Introduction to Renderman

using the Python API

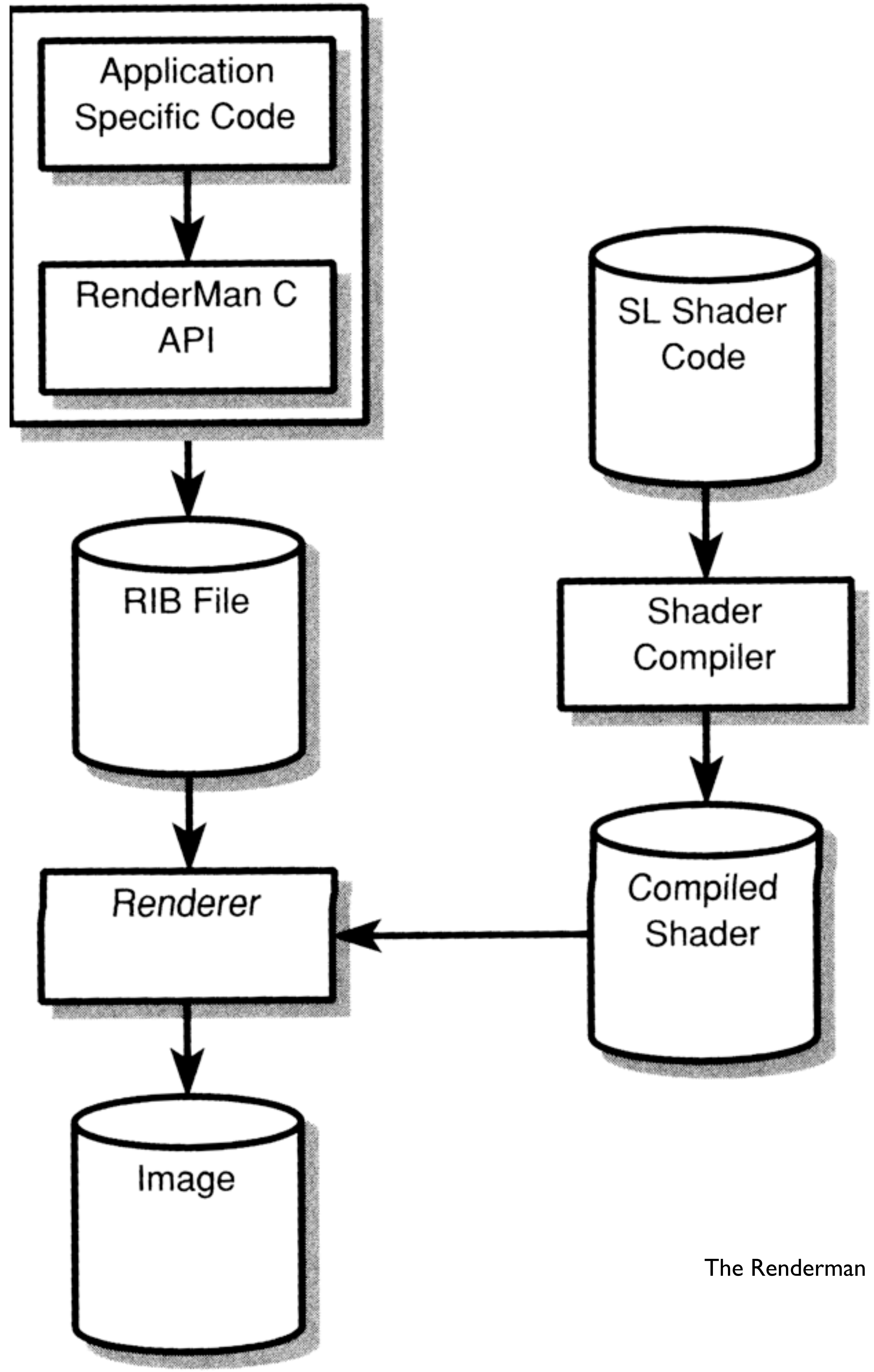
Introduction to Renderman

- When Renderman was first proposed it was a C like API for the development of scene descriptions to be rendered
- The description file produced is usually called a RIB (Renderman Interface Byte stream) file and this is then passed to the renderer
- The description of how the surface is to be textured and shaded is determined by a number of files called shaders
- These can describe surfaces, displacements, lights, volumes.

Renderman Python

- As of Version 14 (2008) renderman now has a python API
- It is similar to the C API and running a python script will output a rib file
- Alternatively we can render directly from within the python script
- All of the notes presented will use the Python API to generate the rib files so we have the dual advantage of learning Python and prman at the same time.

The Renderman Pipeline



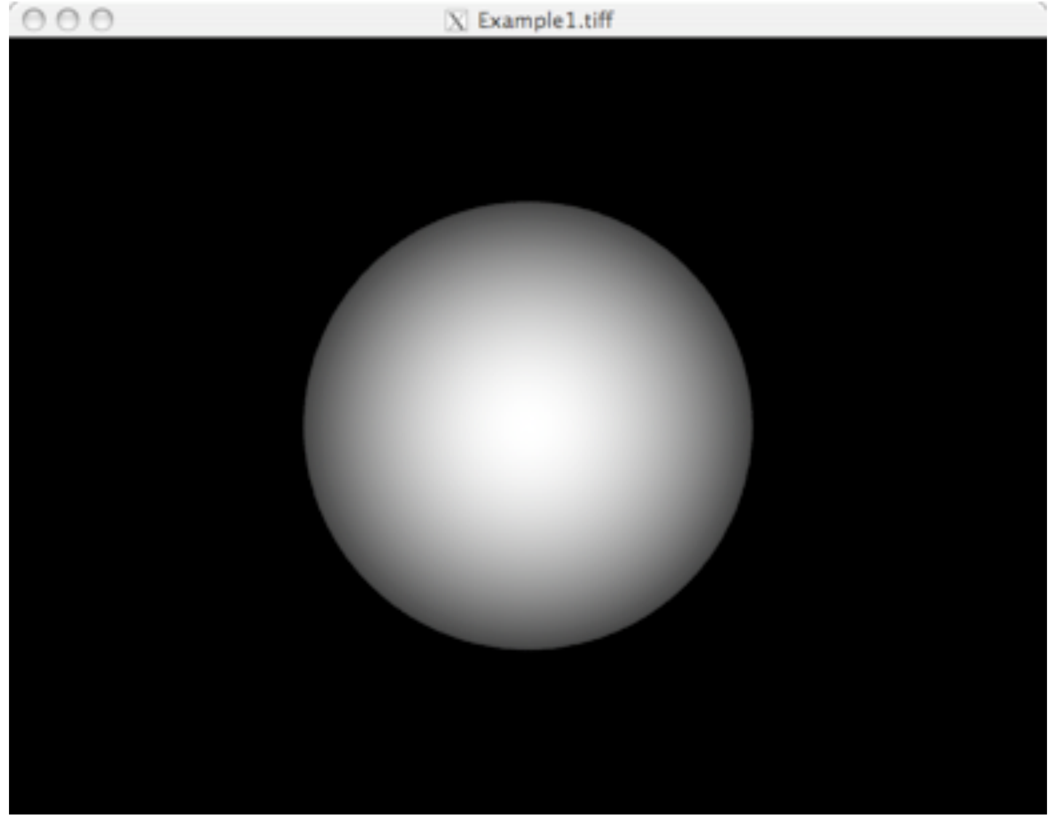
The Renderman Pipeline [1]

Hello World Renderman style

- A rib file is a simple text file, using an editor, type in the following

```
##RenderMan RIB
#Comments start with a #
#File HelloWorld.rib #
#Created by jmacey
#Creation Date: Thu Sep 25 09:43:12 2008
version 3.04
Display "HelloWorld.exr" "framebuffer" "rgba"
Format 720 575 1
Projection "perspective"
WorldBegin
  #move our world back 2 in the z so we can see it
  Translate 0 0 2
  #draw a sphere primitive
  Sphere 1 -1 1 360
  #end our world
WorldEnd
```

Rendering the file



- To render the file use the following command line

```
render -t:2 simple.rib
```

- This will render the rib file to the framebuffer (i.e. the screen)
- • -t:2 tells renderman to use both cpu's for rendering (if you have more increase the value -t:8 !
- To render to file change the Display line to the following

```
Display "HelloWorld.exr" "file" "rgba"
```

- Which will create a file called HelloWorld.exr
- We can then use the sho program to view it (`sho HelloWorld.exr`)

Python Version

- The python script to generate the rib file is a lot larger as we need to do some initial setup for the interface
- All rib commands belong to the namespace ri and are prefixed with ri
- Apart from that the function names are the same as the raw rib commands
- The following file was used to create the HelloWorld rib file

```python
#!/usr/bin/python
# for bash we need to add the following to our .bashrc
# export PYTHONPATH=$PYTHONPATH:$RMANTREE/bin
import getpass
import time
# import the python renderman library
import prman

ri = prman.Ri() # create an instance of the RenderMan interface

filename = "HelloWorld.rib"
# this is the begining of the rib archive generation we can only
# make RI calls after this function else we get a core dump
ri.Begin(filename)
# ArchiveRecord is used to add elements to the rib stream in this case comments
# note the function is overloaded so we can concatinate output
ri.ArchiveRecord(ri.COMMENT, 'Comments␣start␣with␣a␣#')
ri.ArchiveRecord(ri.COMMENT, 'File␣HelloWorld.rib␣#')
ri.ArchiveRecord(ri.COMMENT, "Created␣by␣" + getpass.getuser())
ri.ArchiveRecord(ri.COMMENT, "Creation␣Date:␣" +time.ctime(time.time()))

# now we add the display element using the usual elements
# FILENAME DISPLAY Type Output format
ri.Display("HelloWorld.exr", "framebuffer", "rgba")
# Specify PAL resolution 1:1 pixel Aspect ratio
ri.Format(720,575,1)
# now set the projection to perspective
ri.Projection(ri.PERSPECTIVE)

# now we start our world
ri.WorldBegin()
# move back 2 in the z so we can see what we are rendering
ri.ArchiveRecord(ri.COMMENT, 'move␣our␣world␣back␣2␣in␣the␣z␣so␣we␣can␣see␣it')
ri.Translate(0,0,2)
ri.ArchiveRecord(ri.COMMENT, 'draw␣a␣sphere␣primitive')
ri.Sphere (1,-1, 1, 360)
# end our world
ri.ArchiveRecord(ri.COMMENT, 'end␣our␣world')
ri.WorldEnd()
# and finally end the rib file
ri.End()
```

As you can see The rib file created from the python API has no indentation

```
##RenderMan RIB
#Comments start with a #
#File HelloWorld.rib #
#Created by jmacey
#Creation Date: Thu Sep 25 09:51:00 2008
version 3.04
Display "HelloWorld.exr" "framebuffer" "rgba"
Format 720 575 1
Projection "perspective"
WorldBegin
#move our world back 2 in the z so we can see it
Translate 0 0 2
#draw a sphere primitive
Sphere 1 -1 1 360
#end our world
WorldEnd
```

Python Path

```
1 #!/usr/bin/python
```

- The first line of the program is called a hash bang (#!)
- it is used to tell the shell where to look for the interpreter for the current script (in this case python)
- On most systems python lives in the /usr/bin/python directory so we use put this as the first line of the python script

Finding renderman

```
export PYTHONPATH=$PYTHONPATH:$RMANTREE/bin
```

- Renderman ships with a python interface to the renderman library, we need to tell the python shell where to find this interface
- The python shell uses an environment variable called PYTHONPATH to search for python libraries, when using python for renderman we must tell python to search in the $RMANTREE/bin directory for the python library
- This can be done by setting the line above for our shell (usually in .profile or .bashrc)

Basic Renderman commands

```python
import prman

ri = prman.Ri()


filename = "TeapotPY.rib"
ri.Begin(filename)

    Scene generation
    code goes here


ri.End()
```

- `import prman` — Load the prman library
- `ri = prman.Ri()` — Create an instance of the prman interface and assign it to the variable ri
- `ri.Begin(filename)` — Begin our scene called “TeapotPY.rib”
- `ri.End()` — finally finish our scene will cause the rib file to be written

direct rendering

- In the previous example a file name is passed to the ri.Begin() function
- If we wish to render the scene directly without generating the rib file we can do the following

```
filename = "__render"
ri.Begin(filename)
```

Rib layout

- When writing rib files it is best to use indentation for the different Begin/End block to make it more human readable
- This is not needed by prman but for us when de-bugging ribs
- To make the ribs easier to read we can add the following code

```
# Add Tabs to the rib output
ri.Option("rib", {"string_asciistyle": "indented"})
```

ri. Commands

- There is usually a direct correlation between the rib commands and the prman_for_python commands
- The prman_for_python commands belong to the class prman which we usually assign the prefix pi. and are now functions which may require parameters
- In some cases the commands will also require extra parameter lists which are pass using a python dictionary
- The following code show some of the commands used to initialise the display and the perspective projection

