture vertices
vt 0.125000 0.304087 0.000000
vt 0.125000 0.695913 0.000000
vt -0.125000 0.695913 0.000000
vt -0.125000 0.304087 0.000000
vt 0.125000 0.304087 0.000000
vt 0.875000 0.304087 0.000000
vt 0.625000 0.304087 0.000000
vt 0.375000 0.304087 0.000000
vt 0.125000 0.304087 0.000000
vt 0.375000 0.304087 0.000000
vt 0.375000 0.695913 0.000000
vt 0.125000 0.695913 0.000000
vt 0.875000 0.304087 0.000000
vt 0.875000 0.695913 0.000000
vt 0.625000 0.695913 0.000000
vt 0.625000 0.304087 0.000000
vt 0.125000 0.695913 0.000000
vt 0.375000 0.695913 0.000000
vt 0.625000 0.695913 0.000000
vt 0.875000 0.695913 0.000000
vt 0.375000 0.304087 0.000000
vt 0.625000 0.304087 0.000000
vt 0.625000 0.695913 0.000000
vt 0.375000 0.695913 0.000000
#end 24 texture vertices

```

A Python Obj Loader

Obj
verts : list norm : list text : list face : list
__init__(self, File,Round): getExtents(self) : Polygon(self,ri,Round) PointsPolygon(self,ri,Round) SubDivisionMesh(self,ri,Round)

- The python obj class is passed a file name and loads the obj file data into a number of lists
- These are then made available to a number of methods which produced renderman output for rendering
- The Round parameter specifies a rounding for the floating point values

Methods

- __init__ File the filename Round a rounding function for the data values read in

- Polygon output renderman polygons with normals and texture co-ordinates if in the obj file
- PointsPolygon outputs simple points polygons without normals and texture co-ordinates
- SubDivisionMesh outputs a sub division mesh without normals or texture co-ordinates

```
# ctor to assign values
def __init__(self, File,Round):
  print "opening : "+File
  # open the file
  ip = open(File,'r')
  #grab the data as lines
  data=ip.readlines()
  # for each line check for one of our tokens
  for line in data :
    # we assume that our Tokens are always the first element of the line (which IIRC the rispec
        specifies)
    # so we split each line and look at the first element
    tokens=line.split()
    # make sure we have a token to check against
    if(len(tokens) >0 ) :
      if(tokens[0] =="v") :
        #print "found vert"
        # create a tuple of the vertex point values
        vert=[round(float(tokens[1]),Round),round(float(tokens[2]),Round),round(float(tokens[3])
            ,Round)]
        # then add it to our list
        self.verts+=[vert]
      elif(tokens[0] =="vn") :
        #print "found normal"
        # create a tuple of the normal values
        normal=[round(float(tokens[1]),Round),round(float(tokens[2]),Round),round(float(tokens[3
            ]),Round)]
        # then add it to our list
        self.norm+=[normal]
      elif(tokens[0] =="vt") :
        #print "found texture"
        # create a tuple of the texture values
        #*****************************************************************
        # NOTE RENDERMAN Maps Textures in the T from top to bottom so we
        # calculate 1.0 - t here so the image will map properly
        #
        tx=[round(float(tokens[1]),Round),1-round(float(tokens[2]),Round)]
        #
        #*****************************************************************
        # then add it to our list
        self.text+=[tx]
      # now we have a face value
      elif(tokens[0] =="f") :
        # add the face to the list and we will process it later (see below)
        self.face+=[line]

  # close the file
  ip.close()
```

getExtents

- The getExtents method finds the min / max vertex values of the obj
- It uses the multiple return feature of python

```
1  def getExtents(self) :
2    xmin=self.verts[min((n, i) for i, n in enumerate(self.verts))[1]][0]
3    xmax=self.verts[max((n, i) for i, n in enumerate(self.verts))[1]][0]
4    ymin=self.verts[min((n, i) for i, n in enumerate(self.verts))[1]][1]
5    ymax=self.verts[max((n, i) for i, n in enumerate(self.verts))[1]][1]
6    zmin=self.verts[min((n, i) for i, n in enumerate(self.verts))[1]][2]
7    zmax=self.verts[max((n, i) for i, n in enumerate(self.verts))[1]][2]
8
9    print xmin,xmax,ymin,ymax,zmin,zmax
10
11   return xmin,xmax,ymin,ymax,zmin,zmax
```

