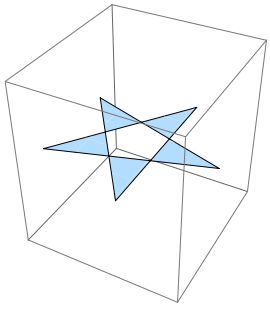
Self-intersecting nonconvex polygons are filled according to an even-odd rule that alternates between filling and not filling at each crossing.

In[11]:=
```
Graphics3D[Polygon[Table[{Cos[2 π k / 5], Sin[2 π k / 5], 0}, {k, 0, 8, 2}]]]
```

Out[11]=



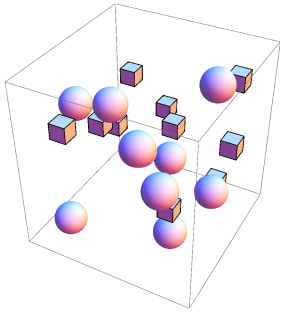
`Cuboid[{x,y,z}]`	a unit cube with opposite corners having coordinates $\{x, y, z\}$ and $\{x+1, y+1, z+1\}$
`Cuboid[{`x_{min}`,`y_{min}`,`z_{min}`},{`x_{max}`,`y_{max}`,`z_{max}`}]`	a cuboid (rectangular parallelepiped) with opposite corners having the specified coordinates
`Cylinder[{`x_1`,`y_1`,`z_1`},{`x_2`,`y_2`,`z_2`}]`	a cylinder of radius 1 with endpoints at $\{x_1, y_1, z_1\}$ and $\{x_2, y_2, z_2\}$
`Cylinder[{`x_1`,`y_1`,`z_1`},{`x_2`,`y_2`,`z_2`},`r`]`	a cylinder of radius r
`Sphere[{x,y,z}]`	a unit sphere centered at $\{x, y, z\}$
`Sphere[{x,y,z},`r`]`	a sphere of radius r

Cuboid graphics elements.

This draws a number of random unit cubes and spheres in three-dimensional space.

In[12]:=
```
Graphics3D[Table[{Cuboid[10 rcoord], Sphere[10 rcoord]}, {10}]]
```

Out[12]=

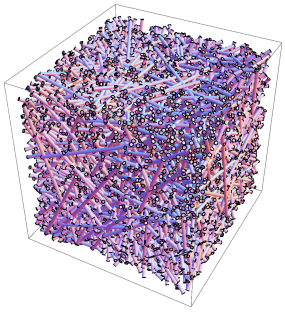


Even though `Cylinder` and `Sphere` produce high-quality renderings, their usage is scalable. A single image can contain thousands of these primitives. When rendering so many primitives, you can increase the efficiency of rendering by using special options to change the number of points used by default to render `Cylinder` and `Sphere`. The `"CylinderPoints" Method` option to `Graphics3D` is used to reduce the rendering quality of each individual cylinder. Sphere quality can be similarly adjusted using `"SpherePoints"`.

Because the cylinders are so small, the number of points used to render them can be reduced with almost no perceptible change.

In[13]:=
```
Graphics3D[Table[Cylinder[{rcoord, rcoord}, .01], {10 000}],
 Method → {"CylinderPoints" → 6}]
```

Out[13]=



Three-Dimensional Graphics Directives

In three dimensions, just as in two dimensions, you can give various graphics directives to specify how the different elements in a graphics object should be rendered.

All the graphics directives for two dimensions also work in three dimensions. There are however some additional directives in three dimensions.

Just as in two dimensions, you can use the directives `PointSize`, `Thickness` and `Dashing` to tell *Mathematica* how to render `Point` and `Line` elements. Note that in three dimensions, the lengths that appear in these directives are measured as fractions of the total width of the display area for your plot.

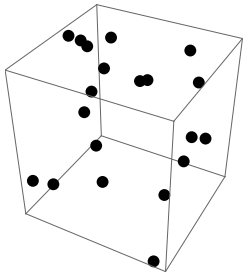
This generates a list of 20 random points in three dimensions.

In[1]:=
```
pts = Table[Point[Table[RandomReal[], {3}]], {20}];
```

This displays the points, with each one being a circle whose diameter is 5% of the display area width.

```
In[2]:= Graphics3D[{PointSize[0.05], pts}]
```

Out[2]=



As in two dimensions, you can use `AbsolutePointSize`, `AbsoluteThickness` and `AbsoluteDashing` if you want to measure length in absolute units.

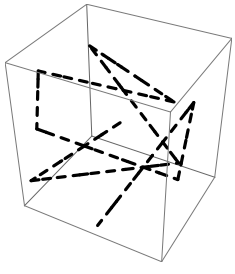
This generates a line through 10 random points in three dimensions.

```
In[3]:= line = Line[Table[RandomReal[], {10}, {3}]];
```

This shows the line dashed, with a thickness of 2 printer's points.

```
In[4]:= Graphics3D[{AbsoluteThickness[2], AbsoluteDashing[{5, 5}], line}]
```

Out[4]=



For `Point` and `Line` objects, the color specification directives also work the same in three dimensions as in two dimensions. For `Polygon` objects, however, they can work differently.

In two dimensions, polygons are always assumed to have an intrinsic color, specified directly by graphics directives such as `RGBColor` and `Opacity`. In three dimensions, however, *Mathematica* generates colors for polygons using a more physical approach based on simulated illumination. Polygons continue to have an intrinsic color defined by color directives, but the final color observed when rendering the graphic may be different based upon the values of the lights shining on the polygon. Polygons are intrinsically white by default.

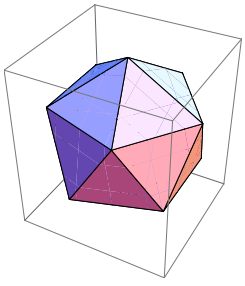
`Lighting->Automatic`	use default light placements and values
`Lighting->None`	disable all lights
`Lighting->"Neutral"`	light using only white light sources

Some schemes for coloring polygons in three dimensions.

This draws an icosahedron with default lighting. The intrinsic color value of the polygons is white.

```
In[5]:= Graphics3D[{PolyhedronData["Icosahedron", "Faces"]}]
```

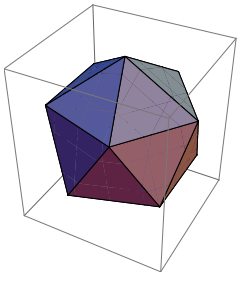
Out[5]=



This draws the icosahedron using the same lighting parameters, but defines the intrinsic color value of the polygons to be gray.

```
In[6]:= Graphics3D[{Gray, PolyhedronData["Icosahedron", "Faces"]}]
```

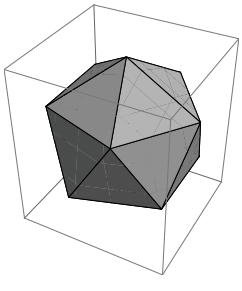
Out[6]=



The intrinsic color value of the polygons becomes more obvious when using the `"Neutral"` lighting scheme.

```
In[7]:= Graphics3D[{Gray, PolyhedronData["Icosahedron", "Faces"]}, Lighting → "Neutral"]
```

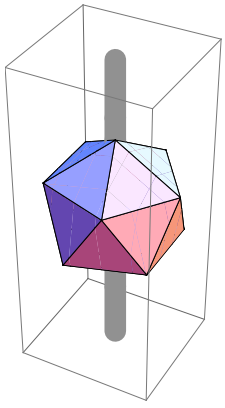
Out[7]=



This applies the gray color only to the line, which is not affected by the lights.

```
In[8]:= Graphics3D[{PolyhedronData["Icosahedron", "Faces"],
          Gray, Thickness[0.1], Line[{{0, 0, -2}, {0, 0, 2}}]}]
```

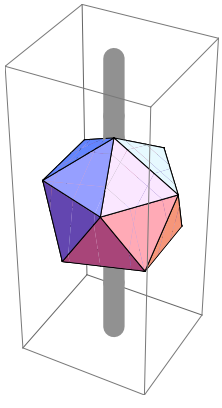
Out[8]=



As with two-dimensional directives, the color directive can be scoped to the line by using a sublist.

```
In[9]:= Graphics3D[{{Gray, Thickness[0.1], Line[{{0, 0, -2}, {0, 0, 2}}]},
          PolyhedronData["Icosahedron", "Faces"]}]
```

Out[9]=



`EdgeForm[]`	draw no lines at the edges of polygons
`EdgeForm[`g`]`	use the graphics directives g to determine how to draw lines at the edges of polygons

Giving graphics directives for all the edges of polygons.

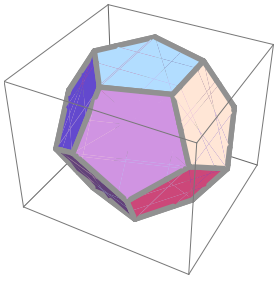
When you render a three-dimensional graphics object in *Mathematica*, there are two kinds of lines that can appear. The first kind are lines from explicit `Line` primitives that you included in the graphics object. The second kind are lines that were generated as the edges of polygons.

You can tell *Mathematica* how to render all lines of the second kind by giving a list of graphics directives inside `EdgeForm`.

This renders a dodecahedron with its edges shown as thick gray lines.

```
In[10]:= Graphics3D[{EdgeForm[{GrayLevel[0.5], Thickness[0.02]}],
           PolyhedronData["Dodecahedron", "Faces"]}]
```

Out[10]=



`FaceForm[`*gfront*`,`*gback*`]`	use *gfront* graphics directives for the front face of each polygon, and *gback* for the back

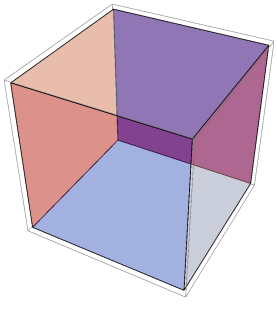
Rendering the fronts and backs of polygons differently.

An important aspect of polygons in three dimensions is that they have both front and back faces. *Mathematica* uses the following convention to define the "front face" of a polygon: if you look at a polygon from the front, then the corners of the polygon will appear counterclockwise, when taken in the order that you specified them.

