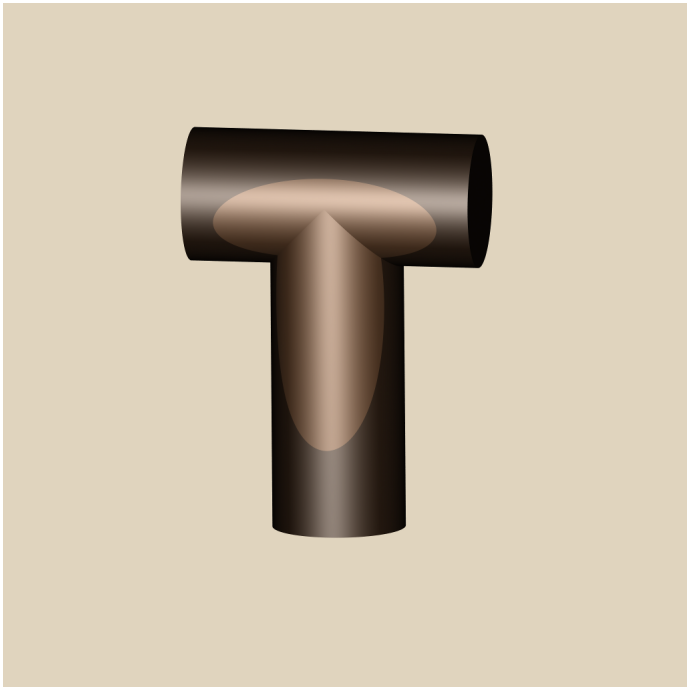

# Tovero Version 0.2.0 Tutorial

# Introduction

Tovero is a 3D modeling system. It uses a technique called constructive solid geometry (CSG) to represent models. Currently, Tovero is layered on top of BRL-CAD using its libraries and tools. This tutorial is based on material from the the document "BRL-CAD Tutorial Series: Volume II Introduction to MGED". It assumes you have some understanding of basic 3D modeling concepts, such as vectors, transformations, and projections.

In this tutorial, you will create a model of a radio. Unlike other 3D modeling systems you may have used, Tovero does not use a GUI to create and interact with models. Rather, you construct a model with a Ruby script. You run the script to generate a BRL-CAD ".g" database file. The database file can then be used to raytrace, view, or analyze the model with tools from BRL-CAD, either by calling these tools from the script (as in the tutorial) or externally. A final step in the scripts developed during the tutorial is to convert the raytraced image to PNG format for viewing.

The tutorial uses Ruby to create Tovero models, although the current version of Tovero also supports C++. The concepts for creating Tovero models in C++ are essentially the same as for Ruby.

# Radio 1: Creating a Simple Radio

The entire working code for this section is in a file called `radio1.rb`, located in the same directory on your system where you installed this tutorial. You can use this file as a reference if you encounter any problems while following the tutorial. A visual differences tool (such as meld) run between the reference version and your version of the script can help you in troubleshooting.

## First Steps

Start a text editor, creating a new file called `radio1.rb`, for example:

```
emacs radio1.rb
```

Now add the following code to this file:

```
# These load the extension libraries
require "libtovero_support_rb_1"
require "libtovero_math_rb_1"
require "libtovero_graphics_rb_1"
```

As the comment indicates, the three require statements load the Tovero libraries. If you have correctly installed Tovero, a script you will use to run the model (tovero_ruby) will set up the path to the libraries. Now add:

```
# These are like "using namespace" in C++
include Libtovero_support_rb_1
include Libtovero_math_rb_1
include Libtovero_graphics_rb_1
```

The include statements effectively allow you to refer to Tovero class names without prefixing them with the module name (e.g. `Libtovero_support_rb_1::`).

Now you will add some Ruby constants which will make the model code more readable:

```
#
# Setup some constants
#

ZERO = Unitless.new(0.0)
CM = Distance::meter * 0.01
ZERO_CM = Distance::meter * 0.0
ORIGIN = Point.new(ZERO_CM,
                   ZERO_CM,
                   ZERO_CM)
```

Geometric objects (e.g. the point used for the origin) in Tovero use explicit units to avoid errors. Tovero uses the `Unitless` class for quantities that do not have associated units.

Next you will add a simple "box" shape which represents the body of the radio.

