Physical properties of light

Physical properties of light

Light consists of photons — "particles" with no mass which travel at the speed of light.

Physical properties of light

Light consists of photons — "particles" with no mass which travel at the speed of light. They have energy, and one measure of this energy is the "wavelength" of the light.

Physical properties of light

Light consists of photons — "particles" with no mass which travel at the speed of light. They have energy, and one measure of this energy is the "wavelength" of the light.

To a very good approximation, light travels in straight lines, and behaves much like a particle.

Physical properties of light

Light consists of photons — "particles" with no mass which travel at the speed of light. They have energy, and one measure of this energy is the "wavelength" of the light.

To a very good approximation, light travels in straight lines, and behaves much like a particle. (In some circumstances, light has wave properties — interference and diffraction effects — but these are rarely important in computer graphics.)

Physical properties of light

Light consists of photons — "particles" with no mass which travel at the speed of light. They have energy, and one measure of this energy is the "wavelength" of the light.

To a very good approximation, light travels in straight lines, and behaves much like a particle. (In some circumstances, light has wave properties — interference and diffraction effects — but these are rarely important in computer graphics.)

Light may be "bent" or refracted in transparent substances, and the degree of bending, or refraction, depends upon a quantity called the refractive index.

Physical properties of light

Light consists of photons — "particles" with no mass which travel at the speed of light. They have energy, and one measure of this energy is the "wavelength" of the light.

To a very good approximation, light travels in straight lines, and behaves much like a particle. (In some circumstances, light has wave properties — interference and diffraction effects — but these are rarely important in computer graphics.)

Light may be "bent" or refracted in transparent substances, and the degree of bending, or refraction, depends upon a quantity called the refractive index.

When light is incident on a shiny, flat surface, it is reflected.

Physical properties of light

Light consists of photons — "particles" with no mass which travel at the speed of light. They have energy, and one measure of this energy is the "wavelength" of the light.

To a very good approximation, light travels in straight lines, and behaves much like a particle. (In some circumstances, light has wave properties — interference and diffraction effects — but these are rarely important in computer graphics.)

Light may be "bent" or refracted in transparent substances, and the degree of bending, or refraction, depends upon a quantity called the refractive index.

When light is incident on a shiny, flat surface, it is reflected. The reflected light leaves the surface at an angle such that the angle between the incident light and a normal to the surface is equal to the same angle for the reflected light.

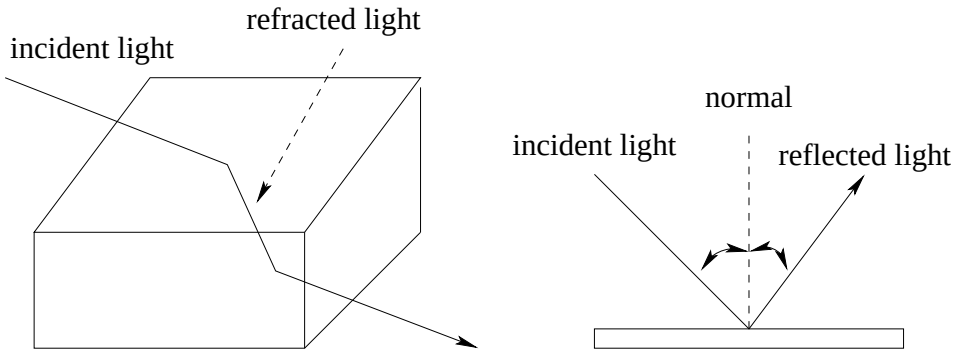
Physical properties of light

Light consists of photons — "particles" with no mass which travel at the speed of light. They have energy, and one measure of this energy is the "wavelength" of the light.

To a very good approximation, light travels in straight lines, and behaves much like a particle. (In some circumstances, light has wave properties — interference and diffraction effects — but these are rarely important in computer graphics.)

Light may be "bent" or refracted in transparent substances, and the degree of bending, or refraction, depends upon a quantity called the refractive index.

When light is incident on a shiny, flat surface, it is reflected. The reflected light leaves the surface at an angle such that the angle between the incident light and a normal to the surface is equal to the same angle for the reflected light.



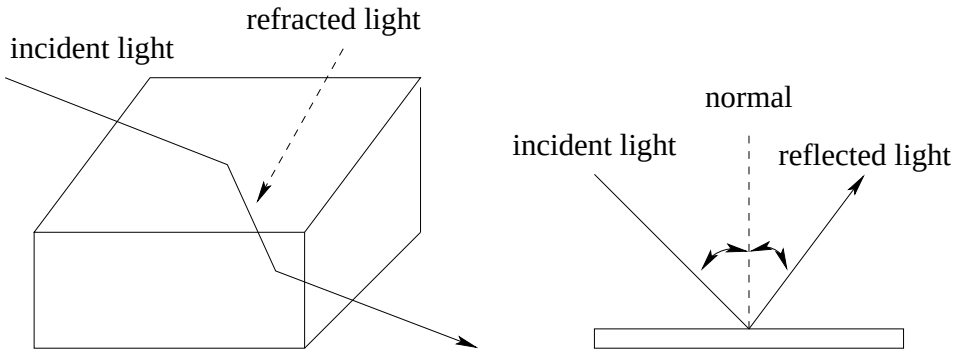
Physical properties of light

Light consists of photons — "particles" with no mass which travel at the speed of light. They have energy, and one measure of this energy is the "wavelength" of the light.

To a very good approximation, light travels in straight lines, and behaves much like a particle. (In some circumstances, light has wave properties — interference and diffraction effects — but these are rarely important in computer graphics.)

Light may be "bent" or refracted in transparent substances, and the degree of bending, or refraction, depends upon a quantity called the refractive index.

When light is incident on a shiny, flat surface, it is reflected. The reflected light leaves the surface at an angle such that the angle between the incident light and a normal to the surface is equal to the same angle for the reflected light.



Physical properties of surfaces

Surfaces have different characteristics with respect to their interaction with light.

Physical properties of surfaces

Surfaces have different characteristics with respect to their interaction with light. Some surfaces are "shiny" and reflect most of the light incident on the surface.

Physical properties of surfaces

Surfaces have different characteristics with respect to their interaction with light. Some surfaces are "shiny" and reflect most of the light incident on the surface. Others are dull and reflect little light at all.

Physical properties of surfaces

Surfaces have different characteristics with respect to their interaction with light. Some surfaces are "shiny" and reflect most of the light incident on the surface. Others are dull and reflect little light at all.

Surfaces have "color" — the color comes from the light incident on the surface, which may reflect some spectral components, and absorb others.

Physical properties of surfaces

Surfaces have different characteristics with respect to their interaction with light. Some surfaces are "shiny" and reflect most of the light incident on the surface. Others are dull and reflect little light at all.

Surfaces have "color" — the color comes from the light incident on the surface, which may reflect some spectral components, and absorb others. A red painted cube reflects red light, but absorbs green and blue light.

Physical properties of surfaces

Surfaces have different characteristics with respect to their interaction with light. Some surfaces are "shiny" and reflect most of the light incident on the surface. Others are dull and reflect little light at all.

Surfaces have "color" — the color comes from the light incident on the surface, which may reflect some spectral components, and absorb others. A red painted cube reflects red light, but absorbs green and blue light. If illuminated by pure green or blue light, the cube would appear black.

Physical properties of surfaces

Surfaces have different characteristics with respect to their interaction with light. Some surfaces are "shiny" and reflect most of the light incident on the surface. Others are dull and reflect little light at all.

Surfaces have "color" — the color comes from the light incident on the surface, which may reflect some spectral components, and absorb others. A red painted cube reflects red light, but absorbs green and blue light. If illuminated by pure green or blue light, the cube would appear black.

Some surfaces, like a CRT screen, for example, emit their own light, in varying colors and intensities.

Physical properties of surfaces

Surfaces have different characteristics with respect to their interaction with light. Some surfaces are "shiny" and reflect most of the light incident on the surface. Others are dull and reflect little light at all.

Surfaces have "color" — the color comes from the light incident on the surface, which may reflect some spectral components, and absorb others. A red painted cube reflects red light, but absorbs green and blue light. If illuminated by pure green or blue light, the cube would appear black.

Some surfaces, like a CRT screen, for example, emit their own light, in varying colors and intensities.

Characteristics like "roughness" are also important in determining the interaction between light and a surface.

Physical properties of surfaces

Surfaces have different characteristics with respect to their interaction with light. Some surfaces are "shiny" and reflect most of the light incident on the surface. Others are dull and reflect little light at all.

Surfaces have "color" — the color comes from the light incident on the surface, which may reflect some spectral components, and absorb others. A red painted cube reflects red light, but absorbs green and blue light. If illuminated by pure green or blue light, the cube would appear black.

Some surfaces, like a CRT screen, for example, emit their own light, in varying colors and intensities.

Characteristics like "roughness" are also important in determining the interaction between light and a surface. Also, some materials can be transparent and this transparency may be wavelength dependent (a red plastic screen transmits only red light, it absorbs the rest).

Physical properties of surfaces

Surfaces have different characteristics with respect to their interaction with light. Some surfaces are "shiny" and reflect most of the light incident on the surface. Others are dull and reflect little light at all.

Surfaces have "color" — the color comes from the light incident on the surface, which may reflect some spectral components, and absorb others. A red painted cube reflects red light, but absorbs green and blue light. If illuminated by pure green or blue light, the cube would appear black.

Some surfaces, like a CRT screen, for example, emit their own light, in varying colors and intensities.

Characteristics like "roughness" are also important in determining the interaction between light and a surface. Also, some materials can be transparent and this transparency may be wavelength dependent (a red plastic screen transmits only red light, it absorbs the rest).

Illumination

Illumination

Indirect light which has been scattered by many sources is known as ambient light.

Illumination

Indirect light which has been scattered by many sources is known as ambient light. You can think of ambient light as background light.

Illumination

Indirect light which has been scattered by many sources is known as ambient light. You can think of ambient light as background light. When ambient light interacts with a surface, it is scattered in all directions.

Illumination

Indirect light which has been scattered by many sources is known as ambient light. You can think of ambient light as background light. When ambient light interacts with a surface, it is scattered in all directions.

Light which is strongly directional (e.g. a laser beam) is said to be specular.

Illumination

Indirect light which has been scattered by many sources is known as ambient light. You can think of ambient light as background light. When ambient light interacts with a surface, it is scattered in all directions.

Light which is strongly directional (e.g. a laser beam) is said to be specular. When interacting with a shiny surface, most of this light is reflected as from a mirror.

Illumination

Indirect light which has been scattered by many sources is known as ambient light. You can think of ambient light as background light. When ambient light interacts with a surface, it is scattered in all directions.

Light which is strongly directional (e.g. a laser beam) is said to be specular. When interacting with a shiny surface, most of this light is reflected as from a mirror. (Specularity is really a material property, but in graphics the term is also applied to light sources.)

Illumination

Indirect light which has been scattered by many sources is known as ambient light. You can think of ambient light as background light. When ambient light interacts with a surface, it is scattered in all directions.

Light which is strongly directional (e.g. a laser beam) is said to be specular. When interacting with a shiny surface, most of this light is reflected as from a mirror. (Specularity is really a material property, but in graphics the term is also applied to light sources.)

The diffuse component of light comes from one direction, so it is brighter if it shines directly on a surface than if it meets the surface at an angle.

Illumination

Indirect light which has been scattered by many sources is known as ambient light. You can think of ambient light as background light. When ambient light interacts with a surface, it is scattered in all directions.

Light which is strongly directional (e.g. a laser beam) is said to be specular. When interacting with a shiny surface, most of this light is reflected as from a mirror. (Specularity is really a material property, but in graphics the term is also applied to light sources.)