This makes the front (outside) face of each polygon mostly transparent, and the back (inside) face fully opaque.

```
In[11]:= Graphics3D[{FaceForm[{Opacity[0.3]}, White], PolyhedronData["Cube", "Faces"]}]
```

Out[11]=



Coordinate Systems for Three-Dimensional Graphics

Whenever *Mathematica* draws a three-dimensional object, it always effectively puts a cuboidal box around the object. With the default option setting `Boxed -> True`, *Mathematica* in fact draws the edges of this box explicitly. But in general, *Mathematica* automatically "clips" any parts of your object that extend outside of the cuboidal box.

The option `PlotRange` specifies the range of x, y and z coordinates that *Mathematica* should include in the box. As in two dimensions the default setting is `PlotRange -> Automatic`, which makes *Mathematica* use an internal algorithm to try and include the "interesting parts" of a plot, but drop outlying parts. With `PlotRange -> All`, *Mathematica* will include all parts.

This loads a package defining polyhedron operations.

In[1]:=
```
<< PolyhedronOperations`
```

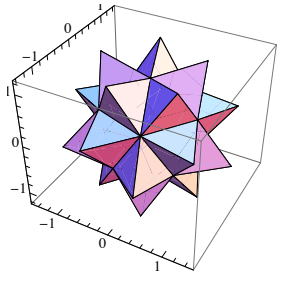
This creates a stellated icosahedron.

In[2]:=
```
stel = Stellate[PolyhedronData["Icosahedron", "Faces"]];
```

This shows the stellated icosahedron, drawn in a box.

In[3]:=
```
Graphics3D[stel, Axes → True]
```

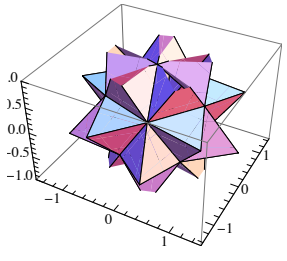
Out[3]=



With this setting for `PlotRange`, many parts of the stellated icosahedron lie outside the box, and are clipped.

In[4]:=
```
Show[%, PlotRange -> {-1, 1}]
```

Out[4]=



Much as in two dimensions, you can use either "original" or "scaled" coordinates to specify the positions of elements in three-dimensional objects. Scaled coordinates, specified as `Scaled[{sx, sy, sz}]` are taken to run from 0 to 1 in each dimension. The coordinates are set up to define a right-handed coordinate system on the box.

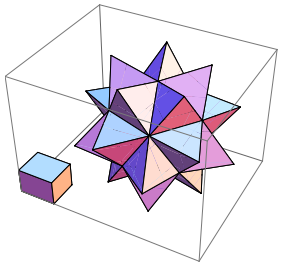
`{x,y,z}`	original coordinates
`Scaled[{sx,sy,sz}]`	scaled coordinates, running from 0 to 1 in each dimension

Coordinate systems for three-dimensional objects.

This puts a cuboid in one corner of the box.

```
In[5]:= Graphics3D[{stel, Cuboid[Scaled[{0, 0, 0}], Scaled[{0.2, 0.2, 0.2}]]}]
```

Out[5]=



Once you have specified where various graphical elements go inside a three-dimensional box, you must then tell *Mathematica* how to draw the box. The first step is to specify what shape the box should be. This is analogous to specifying the aspect ratio of a two-dimensional plot. In three dimensions, you can use the option `BoxRatios` to specify the ratio of side lengths for the box. For `Graphics3D` objects, the default is `BoxRatios -> Automatic`, specifying that the shape of the box should be determined from the ranges of actual coordinates for its contents.

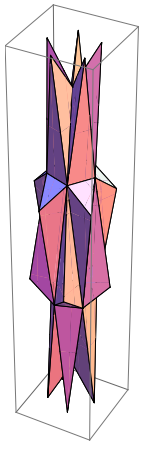
`BoxRatios->{xr,yr,zr}`	specify the ratio of side lengths for the box
`BoxRatios->Automatic`	determine the ratio of side lengths from the range of actual coordinates (default for `Graphics3D`)

Specifying the shape of the bounding box for three-dimensional objects.

This displays the stellated icosahedron in a tall box.

```
In[6]:= Graphics3D[stel, BoxRatios -> {1, 1, 5}]
```

Out[6]=



To produce an image of a three-dimensional object, you have to tell *Mathematica* from what view point you want to look at the object. You can do this using the option `ViewPoint`.

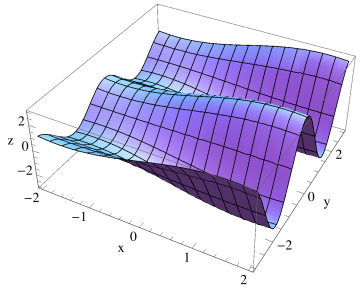
Some common settings for this option were given in "Three-Dimensional Surface Plots". In general, however, you can tell *Mathematica* to use any view point.

View points are specified in the form `ViewPoint -> {`sx`, `sy`, `sz`}`. The values si are given in a special coordinate system, in which the center of the box is `{0, 0, 0}`. The special coordinates are scaled so that the longest side of the box corresponds to one unit. The lengths of the other sides of the box in this coordinate system are determined by the setting for the `BoxRatios` option. For a cubical box, therefore, each of the special coordinates runs from $-1/2$ to $1/2$ across the box. Note that the view point must always lie outside the box.

This generates a picture using the default view point `{1.3, -2.4, 2}`.

```
In[7]:= surf = Plot3D[(2 + Sin[x]) Cos[2 y],
          {x, -2, 2}, {y, -3, 3}, AxesLabel -> {"x", "y", "z"}]
```

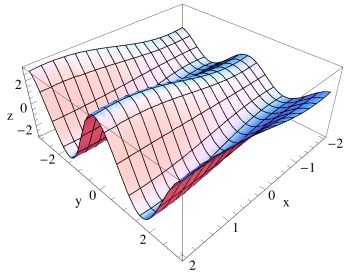
Out[7]=



This is what you get with a view point close to one of the corners of the box.

```
In[8]:= Show[surf, ViewPoint -> {1.2, 1.2, 1.2}]
```

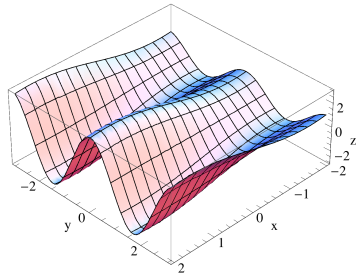
Out[8]=



As you move away from the box, the perspective effect gets smaller.

```
In[9]:= Show[surf, ViewPoint -> {5, 5, 5}]
```

Out[9]=



option name	*default value*	
`ViewPoint`	`{1.3,-2.4,2}`	the point in a special scaled coordinate system from which to view the object
`ViewCenter`	`Automatic`	the point in the scaled coordinate system which appears at the center of the final image
`ViewVertical`	`{0,0,1}`	the direction in the scaled coordinate system which appears as vertical in the final image
`ViewAngle`	`Automatic`	the opening half-angle for a simulated camera used to view the graphic
`ViewVector`	`Automatic`	the position and direction of the simulated camera in the graphic's regular coordinate system

Specifying the position and orientation of three-dimensional objects.

In making a picture of a three-dimensional object you have to specify more than just *where* you want to look at the object from. You also have to specify how you want to "frame" the object in your final image. You can do this using the additional options `ViewCenter`, `ViewVertical` and `ViewAngle`.

`ViewCenter` allows you to tell *Mathematica* what point in the object should appear at the center of your final image. The point is specified by giving its scaled coordinates, running from 0 to 1 in each direction across the box. With the setting `ViewCenter -> {1 / 2, 1 / 2, 1 / 2}`, the center of the box will therefore appear at the center of your final image. With many choices of view point, however, the box will not appear symmetrical, so this setting for `ViewCenter` will not center the whole box in the final image area. You can do this by setting `ViewCenter -> Automatic`.

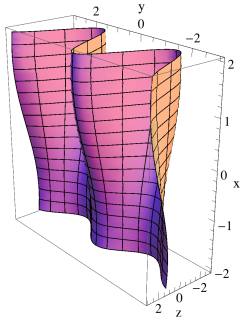
`ViewVertical` specifies which way up the object should appear in your final image. The setting for `ViewVertical` gives the direction in scaled coordinates which ends up vertical in the final image. With the default setting `ViewVertical -> {0, 0, 1}`, the z direction in your original coordinate system always ends up vertical in the final image.

Mathematica uses the properties of a simulated camera to visualize the final image. The position, orientation, and facing of the camera are determined by the `ViewCenter`, `ViewVertical`, and `ViewPoint` options. The `ViewAngle` option specifies the width of the opening of the camera lens. The `ViewAngle` specifies, in radians, the maximum angle from the line stretching from the `ViewPoint` to the `ViewCenter` which can be viewed by the camera. The effective viewing angle is double the value of `ViewAngle`. This means that ViewAngle can effectively be used to zoom in on a part of the image. The default value of `ViewAngle` resolves to 35°, which is the typical viewing angle for the human eye.

This setting for `ViewVertical` makes the x axis of the box appear vertical in your image.

In[10]:=
```
Show[surf, ViewVertical -> {1, 0, 0}]
```

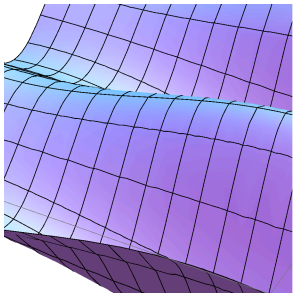
Out[10]=



This uses ViewAngle to effectively zoom in on the center of the image.

```
In[11]:= Show[surf, ViewAngle → 10 Degree]
```

Out[11]=



When you set the options ViewPoint, ViewCenter and ViewVertical, you can think about it as specifying how you would look at a physical object. ViewPoint specifies where your head is relative to the object. ViewCenter specifies where you are looking (the center of your gaze). And ViewVertical specifies which way up your head is.

In terms of coordinate systems, settings for ViewPoint, ViewCenter and ViewVertical specify how coordinates in the three-dimensional box should be transformed into coordinates for your image in the final display area.