The model is created in a coordinate system where the XY plane is horizontal and the Z axis is vertical. When the model is later rendered to create an image, this modeling coordinate system will be projected to a plane using available options to specify the angle from which the model is viewed. (See the `azimuth` and `elevation` options to the rt tool shown later in this section.)

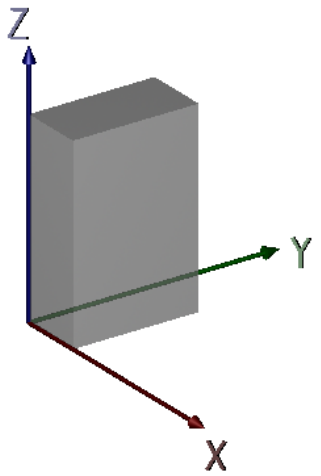


Figure 1: Modeling Coordinate System

The box will be positioned as shown in Figure 1. Add the box shape with:

```
#
# Create the radio
#

#
#   radio body
#
body = AxisAlignedBox.new(ORIGIN,
                          Point.new(CM * 16.0,
                                    CM * 32.0,
                                    CM * 48.0),
                          "body.s")
```

The shape you create with this object is an axis-aligned rectangular prism. The sides are parallel to the X, Y, and Z axes of its modeling coordinate system. Two points (here the origin and the second point parameter) specify the extents of the box, i.e. its minimum and maximum vertices. Note that the point object is created using centimeter units with the constant you defined earlier.

The `"body.s"` parameter gives the shape a name in the BRL-CAD database, which makes it more convenient to use the model with BRL-CAD's tools. The ".s" at the end of the name is a convention indicating that the object is a primitive shape.

Add the following lines to create a combination shape object:

```
#
#   combine parts to make the radio
#
radio = SolidCombination.new("radio.c")
radio << body
```

A combination shape, as you will see later, allows you to combine primitive shapes (like the axis-aligned box) together using boolean operations. For now, you are just adding the body as a single member of the combination. The name ends with ".c", again by convention, to indicate that it is a combination.

To actually create the BRL-CAD database object, add the following lines to the Ruby script:

```
#
# Create a BRL-CAD ".g" database
#

database = BCDatabase.new
```

After the database is created, add the body to its list of shapes with:

```
#
# Add the radio to the database
#

solid_list = database.top_solids;
solid_list << radio
```

Then, to write the database to a file named `radio1.g`:

```
#
# Write the database to a file
#

database_file = "radio1.g"
error = ErrorParam.new
success = database.write(database_file, true, error)
if (not success)
  puts("Could not write #{database_file}:")
  puts(error.base.to_s)
  abort()
end
```

After the ".g" database exists as a file, you can raytrace it using:

```
#
# Render the database with BRL-CAD's tools
#

image_height = 512
image_width = 512
azimuth = -35
elevation = 25

#   display raytraced model
#   options to "rt":
#     -C [background color (R/G/B)]
#     -a azimuth
#     -e elevation
#     -o [output .pix file]
#     -w [width in pixels of rendering]
#     -n [height in pixels (number of lines) of rendering]
#     <first positional arg> [database]
#     <second positional arg> [entity in database to trace]

command = "rm -f #{database_file}.pix #{database_file}.png ; \
          rt -C255/255/255 -a #{azimuth} -e #{elevation} \
             -o #{database_file}.pix \
             -w#{image_width} -n#{image_height} #{database_file} radio.c"
%x[#{command}]

#   convert image and cleanup
command = "pix-png -w#{image_width} -n#{image_height} \
                   -o #{database_file}.png \
                   #{database_file}.pix ; \
          rm -f #{database_file}.pix"
%x[#{command}]
```

The code first uses BRL-CAD's command line tool rt to raytrace the database, outputting to a 2D graphics format called ".pix". It then uses a command line tool caled pix-png to convert the .pix file to a PNG file which you can view with an image viewer.

Save the `radio1.rb` script. Run the script with:

```
tovero_ruby ./radio1.rb
```

You can either run this in a terminal window or in your editor. For example, emacs has a compile function, for which you can supply the above command. In emacs, make sure you are in the script's buffer when you execute the compile function so tovero_ruby can find your Ruby script. If you use a terminal to run the command, make sure the current directory is where you have saved the Ruby script.