```
1  xmin,xmax,ymin,ymax,zmin,zmax= obj.getExtents()
```

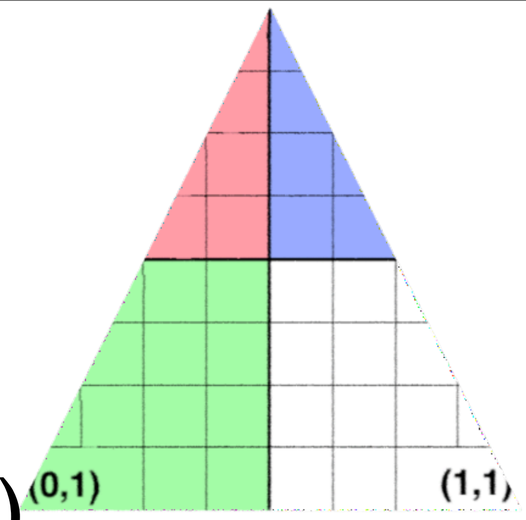
Polygons

- The RenderMan Interface supports two basic types of polygons:
  - a convex polygon and a general concave polygon with holes.
- In both cases the polygon must be planar.
- Collections of polygons can be passed by giving a list of points and an array that indexes these points.
- The geometric normal of the polygon is computed by computing the normal of the plane containing the polygon (unless it is explicitly specified).

Normals

- If the current orientation is left-handed, then a polygon whose vertices were specified in clockwise order (from the point of view of the camera) will be a front-facing polygon (that is, will have a normal vector which points toward the camera).
- If the current orientation is right-handed, then polygons whose vertices were specified in counterclockwise order will be front-facing. The shading normal is set to the geometric normal unless it is explicitly specified at the vertices.

ri.Polygon



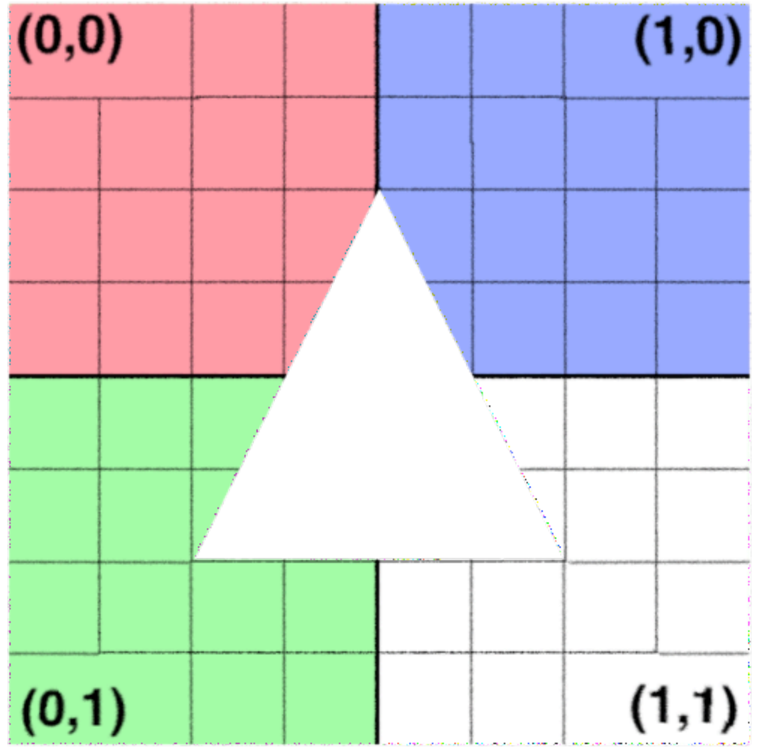
- Polygons are specified with the riPolygon function.
- The parameter list must include at least position ("P") information.

```
1 points=[-1,-1,0,0,1,0,1,-1,0]
2 normals=[0,0,-1,0,0,-1,0,0,-1]
3 tx=[0,1,0.5,0,1,1]
4
5 ri.Surface("texmap",{"string␣texname":"ratGrid.tex","float␣maptype" :[3]})
6 ri.Polygon({ri.P:points,ri.N:normals,ri.ST:tx})
```

```
1 Polygon "varying␣float[2]␣st" [0 1 0.5 0 1 1] "vertex␣point␣P" [-1 -1 -2 0
      1 1 1 -1 0]
2     "varying␣normal␣N" [0 0 -1 0 0 -1 0 0 -1]
```

ri.GeneralPolygon

- Define a general planar concave polygon with holes. This polygon is specified by giving nloops lists of vertices. The first loop is the outer boundary of the polygon; all additional loops are holes.




```
points=[-2,-2,0,-2,2,0,2,2,0,2,-2,0,-1,-1,0,0,1,0,1,-1,0]
tx=[0,1, 0,0, 1,0, 1,1,  0.25,0.75, 0.5,0.25, 0.75,0.75]

ri.Surface("texmap",{"string␣texname":"ratGrid.tex","float␣maptype" :[3]})
ri.GeneralPolygon([4,3],{ri.P:points,ri.ST:tx})
```

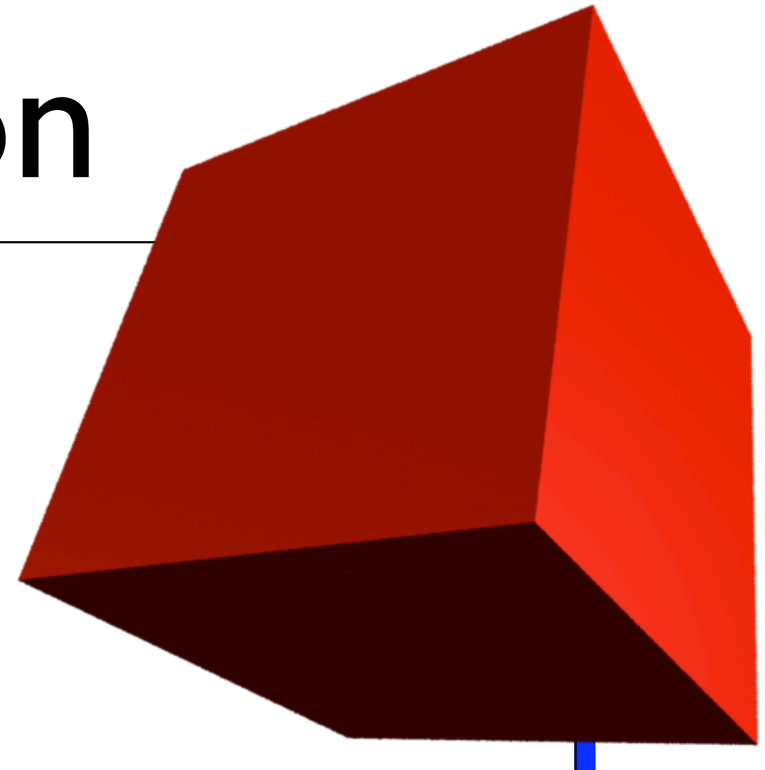
```
GeneralPolygon [4 3]
      "varying␣float[2]␣st" [0 1 0 0 1 0 1 1 0.25 0.75 0.5 0.25 0.75 0.75]
      "vertex␣point␣P" [-2 -2 0 -2 2 0 2 2 0 2 -2 0 -1 -1 0 0 1 0 1 -1 0]
```

ri.PointsPolygon

- Define npolys planar convex polygons that share vertices.
- The array nvertices contains the number of vertices in each polygon and has length npolys.
- The array vertices contains, for each polygon vertex, an index into the varying primitive variable arrays.
- The varying arrays are 0-based. vertices has length equal to the sum of all of the values in the nvertices array.
- Individual vertices in the parameterlist are thus accessed indirectly through the indices in the array vertices.

ri.PointsPolygon