ViewVector->Automatic	uses the values of the ViewPoint and ViewCenter options to determine the position and facing of the simulated camera
ViewVector->{x,y,z}	position of the camera in the coordinates used for objects; the facing of the camera is determined by the ViewCenter option
ViewVector->{{x,y,z},{tx,ty,tz}}	position of the camera and of the point the camera is focused on in the coordinates used for objects

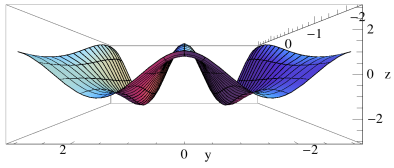
Possible values of the ViewVector option.

The position and facing of the camera can be fully determined by the ViewPoint and ViewCenter options, but the ViewVector option offers a useful generalization. Instead of specifying the position and facing of the camera using scaled coordinates, ViewVector provides the ability to position the camera using the same coordinate system used to position objects within the graphic.

This specifies that the camera should be placed on the negative x axis and facing toward the center of the graphic.

```
In[12]:= Show[surf, ViewVector → {-5, 0, 0}]
```

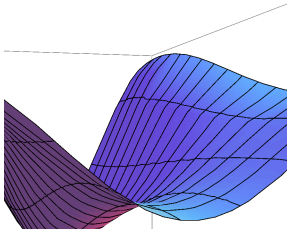
Out[12]=



The camera is in the same position but pointing in a different direction. In combination with `ViewAngle`, this zooms in on a particular section of the graphic.

```
In[13]:= Show[surf, ViewVector → {{-5, 0, 0}, {2, -3, 2}}, ViewAngle → 20 Degree]
```

Out[13]=



Once you have obtained a two-dimensional image of a three-dimensional object, there are still some issues about how this image should be rendered. The issues however are identical to those that occur for two-dimensional graphics. Thus, for example, you can modify the final shape of your image by changing the `AspectRatio` option. And you specify what region of your whole display area your image should take up by setting the `PlotRegion` option.

drag	rotate the graphic about its center
Ctrl+drag	zoom into or out of the graphic
Shift+drag	pan across the graphic in the plane of the screen

Mouse gestures used for interacting with three-dimensional graphics.

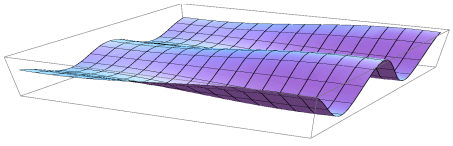
When interactively modifying graphics, *Mathematica* makes changes to the view options. If you have specified the position of the camera using `ViewPoint`, then rotating the graphic causes *Mathematica* to change the value of the `ViewPoint` option. If the position of the camera is specified using `ViewVector`, interactive rotation will instead change the value of that option. In both cases, interactive rotation can also affect the value of the `ViewVertical` option. Interac-

tive zooming of the graphic corresponds directly to changing the `ViewAngle` option. Interactively panning the graphic changes values of the `ViewCenter` option.

This modifies the aspect ratio of the final image.

```
In[14]:= Show[surf, Axes -> False, AspectRatio -> 0.3]
```

Out[14]=

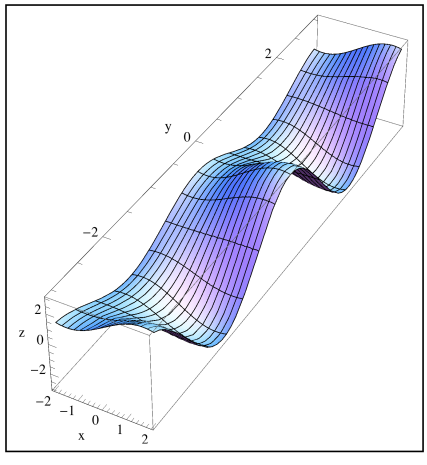


Mathematica usually scales the images of three-dimensional objects to be as large as possible, given the display area you specify. Although in most cases this scaling is what you want, it does have the consequence that the size at which a particular three-dimensional object is drawn may vary with the orientation of the object. You can set the option `SphericalRegion -> True` to avoid such variation. With this option setting, *Mathematica* effectively puts a sphere around the three-dimensional bounding box, and scales the final image so that the whole of this sphere fits inside the display area you specify. The sphere has its center at the center of the bounding box, and is drawn so that the bounding box just fits inside it.

This draws a rather elongated version of the plot.

```
In[15]:= Framed[Show[surf, BoxRatios -> {1, 5, 1}]]
```

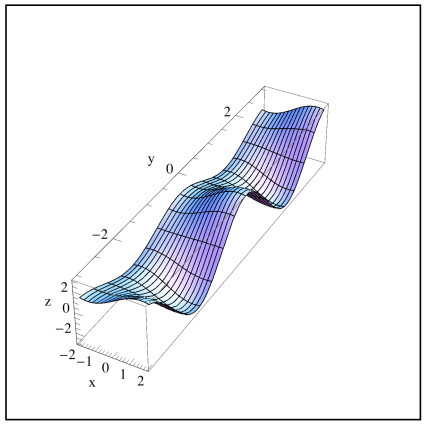
Out[15]=



With `SphericalRegion -> True`, the final image is scaled so that a sphere placed around the bounding box would fit in the display area.

```
In[16]:= Framed[Show[surf, BoxRatios -> {1, 5, 1}, SphericalRegion -> True]]
```

Out[16]=



By setting `SphericalRegion -> True`, you can make the scaling of an object consistent for all orientations of the object. This is useful if you create animated sequences which show a particular object in several different orientations.

`SphericalRegion->False`	scale three-dimensional images to be as large as possible
`SphericalRegion->True`	scale images so that a sphere drawn around the three-dimensional bounding box would fit in the final display area

Changing the magnification of three-dimensional images.

Lighting and Surface Properties

With the default option setting `Lighting -> Automatic`, *Mathematica* uses a simulated lighting model to determine how to color polygons in three-dimensional graphics.

Mathematica allows you to specify various components to the illumination of an object. One component is the "ambient lighting", which produces uniform shading all over the object. Other components are directional, and produce different shading on different parts of the object. "Point lighting" simulates light emanating in all directions from one point in space. "Spot lighting" is similar to point lighting, but emanates a cone of light in a particular direction. "Directional lighting" simulates a uniform field of light pointing in the given direction. *Mathematica* adds together the light from all of these sources in determining the total illumination of a particular polygon.

{"Ambient", *color*}	uniform ambient lighting
{"Directional", *color*, {pos_1, pos_2}}	directional lighting parallel to the vector from pos_1 to pos_2
{"Point", *color*, *pos*}}	spherical point light source at position *pos*
{"Spot", *color*, {*pos*, *tar*}, α}	spotlight at position *pos* aimed at the target position *tar* with a half-angle opening of α
Lighting->{$light_1$, $light_2$, ...}	a number of lights

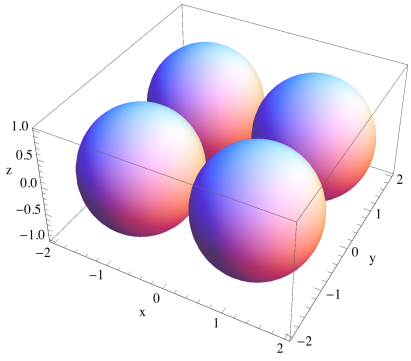
Methods for specifying light sources.

The default lighting used by *Mathematica* involves three point light sources, and no ambient component. The light sources are colored respectively red, green and blue, and are placed at 45° angles on the right-hand side of the object.

Here is a sphere, shaded using simulated lighting using the default set of lights.

```
In[1]:= spheres = Graphics3D[{Sphere[{-1, -1, 0}], Sphere[{-1, 1, 0}], Sphere[{1, 1, 0}],
          Sphere[{1, -1, 0}]}, Axes → True, AxesLabel → {"x", "y", "z"}]
```

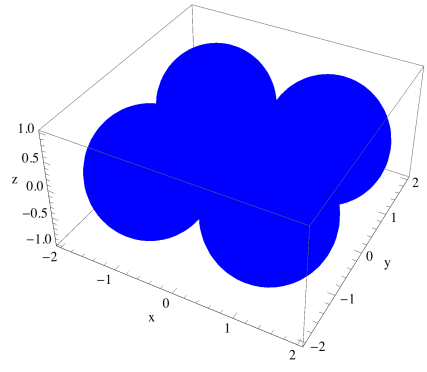
Out[1]=



This shows the result of adding ambient light, and removing all point light sources. Note the Lighting option takes a list of light sources.

```
In[2]:= Show[spheres, Lighting → {{"Ambient", Blue}}]
```

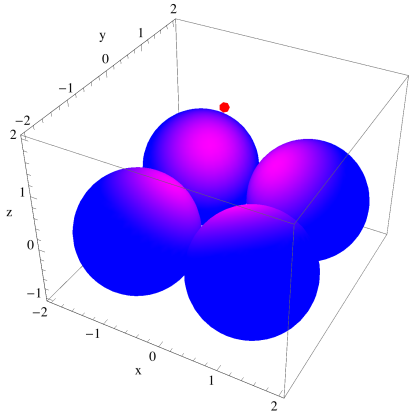
Out[2]=



This adds a single point light source positioned at the red point. The lights are combined as appropriate.

In[3]:=
```
Show[{spheres, Graphics3D[{Red, PointSize[Large], Point[{0, 0, 2}]}]},
 Lighting -> {{"Ambient", Blue}, {"Point", Red, {0, 0, 2}}}]
```

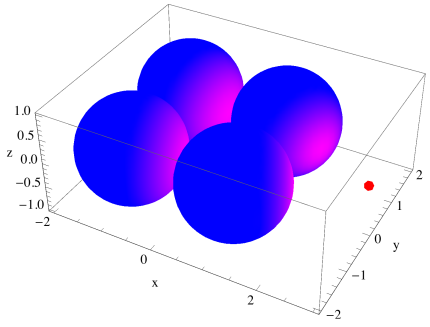
Out[3]=



Objects do not block light sources or cast shadows, so all objects in a scene will be lit evenly by light sources.

