

fig2pov documentation

# Summary

Convert MATLAB figures to ray-traced images using POV-Ray as the rendering engine. If desired, add texture and other rendering elements not supported by MATLAB to your graphics objects.

# Introduction

MATLAB graphics uses openGL or Painters for rendering figures. The quality of the rendering is not as good as what can be provided by e.g ray tracing. POV-Ray is a popular open-source ray tracing program with its own scripting language for describing scenes.

This function provides a way to convert MATLAB figures (or, to be exact, axes) to images rendered using ray tracing. It does so by generating a povray script based on the contents of the axes object. This script can then be executed in POV-Ray to generate the final image.

# Usage

After you've generated an axes object with all the graphics elements you want, this figure can be converted to a POV-Ray script by simply calling:

|  |
| --- |
| fig2pov(h\_axes, script\_name) |

With:

* h\_axes the handle to the axes object you want to convert (use gca for most recent axes)
* script\_name the name you would like for the generated script

If no arguments are provided, the default values are h\_axes = gca and script\_name = 'fig.pov'.

To see the rendered version, you will have to execute this script in POV-Ray. POV-Ray is a free open-source program. It can be downloaded here:

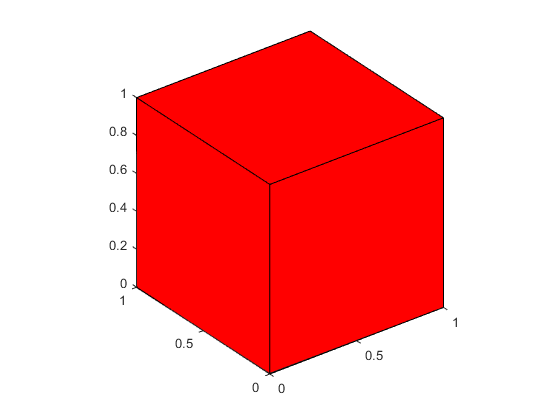
<https://www.povray.org/download/>

# A first example

Consider a red cube, drawn using the MATLAB patch object:

|  |
| --- |
| cube = patch('Vertices',[0 0 0; 0 0 1; 0 1 0; 0 1 1; 1 0 0; 1 0 1; 1 1 0; 1 1 1], ...  'Faces',[1 2 4 3; 5 6 8 7; 1 2 6 5; 3 4 8 7; 1 3 7 5; 2 4 8 6], ...  'FaceColor', [1 0 0]);  view(3)  axis equal  h\_axes = gca; |

The code above will result in the following figure:



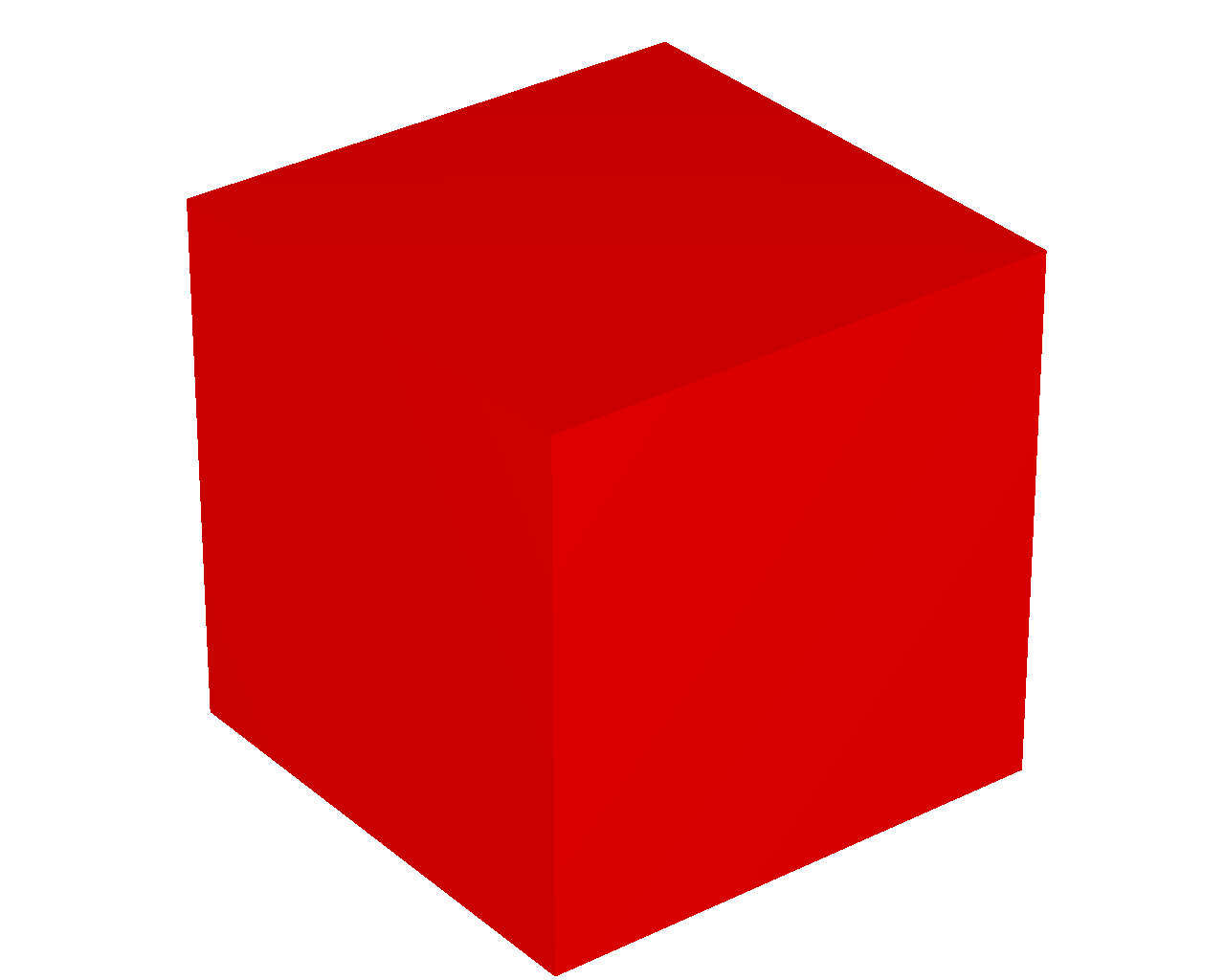
Next, call fig2pov to convert this figure into a POV-Ray script:

|  |
| --- |
| fig2pov(gca, 'cube.pov'); |

The working directory will now contain a file called 'cube.pov', which is a file in the POV-Ray scripting language. The \\*.pov format is readable and can be edited in any text editor. To render the image, the script has to be executed by POV-Ray. Make sure POV-Ray is installed and call it from either the command line or using the GUI version.

|  |
| --- |
| povray cube.pov |

In this case, the script will result in the following figure:



# Adding texture and other elements

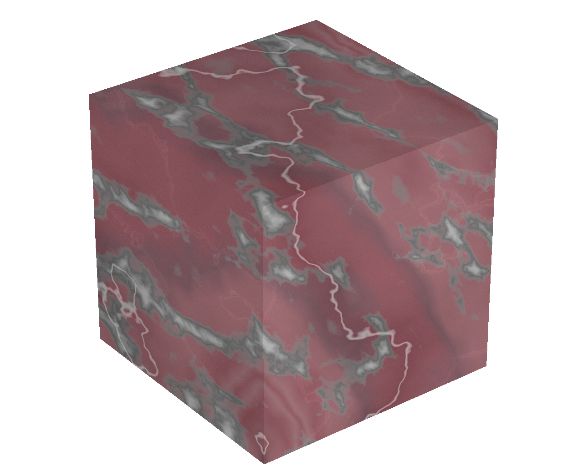
In addition to a straightforward conversion from the MATLAB figure to a ray-traced image, it is possible to specify additional properties and options for the graphics objects. This will enable rendering options that are not natively supported in MATLAB itself, such as adding texture to patch objects, adding shadowing, etc ...

Each graphics object has a property called ‘UserData’, which allows any type of data to be attached to the object. The appearance a graphics object can be changed by adding a structure named 'povray' to the UserData of that object.

For instance, with 'cube' being the handle of the patch created earlier:

|  |
| --- |
| cube.UserData.povray.Texture = 'T\_Stone21'; |

When calling fig2pov with this addition to cube, the same cube will now be rendered with a texture named 'T\_Stone21', giving the following figure:

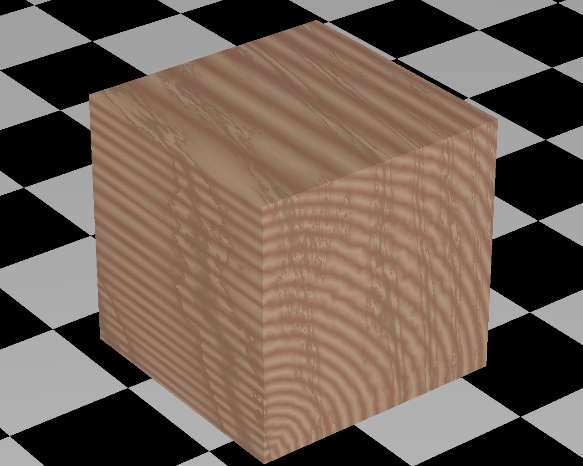


An elaborate list of textures supported by POV-Ray can be found in <https://github.com/sigurd-ss/fig2pov/tree/master/Documents>.