```
points=[-0.5,-0.5,-0.5,
         0.5,-0.5,-0.5,
        -0.5, 0.5,-0.5,
         0.5, 0.5,-0.5,
        -0.5,-0.5, 0.5,
         0.5,-0.5, 0.5,
        -0.5, 0.5, 0.5,
         0.5, 0.5, 0.5]

npolys=[4,4,4,4,4,4]
nvertices=[0,2,3,1,0,1,5,4,0,4,6,2,1,3,7,5,2,6,7,3,4,5,7,6]
ri.Surface("plastic")
ri.PointsPolygons(npolys,nvertices,{ri.P:points})
```



```
Surface "plastic"
PointsPolygons [4 4 4 4 4 4]
          [0 2 3 1 0 1 5 4 0 4 6 2 1 3 7 5 2 6 7 3 4 5 7 6]
"vertex␣point␣P" [-0.5 -0.5 -0.5 0.5 -0.5 -0.5 -0.5 0.5 -0.5 0.5 0.5 -0.5
           -0.5 -0.5 0.5 0.5 -0.5 0.5 -0.5 0.5 0.5 0.5 0.5 0.5]
```

ri.PointsGeneralPolygons

- Define npolys general planar concave polygons, with holes, that share vertices.
- The array nloops indicates the number of loops comprising each polygon and has a length npolys.
- The array nvertices contains the number of vertices in each loop and has a length equal to the sum of all the values in the array nloops.
- The array vertices contains, for each loop vertex, an index into the varying primitive variable arrays.

ri.PointsGeneralPolygons

```
points=[0,0,1,
     0,1,1,
     0,2,1,
     0,0,0,
     0,1,0,
     0,2,0,
     0,0.25,0.5,
     0,0.75,0.75,
     0,0.75,0.25,
     0,1.25,0.5,
     0,1.75,0.75,
     0,1.75,0.25]

npolys=[2,2]
nvertices=[4,3,4,3]
PolyVerts=[0,1,4,3,6,7,8,1,2,5,4,9,10,11]

ri.Surface("plastic")
ri.PointsGeneralPolygons(npolys,nvertices,PolyVerts,{ri.P:points})
```

```
PointsGeneralPolygons   [2 2] [4 3 4 3]
                [0 1 4 3 6 7 8 1 2 5 4 9 10 11]
                "vertex␣point␣P" [0 0 1 0 1 1 0 2 1 0 0 0
                0 1 0 0 2 0 0 0.25 0.5 0 0.75 0.75
                0 0.75 0.25 0 1.25 0.5 0 1.75 0.75 0 1.75 0.25]
```

Obj Polygon Method

```
def Polygon(self,ri,Round) :
  for f in self.face :
    # create some empty data structures to be filled as we go
    vertices=[]
    normals=[]
    points=[]
    tx=[]
    fd=f.split()
    # the face is in the structure shown below Vert / TX / Norm we are gaurenteed to have a
    # Vert but the others may not be there
    #1/1/1 3/2/2 4/3/3 2/4/4
    for perface in fd[1:] :
      index=perface.split("/")
      # get the point array index
      pind=int(index[0])-1
      points.append(round(float(self.verts[pind][0]),Round))
      points.append(round(float(self.verts[pind][1]),Round))
      points.append(round(float(self.verts[pind][2]),Round))
      # check for textures and add if there
      if(index[1] !="") :
        tind=int(index[1])-1
        tx.append(round(float(self.text[tind][0]),Round))
        tx.append(round(float(self.text[tind][1]),Round))
      # checl for normals and check they are there
      if(index[2] !="") :
        nind=int(index[2])-1
        normals.append(round(float(self.norm[nind][0]),Round))
        normals.append(round(float(self.norm[nind][1]),Round))
        normals.append(round(float(self.norm[nind][2]),Round))

    # create a dictionary to store the polygon data, we always have a point so we can add
    #this directly
    PolyData={ri.P:points}
    # now see if we have any texture co-ordinates and add them to the dictionary if we do
    if index[1] !="" :
      PolyData[ri.ST]=tx
    # check for normals and add them to the dictionary as well
    if index[2] !="" :
      PolyData[ri.N]=normals
    # finally we generate the Polygon from the data
    ri.Polygon(PolyData)  #{ri.P:points,ri.N:normals,ri.ST:tx})
```

PointsPolygon Method