In[4]:=
```
Show[{spheres, Graphics3D[{Red, PointSize[Large], Point[{3, 0, 0}]}]},
 Lighting -> {{"Ambient", Blue}, {"Point", Red, {3, 0, 0}}}]
```

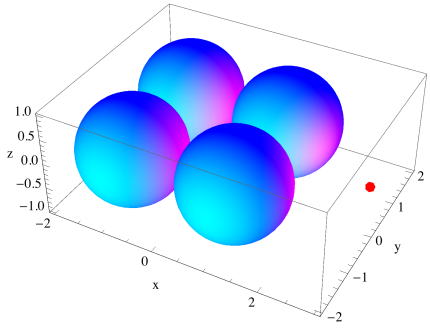
Out[4]=



This adds a directional green light shining from the negative y direction, effectively an infinite distance away.

In[5]:=
```
Show[%, Lighting -> {{"Ambient", Blue},
    {"Point", Red, {3, 0, 0}}, {"Directional", Green, {{0, 0, 0}, {0, 1, 0}}}}]
```

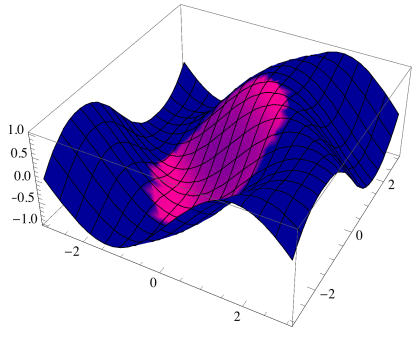
Out[5]=



This shows a spotlight positioned above the plot, combined with ambient lighting.

In[6]:=
```
Plot3D[Sin[x + Sin[y]], {x, -3, 3}, {y, -3, 3}, Lighting →
  {{"Ambient", RGBColor[0, 0, .6]}, {"Spot", Red, {{0, 0, 5}, {0, 0, 0}}, 15 Degree}}]
```

Out[6]=

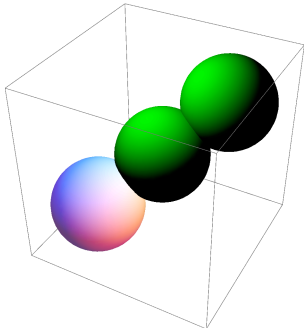


The `Lighting` option controls the lighting of all objects in a scene when used as an option to `Graphics3D` or `Show`. `Lighting` can also be used inline as a directive which specifies lighting for particular objects. The `Lighting` directive replaces the inherited lighting specifications.

The `Lighting` directive replaces the default value of `Lighting` for the two spheres after the directive.

```
In[7]:= Graphics3D[{Sphere[{0, 0, 0}],
          Lighting → {{"Point", Green, {0, 0, 5}}}, Sphere[{1, 1, 1}], Sphere[{2, 2, 2}]}]
```

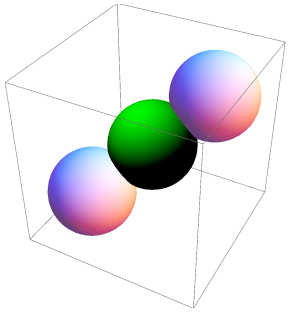
Out[7]=



This example uses list braces to restrict the effect of the `Lighting` directive to the middle sphere.

```
In[8]:= Graphics3D[{Sphere[{0, 0, 0}],
          {Lighting → {{"Point", Green, {0, 0, 5}}}, Sphere[{1, 1, 1}]}, Sphere[{2, 2, 2}]}]
```

Out[8]=



The perceived color of a polygon depends not only on the light which falls on the polygon, but also on how the polygon reflects that light. You can use the graphics directives `RGBColor`, `Specularity`, and `Glow` to specify the way that polygons reflect or emit light.

If you do not explicitly use these coloring directives, *Mathematica* effectively assumes that all polygons have matte white surfaces. Thus the polygons reflect light of any color incident on them, and do so equally in all directions. This is an appropriate model for materials such as uncoated white paper.

Using `RGBColor`, `Specularity`, and `Glow` , however, you can specify more complicated models. These directives separately specify three kinds of light emission: *diffuse reflection, specular reflection*, and *glow*.

In diffuse reflection, light incident on a surface is scattered equally in all directions. When this kind of reflection occurs, a surface has a "dull" or "matte" appearance. Diffuse reflectors obey Lambert's law of light reflection, which states that the intensity of reflected light is $\cos(\alpha)$ times the intensity of the incident light, where α is the angle between the incident light direction and the surface normal vector. Note that when $\alpha > 90^\circ$, there is no reflected light.

In specular reflection, a surface reflects light in a mirror-like way. As a result, the surface has a "shiny" or "gloss" appearance. With a perfect mirror, light incident at a particular angle is reflected at exactly the same angle. Most materials, however, scatter light to some extent, and so lead to reflected light that is distributed over a range of angles. *Mathematica* allows you to specify how broad the distribution is by giving a *specular exponent*, defined according to the Phong lighting model. With specular exponent n, the intensity of light at an angle θ away from the mirror reflection direction is assumed to vary like $\cos(\theta)^n$. As $n \to \infty$, therefore, the surface behaves like a perfect mirror. As n decreases, however, the surface becomes less "shiny", and for $n = 0$, the surface is a completely diffuse reflector. Typical values of n for actual materials range from about 1 to several hundred.

Glow is light radiated from a surface at a certain color and intensity of light that is independent of incident light.

Most actual materials show a mixture of diffuse and specular reflection, and some objects glow in addition to reflecting light. For each kind of light emission, an object can have an intrinsic color. For diffuse reflection, when the incident light is white, the color of the reflected light is the material's intrinsic color. When the incident light is not white, each color component in the reflected light is a product of the corresponding component in the incident light and in the intrinsic color of the material. Similarly, an object may have an intrinsic specular reflection color, which may be different from its diffuse reflection color, and the specularly reflected light is a component-wise product of the incident light and the intrinsic specular color. For glow, the color emitted is determined by intrinsic properties alone, with no dependence on incident light.

In *Mathematica*, you can specify light properties by giving any combination of diffuse reflection, specular reflection, and glow directives. To get no reflection of a particular kind, you may give the corresponding intrinsic color as `Black`, or `GrayLevel[0]`. For materials that are effectively "white", you can specify intrinsic colors of the form `GrayLevel[`a`]`, where a is the reflectance or albedo of the surface.

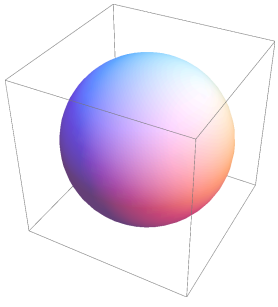
`GrayLevel[`*a*`]`	matte surface with albedo *a*
`RGBColor[`*r*`,`*g*`,`*b*`]`	matte surface with intrinsic color
`Specularity[`*spec*`,`*n*`]`	surface with specularity *spec* and specular exponent *n*; *spec* can be a number between 0 and 1 or an `RGBColor` specification
`Glow[`*col*`]`	glowing surface with color *col*

Specifying surface properties of lighted objects.

This shows a sphere with the default matte white surface, illuminated by several colored light sources.

```
In[9]:= Graphics3D[Sphere[]]
```

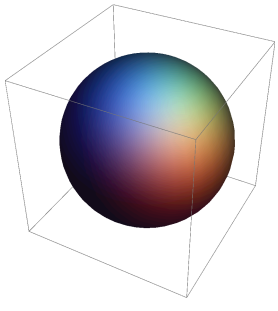
Out[9]=



This makes the sphere have low diffuse reflectance, but high specular reflectance. As a result, the sphere has a "specular highlight" near the light sources, and is quite dark elsewhere.

```
In[10]:= Graphics3D[{GrayLevel[0.2], Specularity[0.8, 5], Sphere[]}]
```

Out[10]=



When you set up light sources and surface colors, it is important to make sure that the total intensity of light reflected from a particular polygon is never larger than 1. You will get strange effects if the intensity is larger than 1.

Labeling Three-Dimensional Graphics

Mathematica provides various options for labeling three-dimensional graphics. Some of these options are directly analogous to those for two-dimensional graphics, discussed in "Labeling Two-Dimensional Graphics". Others are different.

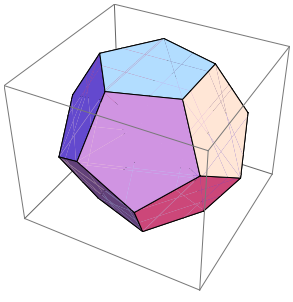
`Boxed->True`	draw a cuboidal bounding box around the graphics (default)
`Axes->True`	draw x, y and z axes on the edges of the box
`Axes->{False,False,True}`	draw the z axis only
`FaceGrids->All`	draw grid lines on the faces of the box
`PlotLabel->`*text*	give an overall label for the plot

Some options for labeling three-dimensional graphics.

The default for `Graphics3D` is to include a box, but no other forms of labeling.

```
In[1]:= Graphics3D[PolyhedronData["Dodecahedron", "Faces"]]
```

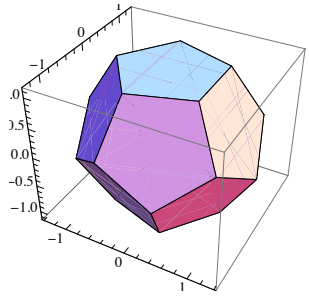
Out[1]=



Setting `Axes -> True` adds x, y and z axes.

```
In[2]:= Show[%, Axes -> True]
```

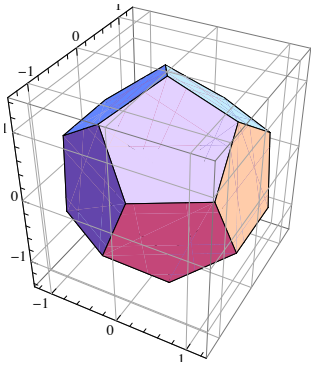
Out[2]=



This adds grid lines to each face of the box.

```
In[3]:= Show[%, FaceGrids -> All]
```

Out[3]=



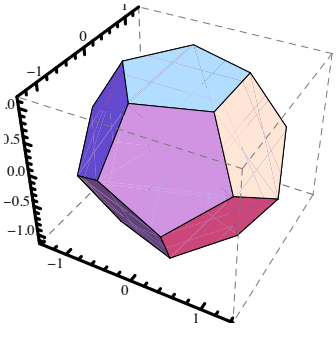
`BoxStyle->`*style*	specify the style for the box
`AxesStyle->`*style*	specify the style for axes
`AxesStyle->{`*xstyle*`,`*ystyle*`,`*zstyle*`}`	specify separate styles for each axis

Style options.