After the Ruby script runs, you can view the resulting image file, `radio1.g.png`, with any image viewer that supports the PNG format. It should look like Figure 2.

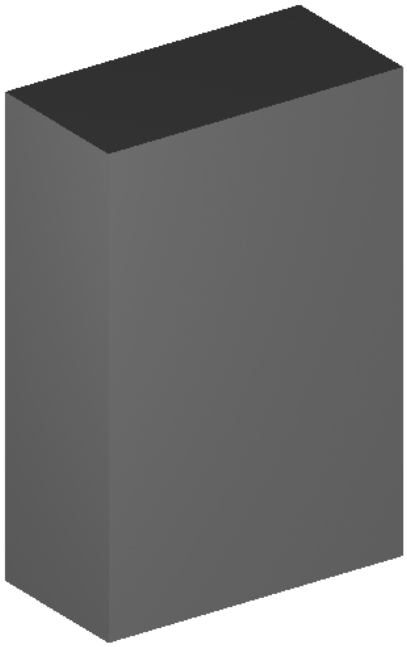

Figure 2: First Steps Radio

For a better understanding of how `azimuth` and `elevation` affect the view of the model, change their values in the script (from 0 to 360). Save the script and run tovero_ruby on it. Open the resulting image file to see how the view has changed.

## Adding the Remaining Shapes to the Radio

### Talk Button

You will now add the remaining shapes for the simple radio model. To start you will create a talk button. It consists of two parts, an elliptical cylinder and an ellipsoid. Insert the following code (and subsequent shapes) directly after the axis-aligned box you created previously:

```
#
#   talk button
#
button = EllipticalCylinder.new(Point.new(CM * 8.0,
                                          CM * 33.0,
                                          CM * 36.0),
                                 Vector.new(CM * 4.0,
                                            ZERO_CM,
                                            ZERO_CM),
                                 Vector.new(ZERO_CM,
                                            ZERO_CM,
                                            CM * 2.0),
                                 CM * 3.0,
                                 "btn.s")
```

In Tovero, an elliptical cylinder is a right cylinder which can have an elliptical cross section. It is created by specifying the center point of the base, two vectors which determine the shape and orientation of the cross section, and a height distance (which can be negative to construct the cylinder's axis in the opposite direction). The cylinder forms the inner part of the talk button.

Create the ellipsoid with:

```
button2 = Ellipsoid.new(Point.new(CM * 8.0,
                                  CM * 33.0,
                                  CM * 36.0),
                        Vector.new(CM * 4.0,
```

```
                                ZERO_CM,
                                ZERO_CM),
                        CM * 2.0,
                        "btn2.s")
```

An ellipsoid can be created several ways in Tovero. Here we use a "radius of revolution". The first point is the center, the vector is one axis of the ellipsoid, and the third distance parameter is the radius of revolution. The ellipsoid shape forms the outer part of the talk button.

### Speaker

A torus represents the speaker of the radio. Add the following code after the ellipsoid:

```
#
#   speaker
#
speaker = Torus.new(Point.new(CM * 16.0,
                              CM * 16.0,
                              CM * 16.0),
                    UnitVector.new(Unitless.new(1.0),
                                   ZERO,
                                   ZERO),
                    CM * 12.0,
                    CM * 1.0,
                    "spkr.s")
```

The first parameter is the center point of the torus. The second parameter is the direction the torus is facing. Notice that this parameter is a unit vector, since it is a direction. The next two distances are the major and minor radii respectively. The major radius is the outer radius of the torus, and the minor radius is smaller radius of the circular cross-section.

### Antenna and Tuning Knob

Two right circular cylinders form the antenna and tuning knob. Add these after the speaker torus:

```
#
#   antenna
#
antenna = Cylinder.new(Point.new(CM * 2.0,
                                 CM * 2.0,
                                 CM * 46.0),
                       Vector.new(ZERO_CM,
                                  ZERO_CM,
                                  CM * 48.0),
                       CM * 1.0,
                       "ant.s")

#
#   volume knob
#
knob = Cylinder.new(Point.new(CM * 4.0,
                              CM * 4.0,
                              CM * 40.0),
                    Vector.new(CM * 8.0,
                               ZERO_CM,
                               ZERO_CM),
                    CM * 5.0,
                    "knob.s")
```

The parameters for both of these cylinders are the center point of the circular base, a height vector, and the radius of the cylinder. You now have all the parts of the radio. Add them after the code where you added the body to the radio combination:

```
radio << button
radio << button2
radio << speaker
radio << antenna
radio << knob
```

Now save and run the model code as before with tovero_ruby (either in emacs or in a terminal). Open `radio1.g.png` in your image viewer. It should look like Figure 3.