Each graphics object can have its own 'UserData.povray' structure. In the example below, both cube and the axes object h\_axes have a structure 'UserData.povray'.

|  |
| --- |
| cube.UserData.povray.Texture = 'T\_Wood1';  h\_axes.UserData.povray.Plane = [0 0 1 -5];  h\_axes.UserData.povray.PlaneColor = 'checker color Black, color White'; |

This gives:



If an axes object contains multiple graphics objects, all of them will be included in the POV-Ray script. Supported graphics objects are patch, surface, line and light.

The section below provides a full list of all options that can be included in the UserData.povray structure. Different graphics objects may have different options.

# Additional POV-Ray options

As explained above, it is possible to add extra rendering instructions to individual graphics objects. Even if these instructions are not supported in MATLAB itself, fig2pov will interpret them when generating the POV-Ray script.

Additional instructions can be added for objects of type axes, patch, surface, line and light. Below, we list the options that are available for each of those types.

## Additional POV-Ray options for object of type ‘axes’

The fields that can be added to the structure UserData.povray for and object of type ‘axes’ are: Plane.Normal, Plane.Color, Plane.Texture, Plane.TextureScale, Define and Zoom.

**Plane**

Adding the field ‘Plane’ to the UserData.povray structure of the axes object draws an infinite plane with the specified orientation. The subfields of Plane determine the appearance of the plane. UserData.povray.Plane can be a single element or an array (to specify multiple planes)

**Plane.Normal**

The value of UserData.povray.Plane.Normal is an array of four numbers. The first three numbers contain the normal of the Plane. The fourth number is the distance from the origin. For instance [0 0 1 -1] is a plane perpendicular to the Z-axis and crosses the Z-axis at z=-1.

The properties of the plane are further specified by the values of ‘Plane.Color’ and ‘Plane.Texture. If neither of these are specified, the default value is a checkered black-and-white plain.

See example under Plane.Color.

**Plane.Color**

If UserData.povray contains a valid value for the field ‘Plane’, UserData.povray.Plane.Color specifies the color of the Plane. The color can be given as an RGB triplet or as a string.

Example:

|  |  |
| --- | --- |
| cube = patch('Vertices',[0 0 0; 0 0 1; 0 1 0; 0 1 1; 1 0 0; 1 0 1; 1 1 0; 1 1 1], ...  'Faces',[1 2 4 3; 5 6 8 7; 1 2 6 5; 3 4 8 7; 1 3 7 5; 2 4 8 6], ...  'FaceColor', [1 0 0]);  view(3)  axis equal  h\_axes = gca;  h\_axes.UserData.povray.Plane.Normal = [0 0 1 -1];  h\_axes.UserData.povray.Plane.Color = [0.8 0.8 0.8]; | |
|  |  |
| No PlaneColor specified | PlaneColor = [0.8 0.8 0.8] |

**Plane.Texture**

If UserData.povray contains a valid value for the field ‘Plane’, UserData.povray.Plane.Texture specifies the texture of the Plane. This value is given as a string containing one of the names of the known POV-Ray textures.

Example:

|  |  |
| --- | --- |
| cube = patch('Vertices',[0 0 0; 0 0 1; 0 1 0; 0 1 1; 1 0 0; 1 0 1; 1 1 0; 1 1 1], ...  'Faces',[1 2 4 3; 5 6 8 7; 1 2 6 5; 3 4 8 7; 1 3 7 5; 2 4 8 6], ...  'FaceColor', [1 0 0]);  view(3)  axis equal  h\_axes = gca;  h\_axes.UserData.povray.Plane.Normal = [0 0 1 -1];  h\_axes.UserData.povray.Plane.Texture = 'T\_Stone28'; | |
|  |  |
| PlaneTexture = 'T\_Stone28' | PlaneTexture = 'Cork' |

**Plane.TextureScale**

UserData.povray.Plane.TextureScale further specifies the texture of the plane. It only takes effect if UserData.povray.Plane.Texture is contained in UserData.povray. Examples of the use of TextureScale are given later for patch objects.