This makes the box dashed, and draws axes which are thicker than normal.

```
In[4]:= Graphics3D[PolyhedronData["Dodecahedron", "Faces"],
         BoxStyle -> Dashing[{0.02, 0.02}], Axes -> True, AxesStyle -> Thickness[0.01]]
```

Out[4]=



By setting the option `Axes -> True`, you tell *Mathematica* to draw axes on the edges of the three-dimensional box. However, for each axis, there are in principle four possible edges on which it can be drawn. The option `AxesEdge` allows you to specify on which edge to draw each of the axes.

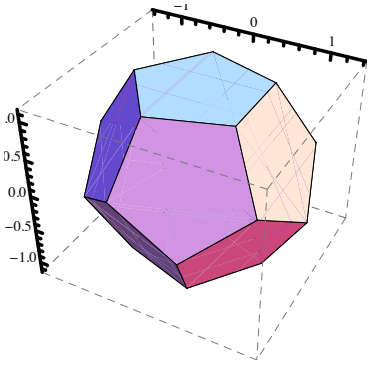
`AxesEdge->Automatic`	use an internal algorithm to choose where to draw all axes
`AxesEdge->`$\{xspec, yspec, zspec\}$	give separate specifications for each of the x, y and z axes
`None`	do not draw this axis
`Automatic`	decide automatically where to draw this axis
$\{dir_i, dir_j\}$	specify on which of the four possible edges to draw this axis

Specifying where to draw three-dimensional axes.

This draws the x on the edge with larger y and z coordinates, draws no y axis, and chooses automatically where to draw the z axis.

```
In[5]:= Show[%, Axes -> True, AxesEdge -> {{1, 1}, None, Automatic}]
```

Out[5]=



When you draw the x axis on a three-dimensional box, there are four possible edges on which the axis can be drawn. These edges are distinguished by having larger or smaller y and z coordinates. When you use the specification $\{dir_y, dir_z\}$ for where to draw the x axis, you can set the dir_i to be $+1$ or -1 to represent larger or smaller values for the y and z coordinates.

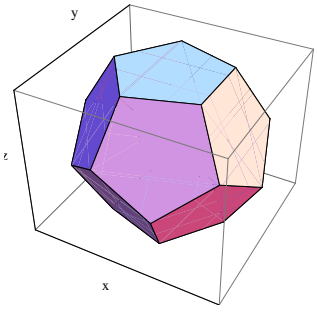
`AxesLabel->None`	give no axis labels
`AxesLabel->`$zlabel$	put a label on the z axis
`AxesLabel->`$\{xlabel, ylabel, zlabel\}$	put labels on all three axes

Axis labels in three-dimensional graphics.

You can use `AxesLabel` to label edges of the box, without necessarily drawing scales on them.

```
In[6]:= Show[PolyhedronData["Dodecahedron", "Image"],
         Axes -> True, AxesLabel -> {"x", "y", "z"}, Ticks -> None]
```

Out[6]=



`Ticks->None`	draw no tick marks
`Ticks->Automatic`	place tick marks automatically
`Ticks->`$\{xticks, yticks, zticks\}$	tick mark specifications for each axis

Settings for the `Ticks` option.

You can give the same kind of tick mark specifications in three dimensions as were described for two-dimensional graphics in "Labeling Two-Dimensional Graphics".

`FaceGrids->None`	draw no grid lines on faces
`FaceGrids->All`	draw grid lines on all faces
`FaceGrids->`$\{face_1, face_2, \ldots\}$	draw grid lines on the faces specified by the $face_i$
`FaceGrids->`$\{\{face_1, \{xgrid_1, ygrid_1\}\}, \ldots\}$	use $xgrid_i$, $ygrid_i$ to determine where and how to draw grid lines on each face

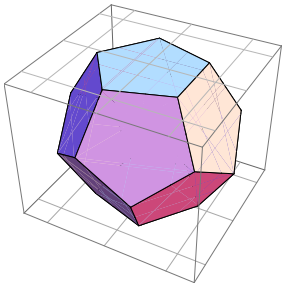
Drawing grid lines in three dimensions.

Mathematica allows you to draw grid lines on the faces of the box that surrounds a three-dimensional object. If you set `FaceGrids -> All`, grid lines are drawn in gray on every face. By setting `FaceGrids ->` $\{face_1, face_2, \ldots\}$ you can tell *Mathematica* to draw grid lines only on specific faces. Each face is specified by a list $\{dir_x, dir_y, dir_z\}$, where two of the dir_i must be `0`, and the third one is `+1` or `-1`. For each face, you can also explicitly tell *Mathematica* where and how to draw the grid lines, using the same kind of specifications as you give for the `GridLines` option in two-dimensional graphics.

This draws grid lines only on the top and bottom faces of the box.

In[7]:=
```
Show[PolyhedronData["Dodecahedron", "Image"], FaceGrids -> {{0, 0, 1}, {0, 0, -1}}]
```

Out[7]=



Efficient Representation of Many Primitives

`Point[{`pt_1`,`pt_2`,…}]`	a multipoint consisting of points at pt_1, pt_2, …
`Line[{`$line_1$`,`$line_2$`,…}]`	a multiline consisting of lines $line_1$, $line_2$, …
`Polygon[{`$poly_1$`,`$poly_2$`,…}]`	a multipolygon consisting of polygons $poly_1$, $poly_2$, …

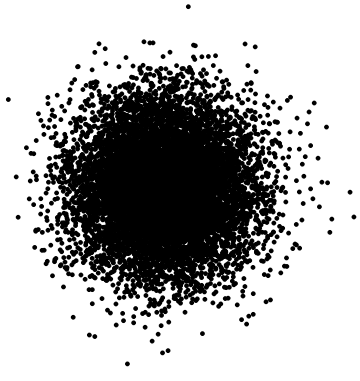
Primitives which can take multiple elements.

Some primitives have multi-element forms that can be processed and rendered more quickly by the *Mathematica* front end than the equivalent individual primitives. For large numbers of primitives, using the multi-element forms can also significantly reduce the sizes of notebook files. Notebooks that use multi-element forms can be less than half the size of those that do not, and render up to ten times faster.

Here is a multipoint random distribution.

In[1]:=
```
Graphics[Point[Table[RandomReal[NormalDistribution[0, 1], 2], {10 000}]]]
```

Out[1]=



`GraphicsComplex[` $\{pt_1, pt_2, \ldots\}$`, data]`	a graphics complex in which coordinates given as integers i in graphics primitives in *data* are taken to be pt_i

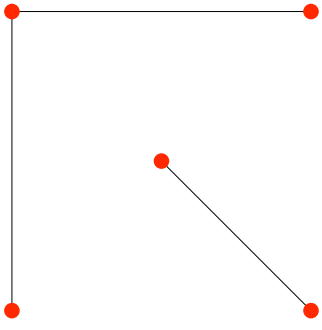
Primitive for sharing coordinate data among primitives.

When many primitives share the same coordinate data, as in meshes and graphs, further efficiency can be gained by using `GraphicsComplex` to factor out the coordinate data. The output of *Mathematica*'s surface- and graph-plotting functions typically use this representation.

Here is a structure of points and lines that share coordinates.

```
In[2]:= Graphics[GraphicsComplex[{{0, 0}, {0, 1}, {1, 1}, {1, 0}, {.5, .5}},
          {Line[{{1, 2, 3}, {5, 4}}], Red, PointSize[.05], Point[{1, 2, 3, 4, 5}]}]]
```

Out[2]=

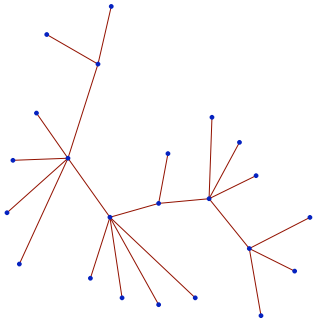


In addition to being efficient, `GraphicsComplex` is useful interactively. Primitives that share coordinates stay connected when one of them is dragged.

Because the output of `GraphPlot` is a `GraphicsComplex`, the graph stays connected when any part of it is dragged.

```
In[3]:= GraphPlot[Table[i → Mod[i^2 - 199, 10], {i, 1, 22}]]
```

Out[3]=

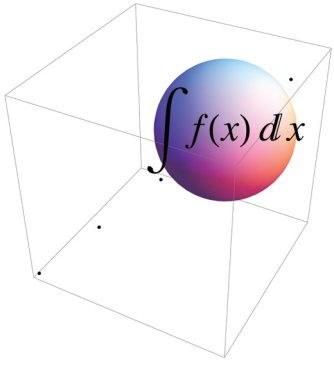


Any primitive may be used within a GraphicsComplex, and GraphicsComplex can be used in both 2D and 3D graphics. Within GraphicsComplex, coordinate positions in primitives are replaced by indices into the coordinate data in the GraphicsComplex.

This GraphicsComplex combines several types of primitives.

In[4]:=
```
Graphics3D[GraphicsComplex[Table[{i, i, i}, {i, 1, 5}], {
  Sphere[4, 1.5],
  Text[Style[∫f[x] dx, 36], 4],
  Point[Range[5]]
}]]
```

Out[4]=

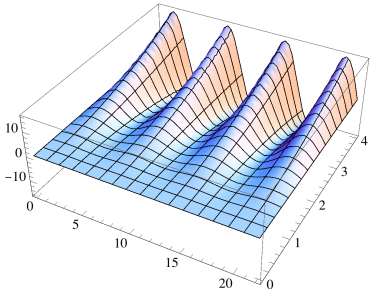


GraphicsComplex is especially useful for representing meshes of polygons. By using GraphicsComplex, numerical errors that could cause gaps between adjacent polygons are avoided.