```
def PointsPolygon(self,ri,Round) :
  npolys=[]
  vertices=[]
  normals=[]
  for f in self.face :
    # create some empty data structures to be filled as we go
    fd=f.split()
    # the face is in the structure shown below Vert / TX / Norm we are gaurenteed to have a
    # Vert but the others may not be there
    #1/1/1 3/2/2 4/3/3 2/4/4
    npolys.append(len(fd)-1)
    for perface in fd[1:] :
      index=perface.split("/")
      # get the point array index
      vertices.append(int(index[0])-1)
  v=[]
  # the verts are in a list of lists at the moment so we extract them
  # into a single list for prman
  for i in range(0,len(self.verts)) :
    for x in range(0,len(self.verts[i])) :
      v.append(self.verts[i][x])

  ri.PointsPolygons(npolys,vertices,{ri.P:v})
```

SubDivisionMesh

```
def SubDivisionMesh(self,ri,Round) :
  npolys=[]
  vertices=[]
  normals=[]
  for f in self.face :
    # create some empty data structures to be filled as we go
    fd=f.split()
    # the face is in the structure shown below Vert / TX / Norm we are gaurenteed to have a
    # Vert but the others may not be there
    #1/1/1 3/2/2 4/3/3 2/4/4
    npolys.append(len(fd)-1)
    for perface in fd[1:] :
      index=perface.split("/")
      # get the point array index
      vertices.append(int(index[0])-1)
  v=[]
  # the verts are in a list of lists at the moment so we extract them
  # into a single list for prman
  for i in range(0,len(self.verts)) :
    for x in range(0,len(self.verts[i])) :
      v.append(self.verts[i][x])

  #ri.PointsPolygons(npolys,vertices,{ri.P:v})
  ri.SubdivisionMesh("loop", npolys, vertices, [ri.CREASE], [2, 1], [3, 0], [20], {ri.P:v})

```

Using Obj Class

```
objFile="cube.obj"

obj=Obj(objFile,5)
.......

ri.Translate(0,-8,22)

ri.TransformBegin()
ri.Surface("texmap",{"string␣texname":"clubbot.tx","float␣maptype" :[3]})
ri.Translate(-9,0,0)
obj.Polygon(ri,5)
ri.TransformEnd()

ri.Color([1,1,1])
ri.Surface("plastic")
ri.ShadingInterpolation("constant")
ri.TransformBegin()
obj.PointsPolygon(ri,5)
ri.TransformEnd()

ri.TransformBegin()
ri.Translate(9,0,0)
obj.SubDivisionMesh(ri,5)
ri.TransformEnd()
```

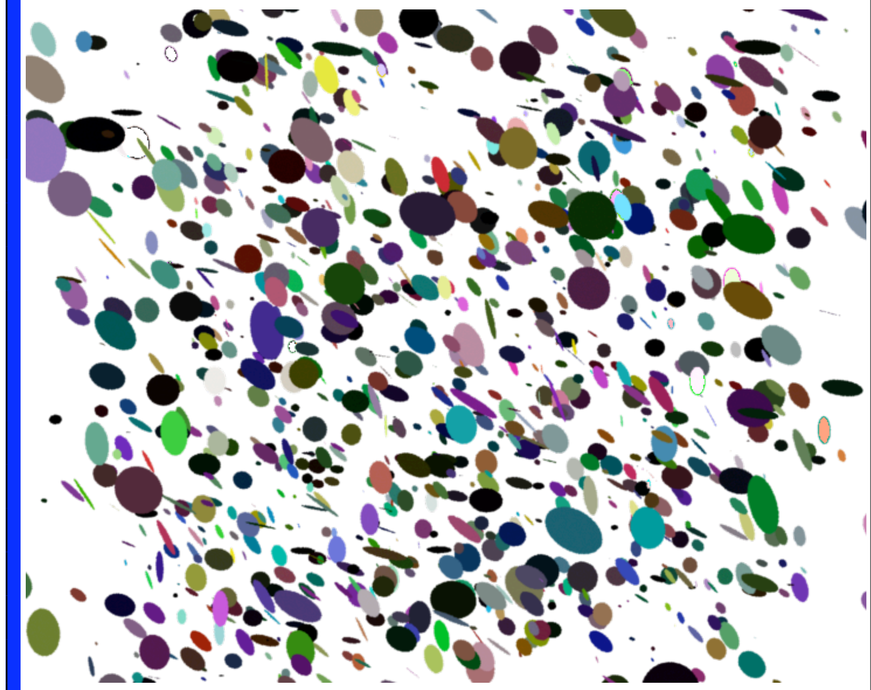
ri.Points & ri.Curves

- The RenderMan Interface includes lightweight primitives for specifying point clouds, lines, curves, or ribbons.
- These primitives are especially useful for representing many particles, hairs, etc.

ri.Points

- Draws npoints number of point-like particles.
- parameterlist is a list of token-array pairs where each token is one of the standard geometric primitive variables, a variable that has been defined with RiDeclare, or is given as an inline declaration.
- The parameter list must include at least position ("P") information, one value for each particle.
- If a primitive variable is of class varying or vertex, the array contains npoints data values of the appropriate type, i.e., one per particle.
- If the variable is uniform or constant, the array contains a single element.

```
from random import uniform as ru

.......


points=[]
width=[]
colour=[]
normals=[]
# get a pointer to the append method as it is faster than calling it
# each time
pappend=points.append
wappend=width.append
cappend=colour.append
nappend=normals.append
# ru is random.uniform brought in by the import statement above
for i in range(0,1500) :
  for ix in range(0,3) :
    cappend(ru(0,1))
    pappend(ru(-2,2))
    nappend(ru(0,1))
  wappend(ru(0.01,0.2))

ri.Surface("plastic")
ri.Points({ri.P:points,ri.CS:colour,ri.WIDTH:width,ri.N:normals})
```



```
Points "varying color Cs" [0.548854 0.880192 0.535627 0.476127 0.189091 0.239006]
       "vertex point P" [-1.13358 -0.638128 1.18703 -0.310617 1.91478 -0.773724]
       "varying float width" [0.131713 0.033679]
       "varying normal N" [0.0171959 0.446503 0.822508 0.338246 0.781155 0.171715]
```

A simple Particle System