Set the display options
(file,display driver,format)

```
ri.Display("Teapot.exr", "framebuffer", "rgba")

ri.Format(720,576,1)

ri.Projection(ri.PERSPECTIVE,{"fov":[35]})
```

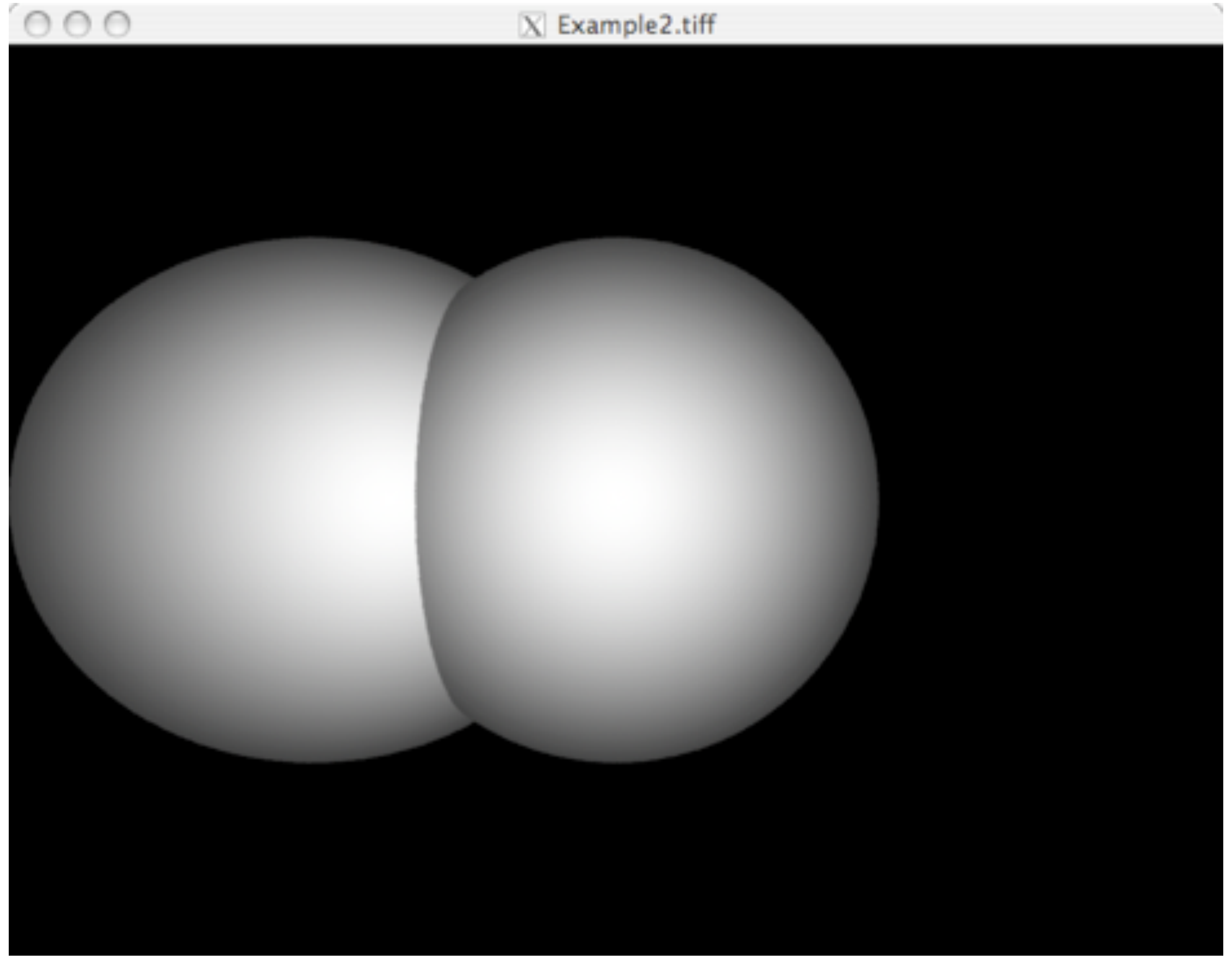
Set image
Format
(W,H, aspect)

Set projection using built in
identifier ri.PERSPECTIVE
and a dictionary for attributes

Moving Things Around

- In the first example the command Translate is used to move the object 2 in the Z.
- Renderman treats +ve Z as going into the screen (opposite to OpenGL)
- Renderman (and ribs) work with a Fixed Camera and the world must be moved to be in the correct position for the fixed camera
- This can be counter intuitive at first but you soon get used to it.

Transforms



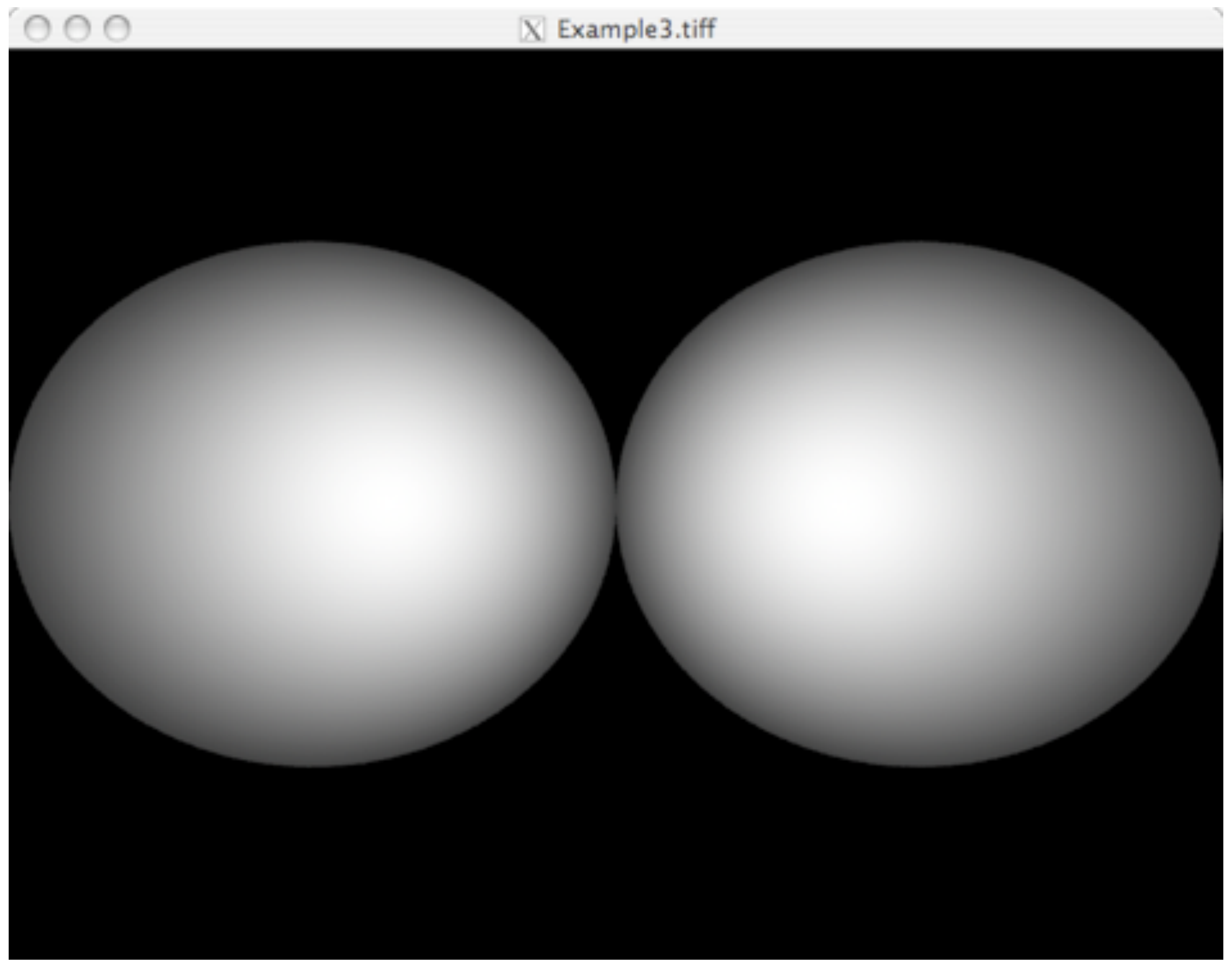

```
##RenderMan RIB
#File transform1.rib
#Created by jmacey
#Creation Date: Thu Sep 25 10:16:50 2008
version 3.04
Display "transform1.exr" "framebuffer" "rgba"
Format 720 575 1
Projection "perspective"
WorldBegin
  Translate 0 0 2
  Translate -1 0 0
  Sphere 1 -1 1 360
  Translate 1 0 0
  Sphere 1 -1 1 360
WorldEnd
```

```python
#!/usr/bin/python
# for bash we need to add the following to our .bashrc
# export PYTHONPATH=$PYTHONPATH:$RMANTREE/bin
import getpass
import time
# import the python renderman library
import prman

ri = prman.Ri() # create an instance of the RenderMan interface

filename = "transform1.rib"
# this is the begining of the rib archive generation we can only
# make RI calls after this function else we get a core dump
ri.Begin(filename)
# ArchiveRecord is used to add elements to the rib stream in this case comments
# note the function is overloaded so we can concatinate output
ri.ArchiveRecord(ri.COMMENT, 'File␣' +filename)
ri.ArchiveRecord(ri.COMMENT, "Created␣by␣" + getpass.getuser())
ri.ArchiveRecord(ri.COMMENT, "Creation␣Date:␣" +time.ctime(time.time()))

# now we add the display element using the usual elements
# FILENAME DISPLAY Type Output format
ri.Display("transform1.exr", "framebuffer", "rgba")
# Specify PAL resolution 1:1 pixel Aspect ratio
ri.Format(720,575,1)
# now set the projection to perspective
ri.Projection(ri.PERSPECTIVE)

# now we start our world
ri.WorldBegin()
# move back 2 in the z so we can see what we are rendering
ri.Translate( 0,0,2)
ri.Translate(-1,0,0)
ri.Sphere(1,-1,1,360)
ri.Translate(1,0,0)
ri.Sphere(1,-1,1,360)
ri.WorldEnd()
# and finally end the rib file
ri.End()
```

Grouping Transforms

- To group transforms we use the TransformBegin and TransformEnd commands
- These are similar to the OpenGL glPushMatrix() and glPopMatrix() and preserve the current transformation state

Grouping Transforms




```
##RenderMan RIB
#File transform2.rib
#Created by jmacey
#Creation Date: Thu Sep 25 10:22:19 2008
version 3.04
Display "transform2.exr" "framebuffer" "rgba"
Format 720 575 1
Projection "perspective"
WorldBegin
  Translate 0 0 2
  TransformBegin
    Translate -1 0 0
    Sphere 1 -1 1 360
  TransformEnd
  TransformBegin
    Translate 1 0 0
    Sphere 1 -1 1 360
  TransformEnd
WorldEnd
```

