■ Graphics`GraphicsOperations`

This package provides functions for general transformations of two and three-dimensional graphics primitives as well as functions for extracting or removing graphics primitives from within existing graphics objects.

`TranslateGraphics[`g`, {`d_x`, `d_y`}]`	translate all graphics primitives in the two-dimensional graphics object g by the specified vector
`TranslateGraphics[`g`, {`d_x`, `d_y`, `d_z`}]`	translate all graphics primitives in the three-dimensional graphics object g by the specified vector

Translating graphics objects.

`ScaleGraphics[`g`, `*scale*`]`	scale all graphics primitives in the graphics object g by the factor *scale*
`ScaleGraphics[`g`, {`$scale_x$`, `$scale_y$`}]`	scale all graphics primitives in the two-dimensional graphics object g by the respective scale factors
`ScaleGraphics[`g`, {`$scale_x$`, `$scale_y$`, `$scale_z$`}]`	scale all graphics primitives in the three-dimensional graphics object g by the respective scale factors
`ScaleGraphics[`g`, `*scale*`, `*origin*`]`	scale all graphics primitives in the graphics object g with respect to the point *origin*

Scaling graphics objects.

`RotateGraphics[`*g*`,` *phi*`]`	rotate all graphics primitives in the two-dimensional graphics object *g* clockwise by the specified angle *phi*
`RotateGraphics[`*g*`, {`*phi*`,` *theta*`,` *psi*`}]`	rotate all graphics primitives in the three-dimensional graphics object *g* by the specified Euler angles *phi*, *theta* and *psi*
`RotateGraphics[`*g*`,` *angles*`,` *origin*`]`	rotate all graphics primitives in the graphics object *g* about the point *origin*
`RotateGraphics[`*g*`,` *mat*`,` *origin*`]`	rotate all graphics primitives in the graphics object *g* using the rotation matrix *mat*

Rotating graphics objects.

option name	*default value*	
`TextRotated`	`False`	determines whether to rotate text orientation

Options for `RotateGraphics`.

`TextRotated` determines whether the orientation of any `Text` graphics primitives is also rotated along with the coordinates of the `Text` primitives. This option only affects the rotation of two-dimensional `Text` primitives since three-dimensional `Text` primitives are always shown in the display plane.

`TransformGraphics[`*g*`,` *f*`]`	applies the function *f* to all lists of coordinates in the graphics object *g*
`SkewGraphics[`*g*`,` *mat*`]`	applies the matrix *mat* to all coordinates in the graphics object *g*

Other transformation functions.

The transformation functions can be used with `Graphics`, `Graphics3D`, `SurfaceGraphics`, `DensityGraphics`, `ContourGraphics` and `GraphicsArray` graphics objects. In most cases, a graphics primitive will retain its original identity after applying any of the transformation functions, however there are a few exceptions. Namely, `Rectangle`, `Cuboid`, `Raster` and

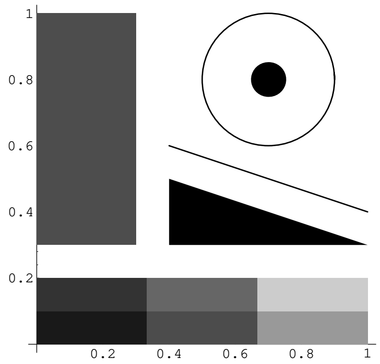
`RasterArray` graphics primitives are converted to equivalent lists of `Polygon` primitives when used with `RotateGraphics`, `TransformGraphics` and `SkewGraphics`.

This loads the package.

```
In[1]:= <<Graphics`GraphicsOperations`
```

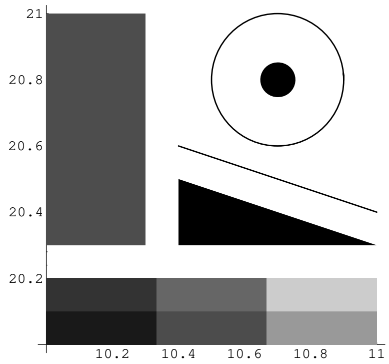
This creates a two-dimensional graphics object of some common graphics primitives.

```
In[2]:= graph = Show[ Graphics[{
        {GrayLevel[0.3], Rectangle[{0,.3},{.3,1}]},
        Raster[{{0.1,.3,.6},{0.2,.4,.8}},{{0,0},{1,.2}}],
        Polygon[{{0.4,.3},{1,.3},{0.4,.5}}],
        Line[{{0.4,.6},{1,.4}}],
        Circle[{.7,.8},.2],
        {PointSize[0.1],Point[{.7,.8}]} }],
            AspectRatio->Automatic, Axes->True]
```