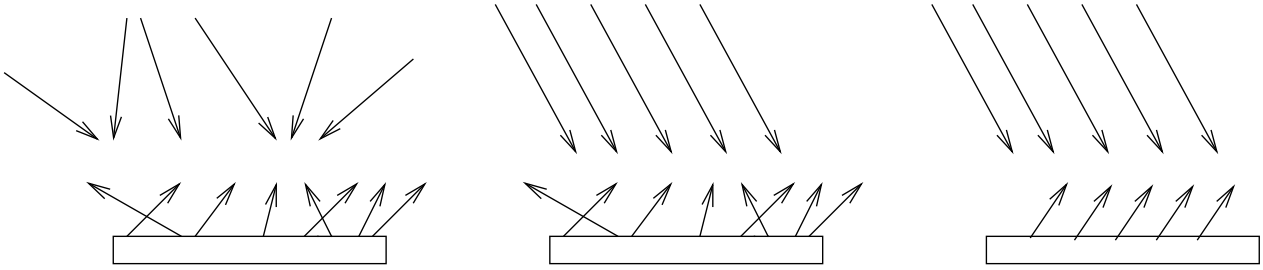
The diffuse component of light comes from one direction, so it is brighter if it shines directly on a surface than if it meets the surface at an angle. Unlike specular light, diffuse light is scattered in all directions after striking a surface.

Illumination

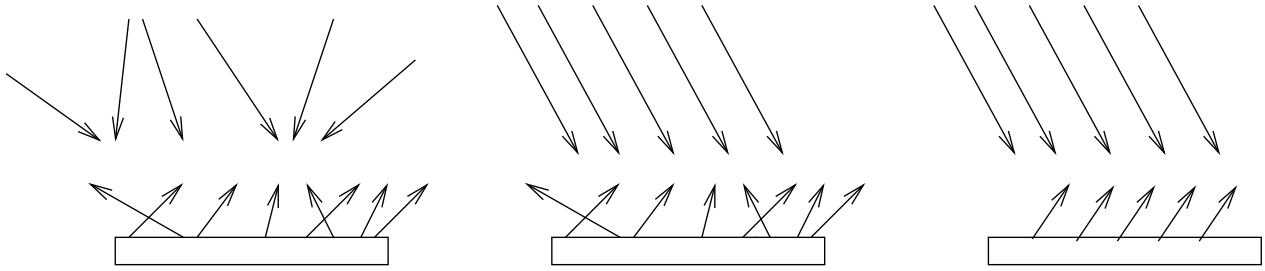
Indirect light which has been scattered by many sources is known as ambient light. You can think of ambient light as background light. When ambient light interacts with a surface, it is scattered in all directions.

Light which is strongly directional (e.g. a laser beam) is said to be specular. When interacting with a shiny surface, most of this light is reflected as from a mirror. (Specularity is really a material property, but in graphics the term is also applied to light sources.)

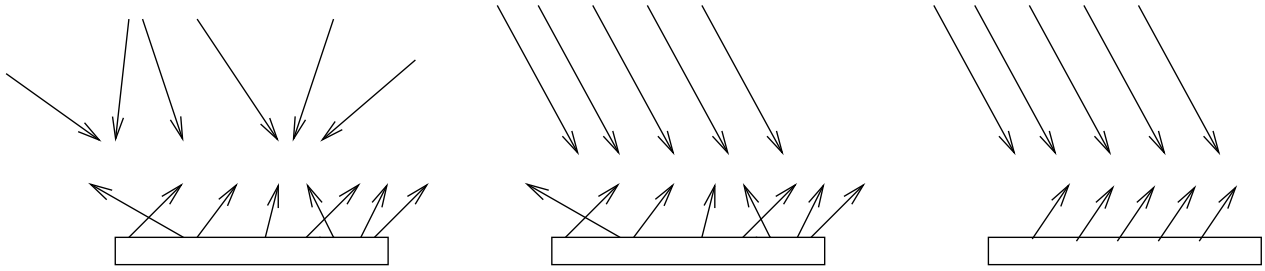
The diffuse component of light comes from one direction, so it is brighter if it shines directly on a surface than if it meets the surface at an angle. Unlike specular light, diffuse light is scattered in all directions after striking a surface.



Ambient Diffuse Specular

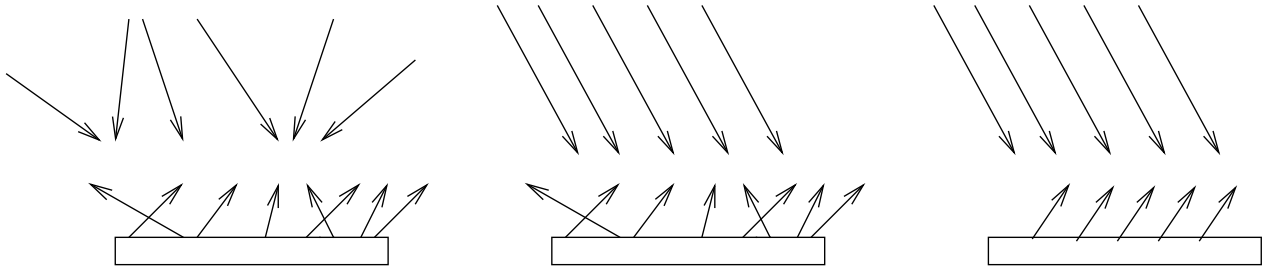


ambient — rays come from and scatter into all directions



ambient — rays come from and scatter into all directions

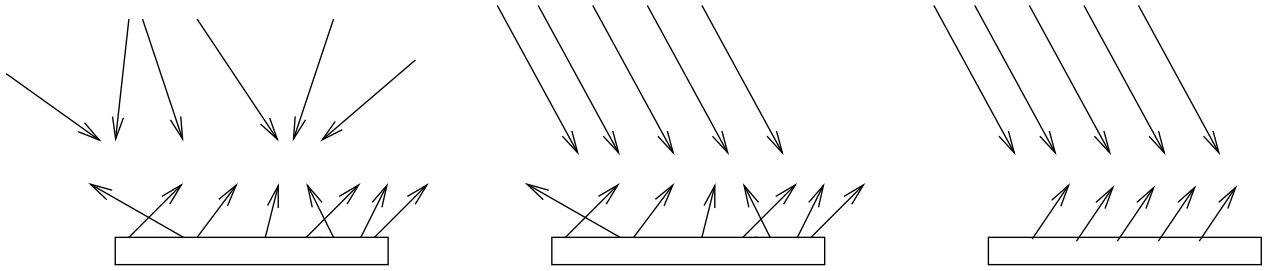
diffuse — rays come from one direction, scatter into all directions



ambient — rays come from and scatter into all directions

diffuse — rays come from one direction, scatter into all directions

specular — rays come from one direction, reflect into one direction

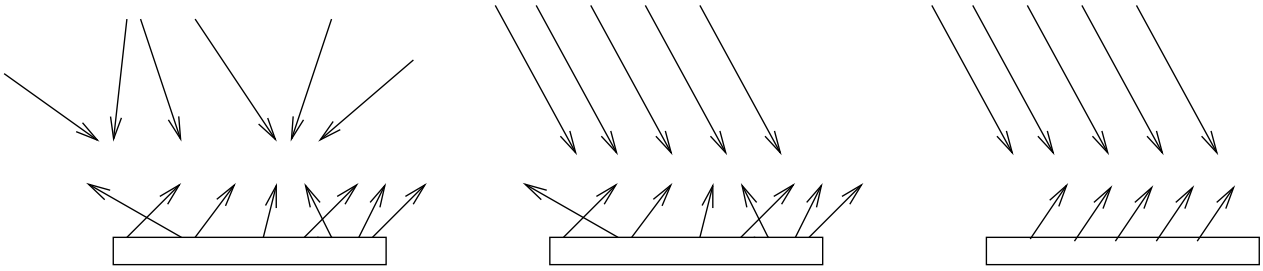


ambient — rays come from and scatter into all directions

diffuse — rays come from one direction, scatter into all directions

specular — rays come from one direction, reflect into one direction

For specular light, the normal to the surface is used to calculate the direction into which the reflected light is scattered.

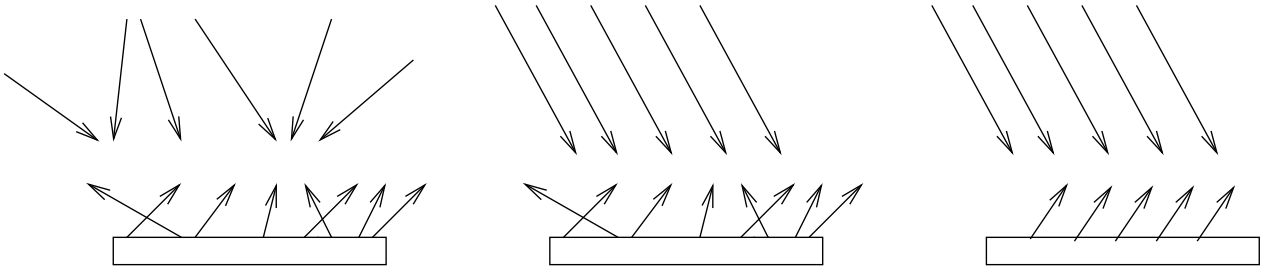


ambient — rays come from and scatter into all directions

diffuse — rays come from one direction, scatter into all directions

specular — rays come from one direction, reflect into one direction

For specular light, the normal to the surface is used to calculate the direction into which the reflected light is scattered. The normal is also used to determine the degree of diffuse lighting.

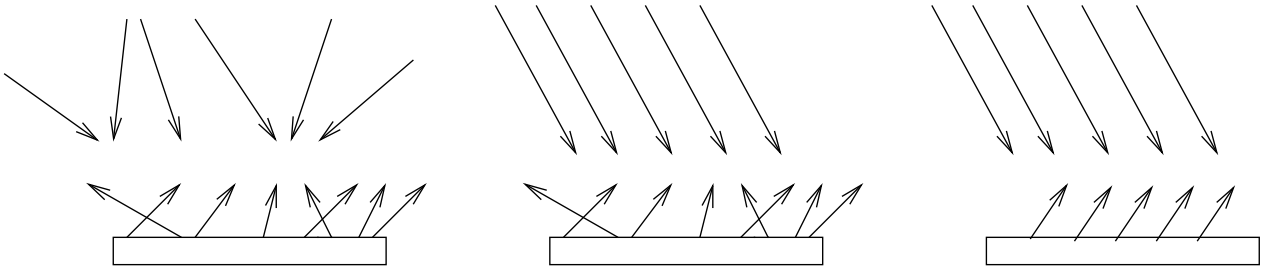


ambient — rays come from and scatter into all directions

diffuse — rays come from one direction, scatter into all directions

specular — rays come from one direction, reflect into one direction

For specular light, the normal to the surface is used to calculate the direction into which the reflected light is scattered. The normal is also used to determine the degree of diffuse lighting. Thus, normals **must** be defined in order to use lighting.



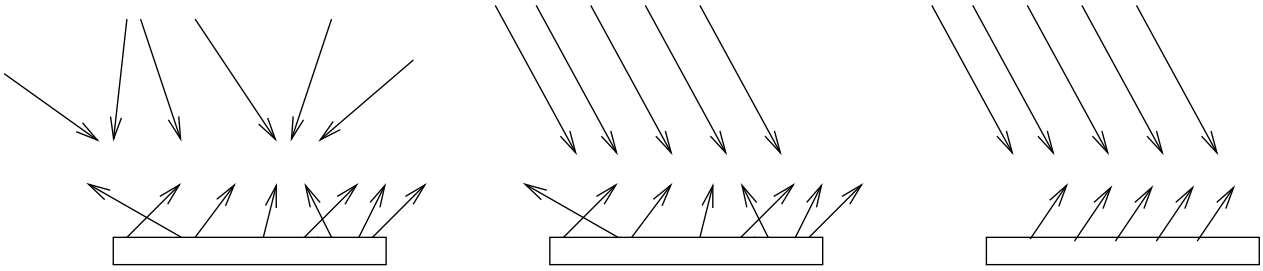
ambient — rays come from and scatter into all directions

diffuse — rays come from one direction, scatter into all directions

specular — rays come from one direction, reflect into one direction

For specular light, the normal to the surface is used to calculate the direction into which the reflected light is scattered. The normal is also used to determine the degree of diffuse lighting. Thus, normals **must** be defined in order to use lighting.