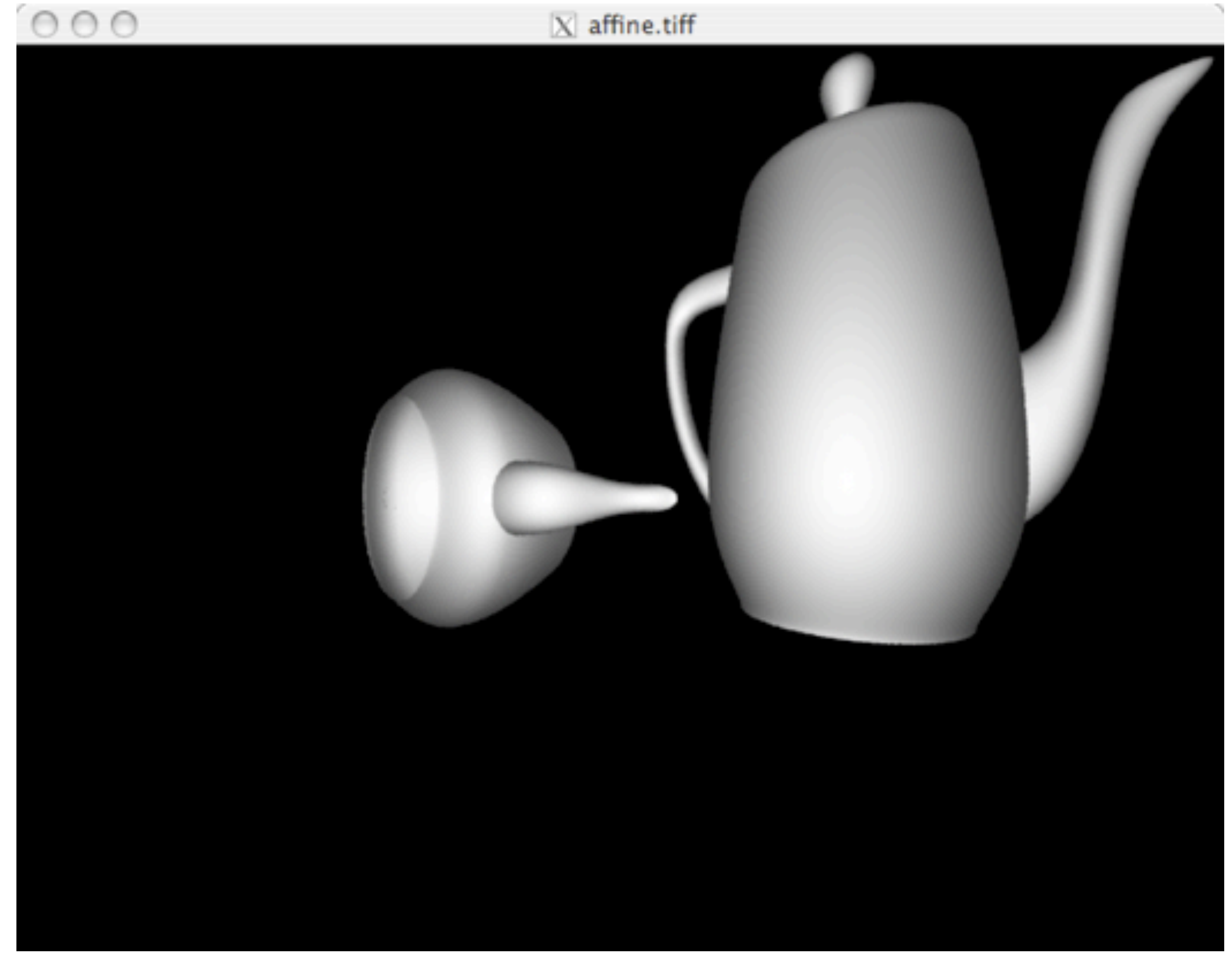
```python
#!/usr/bin/python
# for bash we need to add the following to our .bashrc
# export PYTHONPATH=$PYTHONPATH:$RMANTREE/bin
import getpass
import time
# import the python renderman library
import prman

ri = prman.Ri() # create an instance of the RenderMan interface

filename = "transform2.rib"
# this is the begining of the rib archive generation we can only
# make RI calls after this function else we get a core dump
ri.Begin(filename)
# ArchiveRecord is used to add elements to the rib stream in this case comments
# note the function is overloaded so we can concatinate output
ri.ArchiveRecord(ri.COMMENT, 'File ' +filename)
ri.ArchiveRecord(ri.COMMENT, "Created by " + getpass.getuser())
ri.ArchiveRecord(ri.COMMENT, "Creation Date: " +time.ctime(time.time()))

# now we add the display element using the usual elements
# FILENAME DISPLAY Type Output format
ri.Display("transform2.exr", "framebuffer", "rgba")
# Specify PAL resolution 1:1 pixel Aspect ratio
ri.Format(720,575,1)
# now set the projection to perspective
ri.Projection(ri.PERSPECTIVE)

# now we start our world
ri.WorldBegin()
# move back 2 in the z so we can see what we are rendering
ri.Translate( 0,0,2)
ri.TransformBegin()
ri.Translate(-1,0,0)
ri.Sphere(1,-1,1,360)
ri.TransformEnd()
ri.TransformBegin()
ri.Translate(1,0,0)
ri.Sphere(1,-1,1,360)
ri.TransformEnd()
ri.WorldEnd()
# and finally end the rib file
ri.End()
```

Other Affine Transforms

- Scale x y z : scales the current active elements in x y and z
- Rotate [angle] x y z : rotate around the axis by [angle] degrees
- Identity : restores the transformation matrix to what is was before world begin

```
# scale around the origin x,y,z

ri.Scale(1,2,1)

#rotate -90 degrees around the vector [1 0 0] (x)

ri.Rotate(-90,1,0,0)

# set the transform to the idenity matrix
# [ 1 0 0 0]
# [ 0 1 0 0]
# [ 0 0 1 0]
# [ 0 0 0 1]
ri.Identity()
```

Affine Transforms




```python
import prman

ri = prman.Ri()

filename = "affine.rib"
ri.Begin(filename)
ri.Display("affine.exr", "framebuffer", "rgba")
ri.Format(720,575,1)
ri.Projection(ri.PERSPECTIVE)

ri.WorldBegin()

ri.Translate(0,0,2)
ri.TransformBegin()
ri.Translate(-1,0,0)
ri.Scale(0.3,0.3,0.3)
ri.Rotate(45,0,1,0)
ri.Geometry("teapot")
ri.Identity()
ri.Translate(1,-0.5,2)
ri.Scale(0.3,0.8,0.3)
ri.Rotate(-90,1,0,0)
ri.Rotate(35,0,0,1)
ri.Geometry("teapot")
ri.TransformEnd()

ri.WorldEnd()
ri.End()
```

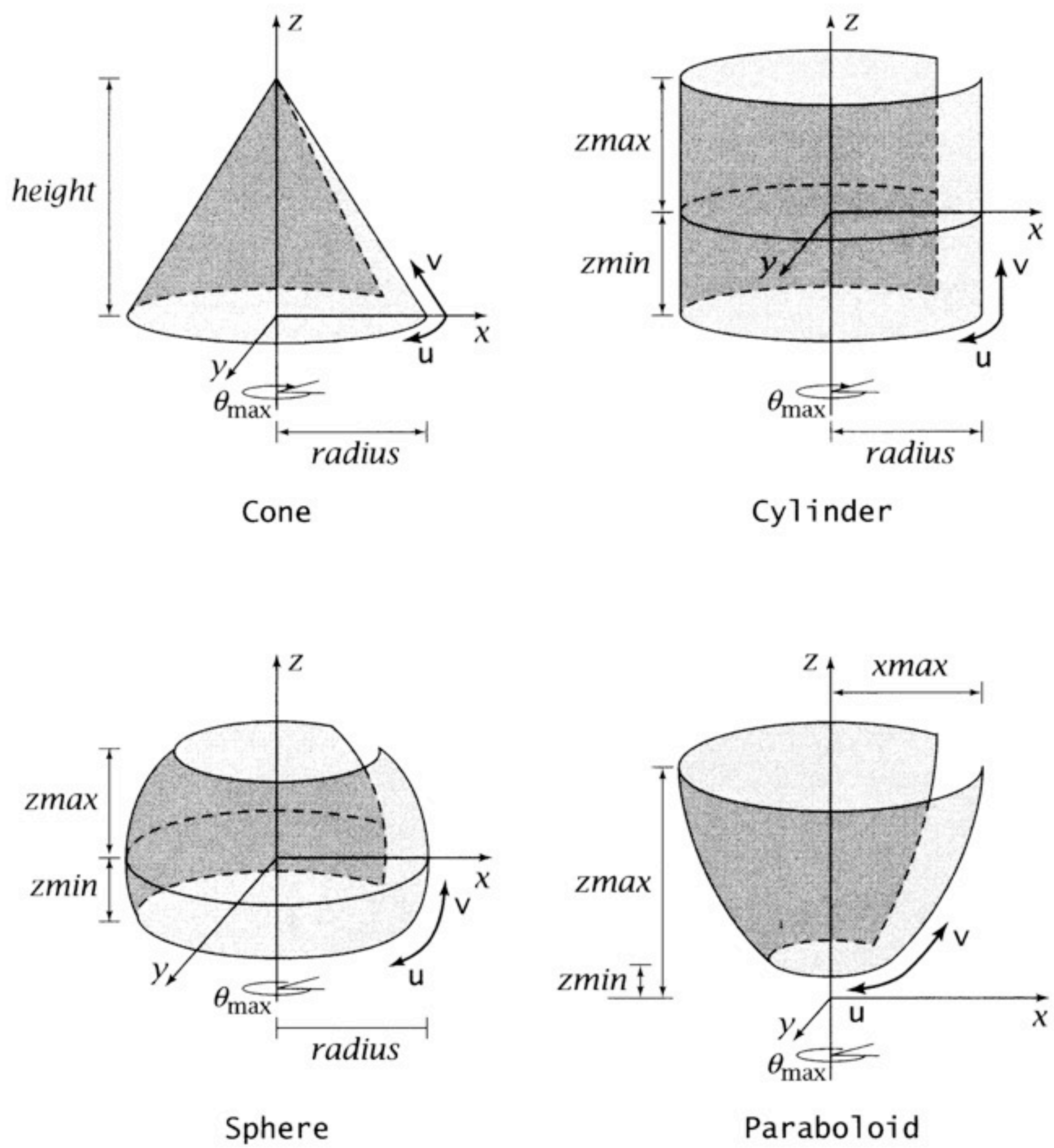
```
Display "affine.exr" "framebuffer" "rgba"
Format 720 575 1
Projection "perspective"
WorldBegin
  Translate 0 0 2
  TransformBegin
    Translate -1 0 0
    Scale 0.3 0.3 0.3
    Rotate 45 0 1 0
    Geometry "teapot"
    Identity
    Translate 1 -0.5 2
    Scale 0.3 0.8 0.3
    Rotate -90 1 0 0
    Rotate 35 0 0 1
    Geometry "teapot"
  TransformEnd
WorldEnd
```

Shape Primitives

- The rib specification has 7 parametric Quadrics commands that allows for the specification of a 7 surfaces
- These are

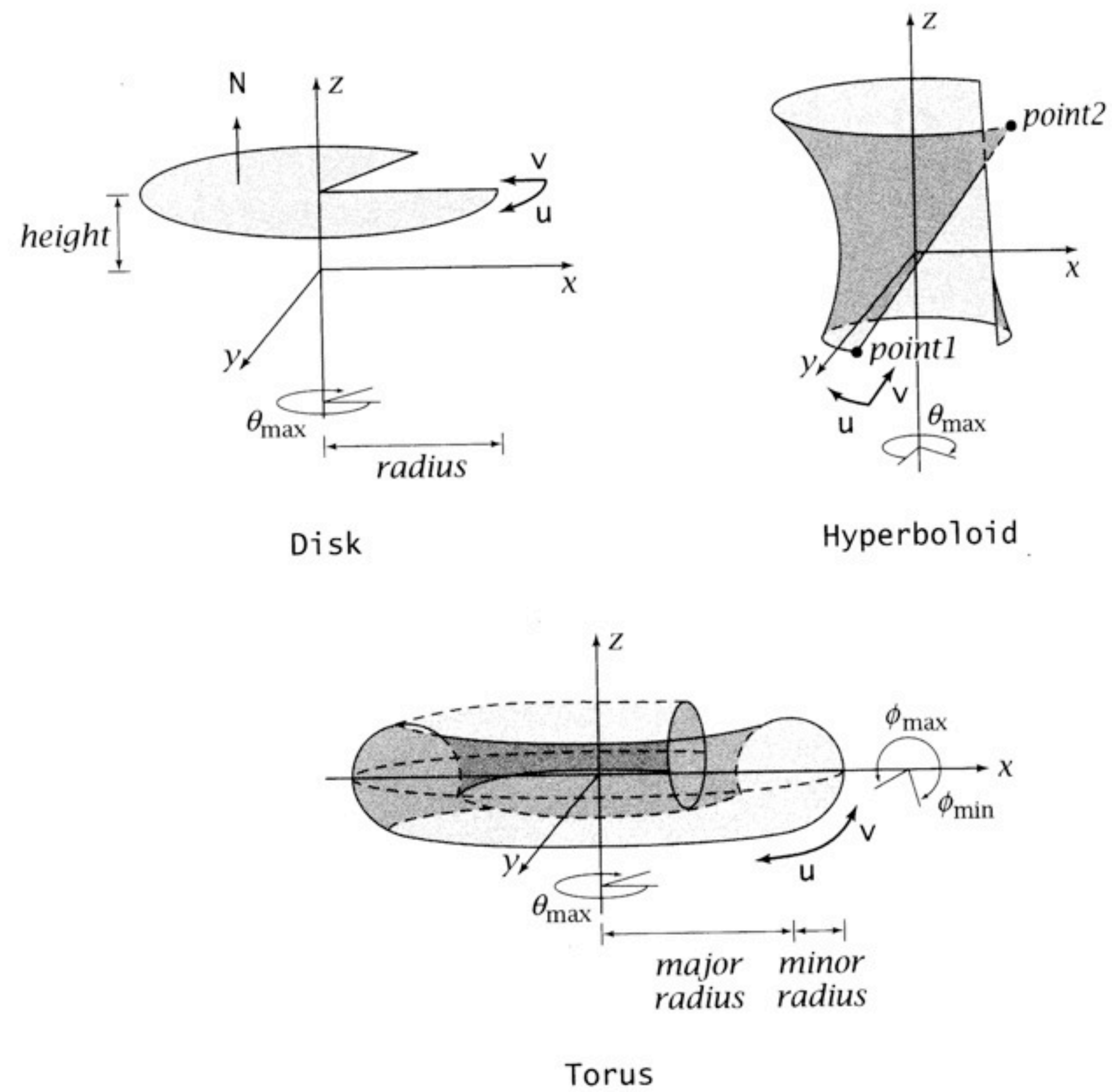
```
Sphere radius zmin zmax sweepangle

Cylinder radius zmin zmax sweepangle

Cone height radius sweepangle

Paraboloid topradius zmin zmax sweepangle

Hyperboloid point1 point2 sweepangle

Disk height radius sweepangle

Torus majorrad minorrad phimin phimax sweepangle
```