Figure 3: Complete Simple Radio

# Radio 2: A More Sophisticated Radio

The entire working code for this section is in a file called `radio2.rb`, located in the same directory on your system where you installed this tutorial.

The first radio you created demonstrated several Tovero geometric primitives unioned together with a combination object. You'll now build a more sophisticated, realistic model. Copy your first radio in the file `radio1.rb` to a new file called `radio2.rb`. Open this new file in a text editor.

One improvement we can make is to "hollow out" the body to create a case which is more like a real radio. As a first step, after creating the `AxisAlignedBox` object you assigned to the `body` variable, add following code:

```
cavity_offset = Vector.new(CM * 1.0,
                           CM * 1.0,
                           CM * 1.0)
cavity = AxisAlignedBox.new(body.minimum + cavity_offset,
                            body.maximum - cavity_offset,
                            "cavity.s")
```

This will create a box inset 1 centimeter from the body box.

The antenna can be improved as well. Locate the section of code where you assigned a cylinder to the variable `antenna`. Change the name `antenna` to `antenna_cylinder`. Add the following code after the creation of the cylinder:

```
antenna_ellipsoid = Ellipsoid.new(Point.new(CM * 2.0,
                                            CM * 2.0,
                                            CM * 94.0),
                                  Vector.new(ZERO_CM,
                                             ZERO_CM,
                                             CM * 1.0),
                                  CM * 3.0,
                                  "ant2.s")
```

This will add a "cap" to the antenna, in the shape of an ellipsoid, a shape you have already seen.

At this point, you will begin to define "regions" for each part of the radio. In Tovero, regions represent unique volumes of physical space. A region is made up of combinations of geometric shapes. As in real physical space, different regions should not overlap. Since they represent physical objects, regions can have graphical attributes such as color.

When you raytrace a model with BRL-CAD, it will output a warning if it detects two overlapping regions. By convention, region names end in ".r".

Locate the section of code where you assigned an elliptical cylinder to the `button` variable. Change the name `button` to `button_cylinder`. Then locate the section of code where you assigned an ellipsoid to the `button2` variable. Change the name `button2` to `button_ellipsoid`. To create the first region in the model, add the following code after the creation of the `button_ellipsoid`:

```
button_color = Color.new(128, 128, 128, 0) # grey
button_attributes = CombinationAttributes.new
button_attributes.color = button_color
button_attributes.is_part = true

button = Combination.new(button_attributes, "button.r")
button << button_cylinder
button << button_ellipsoid
```

The code creates a color object and an attributes object (with default values) for the region, and it assigns the color to the attributes. It then creates a combination with these attributes. Setting the `is_part` flag for the attributes tells Tovero that this combination is a region.

Notice that here you are creating a `Combination` object (for the button), whereas before you had created a `SolidCombination` object (for the whole radio). The `Combination` object is a specialized type of `SolidCombination`, allowing you to assign attributes. The `SolidCombination` cannnot be used for regions, only for simple combinations of primitives. After the combination is created, the code unions the cylinder and ellipsoid into the button region.

Now create a region for the antenna by adding the following code after the creation of the `antenna_ellipsoid`:

```
antenna_shader = PhongShader::mirror
antenna_color = Color.new(128, 128, 128, 0) # grey
antenna_attributes = CombinationAttributes.new
antenna_attributes.shader = antenna_shader
antenna_attributes.color = antenna_color
antenna_attributes.is_part = true

antenna = Combination.new(antenna_attributes, "antenna.r")
antenna << antenna_cylinder
antenna << antenna_ellipsoid
```

This is similar to the code for the button region, with the exception of adding a "shader". A shader is an object that allows you to specify the material properties which determine the appearance of the object other than the color. For the antenna, you are specifying a mirror shader, which derives from a more general plastic or Phong shader. Using the mirror shader for the antenna approximates the appearance of chrome.