The output of Plot3D is a GraphicsComplex.

In[5]:=
```
Plot3D[Sin[x] y^2, {x, 0, 22}, {y, 0, 4}]
```

Out[5]=



Formats for Text in Graphics

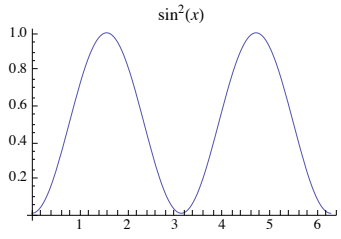
`BaseStyle->`*value*	an option for the text style in a graphic
`FormatType->`*value*	an option for the text format type in a graphic

Specifying formats for text in graphics.

Here is a plot with default settings for all formats.

```
In[1]:= Plot[Sin[x]^2, {x, 0, 2 Pi}, PlotLabel -> Sin[x]^2]
```

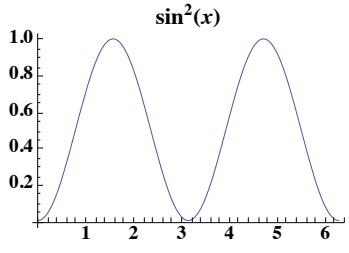
Out[1]=



Here is the same plot, but now using a 12-point bold font.

```
In[2]:= Plot[Sin[x]^2, {x, 0, 2 Pi}, PlotLabel -> Sin[x]^2,
         BaseStyle -> {FontWeight -> "Bold", FontSize → 12}]
```

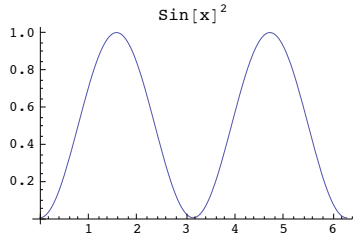
Out[2]=



This uses `StandardForm` rather than `TraditionalForm` .

```
In[3]:= Plot[Sin[x]^2, {x, 0, 2 Pi}, PlotLabel -> Sin[x]^2, FormatType -> StandardForm]
```

Out[3]=



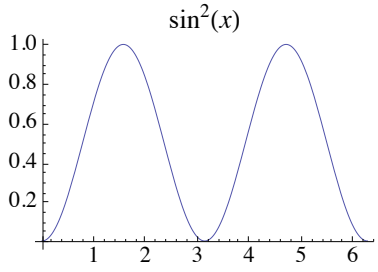
This tells *Mathematica* what default text style to use for all subsequent plots.

```
In[4]:= SetOptions[Plot, BaseStyle -> {FontFamily -> "Times", FontSize → 14}];
```

Now all the text is in 14-point Times font.

```
In[5]:= Plot[Sin[x]^2, {x, 0, 2 Pi}, PlotLabel -> Sin[x]^2]
```

Out[5]=



"*style*"	a named style in your current stylesheet
FontSize->*n*	the size of font to use in printer's points
FontSlant->"Italic"	use an italic font
FontWeight->"Bold"	use a bold font
FontFamily->"*name*"	specify the name of the font family to use (e.g. "Times", "Courier", "Helvetica")

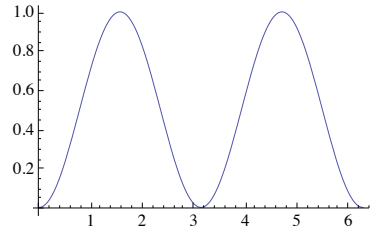
Typical elements used in the setting for BaseStyle.

If you use the standard notebook front end for *Mathematica*, then you can set BaseStyle to be the name of a style defined in your current notebook's stylesheet. You can also explicitly specify how text should be formatted by using options such as FontSize and FontFamily. Note that FontSize gives the absolute size of the font to use, measured in units of printer's points, with one point being $\frac{1}{72}$ inches. If you resize a plot whose font size is specified as a number, the text in it will not by default change size: to get text of a different size you must explicitly specify a new value for the FontSize option. If you resize a plot whose font size is specified as a scaled quantity, the font will scale as the plot is resized. With FontSize -> Scaled[*s*], the effective font size will be *s* scaled units in the plot.

Now all the text resizes as the plot is resized.

```
In[6]:= Plot[Sin[x]^2, {x, 0, 2 Pi}, BaseStyle → {FontSize → Scaled[.05]}]
```

Out[6]=



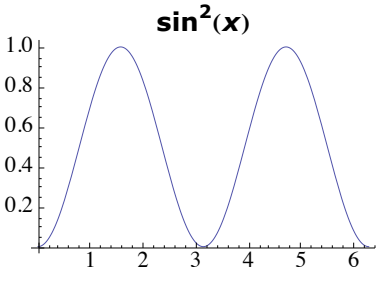
`Style[`*expr*`,"`*style*`"]`	output *expr* in the specified style
`Style[`*expr*`,`*options*`]`	output *expr* using the specified font and style options
`StandardForm[`*expr*`]`	output *expr* in `StandardForm`

Changing the formats of individual pieces of output.

This outputs the plot label using the section heading style in your current notebook.

```
In[7]:= Plot[Sin[x]^2, {x, 0, 2 Pi}, PlotLabel -> Style[Sin[x]^2, "Section"]]
```

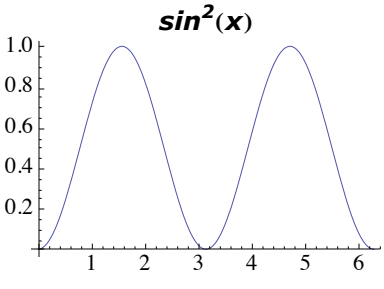
Out[7]=



This uses the section heading style, but modified to be in italics.

```
In[8]:= Plot[Sin[x]^2, {x, 0, 2 Pi},
         PlotLabel -> Style[Sin[x]^2, "Section", FontSlant -> "Italic"]]
```

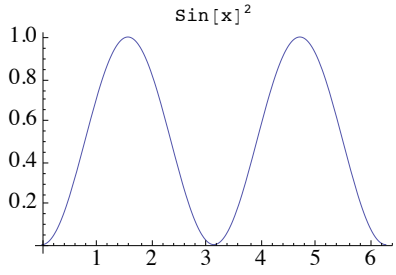
Out[8]=



This produces StandardForm output, with a 12-point font.

```
In[9]:= Plot[Sin[x]^2, {x, 0, 2 Pi},
         PlotLabel -> Style[StandardForm[Sin[x]^2], FontSize -> 12]]
```

Out[9]=



You should realize that the ability to refer to styles such as "Section" depends on using the standard *Mathematica* notebook front end. Even if you are just using a text-based interface to *Mathematica*, however, you can still specify formatting of text in graphics using options such as FontSize. The complete collection of options that you can use is given in "Text and Font Options".

Graphics Primitives for Text

With the Text graphics primitive, you can insert text at any position in two- or three-dimensional *Mathematica* graphics. Unless you explicitly specify a style or font using Style, the text will be given in the graphic's base style.

Text[*expr*, {*x*, *y*}]	text centered at the point {*x*, *y*}
Text[*expr*, {*x*, *y*}, {-1, 0}]	text with its left-hand end at {*x*, *y*}
Text[*expr*, {*x*, *y*}, {1, 0}]	right-hand end at {*x*, *y*}
Text[*expr*, {*x*, *y*}, {0, -1}]	centered above {*x*, *y*}
Text[*expr*, {*x*, *y*}, {0, 1}]	centered below {*x*, *y*}
Text[*expr*, {*x*, *y*}, {*dx*, *dy*}]	text positioned so that {*x*, *y*} is at relative coordinates {*dx*, *dy*} within the box that bounds the text
Text[*expr*, {*x*, *y*}, {*dx*, *dy*}, {0, 1}]	text oriented vertically to read from bottom to top
Text[*expr*, {*x*, *y*}, {*dx*, *dy*}, {0, -1}]	text that reads from top to bottom
Text[*expr*, {*x*, *y*}, {*dx*, *dy*}, {-1, 0}]	text that is upside-down

Two-dimensional text.

This generates five pieces of text, and displays them in a plot.

```
In[1]:= Show[Graphics[Table[Text[Expand[(1 + x)^n], {n, n}], {n, 5}]], PlotRange -> All]
```

Out[1]=

$$x^5 + 5x^4 + 10x^3 + 10x^2 + 5x + 1$$

$$x^4 + 4x^3 + 6x^2 + 4x + 1$$

$$x^3 + 3x^2 + 3x + 1$$

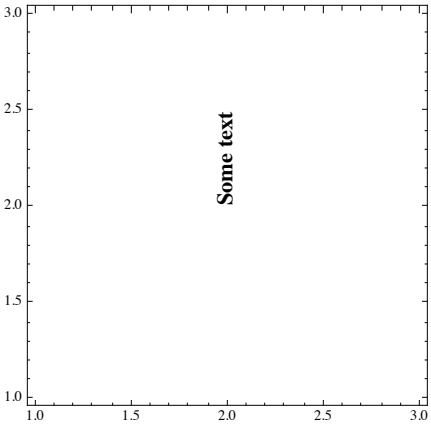
$$x^2 + 2x + 1$$

$$x + 1$$

Here is some vertically oriented text with its left-hand side at the point `{2, 2}`.

```
In[2]:= Show[Graphics[Text[StyleForm["Some text", FontSize -> 14, FontWeight -> "Bold"],
          {2, 2}, {-1, 0}, {0, 1}]], Frame -> True]
```

Out[2]=



When you specify an offset for text, the relative coordinates that are used are taken to run from -1 to 1 in each direction across the box that bounds the text. The point `{0, 0}` in this coordinate system is defined to be center of the text. Note that the offsets you specify need not lie in the range -1 to 1.

Note that you can specify the color of a piece of text by preceding the `Text` graphics primitive with an appropriate `RGBColor` or other graphics directive.