Primitives

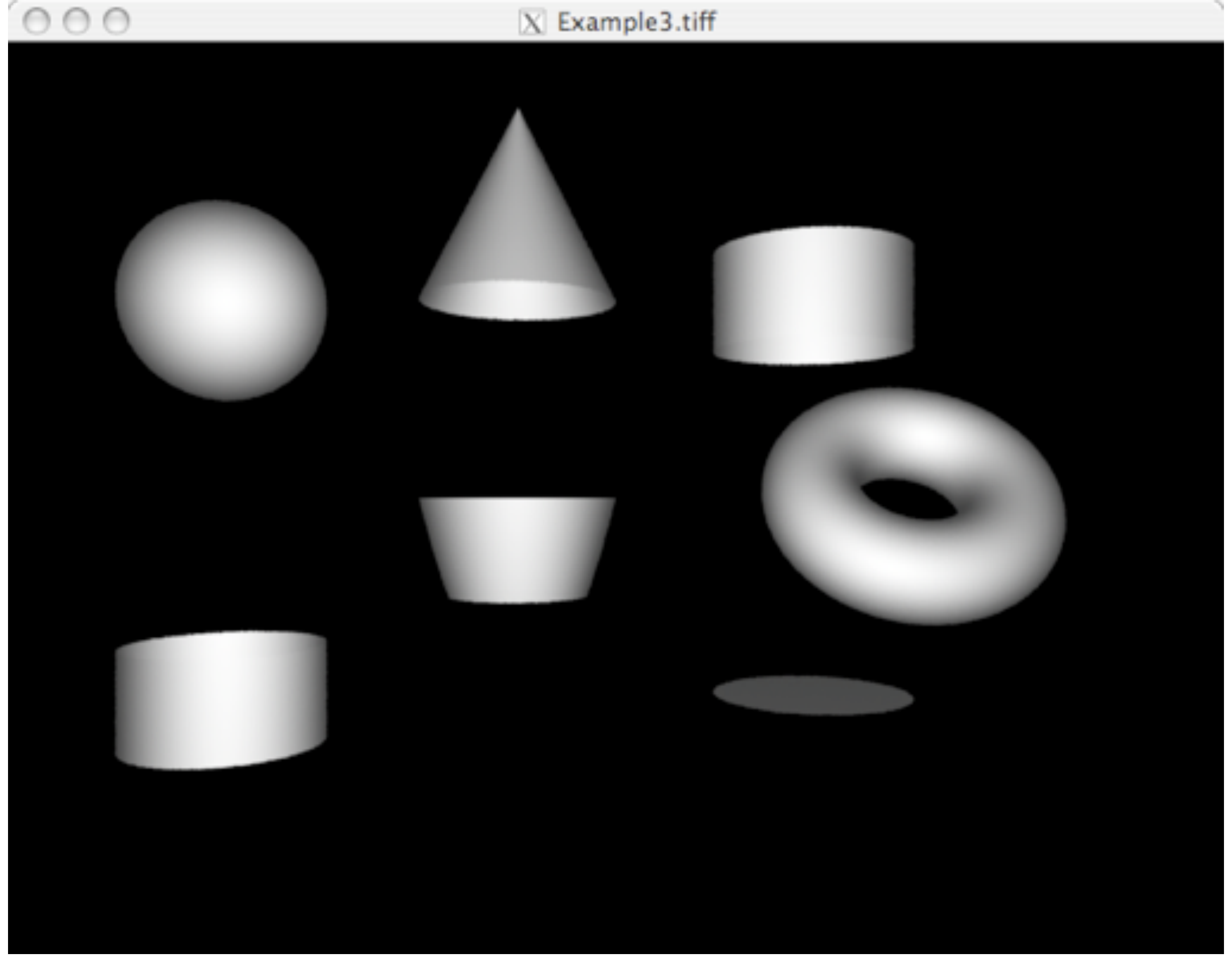


Renderman Primitives [2]

Primitives



Renderman Primitives [2]




```python
ri.WorldBegin()

ri.Translate(0,0,10)
ri.TransformBegin()
ri.Translate(-4,2,0)
ri.Sphere(1,-1,1,360)
ri.TransformEnd()
ri.TransformBegin()
ri.Translate(-4,-2,0)
ri.Rotate(90,1,0,0)
ri.Cylinder(1,-0.5,0.5,360)
ri.TransformEnd()
ri.TransformBegin()
ri.Translate(-1,2,0)
ri.Rotate(-90,1,0,0)
ri.Cone(2,1.0,360)
ri.TransformEnd()
ri.TransformBegin()
ri.Translate(-1,-2,0)
ri.Rotate(-90,1,0,0)
ri.Paraboloid(1.0,1.0,2.0,360)
ri.TransformEnd()
ri.TransformBegin()
ri.Translate(2,2,0)
ri.Rotate(-90,1,0,0)
p1=[1.0,0.0,0.5]
p2=[1.0,0.0,-0.5]
ri.Hyperboloid(p1,p2,270)
ri.TransformEnd()
ri.TransformBegin()
ri.Translate(2,-2,0)
ri.Rotate(-90,1,0,0)
ri.Disk(0,1,360)
ri.TransformEnd()
ri.TransformBegin()
ri.Translate(3 ,0,0)
ri.Rotate(45,1,0,0)
ri.Torus(1.00,0.5,0,360,360)
ri.TransformEnd()

ri.WorldEnd()
```

Note Hyperboloid takes two Points as arguments represented as lists

```
WorldBegin
  Translate 0 0 10
  TransformBegin
    Translate -4 2 0
    Sphere 1 -1 1 360
  TransformEnd
  TransformBegin
    Translate -4 -2 0
    Rotate 90 1 0 0
    Cylinder 1 -0.5 0.5 360
  TransformEnd
  TransformBegin
    Translate -1 2 0
    Rotate -90 1 0 0
    Cone 2 1 360
  TransformEnd
  TransformBegin
    Translate -1 -2 0
    Rotate -90 1 0 0
    Paraboloid 1 1 2 360
  TransformEnd
  TransformBegin
    Translate 2 2 0
    Rotate -90 1 0 0
    Hyperboloid 1 0 0.5 1 0 -0.5 270
  TransformEnd
  TransformBegin
    Translate 2 -2 0
    Rotate -90 1 0 0
    Disk 0 1 360
  TransformEnd
  TransformBegin
    Translate 3 0 0
    Rotate 45 1 0 0
    Torus 1 0.5 0 360 360
  TransformEnd
WorldEnd
```

Parameter Lists

- Each of the primitives have the ability to pass parameters to them

Name	Declared Type	Description
"P"	vertex point	Position
"Pw"	vertex hpoint	Position in homogeneous cords
"N"	varying Normal	Phong shading normals
"Cs"	varying colour	Surface Colour (overrides rib colour)
"Os"	varying colour	Surface opacity (overrides rib opacity)
"st"	varying float[2]	Texture Co-ordinates

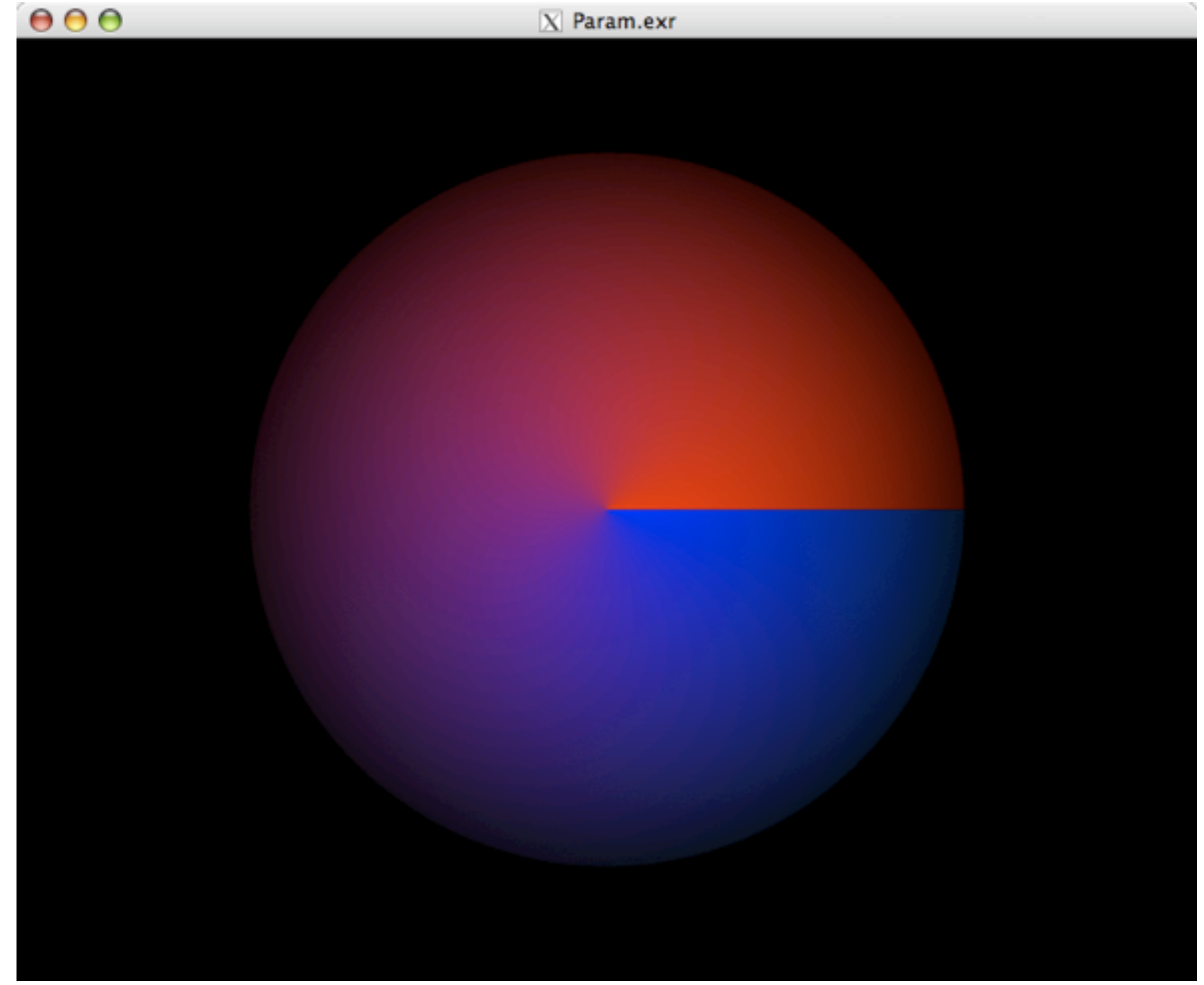
Parameter Lists

- The parameters are passed as a Python Dictionary structure.
- The format is as follows

```
dict ={key : value}

colours=[1,1,1,.5,.9,1,.2,.9,0,.5,.2,0]

ri.Sphere(1,-1,1,360,{"Cs":colours})
```

Coloured Sphere

```
##RenderMan RIB
#File Param.rib
#Created by jmacey
#Creation Date: Thu Sep 25 12:27:52 2008
version 3.04
Display "Param.exr" "framebuffer" "rgba"
Format 720 575 1
Projection "perspective" "uniform␣float␣fov" [50]
WorldBegin
  Translate 0 0 3
  TransformBegin
    Sphere 1 -1 1 360 "Cs" [1 0 0 0 0 1 1 0 0 0 1 0]
  TransformEnd
WorldEnd
```