Light also has the property that it reduces in intensity with distance from the source.



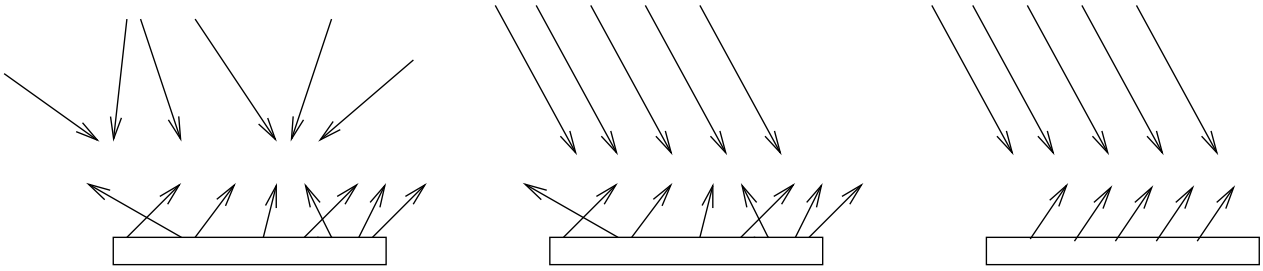
ambient — rays come from and scatter into all directions

diffuse — rays come from one direction, scatter into all directions

specular — rays come from one direction, reflect into one direction

For specular light, the normal to the surface is used to calculate the direction into which the reflected light is scattered. The normal is also used to determine the degree of diffuse lighting. Thus, normals **must** be defined in order to use lighting.

Light also has the property that it reduces in intensity with distance from the source. In particular, light from a point source decreased in intensity proportional to the square of the distance from the source.



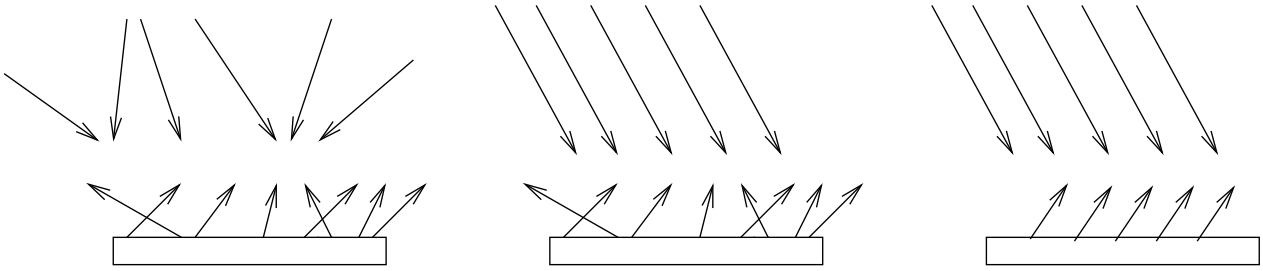
ambient — rays come from and scatter into all directions

diffuse — rays come from one direction, scatter into all directions

specular — rays come from one direction, reflect into one direction

For specular light, the normal to the surface is used to calculate the direction into which the reflected light is scattered. The normal is also used to determine the degree of diffuse lighting. Thus, normals **must** be defined in order to use lighting.

Light also has the property that it reduces in intensity with distance from the source. In particular, light from a point source decreased in intensity proportional to the square of the distance from the source. For a long linear light source, the intensity decreases proportional to the distance from the source.



ambient — rays come from and scatter into all directions

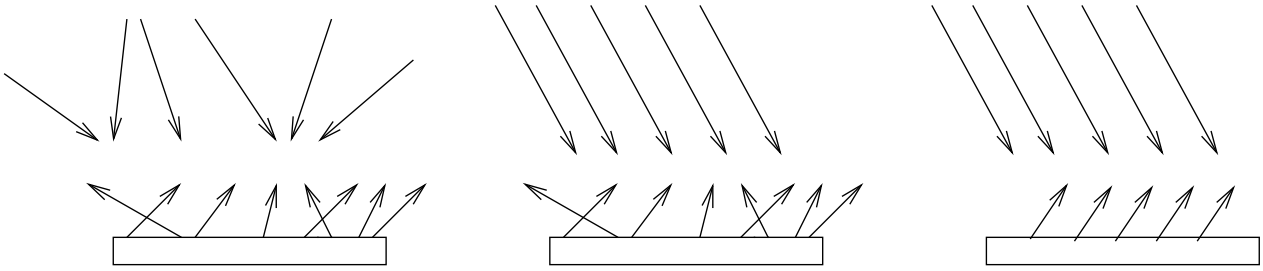
diffuse — rays come from one direction, scatter into all directions

specular — rays come from one direction, reflect into one direction

For specular light, the normal to the surface is used to calculate the direction into which the reflected light is scattered. The normal is also used to determine the degree of diffuse lighting. Thus, normals **must** be defined in order to use lighting.

Light also has the property that it reduces in intensity with distance from the source. In particular, light from a point source decreased in intensity proportional to the square of the distance from the source. For a long linear light source, the intensity decreases proportional to the distance from the source.

Other light sources (spotlights, and other lights containing reflecting or lens elements) may have more complex functions.



ambient — rays come from and scatter into all directions

diffuse — rays come from one direction, scatter into all directions

specular — rays come from one direction, reflect into one direction

For specular light, the normal to the surface is used to calculate the direction into which the reflected light is scattered. The normal is also used to determine the degree of diffuse lighting. Thus, normals **must** be defined in order to use lighting.

Light also has the property that it reduces in intensity with distance from the source. In particular, light from a point source decreased in intensity proportional to the square of the distance from the source. For a long linear light source, the intensity decreases proportional to the distance from the source.

Other light sources (spotlights, and other lights containing reflecting or lens elements) may have more complex functions.

Rendering lit scenes realistically

We require a simple model for lighting which allows us to model both light sources and surface properties efficiently.

Rendering lit scenes realistically

We require a simple model for lighting which allows us to model both light sources and surface properties efficiently.

Ideally, we would assign the appropriate physical properties to each surface and calculate the surface interactions of every visible ray of light from the viewer's eye position back to the source.

Rendering lit scenes realistically

We require a simple model for lighting which allows us to model both light sources and surface properties efficiently.

Ideally, we would assign the appropriate physical properties to each surface and calculate the surface interactions of every visible ray of light from the viewer's eye position back to the source. This would produce a very realistic image (if all properties were well modelled), but would be very computationally expensive.

Rendering lit scenes realistically

We require a simple model for lighting which allows us to model both light sources and surface properties efficiently.

Ideally, we would assign the appropriate physical properties to each surface and calculate the surface interactions of every visible ray of light from the viewer's eye position back to the source. This would produce a very realistic image (if all properties were well modelled), but would be very computationally expensive. This method is known as ray tracing.

Rendering lit scenes realistically

We require a simple model for lighting which allows us to model both light sources and surface properties efficiently.

Ideally, we would assign the appropriate physical properties to each surface and calculate the surface interactions of every visible ray of light from the viewer's eye position back to the source. This would produce a very realistic image (if all properties were well modelled), but would be very computationally expensive. This method is known as ray tracing. It can produce very realistic images, including refraction, shadows and multiple reflections, but at very high computational cost.

Rendering lit scenes realistically

We require a simple model for lighting which allows us to model both light sources and surface properties efficiently.

Ideally, we would assign the appropriate physical properties to each surface and calculate the surface interactions of every visible ray of light from the viewer's eye position back to the source. This would produce a very realistic image (if all properties were well modelled), but would be very computationally expensive. This method is known as ray tracing. It can produce very realistic images, including refraction, shadows and multiple reflections, but at very high computational cost.

OpenGL uses a relatively simple model for lighting.

Rendering lit scenes realistically

We require a simple model for lighting which allows us to model both light sources and surface properties efficiently.

Ideally, we would assign the appropriate physical properties to each surface and calculate the surface interactions of every visible ray of light from the viewer's eye position back to the source. This would produce a very realistic image (if all properties were well modelled), but would be very computationally expensive. This method is known as ray tracing. It can produce very realistic images, including refraction, shadows and multiple reflections, but at very high computational cost.

OpenGL uses a relatively simple model for lighting. It defines a set of properties for materials, a set of light sources, and a lighting model.

Rendering lit scenes realistically

We require a simple model for lighting which allows us to model both light sources and surface properties efficiently.

Ideally, we would assign the appropriate physical properties to each surface and calculate the surface interactions of every visible ray of light from the viewer's eye position back to the source. This would produce a very realistic image (if all properties were well modelled), but would be very computationally expensive. This method is known as ray tracing. It can produce very realistic images, including refraction, shadows and multiple reflections, but at very high computational cost.

OpenGL uses a relatively simple model for lighting. It defines a set of properties for materials, a set of light sources, and a lighting model. This model cannot account for secondary effects like shadows or reflected illumination, but it produces reasonable illumination effects efficiently.

Rendering lit scenes realistically

We require a simple model for lighting which allows us to model both light sources and surface properties efficiently.

Ideally, we would assign the appropriate physical properties to each surface and calculate the surface interactions of every visible ray of light from the viewer's eye position back to the source. This would produce a very realistic image (if all properties were well modelled), but would be very computationally expensive. This method is known as ray tracing. It can produce very realistic images, including refraction, shadows and multiple reflections, but at very high computational cost.

OpenGL uses a relatively simple model for lighting. It defines a set of properties for materials, a set of light sources, and a lighting model. This model cannot account for secondary effects like shadows or reflected illumination, but it produces reasonable illumination effects efficiently.

Lighting models in OpenGL

Lighting models in OpenGL

In order to add lighting to a scene, the following steps are required:

Lighting models in OpenGL

In order to add lighting to a scene, the following steps are required:

- Define normals at each vertex for every object.

Lighting models in OpenGL

In order to add lighting to a scene, the following steps are required:

- Define normals at each vertex for every object. (More on this soon)

Lighting models in OpenGL

In order to add lighting to a scene, the following steps are required:

- Define normals at each vertex for every object. (More on this soon)
- Create and position one or more light sources with `glLight*()`. (OpenGL supports at least 8 light sources).

Lighting models in OpenGL

In order to add lighting to a scene, the following steps are required:

- Define normals at each vertex for every object. (More on this soon)
- Create and position one or more light sources with `glLight*()`. (OpenGL supports at least 8 light sources).
- Select a lighting model; with `glLightModel*()`. This defines the level of (global) ambient light, and the effective location of the viewpoint (for the lighting calculations).

Lighting models in OpenGL

In order to add lighting to a scene, the following steps are required:

- Define normals at each vertex for every object. (More on this soon)
- Create and position one or more light sources with `glLight*()`. (OpenGL supports at least 8 light sources).
- Select a lighting model; with `glLightModel*()`. This defines the level of (global) ambient light, and the effective location of the viewpoint (for the lighting calculations).
- Define the material properties for each object in the scene with `glMaterial*()`.