```
def Draw(self,ri) :
  points=[]
  width=[]
  colour=[]
  # get pointers to the append functions to speed up loop
  pa=points.append
  wa=width.append
  ca=colour.append
  pl=self.ParticleList
  for i in range(0,self.NumParticles) :
    pa(pl[i].Position[0])
    pa(pl[i].Position[1])
    pa(pl[i].Position[2])
    wa(pl[i].Width)
    ca(pl[i].Colour[0])
    ca(pl[i].Colour[1])
    ca(pl[i].Colour[2])

  ri.AttributeBegin()
  print points
  ri.Points({ri.P:points,"varying␣float␣width":width,"
      varying␣color␣Cs":colour})
  ri.AttributeEnd()


```

Emitter
Position : list NumParticles : list ParticleList : list
__init__(self, Pos,NumParticles): Update(self) : Draw(self,ri) :

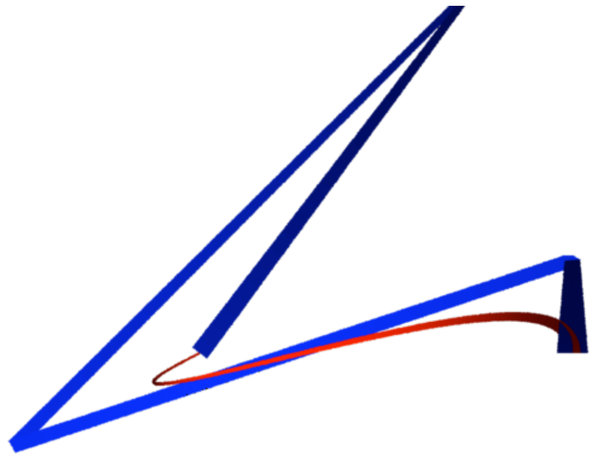
0..1

*

Particle
Colour : list Position : list Direction : list Life : int EndLife : int Width : float
__init__(self, Pos,Colour,Direction,EndLife,Width) Update(self)

```
def Update(self) :
    self.Position[0]+=self.Direction[0]
    self.Position[1]+=self.Direction[1]
    self.Position[2]+=self.Direction[2]
    self.Life+=1
    if self.Life > self.EndLife :
      self.Life=0
      self.Position=[0,0,0]
```

ri.Curves



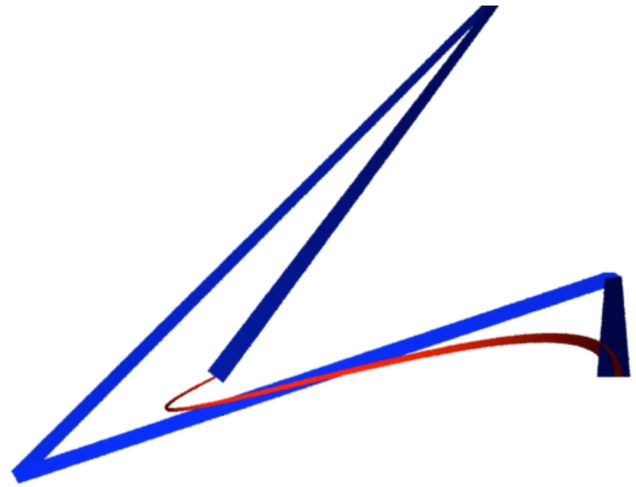
- Draws ncurves number of lines, curves, or ribbon-like particles of specified width through a set of control vertices.
- Multiple disconnected individual curves may be specified using one call to RiCurves.
- The parameter ncurves is the number of individual curves specified by this command
- nvertices is an array of length ncurves integers specifying the number of vertices in each of the respective curves.

ri.Curves

- The interpolation method given by type can be either "linear" or "cubic".
- Cubic curves interpolate using the v basis matrix and step size set by RiBasis.
- The u parameter changes across the width of the curve, whereas the v parameter changes across the length of the curve (i.e., the direction specified by the control vertices).
- Curves may wrap around in the v direction, depending on whether wrap is "periodic" or "nonperiodic".
- Curves that wrap close upon themselves at the ends and the first control points will be automatically repeated. As many as three control points may be repeated, depending on the basis matrix of the curve.

ri.Curves

```
ri.Color([1,0,0])
points= [0, 0, 0 ,-1, -.5 ,1 ,2 ,.5 ,1 ,1 ,0, -1 ]
width=[0.01,0.04]
ri.Curves( "cubic",[4],"nonperiodic",{ri.P:points, ri.WIDTH : width})