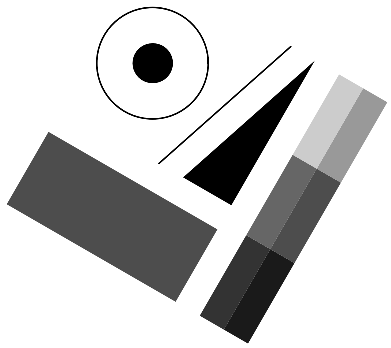
Translate the graphics by the vector `{10,20}`.

```
In[3]:= Show[ TranslateGraphics[graph, {10,20}]]
```



Rotate the graphics counterclockwise about the origin.

```
In[4]:= Show[ RotateGraphics[graph, -Pi/3], Axes->False]
```

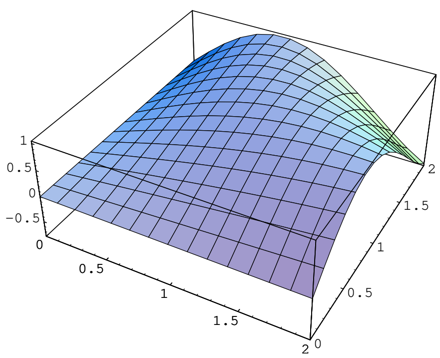


Scale the graphics in the horizontal direction by a factor of 2.

```
In[5]:= Show[ ScaleGraphics[graph, {2,1}], Axes->False]
```

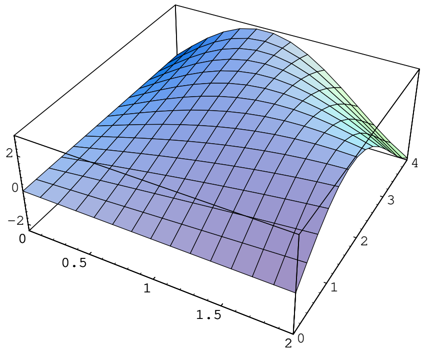
This creates a three-dimensional plot.

```
In[6]:= graph = Plot3D[ Sin[x y], {x,0,2},{y,0,2}]
```



Scale the plot in the y and z directions by factors of 2 and 3 respectively.

```
In[7]:= Show[ ScaleGraphics[graph, {1,2,3}]]
```



The transformation functions will work with any expression and will affect only graphics objects and graphics primitives which are the expression itself, elements of a `List`, or elements of a nested `List`. This is useful when transforming graphics primitives that are not already within `Graphics` or `Graphics3D` graphics objects.

Only the graphics primitives `Line` and `Point` are affected by the translation. Note that you can even apply the translation to an expression that contains both two and three-dimensional graphics primitives.

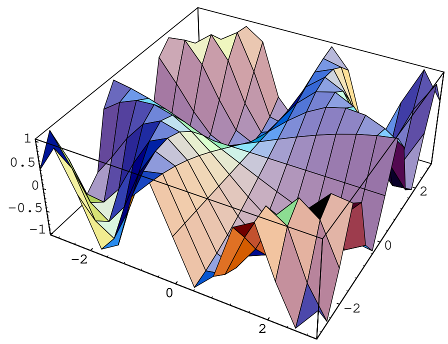
```
In[8]:= TranslateGraphics[{Line[{{0,0},{1,2}}],
                              PointSize[0.01], Point[{1,1,1}]},
          {1,0,7}]
Out[8]= {Line[{{1, 0}, {2, 2}}], PointSize[0.01],
  Point[{2, 1, 8}]}
```