Lighting models in OpenGL

In order to add lighting to a scene, the following steps are required:

- Define normals at each vertex for every object. (More on this soon)
- Create and position one or more light sources with `glLight*()`. (OpenGL supports at least 8 light sources).
- Select a lighting model; with `glLightModel*()`. This defines the level of (global) ambient light, and the effective location of the viewpoint (for the lighting calculations).
- Define the material properties for each object in the scene with `glMaterial*()`.

```
void glLight{if}(GLenum light, GLenum pname,
TYPE param);
```

Lighting models in OpenGL

In order to add lighting to a scene, the following steps are required:

- Define normals at each vertex for every object. (More on this soon)
- Create and position one or more light sources with `glLight*()`. (OpenGL supports at least 8 light sources).
- Select a lighting model; with `glLightModel*()`. This defines the level of (global) ambient light, and the effective location of the viewpoint (for the lighting calculations).
- Define the material properties for each object in the scene with `glMaterial*()`.

```
void glLight{if}(GLenum light, GLenum pname,
TYPE param);
void glLight{if}v(GLenum light, GLenum pname,
TYPE *param);
```

Lighting models in OpenGL

In order to add lighting to a scene, the following steps are required:

- Define normals at each vertex for every object. (More on this soon)
- Create and position one or more light sources with `glLight*()`. (OpenGL supports at least 8 light sources).
- Select a lighting model; with `glLightModel*()`. This defines the level of (global) ambient light, and the effective location of the viewpoint (for the lighting calculations).
- Define the material properties for each object in the scene with `glMaterial*()`.

```
void glLight{if}(GLenum light, GLenum pname,
TYPE param);
```

```
void glLight{if}v(GLenum light, GLenum pname,
TYPE *param);
```

create the light specified by `GL_LIGHT0`, `GL_LIGHT1`, ... with properties given by `pname`;

Lighting models in OpenGL

In order to add lighting to a scene, the following steps are required:

- Define normals at each vertex for every object. (More on this soon)
- Create and position one or more light sources with `glLight*()`. (OpenGL supports at least 8 light sources).
- Select a lighting model; with `glLightModel*()`. This defines the level of (global) ambient light, and the effective location of the viewpoint (for the lighting calculations).
- Define the material properties for each object in the scene with `glMaterial*()`.

```
void glLight{if}(GLenum light, GLenum pname,
TYPE param);
```

```
void glLight{if}v(GLenum light, GLenum pname,
TYPE *param);
```

create the light specified by `GL_LIGHT0`, `GL_LIGHT1`, ... with properties given by `pname`; `param` specifies the set of parameters for the properties `pname`.

Lighting models in OpenGL

In order to add lighting to a scene, the following steps are required:

- Define normals at each vertex for every object. (More on this soon)
- Create and position one or more light sources with `glLight*()`. (OpenGL supports at least 8 light sources).
- Select a lighting model; with `glLightModel*()`. This defines the level of (global) ambient light, and the effective location of the viewpoint (for the lighting calculations).
- Define the material properties for each object in the scene with `glMaterial*()`.

```
void glLight{if}(GLenum light, GLenum pname,
TYPE param);
```

```
void glLight{if}v(GLenum light, GLenum pname,
TYPE *param);
```

create the light specified by `GL_LIGHT0`, `GL_LIGHT1`, ... with properties given by `pname`; `param` specifies the set of parameters for the properties `pname`. The vector version (suffix '`v`') is used to specify an array of parameters.

Lighting models in OpenGL

In order to add lighting to a scene, the following steps are required:

- Define normals at each vertex for every object. (More on this soon)
- Create and position one or more light sources with `glLight*()`. (OpenGL supports at least 8 light sources).
- Select a lighting model; with `glLightModel*()`. This defines the level of (global) ambient light, and the effective location of the viewpoint (for the lighting calculations).
- Define the material properties for each object in the scene with `glMaterial*()`.

```
void glLight{if}(GLenum light, GLenum pname,
TYPE param);
```

```
void glLight{if}v(GLenum light, GLenum pname,
TYPE *param);
```

create the light specified by `GL_LIGHT0`, `GL_LIGHT1`, ... with properties given by `pname`; `param` specifies the set of parameters for the properties `pname`. The vector version (suffix '`v`') is used to specify an array of parameters.

Parameter name	Default Value	Meaning
GL_AMBIENT	(0.0, 0.0, 0.0, 1.0)	ambient color
GL_DIFFUSE	(1.0, 1.0, 1.0, 1.0)	diffuse color
GL_SPECULAR	(1.0, 1.0, 1.0, 1.0)	specular color
GL_POSITION	(0.0, 0.0, 1.0, 0.0)	(x, y, z, w): position or direction
GL_CONSTANT_ATTENUATION	1.0	see equation following
GL_LINEAR_ATTENUATION	0.0	
GL_QUADRATIC_ATTENUATION	0.0	

Parameter name	Default Value	Meaning
`GL_AMBIENT`	(0.0, 0.0, 0.0, 1.0)	ambient color
`GL_DIFFUSE`	(1.0, 1.0, 1.0, 1.0)	diffuse color
`GL_SPECULAR`	(1.0, 1.0, 1.0, 1.0)	specular color
`GL_POSITION`	(0.0, 0.0, 1.0, 0.0)	(x, y, z, w): position or direction
`GL_CONSTANT_ATTENUATION`	1.0	see equation following
`GL_LINEAR_ATTENUATION`	0.0	
`GL_QUADRATIC_ATTENUATION`	0.0	

There are also other parameters which restrict a light to be a spotlight.

Parameter name	Default Value	Meaning
`GL_AMBIENT`	(0.0, 0.0, 0.0, 1.0)	ambient color
`GL_DIFFUSE`	(1.0, 1.0, 1.0, 1.0)	diffuse color
`GL_SPECULAR`	(1.0, 1.0, 1.0, 1.0)	specular color
`GL_POSITION`	(0.0, 0.0, 1.0, 0.0)	(x, y, z, w): position or direction
`GL_CONSTANT_ATTENUATION`	1.0	see equation following
`GL_LINEAR_ATTENUATION`	0.0	
`GL_QUADRATIC_ATTENUATION`	0.0	

There are also other parameters which restrict a light to be a spotlight.

Light color: `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`

Parameter name	Default Value	Meaning
`GL_AMBIENT`	(0.0, 0.0, 0.0, 1.0)	ambient color
`GL_DIFFUSE`	(1.0, 1.0, 1.0, 1.0)	diffuse color
`GL_SPECULAR`	(1.0, 1.0, 1.0, 1.0)	specular color
`GL_POSITION`	(0.0, 0.0, 1.0, 0.0)	(x, y, z, w): position or direction
`GL_CONSTANT_ATTENUATION`	1.0	see equation following
`GL_LINEAR_ATTENUATION`	0.0	
`GL_QUADRATIC_ATTENUATION`	0.0	

There are also other parameters which restrict a light to be a spotlight.

Light color: `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`

These four-dimensional quantities specify the colors of the ambient, diffuse, and specular light emitted from a light source.

Parameter name	Default Value	Meaning
`GL_AMBIENT`	(0.0, 0.0, 0.0, 1.0)	ambient color
`GL_DIFFUSE`	(1.0, 1.0, 1.0, 1.0)	diffuse color
`GL_SPECULAR`	(1.0, 1.0, 1.0, 1.0)	specular color
`GL_POSITION`	(0.0, 0.0, 1.0, 0.0)	(x, y, z, w): position or direction
`GL_CONSTANT_ATTENUATION`	1.0	see equation following
`GL_LINEAR_ATTENUATION`	0.0	
`GL_QUADRATIC_ATTENUATION`	0.0	

There are also other parameters which restrict a light to be a spotlight.

Light color: `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`

These four-dimensional quantities specify the colors of the ambient, diffuse, and specular light emitted from a light source.

The default values for `GL_DIFFUSE` and `GL_SPECULAR` are for `GL_LIGHT0` only.

Parameter name	Default Value	Meaning
`GL_AMBIENT`	(0.0, 0.0, 0.0, 1.0)	ambient color
`GL_DIFFUSE`	(1.0, 1.0, 1.0, 1.0)	diffuse color
`GL_SPECULAR`	(1.0, 1.0, 1.0, 1.0)	specular color
`GL_POSITION`	(0.0, 0.0, 1.0, 0.0)	(x, y, z, w): position or direction
`GL_CONSTANT_ATTENUATION`	1.0	see equation following
`GL_LINEAR_ATTENUATION`	0.0	
`GL_QUADRATIC_ATTENUATION`	0.0	

There are also other parameters which restrict a light to be a spotlight.

Light color: `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`

These four-dimensional quantities specify the colors of the ambient, diffuse, and specular light emitted from a light source.

The default values for `GL_DIFFUSE` and `GL_SPECULAR` are for `GL_LIGHT0` only. Other lights default to black $(0.0, 0.0, 0.0, 1.0)$.

Parameter name	Default Value	Meaning
`GL_AMBIENT`	(0.0, 0.0, 0.0, 1.0)	ambient color
`GL_DIFFUSE`	(1.0, 1.0, 1.0, 1.0)	diffuse color
`GL_SPECULAR`	(1.0, 1.0, 1.0, 1.0)	specular color
`GL_POSITION`	(0.0, 0.0, 1.0, 0.0)	(x, y, z, w): position or direction
`GL_CONSTANT_ATTENUATION`	1.0	see equation following
`GL_LINEAR_ATTENUATION`	0.0	
`GL_QUADRATIC_ATTENUATION`	0.0	

There are also other parameters which restrict a light to be a spotlight.

Light color: `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`

These four-dimensional quantities specify the colors of the ambient, diffuse, and specular light emitted from a light source.

The default values for `GL_DIFFUSE` and `GL_SPECULAR` are for `GL_LIGHT0` only. Other lights default to black $(0.0, 0.0, 0.0, 1.0)$.

Light position: `GL_POSITION`

Parameter name	Default Value	Meaning
`GL_AMBIENT`	(0.0, 0.0, 0.0, 1.0)	ambient color
`GL_DIFFUSE`	(1.0, 1.0, 1.0, 1.0)	diffuse color
`GL_SPECULAR`	(1.0, 1.0, 1.0, 1.0)	specular color
`GL_POSITION`	(0.0, 0.0, 1.0, 0.0)	(x, y, z, w): position or direction
`GL_CONSTANT_ATTENUATION`	1.0	see equation following
`GL_LINEAR_ATTENUATION`	0.0	
`GL_QUADRATIC_ATTENUATION`	0.0	

There are also other parameters which restrict a light to be a spotlight.

Light color: `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`

These four-dimensional quantities specify the colors of the ambient, diffuse, and specular light emitted from a light source.

The default values for `GL_DIFFUSE` and `GL_SPECULAR` are for `GL_LIGHT0` only. Other lights default to black $(0.0, 0.0, 0.0, 1.0)$.

Light position: `GL_POSITION`

The fourth value specified for `GL_POSITION` controls whether the light is directional or positional.

Parameter name	Default Value	Meaning
`GL_AMBIENT`	(0.0, 0.0, 0.0, 1.0)	ambient color
`GL_DIFFUSE`	(1.0, 1.0, 1.0, 1.0)	diffuse color
`GL_SPECULAR`	(1.0, 1.0, 1.0, 1.0)	specular color
`GL_POSITION`	(0.0, 0.0, 1.0, 0.0)	(x, y, z, w): position or direction
`GL_CONSTANT_ATTENUATION`	1.0	see equation following
`GL_LINEAR_ATTENUATION`	0.0	
`GL_QUADRATIC_ATTENUATION`	0.0	

There are also other parameters which restrict a light to be a spotlight.

Light color: `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`

These four-dimensional quantities specify the colors of the ambient, diffuse, and specular light emitted from a light source.

The default values for `GL_DIFFUSE` and `GL_SPECULAR` are for `GL_LIGHT0` only. Other lights default to black $(0.0, 0.0, 0.0, 1.0)$.