**Define**

It is possible to define values for use by POV-Ray, for instance to define a custom texture. This can be done by adding the to-be-defined value as a field in UserData.povray of the axes object. The value of the field is the desired definition.

Example:

axes\_povray.define.T\_Sand = sprintf([

'texture {\n' ...

'\tpigment {color rgb <0.3320, 0.2891, 0.1602> }\n' ...

'\tnormal { granite 0.2 scale 0.02 }\n' ...

'\tnormal { bumps 0.4 scale 1 }\n' ...

'\tfinish {\n' ...

'\t\tbrilliance 1.6\n' ...

'\t\tspecular 0.3\n' ...

'\t\tambient 0.05\n' ...

'\t}\n' ...

'}'

]);

This defines a new texture ‘T\_Sand’ that can then be used as a value in other fields.

**Zoom**

fig2pov reproduces the scene as it is contained in the axes. If the overall scale needs to be changed, this can be done by adding the field Zoom to UserData.povray of the axes object. See example below.

Example:

|  |  |  |
| --- | --- | --- |
| cube = patch('Vertices',[0 0 0; 0 0 1; 0 1 0; 0 1 1; 1 0 0; 1 0 1; 1 1 0; 1 1 1], ...  'Faces',[1 2 4 3; 5 6 8 7; 1 2 6 5; 3 4 8 7; 1 3 7 5; 2 4 8 6], ...  'FaceColor', [1 0 0]);  view(3)  axis equal  h\_axes = gca;  h\_axes.UserData.povray.Zoom = 0.5; | | |
|  |  |  |
| Zoom = 0.5 | Zoom = 1 | Zoom = 1.5 |

Default Value: No Zoom is applied (i.e. Zoom = 1)

## Additional POV-Ray options for object of type ‘patch’

The fields that can be added to the structure UserData.povray for and object of type ‘patch’ are: Texture, InteriorTexture, TextureScale, drawAsSphere, drawAsCylinder, FaceColor, drawEdges, EdgeColor, EdgeTexture and MarkerTexture.

**Texture, InteriorTexture**

UserData.povray.Texture specifies the texture of the patch object. This value is given as a string containing on of the names of the known POV-Ray textures. If the patch has two sides, both Texture and InteriorTexture can be specified. See example below.

Example:

|  |  |
| --- | --- |
| cube = patch('Vertices',[0 0 0; 0 0 1; 0 1 0; 0 1 1; 1 0 0; 1 0 1; 1 1 0; 1 1 1], ...  'Faces',[ 3 4 2 1; 5 6 8 7; 7 8 4 3; 5 7 3 1], ...  'FaceColor', [1 0 0]);  view(3)  axis equal  h\_axes = gca;  cube.UserData.povray.Texture = 'T\_Stone1';  cube.UserData.povray.InteriorTexture = 'T\_Stone18'; | |
|  |  |
| No InteriorTexture | InteriorTexture = 'T\_Stone18' |

Note:

* the notion of “inside” and “outside” depends on the orientation of each face, i.e., the order in which the vertices of that face are linked. This needs to be done consistently to get the proper result.

**TextureScale**

If UserData.povray of the object contains a valid value for Texture, the appearance of the texture can be further changed by adding the field TextureScale. This field determines the size of the patterns in the scale (e.g. grain of the wood or marbling of the stone). See example below.

Example:

|  |  |  |
| --- | --- | --- |
| cube = patch('Vertices',[0 0 0; 0 0 1; 0 1 0; 0 1 1; 1 0 0; 1 0 1; 1 1 0; 1 1 1], ...  'Faces',[1 2 4 3; 5 6 8 7; 1 2 6 5; 3 4 8 7; 1 3 7 5; 2 4 8 6], ...  'FaceColor', [1 0 0]);  view(3)  axis equal  cube.UserData.povray.Texture = 'T\_Stone18';  cube.UserData.povray.TextureScale = 1; | | |
|  |  |  |
| TextureScale = 1 | TextureScale = 5 | TextureScale = 0.2 |

Default Value: TextureScale = 1.

**drawAsSphere, drawAsCylinder**

In MATLAB, patch and surface objects are really collections of polygons, even if the intention is to draw a sphere or a cylinder. The appearance of "roundness" is obtained by using large numbers of faces.