Locate the section of code where you assigned a cylinder to the `knob` variable. Change the name `knob` to `knob_cylinder`. Then add the following code after the point where you create the `knob_cylinder`:

```
knob_shader = PhongShader::plastic
knob_shader.shine = 4
knob_color = Color.new(198, 198, 0, 0) # yellow
knob_attributes = CombinationAttributes.new
knob_attributes.shader = knob_shader
knob_attributes.color = knob_color
knob_attributes.is_part = true

knob = Combination.new(knob_attributes, "knob.r")
knob << knob_cylinder
```

You are now ready to finish up the radio case by adding the following code after the code for the knob:

```
#
#   radio case
#
case_color = Color.new(0, 0, 255, 0) # blue
case_attributes = CombinationAttributes.new
case_attributes.color = case_color
case_attributes.is_part = true

radio_case = Combination.new(case_attributes, "radio_case.r")
radio_case << body
radio_case << SolidMember.new(SolidOperand.new(cavity),
                              SolidOperator::Difference_op)
radio_case << SolidMember.new(SolidOperand.new(antenna_cylinder),
                              SolidOperator::Difference_op)
radio_case << SolidMember.new(SolidOperand.new(knob_cylinder),
                              SolidOperator::Difference_op)
radio_case << SolidMember.new(SolidOperand.new(button_cylinder),
                              SolidOperator::Difference_op)
radio_case << SolidMember.new(SolidOperand.new(speaker),
                              SolidOperator::Union_op)
```

The attributes are similar to previous code. However, creating the radio case region involves some new concepts. Previously, you had only unioned geometry. The radio case region uses union and difference operators. The difference operators for the case subtract out space from it so it will not overlap regions (such as the antenna) later when you create the model for the entire radio.

Unioning in an object (or "member") for a combination was simple. You just used the `<<` operator between the combination and the object. To create a difference operation, you first have to create a `SolidMember` object. The `SolidMember` object is created with an operand (the shape) and an operator (the difference operation) which is then appended to the region (combination) with `<<`. A third operator which performs the intersection of two solids is also available.

When a combination is evaluated as an expression during rendering, the union operator has the least precedence. The union operator effectively groups difference and intersection operators (which have equal precedence) for a single combination. Note that since Tovero does not currently support parentheses, some expressions may need additional combinations to correctly group operators.

The last region you will create for this model is the circuit board. After the code above for the radio case add:

```
#
#   circuit board
#
board_box = AxisAlignedBox.new(Point.new(CM * 3.0,
                                         CM * 1.0,
                                         CM * 1.0),
                               Point.new(CM * 4.0,
                                         CM * 31.0,
                                         CM * 47.0),
                               "board.s")

board_color = Color.new(0, 255, 0, 0) # green
board_attributes = CombinationAttributes.new
board_attributes.color = board_color
board_attributes.is_part = true

board = Combination.new(board_attributes, "board.r")
board << board_box
```

This code is very similar to code you have seen before. It creates an axis-aligned box shape, followed by attributes for the circuit board and a region made from the box.

Replace the following code from your first radio model:

```
radio = SolidCombination.new("radio.c")
radio << body
radio << button
radio << button2
radio << speaker
radio << antenna
radio << knob
```

with:

```
radio = Combination.new("radio.a")
radio << radio_case
radio << button
radio << knob
radio << antenna
radio << board
```

This will create a combination with the regions unioned together into a part assembly, with a name ending in ".a" by convention.

Locate the section of code where you set the variable `database_file`. Change the assigned value to `"radio2.g"`, to modify the name of the BRL-CAD database and to change the name of the image file to `radio2.g.png` .

Make sure you change `radio.c` to `radio.a` in the first rendering command to specify the correct object to raytrace. The code should now be:

```
command = "rm -f #{database_file}.pix #{database_file}.png ; \
          rt -C255/255/255 -a #{azimuth} -e #{elevation} \
             -o #{database_file}.pix \
             -w#{image_width} -n#{image_height} #{database_file} radio.a"
```

Save and run the model with tovero_ruby (either in emacs or in a terminal). View `radio2.g.png` in your image viewer. It should look like Figure 4.