Light position: `GL_POSITION`

The fourth value specified for `GL_POSITION` controls whether the light is directional or positional. A directional light is infinitely far away, such that the rays of light it emanates are

parallel (e.g. like the rays of light from the sun striking a small area on Earth).

parallel (e.g. like the rays of light from the sun striking a small area on Earth). If the w value is zero, the light is considered directional and the (x, y, z) values describe the direction of the light.

parallel (e.g. like the rays of light from the sun striking a small area on Earth). If the w value is zero, the light is considered directional and the (x, y, z) values describe the direction of the light.

If the $w-$value is non-zero the light is positional.

parallel (e.g. like the rays of light from the sun striking a small area on Earth). If the w value is zero, the light is considered directional and the (x, y, z) values describe the direction of the light.

If the $w-$value is non-zero the light is positional. The (x, y, z) values specify the location of the light which radiates in all directions.

parallel (e.g. like the rays of light from the sun striking a small area on Earth). If the w value is zero, the light is considered directional and the (x, y, z) values describe the direction of the light.

If the $w-$value is non-zero the light is positional. The (x, y, z) values specify the location of the light which radiates in all directions.

The direction of a directional light and the position of a positional light are both transformed by the MODELVIEW matrix.

parallel (e.g. like the rays of light from the sun striking a small area on Earth). If the w value is zero, the light is considered directional and the (x, y, z) values describe the direction of the light.

If the $w-$value is non-zero the light is positional. The (x, y, z) values specify the location of the light which radiates in all directions.

The direction of a directional light and the position of a positional light are both transformed by the MODELVIEW matrix. The PROJECTION matrix has no effect on a light source.

parallel (e.g. like the rays of light from the sun striking a small area on Earth). If the w value is zero, the light is considered directional and the (x, y, z) values describe the direction of the light.

If the $w-$value is non-zero the light is positional. The (x, y, z) values specify the location of the light which radiates in all directions.

The direction of a directional light and the position of a positional light are both transformed by the MODELVIEW matrix. The PROJECTION matrix has no effect on a light source.

The tutorial `lightposition` shows a simple use of lighting and the interaction with viewing position.

parallel (e.g. like the rays of light from the sun striking a small area on Earth). If the w value is zero, the light is considered directional and the (x, y, z) values describe the direction of the light.

If the $w-$value is non-zero the light is positional. The (x, y, z) values specify the location of the light which radiates in all directions.

The direction of a directional light and the position of a positional light are both transformed by the MODELVIEW matrix. The PROJECTION matrix has no effect on a light source.

The tutorial `lightposition` shows a simple use of lighting and the interaction with viewing position.

Light attenuation: `GL_CONSTANT_ATTENUATION`, `GL_LINEAR_ATTENUA`
`GL_QUADRATIC_ATTENUATION`

parallel (e.g. like the rays of light from the sun striking a small area on Earth). If the w value is zero, the light is considered directional and the (x, y, z) values describe the direction of the light.

If the $w-$value is non-zero the light is positional. The (x, y, z) values specify the location of the light which radiates in all directions.

The direction of a directional light and the position of a positional light are both transformed by the MODELVIEW matrix. The PROJECTION matrix has no effect on a light source.

The tutorial `lightposition` shows a simple use of lighting and the interaction with viewing position.

Light attenuation: `GL_CONSTANT_ATTENUATION`, `GL_LINEAR_ATTENUA` `GL_QUADRATIC_ATTENUATION`

As mentioned, light has the property that it reduces in intensity with distance.

parallel (e.g. like the rays of light from the sun striking a small area on Earth). If the w value is zero, the light is considered directional and the (x, y, z) values describe the direction of the light.

If the $w-$value is non-zero the light is positional. The (x, y, z) values specify the location of the light which radiates in all directions.

The direction of a directional light and the position of a positional light are both transformed by the MODELVIEW matrix. The PROJECTION matrix has no effect on a light source.

The tutorial `lightposition` shows a simple use of lighting and the interaction with viewing position.

Light attenuation: `GL_CONSTANT_ATTENUATION`, `GL_LINEAR_ATTENUA` `GL_QUADRATIC_ATTENUATION`

As mentioned, light has the property that it reduces in intensity with distance. That is, it attenuates.

parallel (e.g. like the rays of light from the sun striking a small area on Earth). If the w value is zero, the light is considered directional and the (x, y, z) values describe the direction of the light.

If the $w-$value is non-zero the light is positional. The (x, y, z) values specify the location of the light which radiates in all directions.

The direction of a directional light and the position of a positional light are both transformed by the MODELVIEW matrix. The PROJECTION matrix has no effect on a light source.

The tutorial `lightposition` shows a simple use of lighting and the interaction with viewing position.

Light attenuation: `GL_CONSTANT_ATTENUATION`, `GL_LINEAR_ATTENUA` `GL_QUADRATIC_ATTENUATION`

As mentioned, light has the property that it reduces in intensity with distance. That is, it attenuates.

The attenuation is calculated from the expression

parallel (e.g. like the rays of light from the sun striking a small area on Earth). If the w value is zero, the light is considered directional and the (x, y, z) values describe the direction of the light.

If the $w-$value is non-zero the light is positional. The (x, y, z) values specify the location of the light which radiates in all directions.

The direction of a directional light and the position of a positional light are both transformed by the MODELVIEW matrix. The PROJECTION matrix has no effect on a light source.

The tutorial **`lightposition`** shows a simple use of lighting and the interaction with viewing position.

Light attenuation: `GL_CONSTANT_ATTENUATION`, `GL_LINEAR_ATTENUA` `GL_QUADRATIC_ATTENUATION`

As mentioned, light has the property that it reduces in intensity with distance. That is, it attenuates.

The attenuation is calculated from the expression

$$\text{attenuation factor} = \frac{1}{k_c + k_l d + k_q d^2}$$

parallel (e.g. like the rays of light from the sun striking a small area on Earth). If the w value is zero, the light is considered directional and the (x, y, z) values describe the direction of the light.

If the $w-$value is non-zero the light is positional. The (x, y, z) values specify the location of the light which radiates in all directions.

The direction of a directional light and the position of a positional light are both transformed by the MODELVIEW matrix. The PROJECTION matrix has no effect on a light source.

The tutorial `lightposition` shows a simple use of lighting and the interaction with viewing position.

Light attenuation: `GL_CONSTANT_ATTENUATION`, `GL_LINEAR_ATTENUA` `GL_QUADRATIC_ATTENUATION`

As mentioned, light has the property that it reduces in intensity with distance. That is, it attenuates.

The attenuation is calculated from the expression

$$\text{attenuation factor} = \frac{1}{k_c + k_l d + k_q d^2}$$

where d is the distance from the light source, and k_c, k_l, and k_q are the constant, linear, and quadratic attenuation terms.

parallel (e.g. like the rays of light from the sun striking a small area on Earth). If the w value is zero, the light is considered directional and the (x, y, z) values describe the direction of the light.

If the $w-$value is non-zero the light is positional. The (x, y, z) values specify the location of the light which radiates in all directions.

The direction of a directional light and the position of a positional light are both transformed by the MODELVIEW matrix. The PROJECTION matrix has no effect on a light source.

The tutorial `lightposition` shows a simple use of lighting and the interaction with viewing position.

Light attenuation: `GL_CONSTANT_ATTENUATION`, `GL_LINEAR_ATTENUA` `GL_QUADRATIC_ATTENUATION`

As mentioned, light has the property that it reduces in intensity with distance. That is, it attenuates.

The attenuation is calculated from the expression

$$\text{attenuation factor} = \frac{1}{k_c + k_l d + k_q d^2}$$

where d is the distance from the light source, and k_c, k_l, and k_q are the constant, linear, and quadratic attenuation terms.

First example

First example

The following simple program `light.c` shows a lighted sphere:

First example

The following simple program `light.c` shows a lighted sphere:

```
/*  Initialize material property, light source,
*   lighting model, and depth buffer.  */
void init(void)
{
```

First example

The following simple program `light.c` shows a lighted sphere:

```
/*  Initialize material property, light source,
*   lighting model, and depth buffer.  */
void init(void)
{
  GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
  GLfloat mat_shininess[] = { 50.0 };
  GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
  GLfloat white_light_[] = { 1.0, 1.0, 1.0, 0.0 };
```

First example

The following simple program `light.c` shows a lighted sphere:

```
/*  Initialize material property, light source,
*   lighting model, and depth buffer.  */
void init(void)
{
  GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
  GLfloat mat_shininess[] = { 50.0 };
  GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
  GLfloat white_light_[] = { 1.0, 1.0, 1.0, 0.0 };

  glClearColor (0.0, 0.0, 0.0, 0.0);
  glShadeModel (GL_SMOOTH);
```

First example

The following simple program `light.c` shows a lighted sphere:

```
/*  Initialize material property, light source,
*   lighting model, and depth buffer.  */
void init(void)
{
  GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
  GLfloat mat_shininess[] = { 50.0 };
  GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
  GLfloat white_light_[] = { 1.0, 1.0, 1.0, 0.0 };

  glClearColor (0.0, 0.0, 0.0, 0.0);
  glShadeModel (GL_SMOOTH);

  glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
  glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
```

First example

The following simple program `light.c` shows a lighted sphere:

```
/*  Initialize material property, light source,
*   lighting model, and depth buffer.  */
void init(void)
{
  GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
  GLfloat mat_shininess[] = { 50.0 };
  GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
  GLfloat white_light_[] = { 1.0, 1.0, 1.0, 0.0 };

  glClearColor (0.0, 0.0, 0.0, 0.0);
  glShadeModel (GL_SMOOTH);

  glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
  glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
  glLightfv(GL_LIGHT0, GL_POSITION, light_position);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, white_light);
  glLightfv(GL_LIGHT0, GL_SPECULAR, white_light);
```

First example

The following simple program `light.c` shows a lighted sphere:

```
/*  Initialize material property, light source,
*   lighting model, and depth buffer.  */
void init(void)
{
  GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
  GLfloat mat_shininess[] = { 50.0 };
  GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
  GLfloat white_light_[] = { 1.0, 1.0, 1.0, 0.0 };

  glClearColor (0.0, 0.0, 0.0, 0.0);
  glShadeModel (GL_SMOOTH);

  glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
  glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
  glLightfv(GL_LIGHT0, GL_POSITION, light_position);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, white_light);
  glLightfv(GL_LIGHT0, GL_SPECULAR, white_light);

  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glEnable(GL_DEPTH_TEST);
}
```

```
void display(void)
{
  glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glutSolidSphere (1.0, 20, 16);
  glFlush ();
}
```

```
void display(void)
{
  glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glutSolidSphere (1.0, 20, 16);
  glFlush ();
}
```

Second example

```
void display(void)
{
  glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glutSolidSphere (1.0, 20, 16);
  glFlush ();
}
```

Second example

This example, `movelight.c`, illustrates how the position of a light is transformed by the MODELVIEW matrix:

```
void display(void)
{
  glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glutSolidSphere (1.0, 20, 16);
  glFlush ();
}
```

Second example

This example, `movelight.c`, illustrates how the position of a light is transformed by the MODELVIEW matrix:

```
void init(void)
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL_SMOOTH);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
}
```

```
void display(void)
{
  glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glutSolidSphere (1.0, 20, 16);
  glFlush ();
}
```