POV-Ray natively supports commands that draw smooth spheres or cylinders (or any object with a rotation axis). If we know that an object is a sphere or has a symmetry axis, we can include this information in UserData.povray. fig2pov will then use the appropriate povray commands, rather than treating the object as a collection of faces.

Examples are given later for graphics objects of type surface.

**FaceColor**

FaceColor specifies the color of the patch object. It overwrites the values given in the field FaceColor of the object itself. It can be either an RGB triplet of values or a string.

**drawEdges**

By default, fig2pov creates three-dimensional objects, without edges to trace the different faces of the object. If edges are desired, this can be specified by setting UserData.povray.drawEdges to true. The thickness of the edges is determined by the property ‘LineWidth’ of the patch object.

See example below under EdgeTexture.

**EdgeColor**

If UserData.povray.drawEdges is true, UserData.povray.EdgeColor specifies the color of the edges. It can be either an RGB triplet of values or a string. If no value is given, the color is given by the value of the property EdgeColor of the patch object itself.

See example below under EdgeTexture.

**EdgeTexture**

If UserData.povray.drawEdges is true, UserData.povray.EdgeColor specifies the Texture of the edges. This value is given as a string containing on of the names of the known POV-Ray.

See example below.

Example:

|  |  |
| --- | --- |
| cube = patch('Vertices',[0 0 0; 0 0 1; 0 1 0; 0 1 1; 1 0 0; 1 0 1; 1 1 0; 1 1 1], ...  'Faces',[1 2 4 3; 5 6 8 7; 1 2 6 5; 3 4 8 7; 1 3 7 5; 2 4 8 6], ...  'FaceColor', [1 0 0]);  view(3)  axis equal  cube.LineWidth = 5;  cube.UserData.povray.drawEdges = true;  cube.UserData.povray.Texture = 'T\_Stone18';  cube.UserData.povray.EdgeTexture = 'Gold\_Texture'; | |
|  |  |
| EdgeTexture = 'Gold\_Texture'  drawEdges = true | drawEdges = true  no EdgeTexture or EdgeColor specified |

**MarkerTexture**

By default, markers are only drawn if the patch properties Marker is different from ‘none’ and the patch properties MarkerFaceColor and MarkerEdgeColor are not both equal to ‘none’. In that case, the markers are drawn as spheres. If UserData.povray.MarkerTexture is specified, the spheres will have the specified texture. See example below.

Example:

|  |  |
| --- | --- |
| cube = patch('Vertices',[0 0 0; 0 0 1; 0 1 0; 0 1 1; 1 0 0; 1 0 1; 1 1 0; 1 1 1], ...  'Faces',[1 2 4 3; 5 6 8 7; 1 2 6 5; 3 4 8 7; 1 3 7 5; 2 4 8 6], ...  'FaceColor', [1 0 0]);  view(3)  axis equal  cube.Marker = 'o';  cube.MarkerSize = 12;  cube.MarkerFaceColor = 'r';  cube.UserData.povray.Texture = 'T\_Stone18';  cube.UserData.povray.MarkerTexture = 'Gold\_Texture'; | |
|  |  |
| MarkerTexture = 'Gold\_Texture' | no MarkerTexture specified |

## Additional POV-Ray options for object of type ‘surface’

The fields that can be added to the structure UserData.povray for and object of type ‘surface’ are: Texture, InteriorTexture, TextureScale, drawAsSphere, drawAsCylinder, MeshOn and SmoothingOn.

**Texture, InteriorTexture**

See ‘Additional POV-Ray options for object of type ‘patch’’

**TextureScale**

See ‘Additional POV-Ray options for object of type ‘patch’

**drawAsSphere**

In MATLAB, patch and surface objects are really collections of polygons, even if the intention is to draw a sphere or a cylinder. The appearance of "roundness" is obtained by using large numbers of faces.

POV-Ray natively supports commands that draw smooth spheres or cylinders (or any object with a rotation axis). If we know that an object is a sphere or has a symmetry axis, we can include this information in UserData.povray. fig2pov will then use the appropriate povray commands, rather than treating the object as a collection of faces. See example belo.

Example:

|  |  |
| --- | --- |
| [x,y,z]=sphere;  h\_sphere = surf(x, y, z, 'FaceColor', 'r')  view(3)  axis equal  h\_sphere.UserData.povray.drawAsSphere = true; | |
|  |  |
| drawAsSphere = false | drawAsSphere = true |

Default Value: drawAsSphere = false

**drawAsCylinder**

Similar to drawAsSphere, if the object is known to have cylindrical rotational symmetry. See example below.