ri.Color([0,0,1])
points2=[0,0,0,3,4,5,-1,-.5,1,2,.5,1,1,0,-1]
ri.Curves("linear",[5],"nonperiodic",{ ri.P:points2 , ri.CONSTANTWIDTH:[0.075]})
```



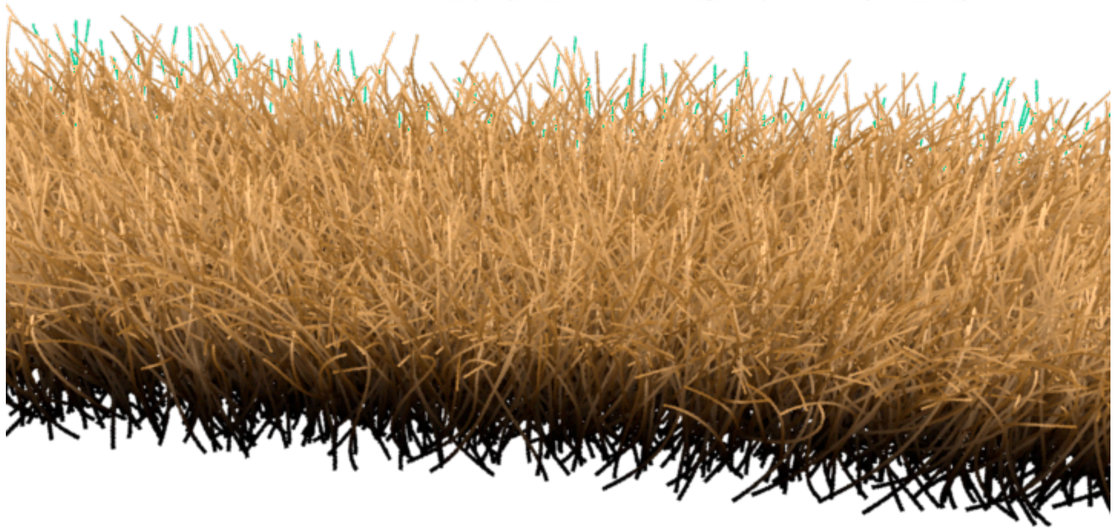
```
Color [1 0 0]
Curves "cubic" [4] "nonperiodic" "vertex␣point␣P" [0 0 0 -1 -0.5 1 2 0.5 1 1 0 -1]
    "varying␣float␣width" [0.01 0.04]

Color [0 0 1]
Curves "linear" [5] "nonperiodic"
    "vertex␣point␣P" [0 0 0 3 4 5 -1 -0.5 1 2 0.5 1 1 0 -1]
    "constant␣float␣constantwidth" [0.075]
```

MultiCurve.py

```
surface hair (float Ka = 1, Kd = .6, Ks = .35, roughness = .15;
        color rootcolor = color (.109, .037, .007);
        color tipcolor = color (.519, .325, .125);
        color specularcolor = (color(1) + tipcolor) / 2;
       )
{
    vector T = normalize (dPdv); /* tangent along length of hair */
    vector V = -normalize(I);    /* V is the view vector */
    color Cspec = 0, Cdiff = 0;  /* collect specular & diffuse light */
    float cosang;

    /* Loop over lights, catch highlights as if this was a thin cylinder */
    illuminance (P) {
  cosang = cos (abs (acos (T.normalize(L)) - acos (-T.V)));
  Cspec += Cl * v * pow (cosang, 1/roughness);
  Cdiff += Cl * v;
  /* We multipled by v to make it darker at the roots.  This
   * assumes v=0 at the root, v=1 at the tip.
   */
    }

    Oi = Os;
    Ci = Oi * (mix(rootcolor, tipcolor, v) * (Ka*ambient() + Kd*Cdiff)
                  + (Ks * Cspec * specularcolor));
}
```

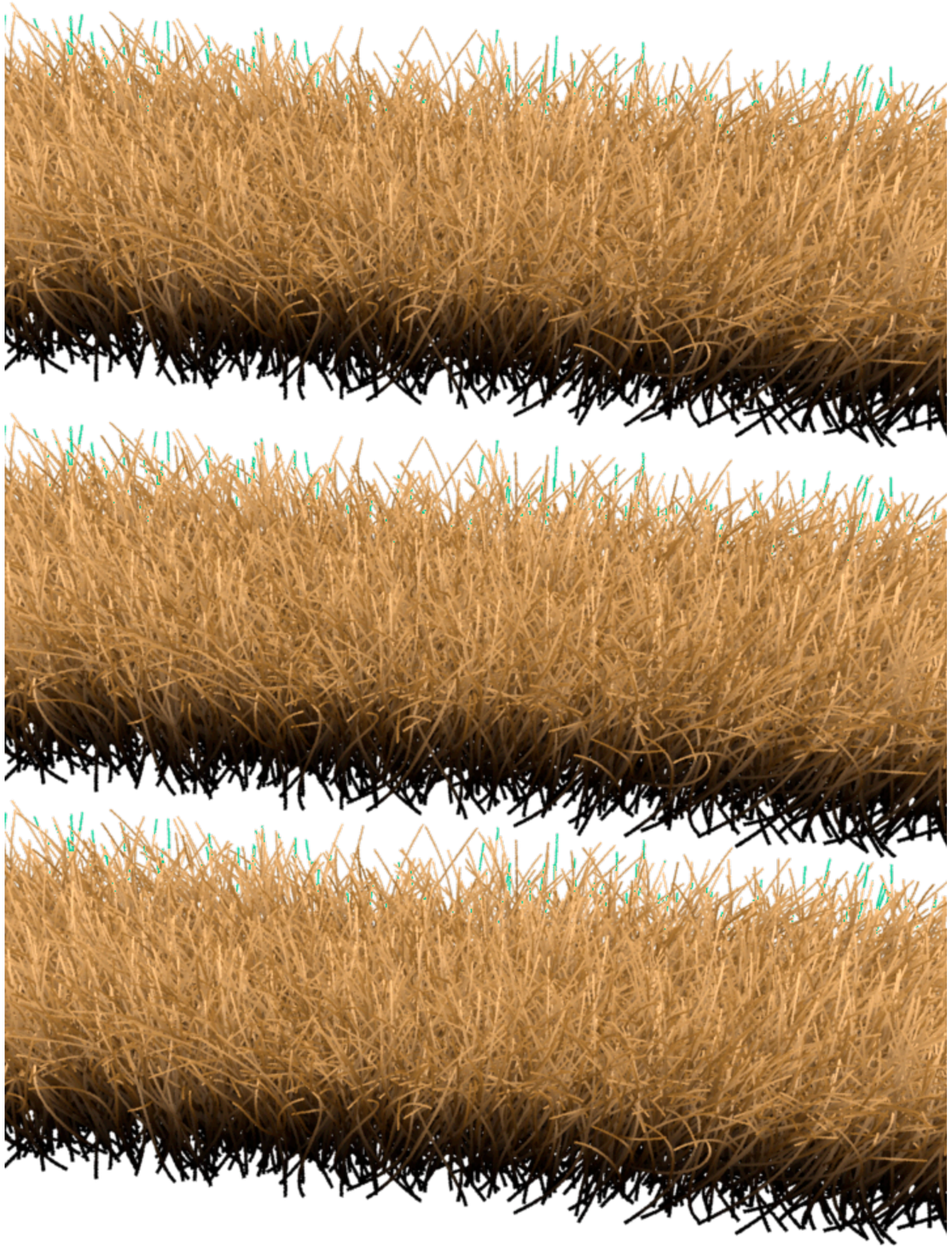
```
points=[]
pappend=points.append
width=[]
wappend=width.append
npoints=[]
npappend=npoints.append
random.seed(129234)
ru=random.uniform
zpos=-2.0
plus=0.1
minus=-0.1
while(zpos < 2.0 ) :
  xpos=-2.0
  while (xpos < 2.0) :
    pappend(xpos+ru(minus,plus))
    pappend(0)
    pappend(zpos+ru(minus,plus))