`TakeGraphics[`*g*`, {{`*xmin*`,`*ymin*`},{`*xmax*`,`*ymax*`}}]`	extract all graphics primitives in the two-dimensional graphics object *g* within the rectangular region `{{`*xmin*`,`*ymin*`},{`*xmax*`,`*ymax*`}}`
`TakeGraphics[`*g*`, {{`*xmin*`,`*ymin*`,`*zmin*`},{`*xmax*`,`*ymax*`,`*zmax*`}}]`	extract all graphics primitives in the three-dimensional graphics object *g* within the cube `{{`*xmin*`,`*ymin*`,`*zmin*`},{`*xmax*`,`*ymax*`,`*zmax*`}}`
`DropGraphics[`*g*`, {{`*xmin*`,`*ymin*`},{`*xmax*`,`*ymax*`}}]`	remove all graphics primitives in the two-dimensional graphics object *g* within the rectangular region `{{`*xmin*`,`*ymin*`},{`*xmax*`,`*ymax*`}}`
`DropGraphics[`*g*`, {{`*xmin*`,`*ymin*`,`*zmin*`},{`*xmax*`,`*ymax*`,`*zmax*`}}]`	remove all graphics primitives in the three-dimensional graphics object *g* within the cube `{{`*xmin*`,`*ymin*`,`*zmin*`},{`*xmax*`,`*ymax*`,`*zmax*`}}`

Extracting and removing graphics.

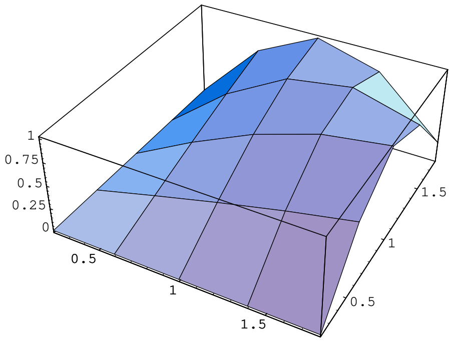
This creates a three-dimensional plot.

```
In[9]:= graph = Plot3D[ Sin[x y], {x,-3,3},{y,-3,3}]
```



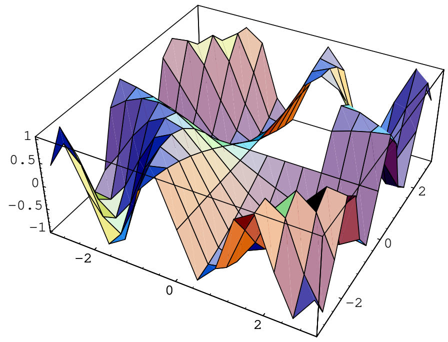
This extracts the graphics within the x-y region `{{0,0},{2,2}}` where the default limits on z are `{-Infinity,Infinity}`.

```
In[10]:= Show[ TakeGraphics[graph, {{0,0},{2,2}}]]
```



This removes the graphics from that same region. Note that `SurfaceGraphics` is converted to `Graphics3D` in order to be able to remove parts of the mesh.

```
In[11]:= Show[ DropGraphics[graph, {{0,0},{2,2}}]]
```



option name	*default value*	
`ModifyGraphics`	`False`	determines whether primitives will be broken up or modified
`RegionTest`	`Inset`	method for determining when primitives are within a region

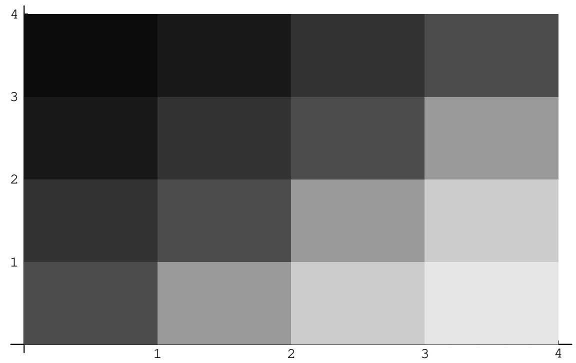
Options for `TakeGraphics` and `DropGraphics`.

Using the default setting of `ModifyGraphics -> False` insures that the original graphics primitives are not broken up in any way. The exception to this is the `Line` graphics primitive because one `Line` primitive is typically generated from a call to two-dimensional plotting functions and we would like to break this single `Line` primitive up even with a setting of `ModifyGraphics -> False`. There are two methods for determining whether a graphics primitive is within the specified region or not. With the default setting of `RegionTest -> Inset`, a graphics primitive is considered within a specified region if all coordinates describing the primitive are within the region. With the setting `RegionTest -> Intersect`, a graphics primitive is considered within a region if at least one of the coordinates describing the primitive is within the region.

When you want more exact cuts through the graphics primitives, use `ModifyGraphics -> True`. In this case, most graphics primitives will be broken up exactly along the border of the region being extracted or removed, except, for instance, in cases such as `Circle`, `Disk` and `Text` primitives which are never broken up. With a setting of `ModifyGraphics -> True` the option `RegionTest` is usually ignored since we are finding an exact intersection with the region boundary and do not need to test whether the entire primitive is within the region or not.