Example:

|  |  |
| --- | --- |
| t = 0:pi/10:2\*pi;  [x, y, z] = cylinder(0.5+0.1\*cos(t));  h\_cyl = surf(x, y, z, 'FaceColor', 'r')  axis equal  h\_cyl.UserData.povray.drawAsCylinder = true; | |
|  |  |
| drawAsCylinder = false | drawAsCylinder = true |

Default Value: drawAsCylinder = false

**MeshOn**

By default, surface objects are drawn as 3D objects, without mesh lines. If mesh lines are desired, this can be achieved by setting UserData.povray.MeshOn to true. The color and thickness of the gridlines are controlled by the surface object properties ‘EdgeColor’ and ‘Linewidth’ respectively. See example below.

Example:

|  |  |
| --- | --- |
| [X,Y,Z] = peaks(25);  h\_surf = surf(X, Y, Z/3);  view(3)  axis equal  h\_surf.UserData.povray.MeshOn = true; | |
|  |  |
| MeshOn= false | MeshOn=true |

Default Value: MeshOn = false

Notes:

* in its current form, ‘MeshOn’ will only work or surfaces defined on an (X, Y)-grid. Not for e,g, spheres or cylinders.

**SmoothingOn**

Surfaces are drawn as collections of faces. When the number of faces is relatively low, the appearance may not be smooth. POV-Ray includes and additional option for smoothing surfaces. This can be enabled by setting UserData.povray.SmoothingOn to true. See example below.

Example:

|  |  |
| --- | --- |
| [X,Y,Z] = peaks(25);  h\_surf = surf(X, Y, Z/3);  view(3)  axis equal  h\_surf.UserData.povray.SmoothingOn = true;  h\_surf.FaceColor=[1 0 0]; | |
|  |  |
| SmoothingOn = false | SmoothingOn = true |

Default Value: SmoothingOn = false

## Additional POV-Ray options for object of type ‘line’

POV-Ray is a ray-tracing program. As such, it has no concept of "lines". Instead, lines are converted into thin cylinders. The fields that can be added to the structure UserData.povray for and object of type ‘line’ are: Texture, TextureScale and SmoothingOn.

Example:

|  |  |
| --- | --- |
| h\_line=plot3(rand(10,1),rand(10,1),rand(10,1),'LineWidth',3);  view(3)  axis equal | |
|  |  |
| MATLAB | POV-Ray |

**Texture, TextureScale**

Determines the texture of the line (similar to the way it is done for objects of types patch and surface). See example below.

Example:

|  |  |
| --- | --- |
| h\_line=plot3(rand(10,1), rand(10,1), rand(10,1),'LineWidth',5);  view(3)  axis equal  h\_line.LineWidth = 10;  h\_line.UserData.povray.Texture = 'T\_Stone9'; | |
|  |  |
| Texture = 'T\_Stone9'  TextureScale = 1 | Texture = 'T\_Stone9'  TextureScale = 0.1 |

Default Value: no Texture, TextureScale = 1

**SmoothingOn**

By default, successive points in a line object are connected with straight segments. By setting UserData.povray.SmoothingOn equal to true, the connecting segments will be smoothed, while still passing through all the original points.

Example:

|  |  |
| --- | --- |
| h\_line=plot3(rand(10,1),rand(10,1),rand(10,1),'LineWidth',3);  view(3)  axis equal  h\_line.UserData.povray.SmoothingOn = true | |
|  |  |
| SmoothingOn = false | SmoothingOn = true |

Default Value: SmoothingOn = false

## Additional POV-Ray options for object of type ‘light’

POV-Ray requires the presence of at least one ‘light’ to be able to render a scene. In MATLAB, lights are optional. If no light is specified, fig2pov assume a light in the direction of the view. The option Shadowless can be specified for objects of type ‘light’.

**Shadowless**

In MATLAB, light illuminates an object, creating lighter and darker areas, but the lights do not cast a shadow. POV-Ray does support shading. By default, no shadows are cast, but this can be changed by setting UserData.povray.Shadowless to false. See example below.

Example:

|  |  |
| --- | --- |
| [X,Y,Z] = peaks(25);  h\_surf = surf(X, Y, Z/3);  hold on  [x,y,z] = sphere;  h\_sphere=surf(x+2, y-2, z+3);  h\_light = camlight('right');  view(3)  axis equal  h\_light.UserData.povray.Shadowless = false; | |
|  |  |
| Shadowless = true | Shadowless = false |

Default Value: Shadowless = true