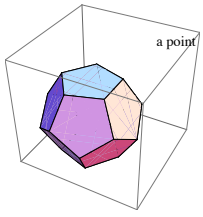
`Text[`*expr*`,{`*x*`,`*y*`,`*z*`}]`	text centered at the point {*x*, *y*, *z*}
`Text[`*expr*`,{`*x*`,`*y*`,`*z*`},{`*sdx*`,`*sdy*`}]`	text with a two-dimensional offset

Three-dimensional text.

This puts text at the specified position in three dimensions.

```
In[3]:= Graphics3D[
         {PolyhedronData["Dodecahedron", "Faces"], Text["a point", {2, 2, 2}, {1, 1}]}]
```

Out[3]=



Note that when you use text in three-dimensional graphics, *Mathematica* assumes that the text is never hidden by any polygons or other objects.

option name	*default value*	
`Background`	`None`	background color
`BaseStyle`	`{}`	style or font specification
`FormatType`	`StandardForm`	format type

Options for `Text`.

By default the text is just put straight on top of whatever graphics have already been drawn.

```
In[4]:= Graphics[
         {{GrayLevel[0.5], Rectangle[{0, 0}, {1, 1}]}, Text["Some text", {0.5, 0.5}]}]
```

Out[4]=

Now there is a rectangle with the background color of the whole plot enclosing the text.

```
In[5]:= Graphics[{{GrayLevel[0.5], Rectangle[{0, 0}, {1, 1}]},
           Text["Some text", {0.5, 0.5}, Background -> Automatic]}]
```

Out[5]=

The Representation of Sound

"Sound" describes how you can take functions and lists of data and produce sounds from them. Here we discuss how sounds are represented in *Mathematica*.

Mathematica treats sounds much like graphics. In fact, *Mathematica* allows you to combine graphics with sound to create pictures with “sound tracks”.

In analogy with graphics, sounds in *Mathematica* are represented by symbolic sound objects. The sound objects have head `Sound`, and contain a list of sound primitives, which represent sounds to be played in sequence.

`Sound[{`s_1`,`s_2`,...}]`	a sound object containing a list of sound primitives

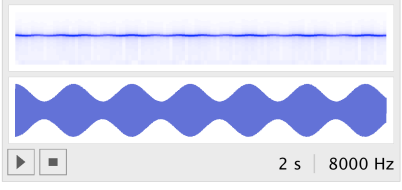
The structure of a sound object.

The functions `Play` and `ListPlay` discussed in "Sound" return `Sound` objects.

`Play` returns a `Sound` object. On appropriate computer systems, it also produces sound.

```
In[1]:= Play[(2 + Cos[20 t]) * Sin[3000 t + 2 Sin[50 t]], {t, 0, 2}]
```

Out[1]=



The `Sound` object contains a `SampledSoundFunction` primitive which uses a compiled function to generate amplitude samples for the sound.

```
In[2]:= Short[InputForm[%]]
```

```
Out[2]//Short= Sound[SampledSoundFunction[CompiledFunction[{_Integer},
                 {<<2>>}, <<3>>, Evaluate], 16384, 8192]]
```

`SampledSoundList[{`a_1`,`a_2`,...},`r`]`	a sound with a sequence of amplitude levels, sampled at rate r
`SampledSoundFunction[`f`,`n`,`r`]`	a sound whose amplitude levels sampled at rate r are found by applying the function f to n successive integers
`SoundNote[`n`,`t`,"`*style*`"]`	a note-like sound with note n, time specification t, with the specified style

Mathematica sound primitives.

At the lowest level, all sounds in *Mathematica* are represented as a sequence of amplitude samples, or as a sequence of MIDI events. In `SampledSoundList`, these amplitude samples are given explicitly in a list. In `SampledSoundFunction`, however, they are generated when the sound is output, by applying the specified function to a sequence of integer arguments. In both cases, all amplitude values obtained must be between -1 and 1. In `SoundNote`, a note-like sound is represented as a sequence of MIDI events that represent the frequency, duration, amplitude and styling of the note.

`ListPlay` generates `SampledSoundList` primitives, while `Play` generates `SampledSoundFunction` primitives. With the default option setting `Compiled -> True`, `Play` will produce a `SampledSoundFunction` object containing a `CompiledFunction`.

Once you have generated a `Sound` object containing various sound primitives, you must then output it as a sound. Much as with graphics, the basic scheme is to take the *Mathematica* representation of the sound, and convert it to a lower-level form that can be handled by an external program, such as a *Mathematica* front end.

The low-level representation of sampled sound used by *Mathematica* consists of a sequence of hexadecimal numbers specifying amplitude levels. Within *Mathematica*, amplitude levels are given as approximate real numbers between -1 and 1. In producing the low-level form, the amplitude levels are "quantized". You can use the option `SampleDepth` to specify how many bits should be used for each sample. The default is `SampleDepth -> 8`, which yields 256 possible amplitude levels, sufficient for most purposes. The low-level representation of note-based sound is as a time-quantized byte stream of MIDI events, which specify various parameters about the note objects. The quantization of time is determined automatically at playback.

You can use the option `SampleDepth` in `Play` and `ListPlay`. In sound primitives, you can specify the sample depth by replacing the sample rate argument by the list $\{rate, depth\}$.

Exporting Graphics and Sounds

Mathematica allows you to export graphics and sounds in a wide variety of formats. If you use the notebook front end for *Mathematica*, then you can typically just copy and paste graphics and sounds directly into other programs using the standard mechanism available on your computer system.

`Export["`*name*`.`*ext*`",`*graphics*`]`	export graphics to a file in a format deduced from the file name
`Export["`*file*`",`*graphics*`,"`*format*`"]`	export graphics in the specified format
`Export["!`*command*`",`*graphics*`,"`*format*`"]`	export graphics to an external command
`Export["`*file*`",{`g_1`,`g_2`,...},...]`	export a sequence of graphics for an animation
`ExportString[`*graphics*`,"`*format*`"]`	generate a string representation of exported graphics

Exporting *Mathematica* graphics and sounds.

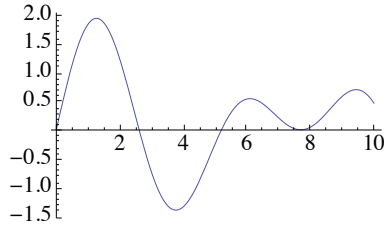
`"EPS"`	Encapsulated PostScript (`.eps`)
`"PDF"`	Adobe Acrobat portable document format (`.pdf`)
`"SVG"`	Scalable Vector Graphics (`.svg`)
`"PICT"`	Macintosh PICT
`"WMF"`	Windows metafile format (`.wmf`)
`"TIFF"`	TIFF (`.tif`, `.tiff`)
`"GIF"`	GIF and animated GIF (`.gif`)
`"JPEG"`	JPEG (`.jpg`, `.jpeg`)
`"PNG"`	PNG format (`.png`)
`"BMP"`	Microsoft bitmap format (`.bmp`)
`"PCX"`	PCX format (`.pcx`)
`"XBM"`	X window system bitmap (`.xbm`)
`"PBM"`	portable bitmap format (`.pbm`)
`"PPM"`	portable pixmap format (`.ppm`)
`"PGM"`	portable graymap format (`.pgm`)
`"PNM"`	portable anymap format (`.pnm`)
`"DICOM"`	DICOM medical imaging format (`.dcm`, `.dic`)
`"AVI"`	Audio Video Interleave format (`.avi`)

Typical graphics formats supported by *Mathematica*. Formats in the first group are resolution independent.

This generates a plot.

```
In[1]:= Plot[Sin[x] + Sin[Sqrt[2] x], {x, 0, 10}]
```

Out[1]=



This exports the plot to a file in Encapsulated PostScript format.

```
In[2]:= Export["sinplot.eps", %]
Out[2]= sinplot.eps
```

When you export a graphic outside of *Mathematica*, you usually have to specify the absolute size at which the graphic should be rendered. You can do this using the `ImageSize` option to `Export`.

`ImageSize -> x` makes the width of the graphic be x printer's points; `ImageSize -> 72 xi` thus makes the width xi inches. The default is to produce an image that is four inches wide. `ImageSize -> {x, y}` scales the graphic so that it fits in an $x \times y$ region.

`ImageSize`	`Automatic`	absolute image size in printer's points
`"ImageTopOrientation"`	`Top`	how the image is oriented in the file
`ImageResolution`	`Automatic`	resolution in dpi for the image

Options for `Export`.

Within *Mathematica*, graphics are manipulated in a way that is completely independent of the resolution of the computer screen or other output device on which the graphics will eventually be rendered.

Many programs and devices accept graphics in resolution-independent formats such as Encapsulated PostScript (EPS). But some require that the graphics be converted to rasters or bitmaps with a specific resolution. The `ImageResolution` option for `Export` allows you to determine what resolution in dots per inch (dpi) should be used. The lower you set this resolution, the lower the quality of the image you will get, but also the less memory the image will take to store. For screen display, typical resolutions are 72 dpi and above; for printers, 300 dpi and above.

`"DXF"`	AutoCAD drawing interchange format (`.dxf`)
`"STL"`	STL stereolithography format (`.stl`)

Typical 3D geometry formats supported by *Mathematica*.

`"WAV"`	Microsoft wave format (`.wav`)
`"AU"`	μ law encoding (`.au`)
`"SND"`	sound file format (`.snd`)
`"AIFF"`	AIFF format (`.aif`, `.aiff`)

Typical sound formats supported by *Mathematica*.

Importing Graphics and Sounds

Mathematica allows you not only to export graphics and sounds, but also to import them. With `Import` you can read graphics and sounds in a wide variety of formats, and bring them into *Mathematica* as *Mathematica* expressions.

`Import["`*name*`.`*ext*`"]`	import graphics from the file *name*`.`*ext* in a format deduced from the file name
`Import["`*file*`","`*format*`"]`	import graphics in the specified format
`ImportString["`*string*`","`*format*`"]`	import graphics from a string

Importing graphics and sounds.