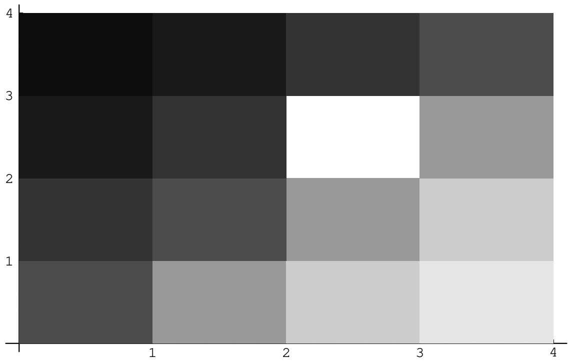
This creates a graphics object of one `Raster` primitive.

```
In[12]:= graph = Show[ Graphics[
           Raster[{{0.3,0.6,0.8,0.9},{0.2,0.3,0.6,0.8},
             {0.1,0.2,0.3,0.6},{0.05,0.1,0.2,0.3}}]], Axes->True]
```



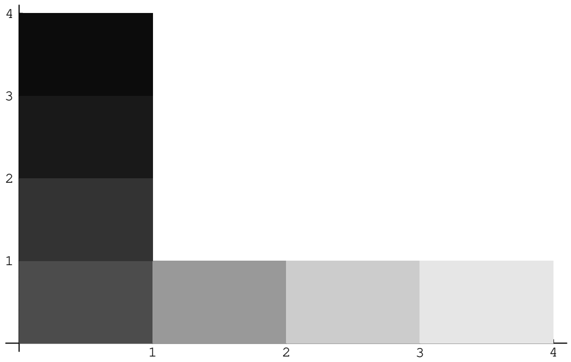
This shows the result of removing the graphics within the region `{{1.5,1.5},{3.5,3.5}}` with the default setting of `RegionTest`.

```
In[13]:= Show[ DropGraphics[graph, {{1.5,1.5},{3.5,3.5}},
         RegionTest->Inset]]
```



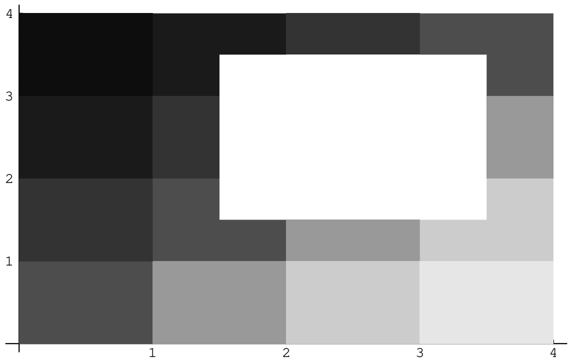
Notice the difference of what is removed with the same region by using the `Intersect` method for `RegionTest`.

```
In[14]:= Show[ DropGraphics[graph, {{1.5,1.5},{3.5,3.5}},
         RegionTest->Intersect]]
```



Here is an exact removal of the specified region. In this case, the `Raster` primitive is converted to an equivalent list of `Polygon` primitives in order to get an exact removal of the region.

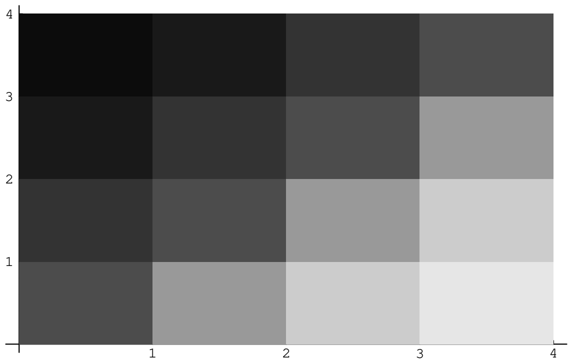
```
In[15]:= Show[ DropGraphics[graph, {{1.5,1.5},{3.5,3.5}},
           ModifyGraphics->True]]
```



The graphics object `DensityGraphics` and the graphics primitive `Raster` are both represented by two-dimensional graphics. However, they both also contain information about a third dimension. For this reason, a range from the third dimension can also be extracted or removed from `DensityGraphics` and `Raster` using `TakeGraphics` or `DropGraphics`. The resulting graphics will be `Polygon` primitives contained in a `Graphics` graphics object. There is currently no option for extracting or removing a z range from `ContourGraphics`.