Second example

This example, `movelight.c`, illustrates how the position of a light is transformed by the MODELVIEW matrix:

```
void init(void)
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL_SMOOTH);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
}
```

```
void display(void)
{
   GLfloat position[] = { 0.0, 0.0, 1.5, 1.0 };
```

```
void display(void)
{
   GLfloat position[] = { 0.0, 0.0, 1.5, 1.0 };

   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glPushMatrix ();
```

```
void display(void)
{
   GLfloat position[] = { 0.0, 0.0, 1.5, 1.0 };

   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glPushMatrix ();

   glPushMatrix ();
   glRotated ((GLdouble) spin, 1.0, 0.0, 0.0);
   glLightfv (GL_LIGHT0, GL_POSITION, position);
   glTranslated (0.0, 0.0, 1.5);
```

```
void display(void)
{
   GLfloat position[] = { 0.0, 0.0, 1.5, 1.0 };

   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glPushMatrix ();

   glPushMatrix ();
   glRotated ((GLdouble) spin, 1.0, 0.0, 0.0);
   glLightfv (GL_LIGHT0, GL_POSITION, position);
   glTranslated (0.0, 0.0, 1.5);

   glDisable (GL_LIGHTING);   /* Draw an unlit wire cube at
   glColor3f (0.0, 1.0, 1.0); /* the position of the light.
   glutWireCube (0.1);
   glEnable (GL_LIGHTING);
   glPopMatrix ();
```

```
void display(void)
{
   GLfloat position[] = { 0.0, 0.0, 1.5, 1.0 };

   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glPushMatrix ();

   glPushMatrix ();
   glRotated ((GLdouble) spin, 1.0, 0.0, 0.0);
   glLightfv (GL_LIGHT0, GL_POSITION, position);
   glTranslated (0.0, 0.0, 1.5);

   glDisable (GL_LIGHTING);   /* Draw an unlit wire cube at
   glColor3f (0.0, 1.0, 1.0); /* the position of the light.
   glutWireCube (0.1);
   glEnable (GL_LIGHTING);
   glPopMatrix ();
```

```
void display(void)
{
   GLfloat position[] = { 0.0, 0.0, 1.5, 1.0 };

   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glPushMatrix ();

   glPushMatrix ();
   glRotated ((GLdouble) spin, 1.0, 0.0, 0.0);
   glLightfv (GL_LIGHT0, GL_POSITION, position);
   glTranslated (0.0, 0.0, 1.5);

   glDisable (GL_LIGHTING);   /* Draw an unlit wire cube at
   glColor3f (0.0, 1.0, 1.0); /* the position of the light.
   glutWireCube (0.1);
   glEnable (GL_LIGHTING);
   glPopMatrix ();

   glutSolidTorus (0.275, 0.85, 8, 15);
```

```
void display(void)
{
   GLfloat position[] = { 0.0, 0.0, 1.5, 1.0 };

   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glPushMatrix ();

   glPushMatrix ();
   glRotated ((GLdouble) spin, 1.0, 0.0, 0.0);
   glLightfv (GL_LIGHT0, GL_POSITION, position);
   glTranslated (0.0, 0.0, 1.5);

   glDisable (GL_LIGHTING);   /* Draw an unlit wire cube at
   glColor3f (0.0, 1.0, 1.0); /* the position of the light.
   glutWireCube (0.1);
   glEnable (GL_LIGHTING);
   glPopMatrix ();

   glutSolidTorus (0.275, 0.85, 8, 15);

   glPopMatrix ();
   glFlush ();
}
```

```
void display(void)
{
   GLfloat position[] = { 0.0, 0.0, 1.5, 1.0 };

   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glPushMatrix ();

   glPushMatrix ();
   glRotated ((GLdouble) spin, 1.0, 0.0, 0.0);
   glLightfv (GL_LIGHT0, GL_POSITION, position);
   glTranslated (0.0, 0.0, 1.5);

   glDisable (GL_LIGHTING);   /* Draw an unlit wire cube at
   glColor3f (0.0, 1.0, 1.0); /* the position of the light.
   glutWireCube (0.1);
   glEnable (GL_LIGHTING);
   glPopMatrix ();

   glutSolidTorus (0.275, 0.85, 8, 15);

   glPopMatrix ();
   glFlush ();
}
```

```
void reshape (int w, int h)
{
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(40.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0)
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
}
```

```
void reshape (int w, int h)
{
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(40.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0)
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
}


void mouse(int button, int state, int x, int y) {
   switch (button) {
      case GLUT_LEFT_BUTTON:
         if (state == GLUT_DOWN) {
            spin = (spin + 30) % 360;
            glutPostRedisplay();
         }
         break;
      default:
         break;
   }
}
```

```
void keyboard(unsigned char key, int x, int y) {
   switch (key) {
      case 27:
         exit(0);
         break;
   }
}
```

```
void keyboard(unsigned char key, int x, int y) {
   switch (key) {
      case 27:
         exit(0);
         break;
   }
}


int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH
   glutInitWindowSize (500, 500);
   glutInitWindowPosition (100, 100);
   glutCreateWindow (argv[0]);
   init ();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMouseFunc(mouse);
   glutKeyboardFunc(keyboard);
   glutMainLoop();
   return 0;
}
```

Material properties

Material properties

The colors of light reflected by a primitive are set with the function `glMaterial*()`.

Material properties

The colors of light reflected by a primitive are set with the function `glMaterial*()`. It has the form:

Material properties

The colors of light reflected by a primitive are set with the function `glMaterial*()`. It has the form:

```
void glMaterial{if}(GLenum face, GLenum pname, TYPE
param);
```

Material properties

The colors of light reflected by a primitive are set with the function `glMaterial*()`. It has the form:

```
void glMaterial{if}(GLenum face, GLenum pname, TYPE
param);
```

```
void glMaterial{if}v(GLenum face, GLenum pname, TYPE
*param);
```

Material properties

The colors of light reflected by a primitive are set with the function `glMaterial*()`. It has the form:

```
void glMaterial{if}(GLenum face, GLenum pname, TYPE
param);
```

```
void glMaterial{if}v(GLenum face, GLenum pname, TYPE
*param);
```

where `face` can be `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`, and `pname` and `param` are defined in the following table:

Material properties

The colors of light reflected by a primitive are set with the function `glMaterial*()`. It has the form:

```
void glMaterial{if}(GLenum face, GLenum pname, TYPE param);
```

```
void glMaterial{if}v(GLenum face, GLenum pname, TYPE *param);
```

where `face` can be `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`, and `pname` and `param` are defined in the following table:

Parameter name	Default Value	Meaning
`GL_AMBIENT`	(0.2, 0.2, 0.2, 1.0)	ambient color of material
`GL_DIFFUSE`	(0.8, 0.8, 0.8, 1.0)	diffuse color of material
`GL_SPECULAR`	(0.0, 0.0, 0.0, 1.0)	specular color of material
`GL_EMISSION`	(0.0, 0.0, 0.0, 1.0)	emissive color of material
`GL_SHININESS`	0.0	specular exponent

Material properties

The colors of light reflected by a primitive are set with the function `glMaterial*()`. It has the form:

```
void glMaterial{if}(GLenum face, GLenum pname, TYPE param);
```

```
void glMaterial{if}v(GLenum face, GLenum pname, TYPE *param);
```

where `face` can be `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`, and `pname` and `param` are defined in the following table:

Parameter name	Default Value	Meaning
`GL_AMBIENT`	(0.2, 0.2, 0.2, 1.0)	ambient color of material
`GL_DIFFUSE`	(0.8, 0.8, 0.8, 1.0)	diffuse color of material
`GL_SPECULAR`	(0.0, 0.0, 0.0, 1.0)	specular color of material
`GL_EMISSION`	(0.0, 0.0, 0.0, 1.0)	emissive color of material
`GL_SHININESS`	0.0	specular exponent

The parameter `GL_EMISSION` allows a body to emit light (for modelling lamps, etc.).

Material properties

The colors of light reflected by a primitive are set with the function `glMaterial*()`. It has the form:

```
void glMaterial{if}(GLenum face, GLenum pname, TYPE param);
```

```
void glMaterial{if}v(GLenum face, GLenum pname, TYPE *param);
```

where `face` can be `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`, and `pname` and `param` are defined in the following table:

Parameter name	Default Value	Meaning
`GL_AMBIENT`	(0.2, 0.2, 0.2, 1.0)	ambient color of material
`GL_DIFFUSE`	(0.8, 0.8, 0.8, 1.0)	diffuse color of material
`GL_SPECULAR`	(0.0, 0.0, 0.0, 1.0)	specular color of material
`GL_EMISSION`	(0.0, 0.0, 0.0, 1.0)	emissive color of material
`GL_SHININESS`	0.0	specular exponent

The parameter `GL_EMISSION` allows a body to emit light (for modelling lamps, etc.). Light from this source does not illuminate any other part of the scene.

Material properties

The colors of light reflected by a primitive are set with the function `glMaterial*()`. It has the form:

```
void glMaterial{if}(GLenum face, GLenum pname, TYPE param);
```

```
void glMaterial{if}v(GLenum face, GLenum pname, TYPE *param);
```

where `face` can be `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`, and `pname` and `param` are defined in the following table:

Parameter name	Default Value	Meaning
`GL_AMBIENT`	(0.2, 0.2, 0.2, 1.0)	ambient color of material
`GL_DIFFUSE`	(0.8, 0.8, 0.8, 1.0)	diffuse color of material
`GL_SPECULAR`	(0.0, 0.0, 0.0, 1.0)	specular color of material
`GL_EMISSION`	(0.0, 0.0, 0.0, 1.0)	emissive color of material
`GL_SHININESS`	0.0	specular exponent

The parameter `GL_EMISSION` allows a body to emit light (for modelling lamps, etc.). Light from this source does not illuminate any other part of the scene.

The tutorial `lightmaterial` shows interaction between light sources and material properties.

Material properties

The colors of light reflected by a primitive are set with the function `glMaterial*()`. It has the form:

```
void glMaterial{if}(GLenum face, GLenum pname, TYPE param);
```

```
void glMaterial{if}v(GLenum face, GLenum pname, TYPE *param);
```

where `face` can be `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`, and `pname` and `param` are defined in the following table:

Parameter name	Default Value	Meaning
`GL_AMBIENT`	(0.2, 0.2, 0.2, 1.0)	ambient color of material
`GL_DIFFUSE`	(0.8, 0.8, 0.8, 1.0)	diffuse color of material
`GL_SPECULAR`	(0.0, 0.0, 0.0, 1.0)	specular color of material
`GL_EMISSION`	(0.0, 0.0, 0.0, 1.0)	emissive color of material
`GL_SHININESS`	0.0	specular exponent

The parameter `GL_EMISSION` allows a body to emit light (for modelling lamps, etc.). Light from this source does not illuminate any other part of the scene.

The tutorial `lightmaterial` shows interaction between light sources and material properties.

The lighting model

The lighting model

The OpenGL lighting model has four components:

The lighting model

The OpenGL lighting model has four components:

1. The global ambient light intensity

The lighting model

The OpenGL lighting model has four components:

1. The global ambient light intensity
2. The viewpoint position (local or at infinity)

The lighting model

The OpenGL lighting model has four components:

1. The global ambient light intensity
2. The viewpoint position (local or at infinity)
3. Whether front and back surfaces should have different lighting calculations

The lighting model

The OpenGL lighting model has four components:

1. The global ambient light intensity
2. The viewpoint position (local or at infinity)
3. Whether front and back surfaces should have different lighting calculations
4. Whether or not the specular color should be separated from ambient and diffuse colors and applied after texturing.

The lighting model

The OpenGL lighting model has four components:

1. The global ambient light intensity
2. The viewpoint position (local or at infinity)
3. Whether front and back surfaces should have different lighting calculations
4. Whether or not the specular color should be separated from ambient and diffuse colors and applied after texturing.

All those properties can be set individually with the function `glLightModel*()`.

The lighting model

The OpenGL lighting model has four components:

1. The global ambient light intensity
2. The viewpoint position (local or at infinity)
3. Whether front and back surfaces should have different lighting calculations
4. Whether or not the specular color should be separated from ambient and diffuse colors and applied after texturing.