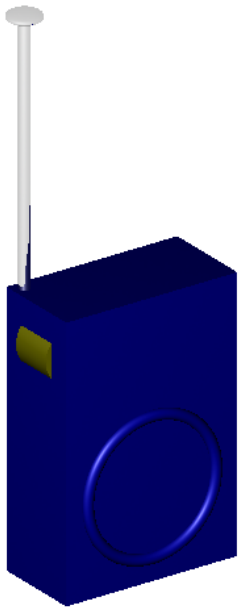

Figure 4: A More Sophisticated Radio

## Radio 3: Alternate Views of the Radio using Specialized Models

In the second section you developed an improved radio model, including an internal printed circuit board. You have likely realized, however, that you were unable to view inside the radio with this version of the model. In this section, in addition to the original model, you will create two specialized versions which provide a translucent view and a cutaway view. This will allow you to see inside the radio. One script will create all three models stored as separate combinations in a single BRL-CAD database. When you run the script, it will create image files for all three views: normal, translucent, and cutaway.

The entire working code for this section is in a file called `radio3.rb`, located in the same directory on your system where you installed this tutorial.

## Creating a Translucent View of the Radio

Make a copy of your second radio in the file `radio2.rb` to a new file called `radio3.rb`. Open the new file in a text editor.

Add this code in the constant setup section after the `ORIGIN` constant:

```
Z_AXIS = UnitVector.new(ZERO,
                        ZERO,
                        Unitless.new(1.0))
```

You will use the `Z_AXIS` constant later in the tutorial.

The first specialized model modifies the case to be translucent in order to show the parts inside. All the other parts are identical and are unioned together with the new case as a combination.

Locate the section in the code where you created the `radio` combination and added the parts. After all of the parts have been combined with the `<<` operator, add the following code:

```
#
# Make a translucent view
#

#
#   make a copy of the attributes
#
radio_case_translucent_shader = PhongShader.new
radio_case_translucent_shader.transmitted = Unitless.new(0.6)
radio_case_translucent_attributes = CombinationAttributes.new(case_attributes)
radio_case_translucent_attributes.shader = radio_case_translucent_shader
```

The code copies the original attributes of the case (`case_attributes`, including the shader) to a new attributes object using the "copy constructor" of `CombinationAttributes`. Changing the `transmitted` value of the shader to `0.6`, causes it to transmit a fraction of light through the case, making it translucent.

After copying the attributes, add:

```
#
#   make a copy of the case
#
radio_case_translucent = Combination.new(radio_case)
radio_case_translucent.name = "case_translucent.r"
radio_case_translucent.attributes = radio_case_translucent_attributes
```

This copies the `radio_case` object to the translucent version. To complete the translucent radio, add:

```
#
#   make the translucent radio
#
radio_translucent = Combination.new("radio_translucent.a")
radio_translucent << radio_case_translucent
radio_translucent << button
radio_translucent << knob
radio_translucent << antenna
radio_translucent << board
```

Again, all the other parts are the same. However they are unioned together into a new combination (assembly) with the translucent case.

The new translucent view needs to be added to the BRL-CAD database. So far, one shape is added to the database with the line:

```
solid_list << radio
```

Immediately after this line, add the line:

```
solid_list << radio_translucent
```

Locate the place where you set the variable `database_file` and change its assigned value to `"radio3.g"` to modify the name of the database.

To render the translucent view, go to the end of the code where the raytracing is performed. Replace the following code starting with the lines:

```
#
# Render the database with BRL-CAD's tools
#
```

to the end of the file with:

```
#
# Render the database with BRL-CAD's tools
#

image_height = 512
image_width = 512
azimuth = 35
elevation = 25

#   display raytraced model
#   options to "rt":
#     -C [background color (R/G/B)]
#     -a azimuth
#     -e elevation
#     -o [output .pix file]
#     -w [width in pixels of rendering]
#     -n [height in pixels (number of lines) of rendering]
#     <first positional arg> [database]
#     <second positional arg> [entity in database to trace]
views = ["translucent"]
views.each do |view|
  if view == ""
    image_file_name = "radio3.g"
    view_assembly = "radio.a"
  else
```

```
    image_file_name = "radio3_#{view}.g"
    view_assembly = "radio_#{view}.a"
  end
  command = "rm -f #{image_file_name}.pix #{image_file_name}.png ; \
            rt -C255/255/255 -a #{azimuth} -e #{elevation} \
               -o #{image_file_name}.pix \
               -w#{image_width} -n#{image_height} \
               #{database_file} #{view_assembly}"
  %x[#{command}]

  # convert image and cleanup
  command = "pix-png -w#{image_width} -n#{image_height} \
                     -o #{image_file_name}.png \
                     #{image_file_name}.pix ; \
            rm -f #{image_file_name}.pix"
  %x[#{command}]
end
```

The `views` array will eventually hold the names of models you want to be raytraced. For now, you only need the translucent model to be rendered.