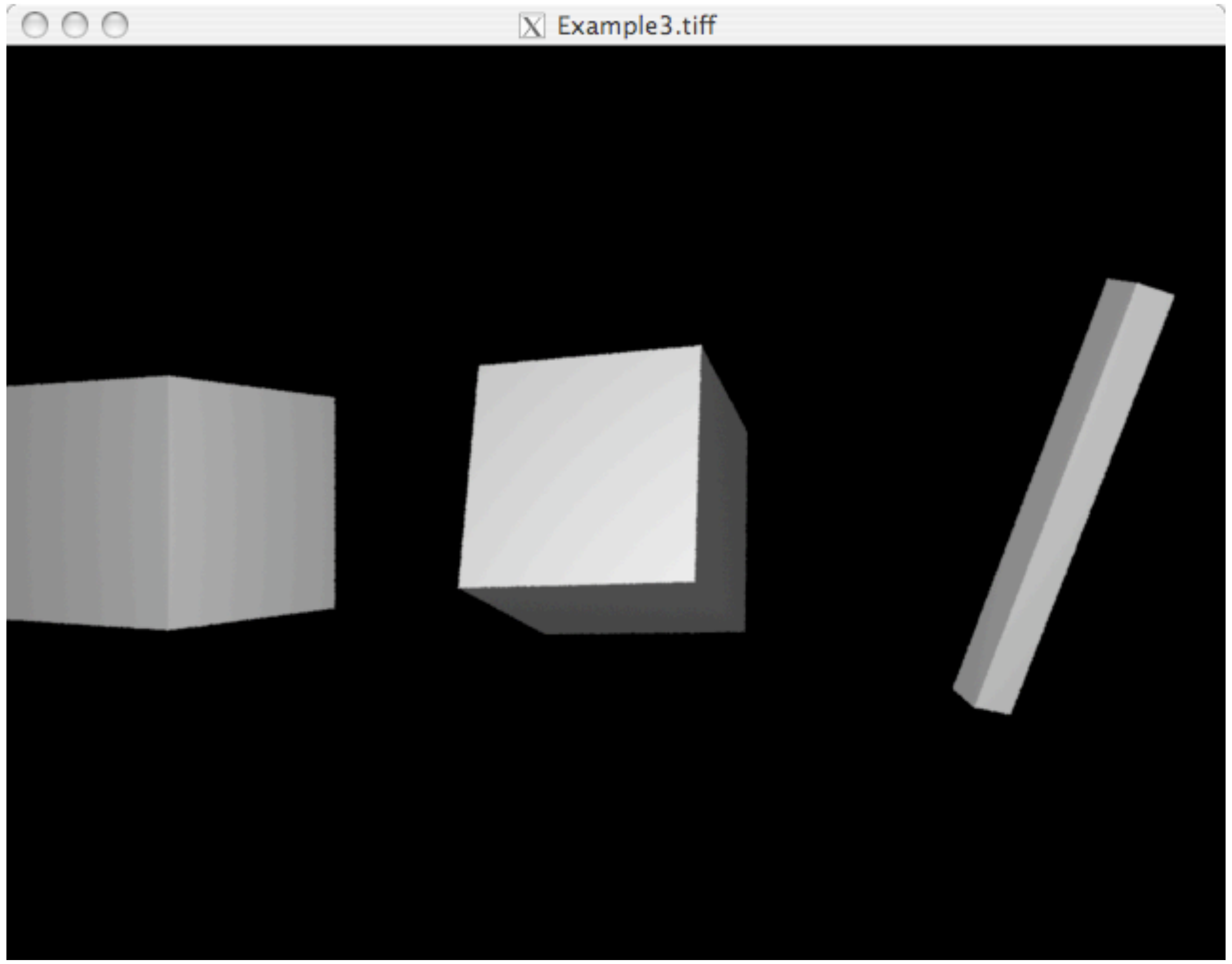

```
# now we start our world
ri.WorldBegin()

ri.Translate(0,0,3)
ri.TransformBegin()
colours=[1,0,0,0,0,1,1,0,0,0,1,0]
#ri.Rotate(90,1,1,1)
ri.Sphere(1,-1,1,360,{"Cs":colours})
ri.TransformEnd()
ri.WorldEnd()
# and finally end the rib file
ri.End()
```

What No Cube?

- PRman uses patches and we can combine them to make a cube.

```
Patch "type" [parameterlist]
```

- Define a single patch. type can be either "bilinear" or "bicubic". parameterlist is a list of token-array pairs where each token is one of the standard geometric primitive variables
- Four points define a bilinear patch, and 16 define a bicubic patch. The order of vertices for a bilinear patch is (0,0),(1,0),(0,1),(1,1).
- Note that the order of points defining a quadrilateral is different depending on whether it is a bilinear patch or a polygon.
- The vertices of a polygon would normally be in clockwise (0,0),(0,1),(1,1),(1,0) order.




```
TransformBegin
  Translate -2 0 0
  Rotate 25 0 1 0
      ObjectInstance "Cube"
  TransformEnd

TransformBegin
  Translate 0 0 0
  Rotate 25 1 1 0
      ObjectInstance "Cube"
  TransformEnd

TransformBegin
  Translate 2 0 0
  Rotate -25 1 1 1
  Scale 0.2 2.0 0.2
      ObjectInstance "Cube"
  TransformEnd
```

```
Declare "Cube" "string"

ObjectBegin "Cube"
    Patch "bilinear" "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]
    Patch "bilinear" "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]
    Patch "bilinear" "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]
    Patch "bilinear" "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]
    Patch "bilinear" "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]
    Patch "bilinear" "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]
ObjectEnd
```

Python Cube Function

- The previous example used the Object instance rib command
- This allowed us to repeat a series of rib commands.
- With python this can be replaced with a python function instead

```python
def Cube(width,height,depth) :
  w=width/2.0
  h=height/2.0
  d=depth/2.0
  ri.ArchiveRecord(ri.COMMENT, 'Cube␣Generated␣by␣Cube␣Function')
  #rear
  face=[-w,-h,d,-w,h,d,w,-h,d,w,h,d]
  ri.Patch("bilinear",{'P':face})
  #front
  face=[-w,-h,-d,-w,h,-d,w,-h,-d,w,h,-d]
  ri.Patch("bilinear",{'P':face})
  #left
  face=[-w,-h,-d,-w,h,-d,-w,-h,d,-w,h,d]
  ri.Patch("bilinear",{'P':face})
  #right
  face=[w,-h,-d,w,h,-d,w,-h,d,w,h,d]
  ri.Patch("bilinear",{'P':face})
  #bottom
  face=[w,-h,d,w,-h,-d,-w,-h,d,-w,-h,-d]
  ri.Patch("bilinear",{'P':face})
  #top
  face=[w,h,d,w,h,-d,-w,h,d,-w,h,-d]
  ri.Patch("bilinear",{'P':face})
  ri.ArchiveRecord(ri.COMMENT, '--End␣of␣Cube␣Function--')



# now we start our world
ri.WorldBegin()

ri.Translate(0,0,5)
ri.TransformBegin()
ri.Translate(-2,0,0)
ri.Rotate(25,0,1,0)
Cube(1,1,1)
ri.TransformEnd()
ri.TransformBegin()
ri.Translate( 0,0,0)
ri.Rotate( 25,1,1,0)
Cube(1,1,1)
ri.TransformEnd()
ri.TransformBegin()
ri.Translate(2,0,0)
ri.Rotate(-25,1,1,1)
Cube(0.2,2,0.2);
ri.TransformEnd()

ri.WorldEnd()
```

```
WorldBegin
  Translate 0 0 5
  TransformBegin
    Translate -2 0 0
    Rotate 25 0 1 0
    #Cube Generated by Cube Function
    Patch "bilinear" "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]
    Patch "bilinear" "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]
    Patch "bilinear" "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]
    Patch "bilinear" "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]
    Patch "bilinear" "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]
    Patch "bilinear" "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]
    #--End of Cube Function--
  TransformEnd
  TransformBegin
    Translate 0 0 0
    Rotate 25 1 1 0
    #Cube Generated by Cube Function
    Patch "bilinear" "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]
    Patch "bilinear" "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]
    Patch "bilinear" "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]
    Patch "bilinear" "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]
    Patch "bilinear" "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]
    Patch "bilinear" "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]
    #--End of Cube Function--
  TransformEnd
  TransformBegin
    Translate 2 0 0
    Rotate -25 1 1 1
    #Cube Generated by Cube Function
    Patch "bilinear" "P" [-0.1 -1 0.1 -0.1 1 0.1 0.1 -1 0.1 0.1 1 0.1]
    Patch "bilinear" "P" [-0.1 -1 -0.1 -0.1 1 -0.1 0.1 -1 -0.1 0.1 1 -0.1]
    Patch "bilinear" "P" [-0.1 -1 -0.1 -0.1 1 -0.1 -0.1 -1 0.1 -0.1 1 0.1]
    Patch "bilinear" "P" [0.1 -1 -0.1 0.1 1 -0.1 0.1 -1 0.1 0.1 1 0.1]
    Patch "bilinear" "P" [0.1 -1 0.1 0.1 -1 -0.1 -0.1 -1 0.1 -0.1 -1 -0.1]
    Patch "bilinear" "P" [0.1 1 0.1 0.1 1 -0.1 -0.1 1 0.1 -0.1 1 -0.1]
    #--End of Cube Function--
  TransformEnd
WorldEnd
```

Python Dictionaries

- Python dictionaries are a powerful key / value data structure which allows the storing of different data types in the same data set
- RenderMan's variable-length parameter list is represented in prman_for_python as a standard Python dictionary whose keys are the parameter declaration and whose values are scalars or sequences whose length is governed by the declaration and standard binding semantics

```python
#!/usr/bin/python

Dictionary={
            "red":[1.0,0.0,0.0],
            "green":[0.0,1.0,0.0],
            "blue":[0.0,0.0,1.0],
            "white":[1.0,1.0,1.0],
            "black":[0.0,0.0,0.0]
            }

print Dictionary.get("red")
print Dictionary.get("white")
print Dictionary.get("purple")
```

Create a dictionary of colour lists “key”:[r,g,b]

Use the .get(“key”) method to find the value

```
[1.0, 0.0, 0.0]
[1.0, 1.0, 1.0]
None
```

note “None” returned if “key” not found

Adding Colour

- To change the colour of a primitive we use the Color command passing in the RGB components
- For example to create a red object we use

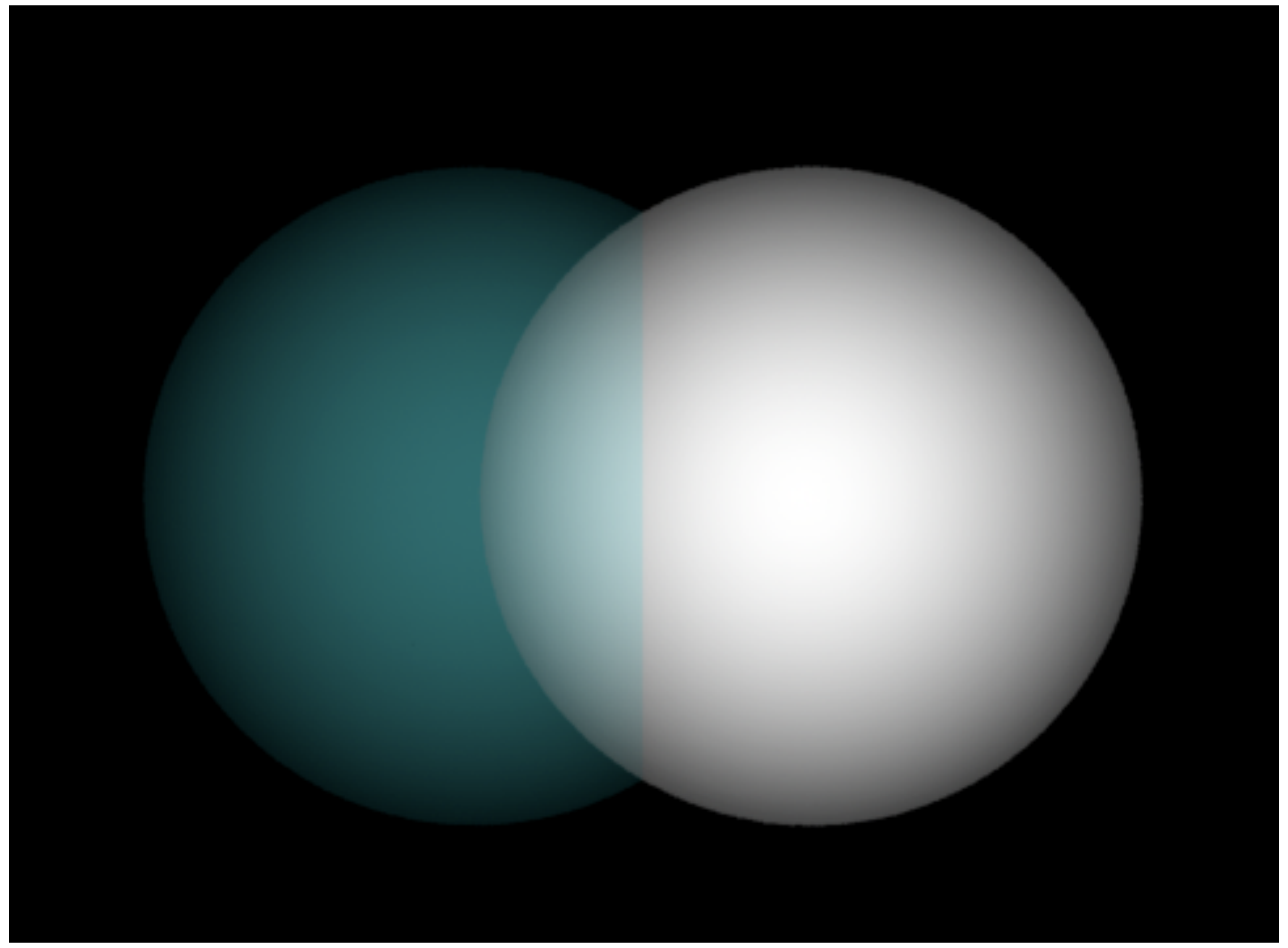
```
[RIB] Color 1 0 0 [Python] ri.Color([1,0,0])
```

- Colour is an attribute and as such will remain the currently active colour until changed.
- To group colours (or any other attributes) we use the AttributeBegin and AttributeEnd block