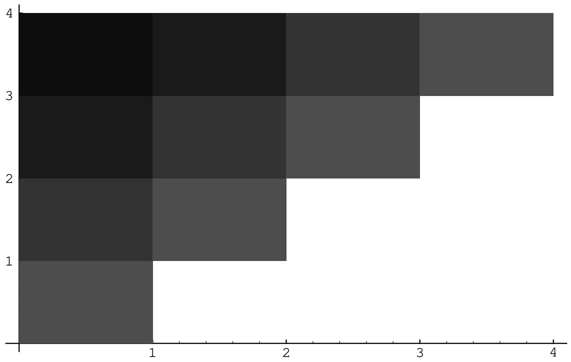
This creates a graphics object of one `Raster` primitive.

```
In[16]:= graph = Show[ Graphics[
           Raster[{{0.3,0.6,0.8,0.9},{0.2,0.3,0.6,0.8},
             {0.1,0.2,0.3,0.6},{0.05,0.1,0.2,0.3}}]], Axes->True]
```



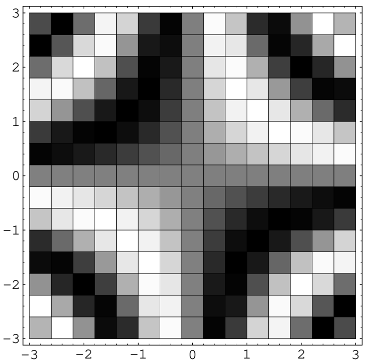
This shows the result of taking the graphics within the z range `{-Infinity,0.5}`.

```
In[17]:= Show[ TakeGraphics[graph,
           {{-Infinity,-Infinity,-Infinity},
              {Infinity, Infinity, 0.5}}]]
```



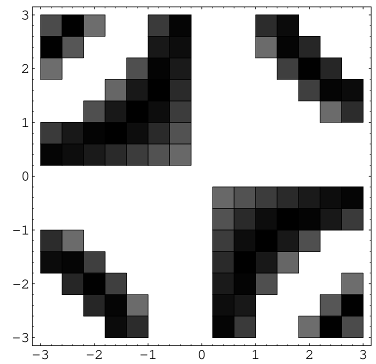
This creates a `DensityGraphics` graphics object.

```
In[18]:= graph = DensityPlot[ Sin[x y], {x,-3,3}, {y,-3,3}]
```



This shows the result of taking the cells within the z range `{-Infinity,-0.01}`.

```
In[19]:= Show[ TakeGraphics[graph,
            {{-Infinity,-Infinity,-Infinity},
               {Infinity,Infinity,-0.01}}]]
```



Just as with the transformation functions, `TakeGraphics` and `DropGraphics` will work with any expression and will affect only graphics objects and graphics primitives which are the expression itself, elements of a `List`, or elements of a nested `List`. This is useful when working with graphics primitives that are not already within `Graphics` or `Graphics3D` graphics objects.

Only the graphics primitives `Line` and `Point` are affected by `DropGraphics`. Note that `DropGraphics` cen even be applied to expressions that contain both two and three-dimensional graphics primitives.

```
In[20]:= DropGraphics[{Line[{{0,0},{7,7}}], PointSize[0.01],
            Point[{1,3,1}]},
               {{1,1},{4,4}}, ModifyGraphics->True]
Out[20]= {{Line[{{0, 0}, {1, 1}}],
    Line[{{4, 4}, {7, 7}}]}, PointSize[0.01]}
```

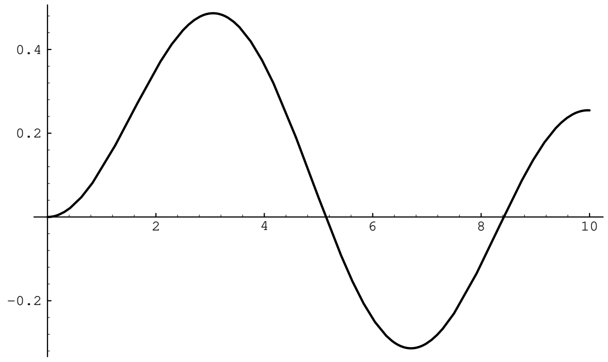
`SliceGraphics[`g`, `n`]`	slice the graphics object g into n slices along the x axis
`SliceGraphics[`g`, `n`, Axis->{`x`,`y`,`z`}]`	slice the graphics object g into n slices along the axis specified by the vector from the origin to the point `{`x`,`y`,`z`}`

Slicing graphics objects.