Run the model code in `radio3.rb` with tovero_ruby and view the resulting image file (`radio3_translucent.g.png`). It should look like Figure 5. As you can see, the the radio has a translucent case in order to see inside it.

## Creating a Cutaway View of the Radio

The cutaway version of the model takes a different approach than the translucent version. It uses all the original parts, including the opaque radio case. To view inside the radio, it cuts a portion of the radio off using a box and a CSG difference operation (sometimes called subtraction).

Return to the point in the code after the line where you finished the translucent radio:

```
radio_translucent << board
```

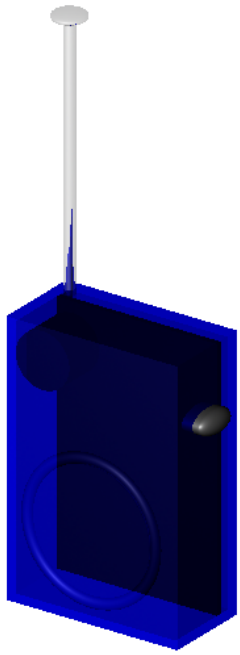

Figure 5: Translucent View Radio

Add the code:

```
#
# Make a cutaway view
#

cutaway_box = AxisAlignedBox.new(Point.new(CM * -16.0,
                                           CM * -16.0,
                                           CM * -1.0),
                                 Point.new(CM * 16.0,
                                           CM * 16.0,
                                           CM * 49.0),
                                 "cutaway_box.s")
```

The axis-aligned box is the shape you will use later to "remove" material from the radio using the difference operator. However before you do this, you have to position the box with a `Transformation` object. Create the transformation by adding the lines:

```
cutaway_transform = Transformation.new
cutaway_transform.set_translation(CM * -8.0,
                                  CM * 36.0,
                                  ZERO_CM)
cutaway_transform.rotate(Z_AXIS, Angle::degree * -45.0)
```

The code applies a translation and rotation (around the Z axis, using the constant you created earlier) to correctly orient the box for the subtraction. To apply the transformation, add:

```
cutaway = SolidCombination.new("cutaway.c")
cutaway << SolidMember.new(SolidOperand.new(cutaway_box, cutaway_transform))
```

You previously created `SolidOperand` objects (without a transformation) when you needed to apply a `SolidOperator` in a combination. `SolidOperand` objects also allow you to associate a transformation to members of the a combination. The previous two lines create a new `SolidCombination` to hold the (only) member of a combination whose operand has a transformation applied to it.

To finish the cutaway view, add:

```
radio_cutaway = Combination.new("radio_cutaway.a")
radio_cutaway << radio
radio_cutaway << SolidMember.new(cutaway, SolidOperator::Difference_op)
```

This creates a combination consisting of the second radio with the cutaway box subtracted out. Add this new view to the BRL-CAD database with the line (after the similar lines which add `radio` and `radio_translucent` to the database):

```
solid_list << radio_cutaway
```

Change the line near the raytracing code:

```
views = ["translucent"]
```

to

```
views = ["cutaway"]
```

Run the model again with tovero_ruby and view the `radio3_cutaway.g.png` image file. It should look like Figure 6. It has a portion of the radio removed in order to see inside it.

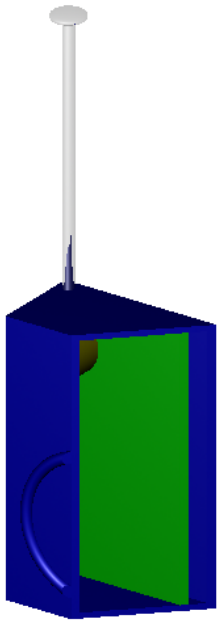

Figure 6: Cutaway View Radio

To finish the script, change the line:

```
views = ["cutaway"]
```

to:

```
views = ["", "translucent", "cutaway"]
```

which will create image files for each of the three views every time you run the script.