Attributes

```
ri.AttributeBegin()
ri.Color([1,0,0])
ri.Geometry("teapot")
ri.AttributeEnd()
```

Attributes



```
WorldBegin
  Translate 0 0 5
  Color [1 1 1]
  Opacity [1 1 1]
  TransformBegin
    Translate -0.5 0 0
    AttributeBegin
      Color [0 1 1]
      Opacity [0.2 0.2 0.2]
      Sphere 1 -1 1 360
    AttributeEnd
  TransformEnd
  TransformBegin
    Translate 0.5 0 0
    Sphere 1 -1 1 360
  TransformEnd
WorldEnd
```

```
# now we start our world
ri.WorldBegin()

ri.Translate(0,0,5)
ri.Color([1,1,1])
ri.Opacity([1,1,1])
ri.TransformBegin()
ri.Translate(-0.5,0,0)
ri.AttributeBegin()
ri.Color([0,1,1])
ri.Opacity([0.2,0.2,0.2])
ri.Sphere(1,-1,1,360)
ri.AttributeEnd()
ri.TransformEnd()
ri.TransformBegin()
ri.Translate(0.5,0,0)
ri.Sphere(1,-1,1,360)
ri.TransformEnd()

ri.WorldEnd()
```

Rib file Structure Conventions

- Following is a structured list of components for a conforming RIB file that diagrams the "proper" use of RIB.
- Some of the components are optional and will depend greatly on the resource requirements of a given scene.
- Indentation indicates the scope of the following command.

```
Preamble and global variable declarations (RIB requests:version,declare)

Static options and default attributes (image and display options,camera options)

Static camera transformations (camera location and orientation)

Frame block (if more than one frame)

    Frame-specific variable declarations

    Variable options and default attributes

    Variable camera transforms

    World block

      (scene description)
      User Entity (enclosed within AttributeBegin/AttributeEnd)
      User Entity (enclosed within AttributeBegin/AttributeEnd)
      User Entity
more frame blocks
```

Rib file Structure

- This structure results from the vigourous application of the following Scoping Conventions:
- No attribute inheritance should be assumed unless implicit in the definition of the User Entity (i.e., within a hierarchy).
- No attribute should be exported except to establish either global or local defaults.
- The RenderMan Specification provides block structuring to organize the components of a RIB file.
- Although the use of blocks is only required for frame and world constructs by the Specification, the liberal use of attribute and transform blocks is encouraged.

Attributes

- Attributes are flags and values that are part of the graphics state, and are therefore associated with individual primitives.
- The values of these attributes are pushed and popped with the graphics state.
- This is done with the AttributeBegin and AttributeEnd commands
- The attribute block is the fundamental block for encapsulating user entities.

Attributes II

- Within an attribute block, the structure is simple. All attribute settings should follow immediately after the AttributeBegin request.
- Geometric transformations are considered attributes in the RenderMan Interface and should also precede any geometry.
- Depending on the internal architecture of the modeling software, user entities may be described around a local origin. In this case, a modeling transformation commonly transforms the entity from object space to world space.
- If this is not the case, the modeler will probably be working entirely in world space and no modeling transform will be present.
- After setting all of the attributes for the entity, the geometry should immediately follow

Shading Rate

- This is probably the second most critical factor in the speed performance of RenderMan (exceeded only by the resolution).
- This is due to two factors.
- First, it governs how often the shading language interpreter runs. Smaller numbers mean the shaders must be evaluated at more places on the surface of the primitives.
- Second, it governs how many polygons (micropolygons) are passed through the hidden-surface algorithm.
- Smaller numbers mean more micropolygons, requiring more hidden-surface evaluation and more memory to store temporary results.
- The end result of all this is that doubling the Shading Rate usually gets you nearly twice the rendering speed. Pretty good!

Shading Rate II

- The default for shading rate is 0.25, which is much smaller than is necessary for most images.
- A much more typical number for final rendering is 1.0 to 4.0, and test renderings can usually be done at 16.0 or even larger.
- What is the disadvantage?
- A shading rate that is too large tends to give blocky looking colours and excessive blur on textures.
- The blockiness can often be alleviated by turning on Gouraud shading with the ShadingInterpolation "smooth" call.

Shading Rate III

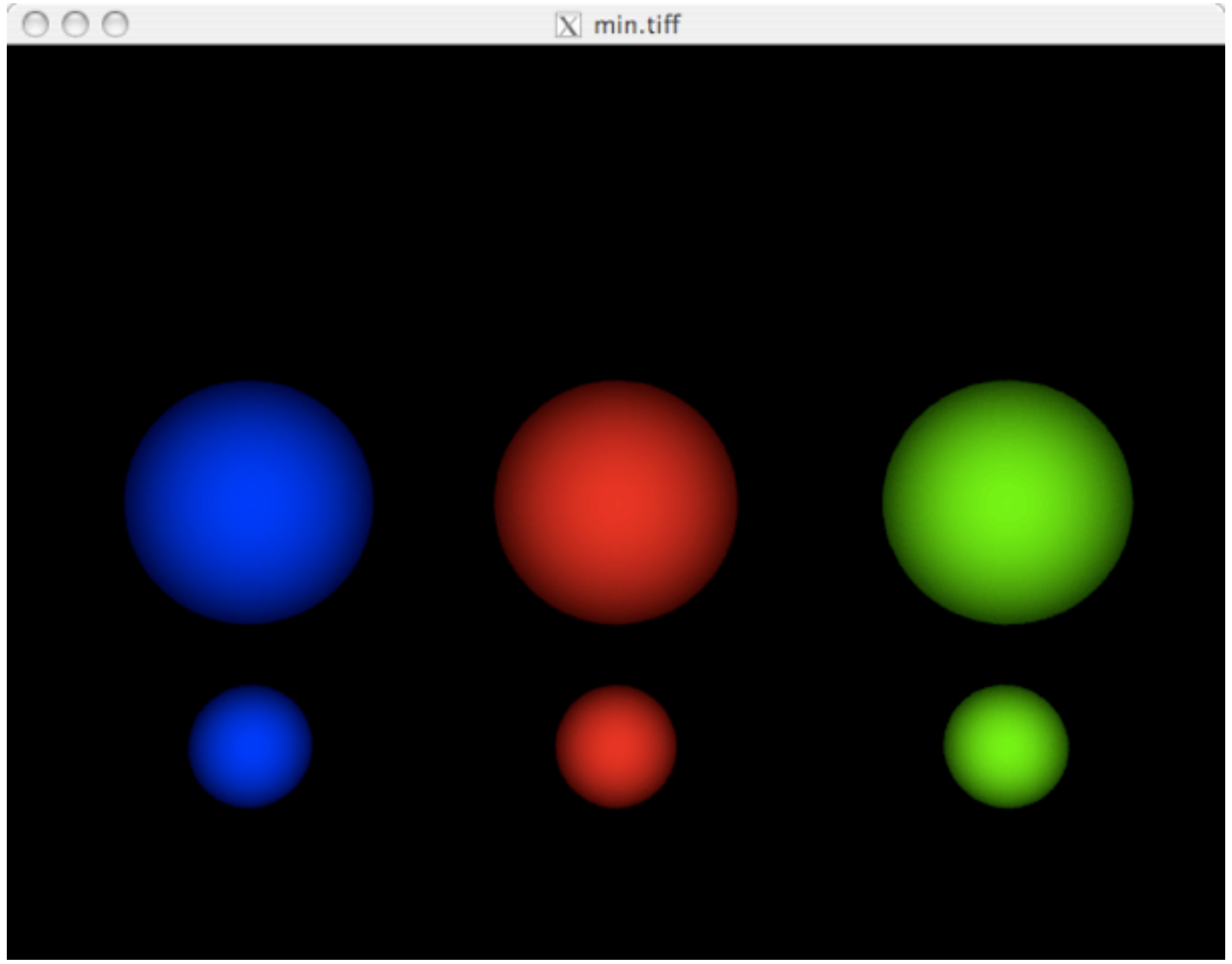
- As long as the colour of an object changes slowly and smoothly across its surface, this will look fine.
- Only if the surface has drastic colour changes, such as sharp-edged patterns in its textures, will these results be unsatisfactory.
- And an object with a shading rate of 16.0 and Gouraud shading will render much faster than an object with a shading rate of 1.0.

Shading Rate III

- One of the most important things to remember about the Shading Rate and Shading Interpolation values is that they are Attributes.
- That is, they can be changed from one primitive to the next.
- So, if you have a finely patterned vase sitting in a room with flat white walls, the vase can have a small shading rate (like 1.0) to capture its detail
- while the walls can have a very large shading rate (like 64.0) to save time (with no visible problems).
- This is a very powerful technique that amounts to telling the renderer which objects to spend time getting right and which objects are boring and can be handled simply.

Objects

- A single geometric primitive or a list of geometric primitives may be retained by enclosing them with ObjectBegin and ObjectEnd.
- The RenderMan Interface allocates and re-turns an ObjectHandle for each retained object defined in this way.
- This handle can subsequently be used to reference the object when creating instances with ObjectInstance.
- Objects are not rendered when they are defined within an ObjectBegin-ObjectEnd block; only an internal definition is created.
- Transformations, and even Motion blocks, may be used inside an Object block, though they obviously imply a relative transformation to the coordinate system active when the Object is instanced.
- All of an object's attributes are inherited at the time it is instanced, not at the time at which it is created.




```
#declare a string so we can refer to the Object by name
Declare "Spheres" "string"
# Now we actually create  the Object

ObjectBegin "Spheres"
  Sphere 1 -1 1 360
  Translate 0 0 2
  Scale 0.5 0.5 0.5
  Sphere 1 -1 1 360
ObjectEnd

Display "min.tiff" "framebuffer" "rgba"
Projection "perspective" "fov" [30]

# start our world
WorldBegin
  Translate 0 0 14 #move the global view position
  Rotate 90 1 0 0
  Color [1 0 0]
  Attribute "identifier" "name" ["Spheres1"]
  ObjectInstance "Spheres"
  Color [0 1 0]
  Translate 3.2 0 0
  Attribute "identifier" "name" ["Spheres2"]
  ObjectInstance "Spheres"
  Color [0 0 1]
  Translate -6.2 0 0
  Attribute "identifier" "name" ["Spheres3"]
  ObjectInstance "Spheres"
#end our world
WorldEnd
```

Named Primitives

- It is occasionally useful to give names to individual primitives. For example, when a primitive won't split at the eye plane (see Section 4.8 prman docs) it can be desirable to know which primitive is causing the problem. This can be done using the attribute identifier with the parameter name, as in:

```
RtString name[1] = {"Gigi"};
RiAttribute("identifier","name",(RtPointer)name,RI_NULL);

or

Attribute "identifier" "name" ["Spheres3"]
```

- All defined primitives will have this name until the graphics stack is popped (with RiAttributeEnd) or another such RiAttribute call is made.

- The error message would then contain a reference to a specific primitive name instead of the mysterious <unnamed>.

Python ObjectBegin / End

- At present there is a bug in the python version of ObjectInstance which does not allow rib file generation
- However it will work in direct mode where the rib stream is fed directly into the renderer
- To do this we use the following

```python
# if we use __render as the file name we go to
# immediate mode and the rib stream is passed directly to
# the renderer.
# if we specify framebuffer in the Dispalay option we render to screen
# if we specify file we render to file
filename = "__render"

ri.Begin(filename)
```

```python
ri = prman.Ri() # create an instance of the RenderMan interface
ri.Option("rib", {"string_asciistyle": "indented"})

filename = "__render"
ri.Begin(filename)


#declare a string so we can refer to the Object by name
ri.Declare( "Spheres" ,"string")
# Now we actually create  the Object
ObjHandle=ri.ObjectBegin()
print ObjHandle
ri.Sphere(1,-1,1,360)
ri.Translate(0,0,2)
ri.Scale(0.5,0.5,0.5)
ri.Sphere(1,-1,1,360)
ri.ObjectEnd()


# start our world
ri.WorldBegin()
ri.Translate(0,0,14) #move the global view position
ri.Rotate(90,1,0,0)
ri.Color(colours["red"])
ri.Attribute ("identifier",{"name": "Spheres1"})
ri.ObjectInstance(ObjHandle)
ri.Color(colours["green"])
ri.Translate(3.2,0,0)
ri.Attribute( "identifier",{ "name" :"Spheres2"})
ri.ObjectInstance(ObjHandle)
ri.Color(colours["blue"])
ri.Translate(-6.2,0,0)
ri.Attribute("identifier",{ "name" : "Spheres3"})
ri.ObjectInstance("%s"%(ObjHandle))
ri.ArchiveRecord("ribfile", "ObjectInstance " +ObjHandle)

#end our world

ri.WorldEnd()
# and finally end the rib file
ri.End()
```

ObjectBegin returns a handle
This is generated by prman and is unique each time :
8a5644f8-8bae-11dd-9428-001b639ea4ff

We then use the Object Handle in the instance call

Options

- Options are parameters that affect the rendering of an entire image.
- They must be set before calling WorldBegin, since at that point options for a specific frame are frozen.
- The PRMan Quick Reference includes a table that summarizes summarizes the options available in PhotoRealistic RenderMan.
- Note that some of the defaults listed can be overridden by configuration files.