All those properties can be set individually with the function `glLightModel*()`.

```
void glLightModel{if}(GLenum pname, TYPE param);
```

The lighting model

The OpenGL lighting model has four components:

1. The global ambient light intensity
2. The viewpoint position (local or at infinity)
3. Whether front and back surfaces should have different lighting calculations
4. Whether or not the specular color should be separated from ambient and diffuse colors and applied after texturing.

All those properties can be set individually with the function `glLightModel*()`.

```
void glLightModel{if}(GLenum pname, TYPE param);
void glLightModel{if}v(GLenum pname, TYPE *param);
```

The lighting model

The OpenGL lighting model has four components:

1. The global ambient light intensity
2. The viewpoint position (local or at infinity)
3. Whether front and back surfaces should have different lighting calculations
4. Whether or not the specular color should be separated from ambient and diffuse colors and applied after texturing.

All those properties can be set individually with the function `glLightModel*()`.

```
void glLightModel{if}(GLenum pname, TYPE param);
void glLightModel{if}v(GLenum pname, TYPE *param);
```

Again, `pname` specifies the property; `param` specifies the set of parameters for the property `pname`.

The lighting model

The OpenGL lighting model has four components:

1. The global ambient light intensity
2. The viewpoint position (local or at infinity)
3. Whether front and back surfaces should have different lighting calculations
4. Whether or not the specular color should be separated from ambient and diffuse colors and applied after texturing.

All those properties can be set individually with the function `glLightModel*()`.

```
void glLightModel{if}(GLenum pname, TYPE param);
void glLightModel{if}v(GLenum pname, TYPE *param);
```

Again, `pname` specifies the property; `param` specifies the set of parameters for the property `pname`.

In the following, `GL_LIGHT_MODEL` precedes the parameter name:

The lighting model

The OpenGL lighting model has four components:

1. The global ambient light intensity
2. The viewpoint position (local or at infinity)
3. Whether front and back surfaces should have different lighting calculations
4. Whether or not the specular color should be separated from ambient and diffuse colors and applied after texturing.

All those properties can be set individually with the function `glLightModel*()`.

```
void glLightModel{if}(GLenum pname, TYPE param);
void glLightModel{if}v(GLenum pname, TYPE *param);
```

Again, `pname` specifies the property; `param` specifies the set of parameters for the property `pname`.

In the following, `GL_LIGHT_MODEL` precedes the parameter name:

Parameter name	Default Value	Meaning
`_AMBIENT`	(0.2, 0.2, 0.2, 1.0)	ambient intensity for scene
`_LOCAL_VIEWER`	0.0	how specular reflection angles are calculated
`_TWO_SIDE`	0.0	one sides or two sided lighting
`_COLOR_CONTROL`	`GL_SINGLE_COLOR`	whether specular color is calculated separately

Parameter name	Default Value	Meaning
`_AMBIENT`	(0.2, 0.2, 0.2, 1.0)	ambient intensity for scene
`_LOCAL_VIEWER`	0.0	how specular reflection angles are calculated
`_TWO_SIDE`	0.0	one sides or two sided lighting
`_COLOR_CONTROL`	`GL_SINGLE_COLOR`	whether specular color is calculated separately

How a vertex's color is calculated in OpenGL

How a vertex's color is calculated in OpenGL

The color of a vertex under lighting in OpenGL is:

How a vertex's color is calculated in OpenGL

The color of a vertex under lighting in OpenGL is:

color = emission at the vertex +

How a vertex's color is calculated in OpenGL

The color of a vertex under lighting in OpenGL is:

color = emission at the vertex +
global ambient light scaled by the ambient coefficient of the material +

How a vertex's color is calculated in OpenGL

The color of a vertex under lighting in OpenGL is:

color = emission at the vertex +
global ambient light scaled by the ambient coefficient of the material +
the ambient, diffuse, and specular components from all light sources, properly attenuated

How a vertex's color is calculated in OpenGL

The color of a vertex under lighting in OpenGL is:

color = emission at the vertex +
global ambient light scaled by the ambient coefficient of the material +
the ambient, diffuse, and specular components from all light sources, properly attenuated

The contribution from individual light sources is calculated as follows:

How a vertex's color is calculated in OpenGL

The color of a vertex under lighting in OpenGL is:

color = emission at the vertex +
global ambient light scaled by the ambient coefficient of the material +
the ambient, diffuse, and specular components from all light sources, properly attenuated

The contribution from individual light sources is calculated as follows:

color = $\text{emission}_{\text{material}}$+

How a vertex's color is calculated in OpenGL

The color of a vertex under lighting in OpenGL is:

color = emission at the vertex +
global ambient light scaled by the ambient coefficient of the material +
the ambient, diffuse, and specular components from all light sources, properly attenuated

The contribution from individual light sources is calculated as follows:

color = $\text{emission}_{\text{material}}$ +
$\text{ambient}_{\text{light model}} * \text{ambient}_{\text{material}}$ +

How a vertex's color is calculated in OpenGL

The color of a vertex under lighting in OpenGL is:

color = emission at the vertex +
global ambient light scaled by the ambient coefficient of the material +
the ambient, diffuse, and specular components from all light sources, properly attenuated

The contribution from individual light sources is calculated as follows:

$$\begin{aligned}\text{color} = {} & \text{emission}_{\text{material}} + \\ & \text{ambient}_{\text{light model}} * \text{ambient}_{\text{material}} + \\ & \Sigma_{i=0}^{n-1} \left(\frac{1}{k_c + k_l d + k_q d^2} \right) * (\text{spotlight effect})_i *\end{aligned}$$

How a vertex's color is calculated in OpenGL

The color of a vertex under lighting in OpenGL is:

color = emission at the vertex +
global ambient light scaled by the ambient coefficient of the material +
the ambient, diffuse, and specular components from all light sources, properly attenuated

The contribution from individual light sources is calculated as follows:

$$
\begin{aligned}
\text{color} = {} & \text{emission}_{\text{material}} + \\
& \text{ambient}_{\text{light model}} * \text{ambient}_{\text{material}} + \\
& \Sigma_{i=0}^{n-1} \left(\frac{1}{k_c + k_l d + k_q d^2} \right) * (\text{spotlight effect})_i * \\
& [\text{ambient}_{\text{light}} * \text{ambient}_{\text{material}} +
\end{aligned}
$$

How a vertex's color is calculated in OpenGL

The color of a vertex under lighting in OpenGL is:

color = emission at the vertex +
global ambient light scaled by the ambient coefficient of the material +
the ambient, diffuse, and specular components from all light sources, properly attenuated

The contribution from individual light sources is calculated as follows:

$$
\begin{aligned}
\text{color} = \; & \text{emission}_{\text{material}} + \\
& \text{ambient}_{\text{light model}} * \text{ambient}_{\text{material}} + \\
& \Sigma_{i=0}^{n-1} \left(\frac{1}{k_c + k_l d + k_q d^2} \right) * (\text{spotlight effect})_i * \\
& [\text{ambient}_{\text{light}} * \text{ambient}_{\text{material}} + \\
& (\max\{\mathbf{L}_i \cdot \mathbf{n}, 0\}) * \text{diffuse}_{\text{light}} * \text{diffuse}_{\text{material}} +
\end{aligned}
$$

How a vertex's color is calculated in OpenGL

The color of a vertex under lighting in OpenGL is:

color = emission at the vertex +
global ambient light scaled by the ambient coefficient of the material +
the ambient, diffuse, and specular components from all light sources, properly attenuated

The contribution from individual light sources is calculated as follows:

$$
\begin{aligned}
\text{color} = \; & \text{emission}_{\text{material}} + \\
& \text{ambient}_{\text{light model}} * \text{ambient}_{\text{material}} + \\
& \Sigma_{i=0}^{n-1} \left(\frac{1}{k_c + k_l d + k_q d^2} \right) * (\text{spotlight effect})_i * \\
& [\text{ambient}_{\text{light}} * \text{ambient}_{\text{material}} + \\
& (\max\{\mathbf{L}_i \cdot \mathbf{n}, 0\}) * \text{diffuse}_{\text{light}} * \text{diffuse}_{\text{material}} + \\
& \text{specular effect }]_i
\end{aligned}
$$

How a vertex's color is calculated in OpenGL

The color of a vertex under lighting in OpenGL is:

color = emission at the vertex +
global ambient light scaled by the ambient coefficient of the material +
the ambient, diffuse, and specular components from all light sources, properly attenuated

The contribution from individual light sources is calculated as follows:

$$
\begin{aligned}
\text{color} = \; & \text{emission}_{\text{material}} + \\
& \text{ambient}_{\text{light model}} * \text{ambient}_{\text{material}} + \\
& \Sigma_{i=0}^{n-1} \left(\frac{1}{k_c + k_l d + k_q d^2} \right) * (\text{spotlight effect})_i * \\
& [\text{ambient}_{\text{light}} * \text{ambient}_{\text{material}} + \\
& (\max\{\mathbf{L}_i \cdot \mathbf{n}, 0\}) * \text{diffuse}_{\text{light}} * \text{diffuse}_{\text{material}} + \\
& \text{specular effect }]_i
\end{aligned}
$$

where $\mathbf{n} = (\mathbf{n}_x, \mathbf{n}_y, \mathbf{n}_z)$ is the unit normal vector at the vertex and $\mathbf{L} = (\mathbf{L}_x, \mathbf{L}_y, \mathbf{L}_z)$ is the unit vector pointing from the vertex to the light.

How a vertex's color is calculated in OpenGL

The color of a vertex under lighting in OpenGL is:

color = emission at the vertex +
global ambient light scaled by the ambient coefficient of the material +
the ambient, diffuse, and specular components from all light sources, properly attenuated

The contribution from individual light sources is calculated as follows:

$$
\begin{aligned}
\text{color} = \; & \text{emission}_{\text{material}} + \\
& \text{ambient}_{\text{light model}} * \text{ambient}_{\text{material}} + \\
& \Sigma_{i=0}^{n-1} \left(\frac{1}{k_c + k_l d + k_q d^2} \right) * (\text{spotlight effect})_i * \\
& [\text{ambient}_{\text{light}} * \text{ambient}_{\text{material}} + \\
& (\max\{\mathbf{L}_i \cdot \mathbf{n}, 0\}) * \text{diffuse}_{\text{light}} * \text{diffuse}_{\text{material}} + \\
& \text{specular effect} \,]_i
\end{aligned}
$$

where $\mathbf{n} = (\mathbf{n}_x, \mathbf{n}_y, \mathbf{n}_z)$ is the unit normal vector at the vertex and $\mathbf{L} = (\mathbf{L}_x, \mathbf{L}_y, \mathbf{L}_z)$ is the unit vector pointing from the vertex to the light. Thus, the diffuse component is highest when $\mathbf{n}$ and $\mathbf{L}$ are parallel.

How a vertex's color is calculated in OpenGL

The color of a vertex under lighting in OpenGL is:

$$\begin{aligned} \text{color} \;=\; & \text{emission at the vertex} + \\ & \text{global ambient light scaled by the ambient coefficient of the material} + \\ & \text{the ambient, diffuse, and specular components from all light sources, properly attenuated} \end{aligned}$$

The contribution from individual light sources is calculated as follows:

$$\begin{aligned} \text{color} \;=\; & \text{emission}_{\text{material}} + \\ & \text{ambient}_{\text{light model}} * \text{ambient}_{\text{material}} + \\ & \Sigma_{i=0}^{n-1} \left(\frac{1}{k_c + k_l d + k_q d^2} \right) * (\text{spotlight effect})_i * \\ & [\text{ambient}_{\text{light}} * \text{ambient}_{\text{material}} + \\ & (\max\{\mathbf{L}_i \cdot \mathbf{n}, 0\}) * \text{diffuse}_{\text{light}} * \text{diffuse}_{\text{material}} + \\ & \text{specular effect} \,]_i \end{aligned}$$

where $\mathbf{n} = (\mathbf{n}_x, \mathbf{n}_y, \mathbf{n}_z)$ is the unit normal vector at the vertex and $\mathbf{L} = (\mathbf{L}_x, \mathbf{L}_y, \mathbf{L}_z)$ is the unit vector pointing from the vertex to the light. Thus, the diffuse component is highest when $\mathbf{n}$ and $\mathbf{L}$ are parallel. The calculation for the specular effect is similar but incorporates also the position of the viewpoint.