    pappend(xpos+ru(minus,plus))
    pappend(0.1)
    pappend(zpos+ru(minus,plus))

    pappend(xpos+ru(minus,plus))
    pappend(0.2)
    pappend(zpos+ru(minus,plus))

    pappend(xpos+ru(minus,plus))
    pappend(0.3+ru(-0.1,0.1))
    pappend(zpos+ru(minus,plus))

    wappend(0.006)
    wappend(0.001)
    npappend(4)
    xpos+=0.02
  zpos+=0.02


ri.Surface("hair")
ri.Curves( "cubic",npoints,"nonperiodic",{ri.P:points, ri.WIDTH : width})
```

Animated Curves



```
def UpdateField(points,offset,offset1) :
for i in range(0,len(points),12) :
  points[i+6]+=offset
  points[i+9]+=offset1
```

```
def BuildField(wi,depth,inc,points,width,npoints) :
xmin=-wi/2.0
xmax=wi/2.0
zmin=-depth/2.0
zmax=depth/2.0
pappend=points.append
wappend=width.append
npappend=npoints.append
random.seed(1)
ru=random.uniform
zpos=zmin
plus=0.1
minus=-0.1
while(zpos < zmax ) :
  xpos=xmin
  while (xpos < xmax) :
    pappend(xpos+ru(minus,plus))
    pappend(0)
    pappend(zpos+ru(minus,plus))

    pappend(xpos+ru(minus,plus))
    pappend(0.1)
    pappend(zpos+ru(minus,plus))

    pappend(xpos+ru(minus,plus))
    pappend(0.2)
    pappend(zpos+ru(minus,plus))

    pappend(xpos+ru(minus,plus))
    pappend(0.3+ru(-0.1,0.1))
    pappend(zpos+ru(minus,plus))

    wappend(0.006)
    wappend(0.003)
    npappend(4)
    xpos+=inc
  zpos+=inc
```



```
offset=0
for frame in range(0,100) :
.................
.....Scene setup
.................

  if dir ==0 :
    offset+=0.001
    dircount+=1
    UpdateField(points,offset,offset)
    if(dircount == 10) :
      dir=1
      dircount=0
      offset-=0.0001
  else :
    offset-=0.001
    dircount+=1
    UpdateField(points,-offset,-offset)
    if(dircount == 10) :
      dir=0
      dircount=0
      offset+=0.0001

  ri.Surface("hair")
  ri.Curves( "cubic",npoints,"nonperiodic",{ri.P:
      points, ri.WIDTH : width})
```

Procedural Primitives

- Procedural primitives allow use to call a helper program which generates geometry on-the-fly in response to procedural primitive requests in the RIB stream.
- Each generated procedural primitive is described by a request to the helper program, in the form of an ASCII datablock which describes the primitive to be generated.
- This datablock can be anything which is meaningful and adequate to the helper program, such as a sequence of a few floating point numbers, a filename, or a snippet of code in a interpreted modeling language.
- In addition the renderer supplies the detail of the primitive's bounding box, so that the generating program can make decisions on what to generate based on how large the object will appear on-screen.

Procedural Primitives

- The generation program reads requests on its standard input stream, and emits RIB streams on its standard output stream.
- These RIB streams are read into the renderer as though they were read from a file (as with ReadArchive), and may include any standard RenderMan attributes and primitives (including procedural primitive calls to itself or other helper programs).
- As long as any procedural primitives exist in the rendering database which require the identical helper program for processing, the socket connection to the program will remain open.
- This means that the program should be written with a loop which accepts any number of requests and generates a RIB "snippet" for each one.

Procedural Primitives

- The specific syntax of the request from the renderer to the helper program is extremely simple, as follows:

```
1 fprintf(socket, "%g␣%s\n", detail, datablock);
```

- The detail is provided first, as a single floating-point number, followed by a space, followed by the datablock and finally a newline.
- The datablock is completely uninterpreted by the renderer or by the socket write, so any newlines or special characters should be preserved (but the quotation marks that make it a string in the RIB file will, of course, be missing).

Procedural Primitives

- The helper program's response should be to create a RIB file on stdout,