Frame Buffer Control

- There are several options which can be enabled through the parameter list of the RiDisplay call. These options, naturally enough, influence the use of the display device.
- Output Compression
  - The TIFF driver also accepts an option to set the compression type, which may be "lzw", "packbits", "zip" (the default), "pixarlog", or "none":

```
Display "min.tiff" "TIFF" "rgba" "compression" "lzw"
```

OpenEXR Display Driver

- This driver supports OpenEXR, a high dynamic-range image, floating point file format developed by Industrial Light & Magic.
- When using this display driver for rgba or Z output, you should turn rgba and Z quantization off by using a floating point Quantize statement, ie:

```
Quantize "rgba" 0 0 0 0
Quantize "z"    0 0 0 0

ri.Quantize("rgba",0,0,0,0)
ri.Quantize("z",0,0,0,0)
```

OpenEXR Driver

- This display driver also supports the output of image channels other than rgba using the Arbitrary Output Variable mechanisms.
- This driver maps Renderman's output variables to image channels as follows:

output variable name	image channel name	type
"r"	"R"	preferred type
"g"	"G"	preferred type
"b"	"B"	preferred type
"a"	"A"	preferred type
"z"	"Z"	FLOAT
other	same as output variable name	preferred type

Setting Display Parameters

- By default, the "preferred" channel type is the value float (32-bit).
- The preferred type can be changed by adding an "exrpixeltype" or "type" argument to the Display command in the RIB file.

```
# Store point positions in HALF format
Display "Points.exr" "openexr" "P" "string␣exrpixeltype" "half"
ri.Display("Points.exr", "openexr", "P" ,{"string␣exrpixeltype" :"half"})
```

- Compression defaults to "zip"
- You can select a different compression method by adding an "exrcompression" argument or simply the "compression" argument to the Display command.

```
# Store RGBA using run-length encoding
Display "rle.rgba.exr" "openexr" "rgba" "string␣exrcompression" "rle"
ri.Display("rle.rgba.exr", "openexr", "rgba" ,{"string␣exrcompression" :"rle"})
```

Search Paths

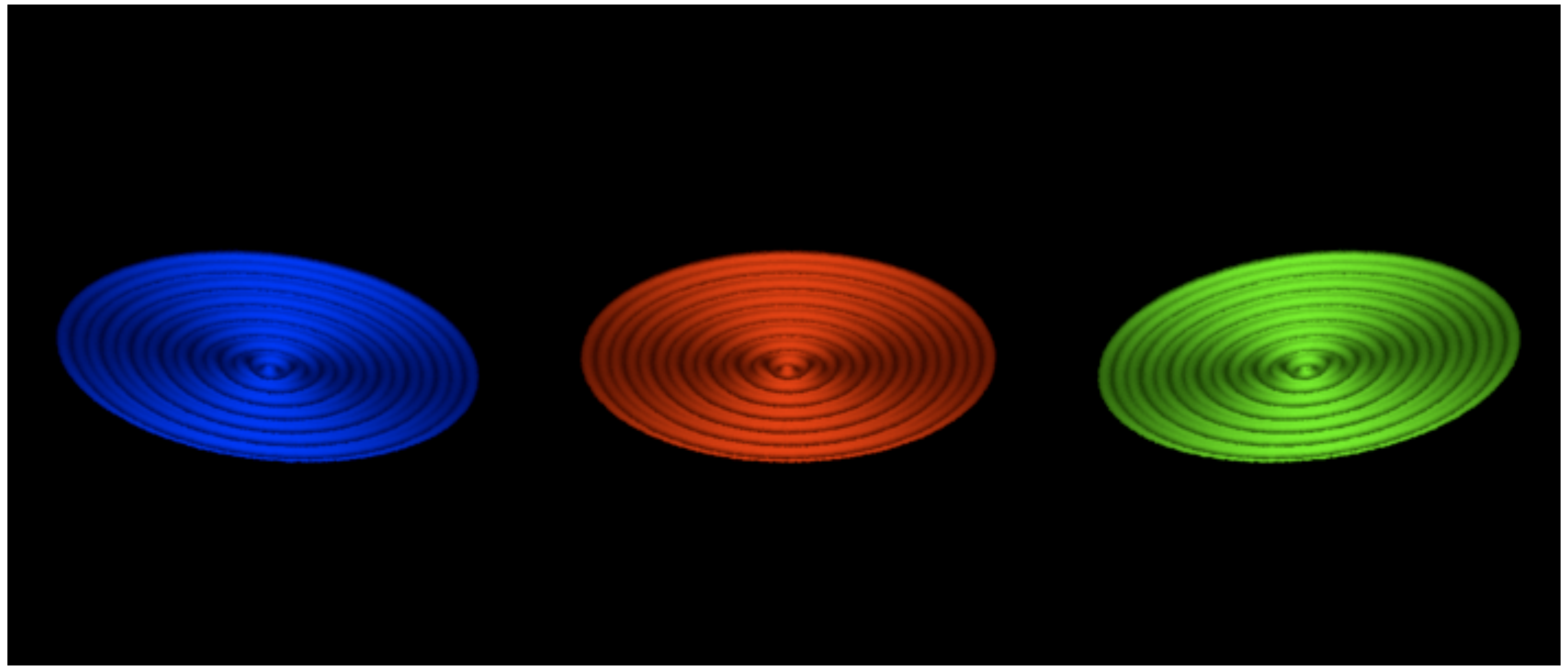
- PhotoRealistic RenderMan searches specific paths for shader definitions, texture map files and Pixar Looks® masters and instances.
- The search path is a colon-separated list of directories that are used in searching for files.
- Example

```
Option "searchpath" "string␣shader" ["/mapublic/shaders"]

ri.Option("searchpath", {"string␣shader":"/mapublic/shaders"})
```

Search Paths

- The valid search paths are:
  - shader :- Used by the renderer to find all shader .slo files.
  - texture :- Used by the renderer to find all texture files.
  - archive :- Used by the renderer to find RIB archives.
  - procedural :- Used by the renderer to find procedural primitive DSOS.
  - display :- Used by the renderer to find display drivers.

ReadArchive



- The ReadArchive command allows us to read another rib file into the current position of the RIB stream

```
ri.Begin(filename)
ri.Option("searchpath", {"string␣archive":"./Archive/"})

ri.Attribute ("identifier",{"name": "Wave1"})
ri.ReadArchive("Archive.rib")


Option "searchpath" "string␣archive" ["./Archive/"]

Attribute "identifier" "name" ["Wave1"]
ReadArchive "Archive.rib"
```

- Archives may also be specified within the current rib file using the following

```
ri.Begin(filename)

ri.ArchiveBegin("Wave")
ri.Rotate(90,1,0,0)
ri.Sphere(0.030303,-0.030303,0,360)
ri.Torus(0.0606061,0.030303,0,180,360)
ri.Torus(0.121212,0.030303,180,360,360)
ri.Torus(0.181818,0.030303,0,180,360)
ri.Torus(0.242424,0.030303,180,360,360)
ri.Torus(0.30303,0.030303,0,180,360)
ri.Torus(0.363636,0.030303,180,360,360)
ri.Torus(0.424242,0.030303,0,180,360)
ri.Torus(0.484848,0.030303,180,360,360)
ri.Torus(0.545455,0.030303,0,180,360)
ri.Torus(0.606061,0.030303,180,360,360)
ri.Torus(0.666667,0.030303,0,180,360)
ri.Torus(0.727273,0.030303,180,360,360)
ri.Torus(0.787879,0.030303,0,180,360)
ri.Torus(0.848485,0.030303,180,360,360)
ri.ArchiveEnd()

ri.Attribute ("identifier",{"name": "Wave1"})
ri.ReadArchive("Wave")
```

```
ArchiveBegin "Wave"
        Rotate 90 1 0 0
        Sphere 0.030303 -0.030303 0 360
        Torus 0.0606061 0.030303 0 180 360
        Torus 0.121212 0.030303 180 360 360
        Torus 0.181818 0.030303 0 180 360
        Torus 0.242424 0.030303 180 360 360
        Torus 0.30303 0.030303 0 180 360
        Torus 0.363636 0.030303 180 360 360
        Torus 0.424242 0.030303 0 180 360
        Torus 0.484848 0.030303 180 360 360
        Torus 0.545455 0.030303 0 180 360
        Torus 0.606061 0.030303 180 360 360
        Torus 0.666667 0.030303 0 180 360
        Torus 0.727273 0.030303 180 360 360
        Torus 0.787879 0.030303 0 180 360
        Torus 0.848485 0.030303 180 360 360
ArchiveEnd

WorldBegin
Attribute "identifier" "name" ["Wave1"]
ReadArchive "Wave"
WorldEnd
```

Procedural Geometry

- The torus wave in the last examples was generated from an example in the renderman companion
- The function was re-written from the original C into python as shown below

```
def TorusWave(ri,nwaves,thetamax) :
  if(nwaves < 1) :
    print "need positive number of waves"
    return
  innerrad = 2.0/(8.0 * nwaves +2)
  ri.Rotate(90.0,1.0,0.0,0.0)
  ri.Sphere(innerrad,-innerrad,0,thetamax)
  outerrad =0.0
  for wave in range(1,nwaves) :
    outerrad=outerrad+(innerrad*2)
    ri.Torus(outerrad,innerrad,0.0,180.0,thetamax)
    outerrad=outerrad+(innerrad*2)
    ri.Torus(outerrad,innerrad,180.0,360.0,thetamax)

```

```
ri = prman.Ri()

filename = "Archive.rib"
ri.Begin(filename)

TorusWave(ri,8,360.0)
# and finally end the rib file
ri.End()
```