This imports an image stored in JPEG format.

```
In[1]:= g = Import["ExampleData/ocelot.jpg"]
```

Out[1]=

This shows an array of four copies of the image.

```
In[2]:= GraphicsGrid[{{g, g}, {g, g}}]
```

Out[2]=

`Import` yields expressions with different structures depending on the type of data it reads. Typically you will need to know the structure if you want to manipulate the data that is returned.

`Graphics[`*primitives*`,`*opts*`]`	resolution-independent graphics
`Graphics[Raster[`*data*`],`*opts*`]`	resolution-dependent bitmap images
{*graphics*$_1$`,`*graphics*$_2$`,…}`	animated graphics
`Sound[SampledSoundList[`*data*`,`*r*`]]`	sounds

Structures of expressions returned by `Import`.

This shows the overall structure of the graphics object imported above.

```
In[3]:= Shallow[InputForm[g]]
Out[3]//Shallow= Graphics[Raster[<< 4 >>], Rule[<< 2 >>], Rule[<< 2 >>]]
```

This extracts the array of pixel values used.

```
In[4]:= d = g[[1, 1]];
```

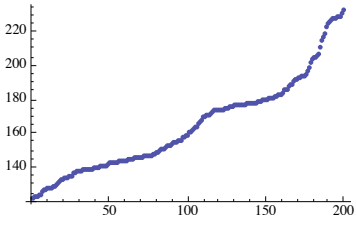
Here are the dimensions of the array.

```
In[5]:= Dimensions[d]
Out[5]= {200, 200}
```

This shows the distribution of pixel values.

```
In[6]:= ListPlot[Sort[Flatten[d]]]
```

Out[6]=



This shows a transformed version of the image.

```
In[7]:= Graphics[Raster[d^2 / Max[d^2]]]
```

Out[7]=

Editing *Mathematica* Graphics

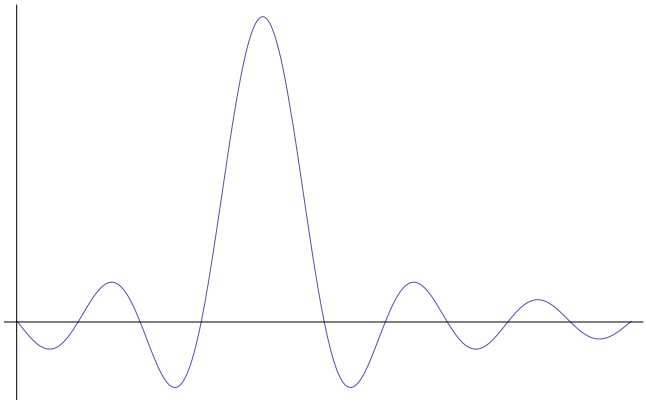
Introduction to Editing *Mathematica* Graphics

An Example of Editing Graphics

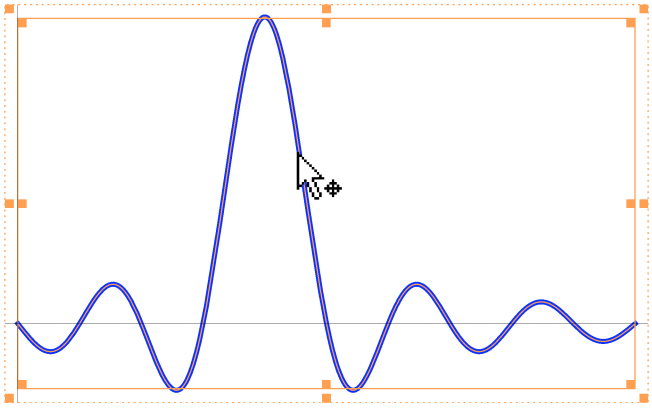
The following graph represents an impulse response of an ideal Low Pass Filter (LPF). This graph illustrates some of the ways of interacting with graphics. Details on each topic follow in the other parts of "Interactive Graphics".

```
In[1]:= Plot[Sin[Pi (t - 4)] / (Pi (t - 4)), {t, 0, 10}, PlotRange → All, Ticks → None]
```

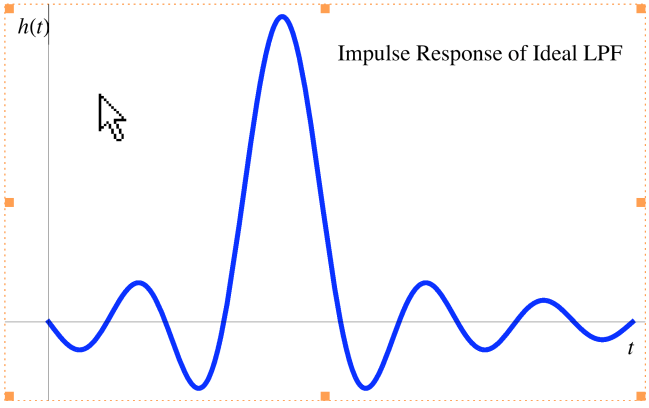
Out[1]=



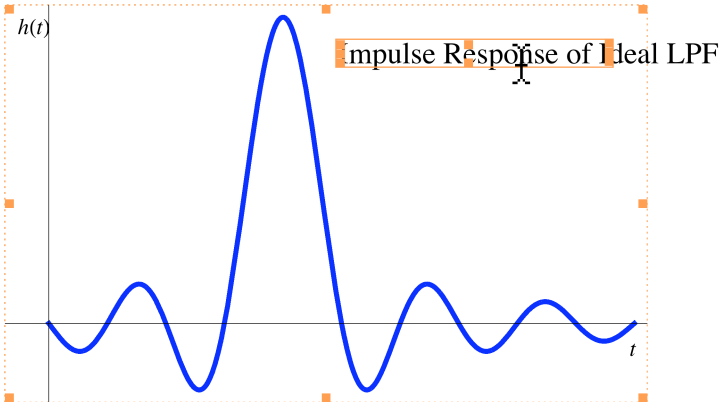
Select the curve and change its color and thickness using the **Graphics Inspector**.



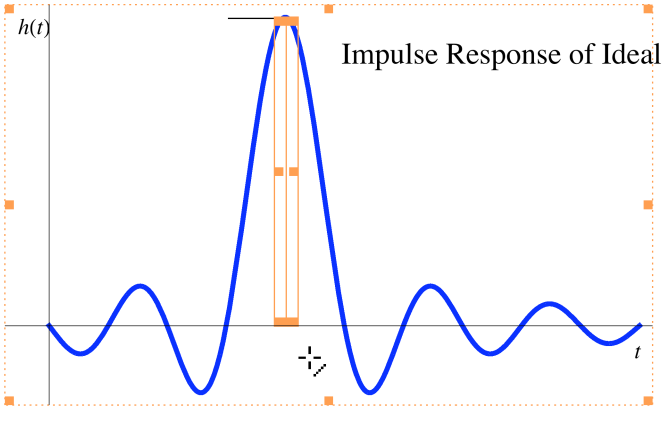
Using the **Text** tool and the **TraditionalForm Text** tool, add a plot label and axis labels.



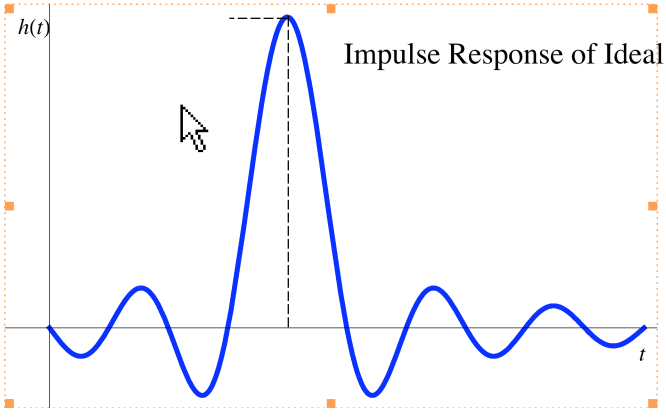
You can edit the title to change its font, color, size, and face.



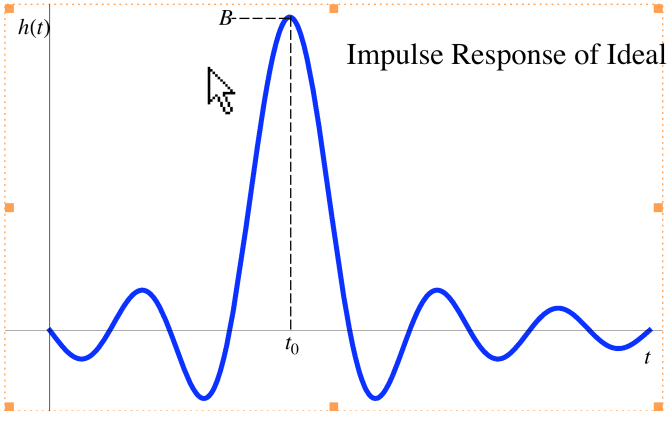
Draw vertical and horizontal lines to the maximum point with the **Line** tool.



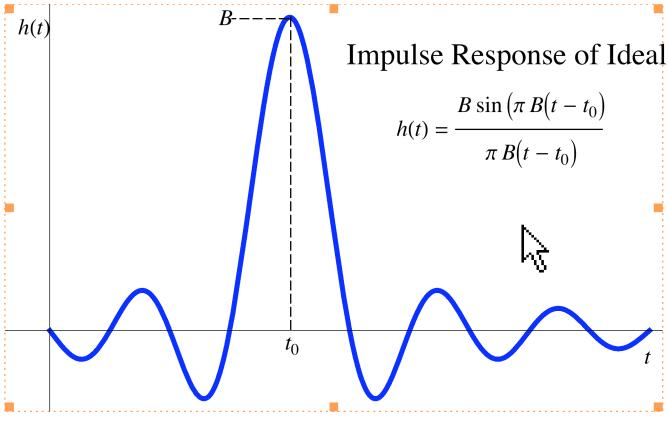
Select the lines and make them dashed using the **Graphics Inspector**.



Label the point with the **TraditionalForm Text** tool.



Add the formula for the curve.



Drawing Tools

To Open the Graphics Palette:

Type Ctrl+T or choose **Graphics ▸ Drawing Tools**.

For more information on each tool, click the words pointing into the palette.