```
1  RiBegin(RI_NULL);
2     RiAttributeBegin();
3
4        // various attributes
5
6        // various primitives
7
8     RiAttributeEnd();
9   RiArchiveRecord(RI_COMMENT,"\377");
10 RiEnd();
```

Procedural Primitives

- Notice, in particular, the special trick which the helper program must use to stay in synchronized communication with the renderer.
- stdout should not be closed when the RIB snippet is complete, but rather a single '\377' character should be emitted and the stdout stream flushed.
- • This will signal the renderer that this particular snippet is complete, yet leave the stream open in order to write the next snippet.
- • The use of RiArchiveRecord and RiEnd as above will accomplish this when the RIB client library is used.
- Warning: if the '\377' character is not emitted, nor is accidentally not flushed through the stdout pipe to the renderer, the render will hang.
- When the renderer is done with the helper program, it will close its end of the IPC socket, so reading an EOF on stdin is the signal for the helper program to exit.

Procedural Primitives

- In RIB, the syntax for specifying a RIB-generating program procedural primitive is:

```
1 Procedural "RunProgram" [ "program" "datablock" ] [ bound ]
```

- program is the name of the helper program to execute, and may include command line options. datablock is the generation request data block. It is an ASCII string which is meaningful to program, and adequately describes the children which are to be generated.

- Notice that program quoted string in the RIB file, so if it contains quote marks or other special characters, these must be escaped in the standard way.

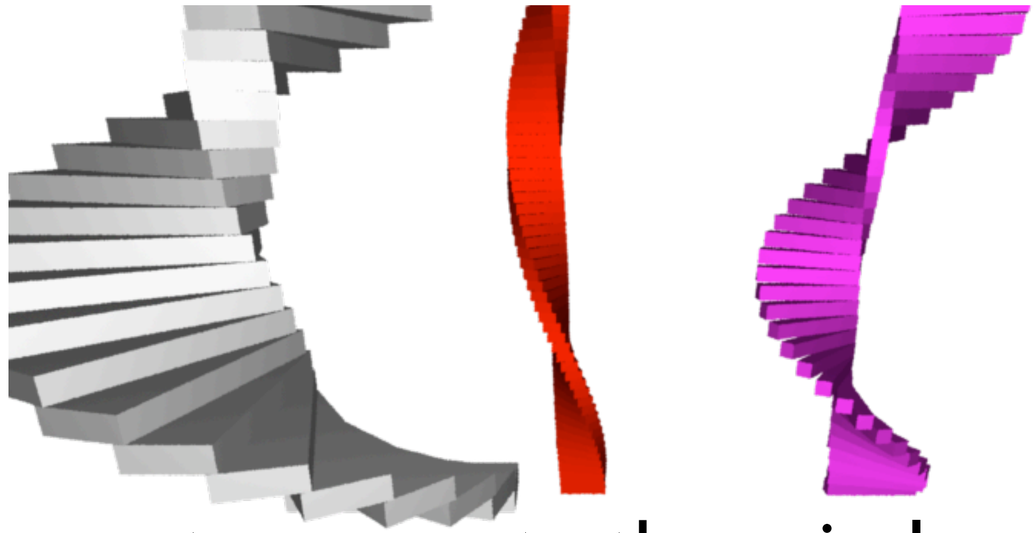
- The bound is an array of six floating point numbers which is xmin, xmax, ymin, ymax, zmin, zmax in the current object space.

Template Helper Program

```
#####################################################################
# prman RunPrograms must read in a the data from stdin in the format
# The detail is provided first, as a single floating-point number, followed by a space,
# followed by the datablock and finally a newline.
# The datablock is completely uninterpreted by the renderer or by the socket write,
# so any newlines or special characters should be preserved
# (but the quotation marks that make it a string in the RIB file will, of course, be
    missing).
#####################################################################

if __name__ == '__main__':
  #get the datablock from prman
  args = sys.stdin.readline()
  # now loop round as we may get more data
  while(args) :
    # split the  data into blocks
    words =args.split()
    # the first is the detail (a float)
    detail = float(words[0])
    # now check  to see if we have enough arguments
    if len(words) <6 :
      NiceExit("Error␣in␣ribfile␣not␣enough␣arguments\n␣Args␣were␣%s" %args)
      break
    # so we have enough arguments so extract the detail we need
    else :
      # first get the datablock from prman
      datablock = words[1:]

      # do our rib output here


      # we now close the rib stream and get the  next datablock  if there  is  one

      Write('\377')
      sys.stdout.flush()
# read the next line
    args = sys.stdin.readline()

```

```
#############################################
# Signal to prman that we have finished (\377) and
# print out error message on stderr
#############################################
def NiceExit(message) :
    sys.stderr.write(message)
    # flush stream so prman knows we are done
    Write('\377')
    sys.stdout.flush()
```

Spiral.py



- The spiral.py script is an external program to generate the spirals.
- It doesn't use the ri python library just output to stdout
- To call it in the RunProgram.py renderman program we use the following code

```
1 program="Procedural␣\"RunProgram\"␣[\"spiral.py\"␣\"5␣0.5␣1.5␣10␣15\"]␣[-5␣5␣-5␣5␣-35␣35]\n"
2 ri.ArchiveRecord(ri.VERBATIM,program)
```

- note we have to use the ri.VERBATIM call to output our data to the rib file as the call to ri.Procedural doesn't work for external programs

references

- Renderman documentation
- Using Procedural Primitives in PhotoRealistic RenderMan
- PRMan for Python — A Bridge from PRMan to Python (and Back)
- http://www.fileformat.info/format/wavefrontobj/
- App not #19 Using the RiCurves Primitive
- http://docs.python.org/
- Appendix D - RenderMan Interface Bytestream Conventions
- Application Note #3 How To Render Quickly Using PhotoRealistic RenderMan