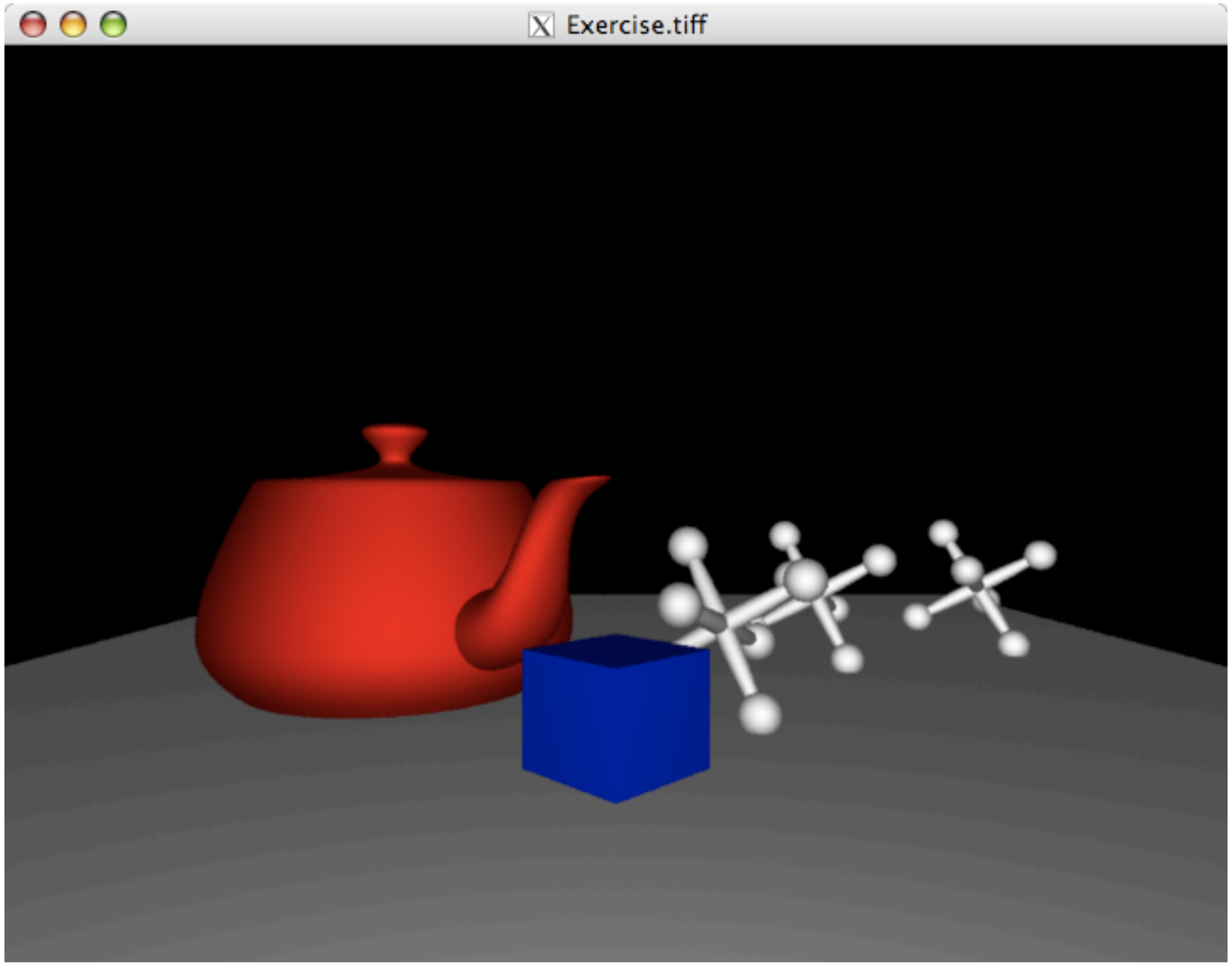
Creating Sequences

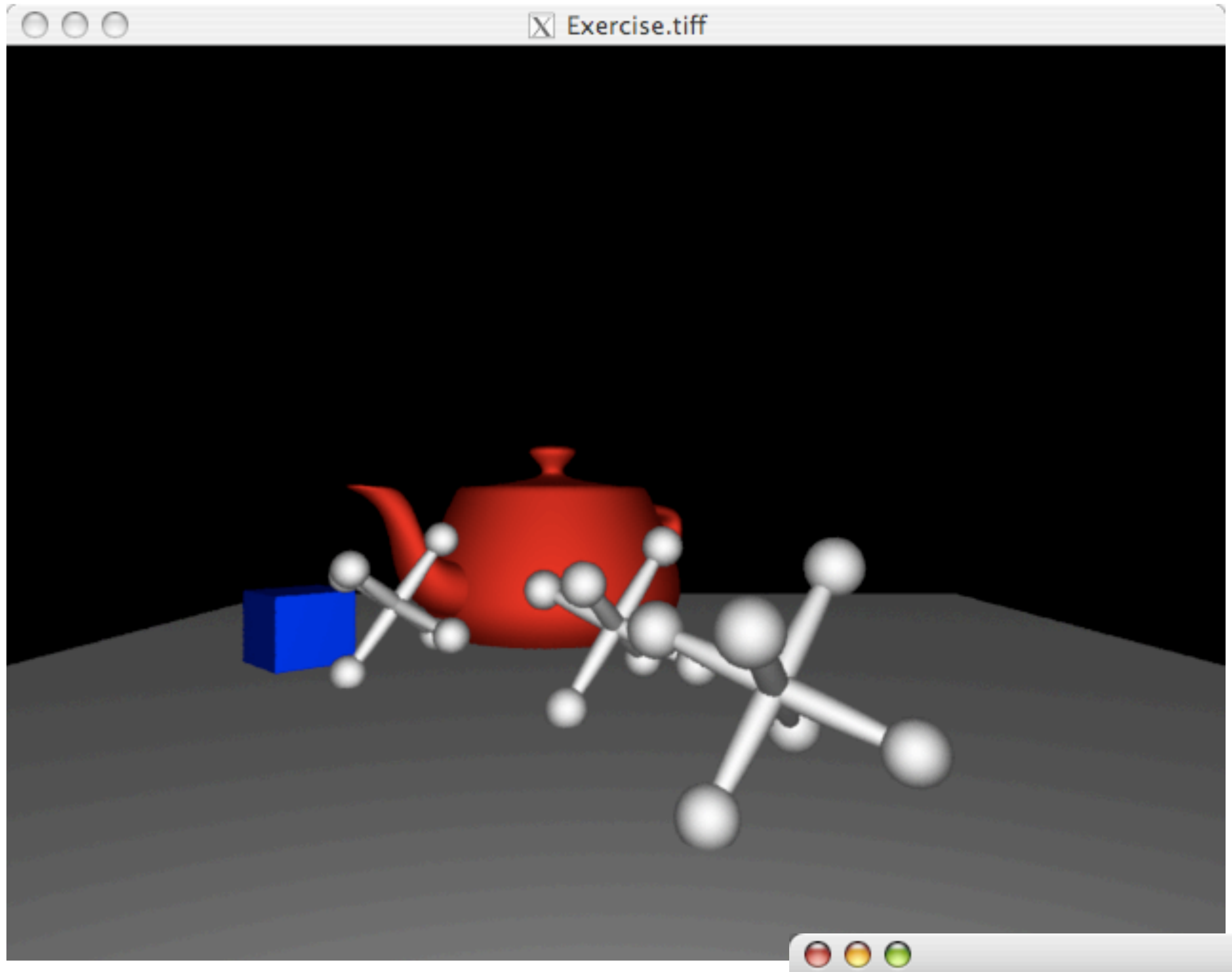
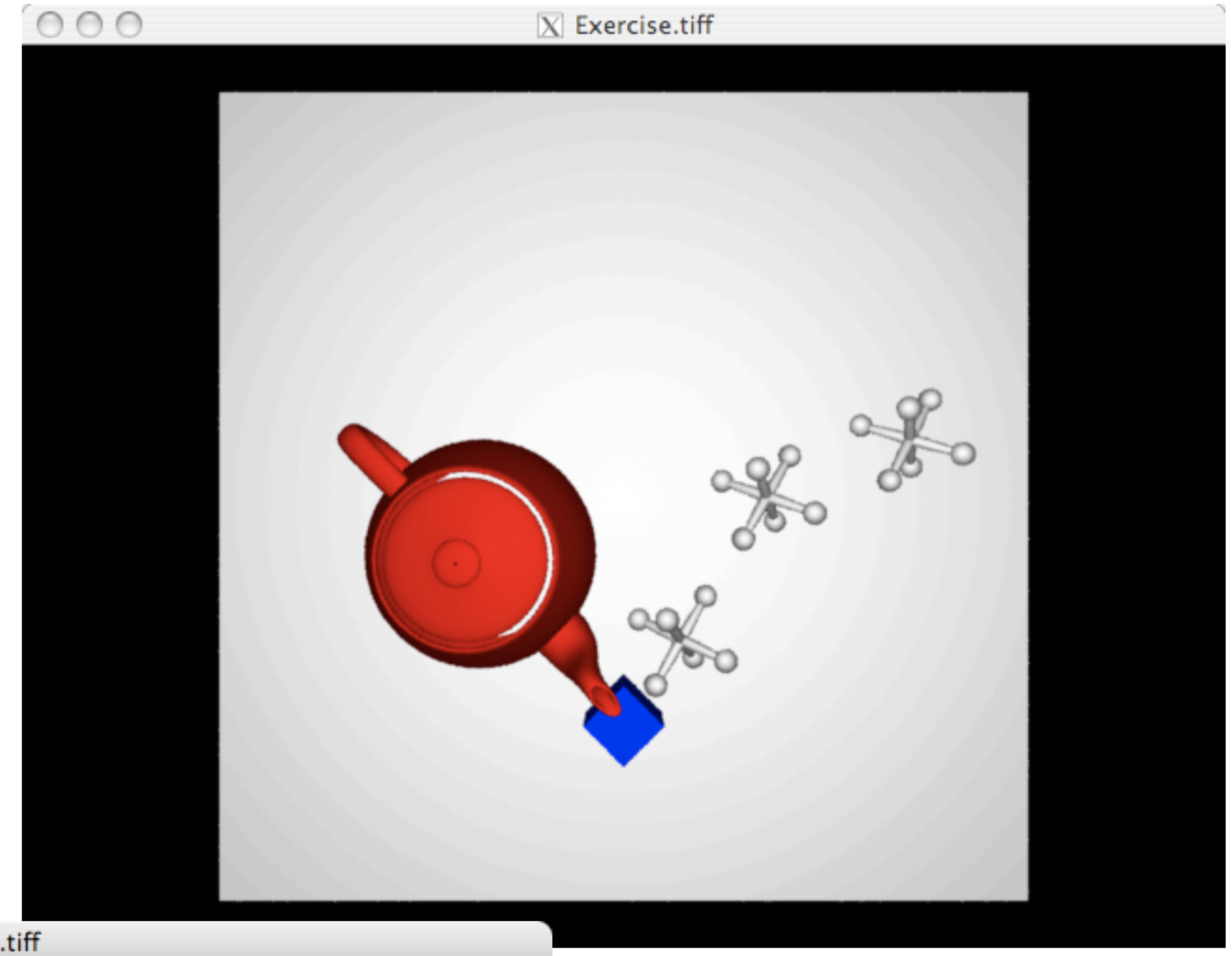
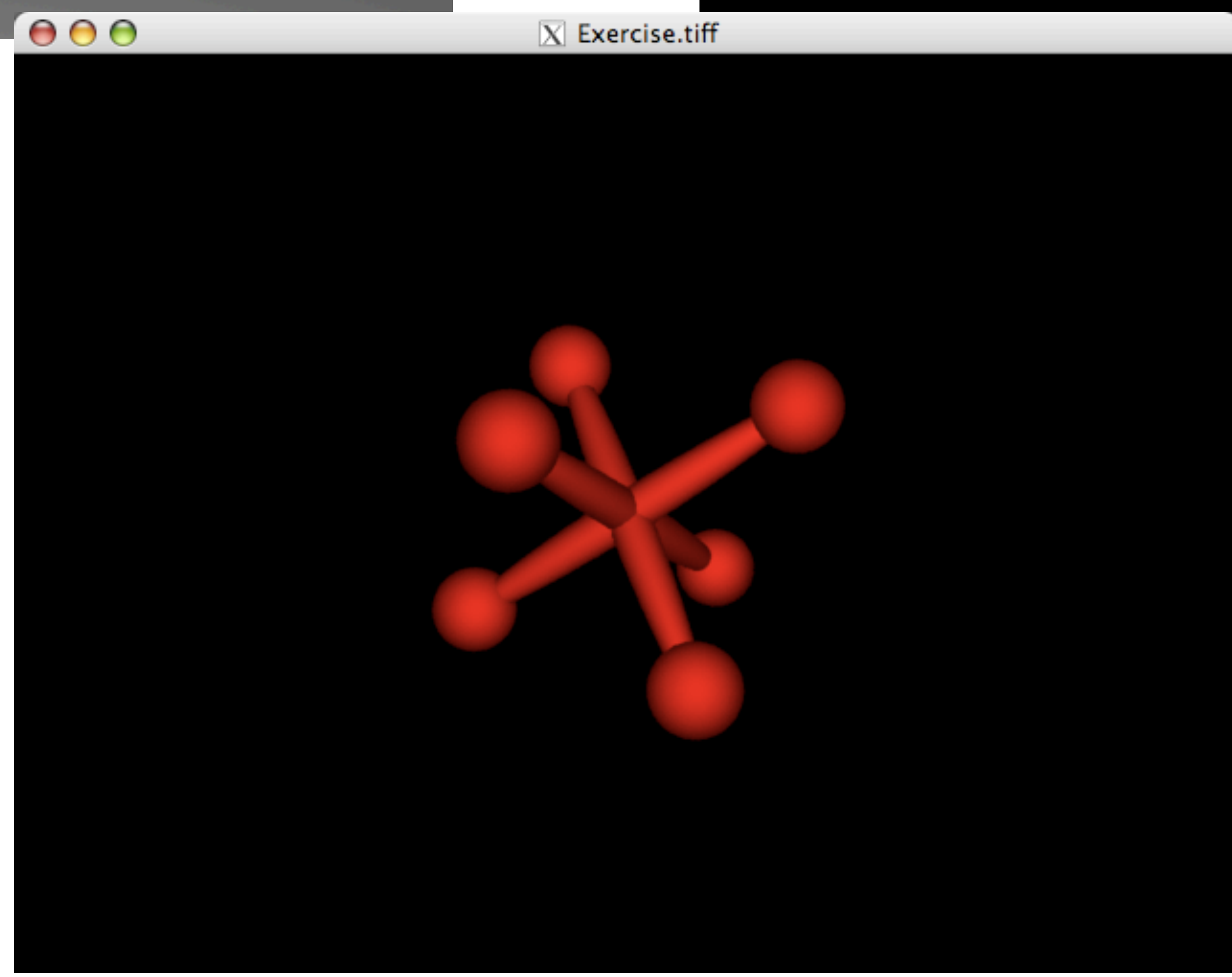
- Renderman allows for sequences of frames to be created within the rib structure by using FrameBegin / FrameEnd
- However it is usually better to create a sequence of individual rib files per frame as these can be distributed on the render farm.
- The best method for doing this is to use a frame counter and export Rib / frame using the format file.####.rib -> image.####.exr

```
for frame in range(1,30) :
  filename = "Wave.%03d.rib" %(frame)
  ri.Begin(filename)

  ri.Display("ProcGeom.%03d.exr" %(frame), "file", "rgba")

```

Exercise

- Try to build this Scene using python functions




Exercise.tiff

Exercise.tiff

Exercise.tiff

References

- [1] Ian Stephenson. Essential Renderman Fast. Springer-Verlag, 2003.
- [2] Larry Gritz Anthony A Apodaca. Advanced Renderman (Creating CGI for Motion Pictures). Morgan Kaufmann, 2000.
- Upstill S "The Renderman Companion" Addison Wesley 1992
- Renderman Documentation Appendix D - RenderMan Interface Bytestream Conventions
- Application Note #3 How To Render Quickly Using PhotoRealistic RenderMan