`SliceGraphics` actually calls `TakeGraphics` and `TranslateGraphics` and, if necessary, `RotateGraphics` to create the slicing of graphics objects. Thus, `SliceGraphics` inherits the options of `TakeGraphics`.

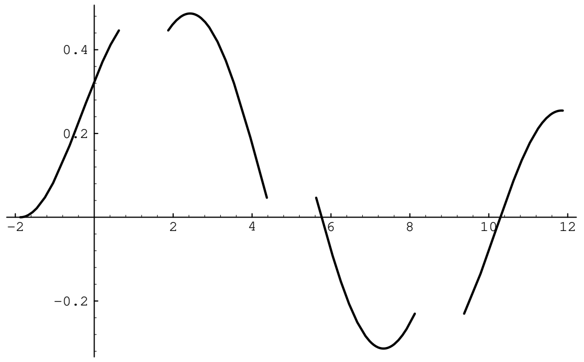
This creates a plot of a Bessel function.

```
In[21]:= graph = Plot[ BesselJ[2,x], {x,0,10}]
```



This breaks the plot up into 4 slices and spreads them apart. `ModifyGraphics` is used to make sure no line segments are lost.

```
In[22]:= Show[ SliceGraphics[graph, 4, ModifyGraphics->True]]
```



option name	*default value*	
`Axis`	`{1,0,0}`	axis along which the slices are taken
`SliceSpacing`	`Automatic`	the spacing between slices
`SliceRatios`	`Automatic`	the size of each individual slice
`ModifyGraphics`	`False`	determines whether primitives will be broken up or modified
`RegionTest`	`Inset`	method for determining when primitives are within a region

Options for `SliceGraphics`.

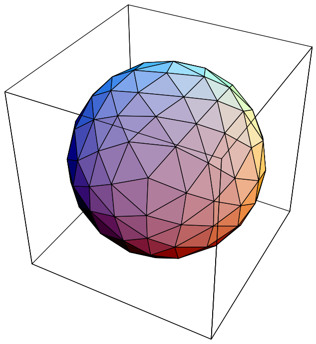
Only the direction of the option `Axis` is used, and thus, only the relative magnitudes of the vector components are significant. The default values of the options `SliceSpacing` and `SliceRatios` are determined from the number of slices and the total spacial extent of all graphics primitives. `SliceSpacing ->` d will evenly space the slices by a distance d. `SliceSpacing -> {`d_1`,` d_2`, ...}`, where `{`d_1`,` d_2`, ...}` is a list of length n ϒ 1 where n is the number of slices, will separate the corresponding slices by a distance d_i. `SliceSpacing` is measured along the direction specified by `Axis`. `SliceRatios -> {`r_1`,` r_2`, ...}`, where `{`r_1`,` r_2`, ...}` is a list of ratios of length n where n is the number of slices, can be used to make a set of slices with varying sizes. `SliceGraphics` returns a list of graphics objects representing each slice which can be displayed together using `Show` or acted on separately if desired. Note that `SliceSpacing -> 0` can be used to get a list of slices without the side effect of translating any of the slices.

This loads a package which defines various polyhedra.

```
In[23]:= <<Graphics`Polyhedra`
```

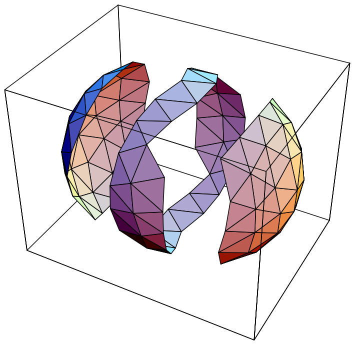
This creates a geodesic from a dodecahedron.

```
In[24]:= graph = Show[ Geodesate[ Polyhedron[Dodecahedron], 2]]
```



Here we slice the sphere up into three sections relying on `RegionTest` to determine the `Polygon` primitives contained in each region.

```
In[25]:= Show[ SliceGraphics[graph, 3]]
```



Here we get an exact slice through the sphere using `ModifyGraphics -> True`, but at the expense of the time required to break up `Polygon` primitives.

```
In[26]:= Show[ SliceGraphics[graph, 3, ModifyGraphics->True]]
```