How a vertex's color is calculated in OpenGL

The color of a vertex under lighting in OpenGL is:

$$
\begin{aligned}
\text{color} \;=\; & \text{emission at the vertex} + \\
& \text{global ambient light scaled by the ambient coefficient of the material} + \\
& \text{the ambient, diffuse, and specular components from all light sources, properly attenuated}
\end{aligned}
$$

The contribution from individual light sources is calculated as follows:

$$
\begin{aligned}
\text{color} \;=\; & \text{emission}_{\text{material}} + \\
& \text{ambient}_{\text{light model}} * \text{ambient}_{\text{material}} + \\
& \Sigma_{i=0}^{n-1} \left(\frac{1}{k_c + k_l d + k_q d^2} \right) * (\text{spotlight effect})_i * \\
& [\text{ambient}_{\text{light}} * \text{ambient}_{\text{material}} + \\
& (\max\{\mathbf{L}_i \cdot \mathbf{n}, 0\}) * \text{diffuse}_{\text{light}} * \text{diffuse}_{\text{material}} + \\
& \text{specular effect} \,]_i
\end{aligned}
$$

where $\mathbf{n} = (\mathbf{n}_x, \mathbf{n}_y, \mathbf{n}_z)$ is the unit normal vector at the vertex and $\mathbf{L} = (\mathbf{L}_x, \mathbf{L}_y, \mathbf{L}_z)$ is the unit vector pointing from the vertex to the light. Thus, the diffuse component is highest when $\mathbf{n}$ and $\mathbf{L}$ are parallel. The calculation for the specular effect is similar but incorporates also the position of the viewpoint.

The result is clamped to $[0, 1]$.

How a vertex's color is calculated in OpenGL

The color of a vertex under lighting in OpenGL is:

color = emission at the vertex +
global ambient light scaled by the ambient coefficient of the material +
the ambient, diffuse, and specular components from all light sources, properly attenuated

The contribution from individual light sources is calculated as follows:

$$
\begin{aligned}
\text{color} = {} & \text{emission}_{\text{material}} + \\
& \text{ambient}_{\text{light model}} * \text{ambient}_{\text{material}} + \\
& \Sigma_{i=0}^{n-1} \left(\frac{1}{k_c + k_l d + k_q d^2} \right) * (\text{spotlight effect})_i * \\
& [\text{ambient}_{\text{light}} * \text{ambient}_{\text{material}} + \\
& (\max\{\mathbf{L}_i \cdot \mathbf{n}, 0\}) * \text{diffuse}_{\text{light}} * \text{diffuse}_{\text{material}} + \\
& \text{specular effect }]_i
\end{aligned}
$$

where $\mathbf{n} = (\mathbf{n}_x, \mathbf{n}_y, \mathbf{n}_z)$ is the unit normal vector at the vertex and $\mathbf{L} = (\mathbf{L}_x, \mathbf{L}_y, \mathbf{L}_z)$ is the unit vector pointing from the vertex to the light. Thus, the diffuse component is highest when $\mathbf{n}$ and $\mathbf{L}$ are parallel. The calculation for the specular effect is similar but incorporates also the position of the viewpoint.

The result is clamped to $[0, 1]$.

Shading — what happens between vertices

Shading — what happens between vertices

Clearly, the color and intensity of light at interior points in a polygon could be calculated similarly.

Shading — what happens between vertices

Clearly, the color and intensity of light at interior points in a polygon could be calculated similarly. However, this would be computationally expensive, so other methods are used.

Shading — what happens between vertices

Clearly, the color and intensity of light at interior points in a polygon could be calculated similarly. However, this would be computationally expensive, so other methods are used.

As mentioned, OpenGL provides an interpolated shading model known as Gouraud shading.

Shading — what happens between vertices

Clearly, the color and intensity of light at interior points in a polygon could be calculated similarly. However, this would be computationally expensive, so other methods are used.

As mentioned, OpenGL provides an interpolated shading model known as Gouraud shading. Gouraud shading linearly interpolates the color of an interior pixel from the color at the vertices.

Shading — what happens between vertices

Clearly, the color and intensity of light at interior points in a polygon could be calculated similarly. However, this would be computationally expensive, so other methods are used.

As mentioned, OpenGL provides an interpolated shading model known as Gouraud shading. Gouraud shading linearly interpolates the color of an interior pixel from the color at the vertices. This interpolation is incorporated directly into the rasterization algorithm.

Shading — what happens between vertices

Clearly, the color and intensity of light at interior points in a polygon could be calculated similarly. However, this would be computationally expensive, so other methods are used.

As mentioned, OpenGL provides an interpolated shading model known as Gouraud shading. Gouraud shading linearly interpolates the color of an interior pixel from the color at the vertices. This interpolation is incorporated directly into the rasterization algorithm.

Another common shading model is Phong shading.

Shading — what happens between vertices

Clearly, the color and intensity of light at interior points in a polygon could be calculated similarly. However, this would be computationally expensive, so other methods are used.

As mentioned, OpenGL provides an interpolated shading model known as Gouraud shading. Gouraud shading linearly interpolates the color of an interior pixel from the color at the vertices. This interpolation is incorporated directly into the rasterization algorithm.

Another common shading model is Phong shading. Instead of interpolating over the intensity, Phong shading interpolates over the normal vectors specified at the vertices.

Shading — what happens between vertices

Clearly, the color and intensity of light at interior points in a polygon could be calculated similarly. However, this would be computationally expensive, so other methods are used.

As mentioned, OpenGL provides an interpolated shading model known as Gouraud shading. Gouraud shading linearly interpolates the color of an interior pixel from the color at the vertices. This interpolation is incorporated directly into the rasterization algorithm.

Another common shading model is Phong shading. Instead of interpolating over the intensity, Phong shading interpolates over the normal vectors specified at the vertices. This generally produces a better specular component, because a better approximation is used to the normal at each point.

Shading — what happens between vertices

Clearly, the color and intensity of light at interior points in a polygon could be calculated similarly. However, this would be computationally expensive, so other methods are used.

As mentioned, OpenGL provides an interpolated shading model known as Gouraud shading. Gouraud shading linearly interpolates the color of an interior pixel from the color at the vertices. This interpolation is incorporated directly into the rasterization algorithm.

Another common shading model is Phong shading. Instead of interpolating over the intensity, Phong shading interpolates over the normal vectors specified at the vertices. This generally produces a better specular component, because a better approximation is used to the normal at each point. (Gouraud shading tends to "average out" specular highlights.)

Shading — what happens between vertices

Clearly, the color and intensity of light at interior points in a polygon could be calculated similarly. However, this would be computationally expensive, so other methods are used.

As mentioned, OpenGL provides an interpolated shading model known as Gouraud shading. Gouraud shading linearly interpolates the color of an interior pixel from the color at the vertices. This interpolation is incorporated directly into the rasterization algorithm.

Another common shading model is Phong shading. Instead of interpolating over the intensity, Phong shading interpolates over the normal vectors specified at the vertices. This generally produces a better specular component, because a better approximation is used to the normal at each point. (Gouraud shading tends to "average out" specular highlights.)

This is computationally expensive, however, because the normal vector must be both interpolated and normalized for each pixel.

Shading — what happens between vertices

Clearly, the color and intensity of light at interior points in a polygon could be calculated similarly. However, this would be computationally expensive, so other methods are used.

As mentioned, OpenGL provides an interpolated shading model known as Gouraud shading. Gouraud shading linearly interpolates the color of an interior pixel from the color at the vertices. This interpolation is incorporated directly into the rasterization algorithm.

Another common shading model is Phong shading. Instead of interpolating over the intensity, Phong shading interpolates over the normal vectors specified at the vertices. This generally produces a better specular component, because a better approximation is used to the normal at each point. (Gouraud shading tends to "average out" specular highlights.)

This is computationally expensive, however, because the normal vector must be both interpolated and normalized for each pixel.

Specifying normal vectors

Specifying normal vectors

In the examples above the objects viewed were generated by calls to `glutSolidSphere` and `glutSolidTorus`.

Specifying normal vectors

In the examples above the objects viewed were generated by calls to `glutSolidSphere` and `glutSolidTorus`. These functions generate the appropriate surface normals for these objects.

Specifying normal vectors

In the examples above the objects viewed were generated by calls to `glutSolidSphere` and `glutSolidTorus`. These functions generate the appropriate surface normals for these objects. However, you often have to specify normal vectors yourself.

Specifying normal vectors

In the examples above the objects viewed were generated by calls to `glutSolidSphere` and `glutSolidTorus`. These functions generate the appropriate surface normals for these objects. However, you often have to specify normal vectors yourself.

For proper lighting, these normal vectors must have unit length.

Specifying normal vectors

In the examples above the objects viewed were generated by calls to `glutSolidSphere` and `glutSolidTorus`. These functions generate the appropriate surface normals for these objects. However, you often have to specify normal vectors yourself.

For proper lighting, these normal vectors must have unit length. Even if you specify unit length normals, their lengths can be changed by subsequent transformations (scaling and shearing).

Specifying normal vectors

In the examples above the objects viewed were generated by calls to `glutSolidSphere` and `glutSolidTorus`. These functions generate the appropriate surface normals for these objects. However, you often have to specify normal vectors yourself.

For proper lighting, these normal vectors must have unit length. Even if you specify unit length normals, their lengths can be changed by subsequent transformations (scaling and shearing). Therefore you may have to renormalize them.

Specifying normal vectors

In the examples above the objects viewed were generated by calls to `glutSolidSphere` and `glutSolidTorus`. These functions generate the appropriate surface normals for these objects. However, you often have to specify normal vectors yourself.

For proper lighting, these normal vectors must have unit length. Even if you specify unit length normals, their lengths can be changed by subsequent transformations (scaling and shearing). Therefore you may have to renormalize them. OpenGL can do this for you automatically (at some computational expense).

Specifying normal vectors

In the examples above the objects viewed were generated by calls to `glutSolidSphere` and `glutSolidTorus`. These functions generate the appropriate surface normals for these objects. However, you often have to specify normal vectors yourself.

For proper lighting, these normal vectors must have unit length. Even if you specify unit length normals, their lengths can be changed by subsequent transformations (scaling and shearing). Therefore you may have to renormalize them. OpenGL can do this for you automatically (at some computational expense). To set this up call `glEnable` with `GL_NORMALIZE` or `GL_RESCALE_NORMAL` as a parameter (see red book for details).

Specifying normal vectors

In the examples above the objects viewed were generated by calls to `glutSolidSphere` and `glutSolidTorus`. These functions generate the appropriate surface normals for these objects. However, you often have to specify normal vectors yourself.

For proper lighting, these normal vectors must have unit length. Even if you specify unit length normals, their lengths can be changed by subsequent transformations (scaling and shearing). Therefore you may have to renormalize them. OpenGL can do this for you automatically (at some computational expense). To set this up call `glEnable` with `GL_NORMALIZE` or `GL_RESCALE_NORMAL` as a parameter (see red book for details).

The function `glNormal*(x,y,z)` sets the normal vector for a vertex.

Specifying normal vectors

In the examples above the objects viewed were generated by calls to `glutSolidSphere` and `glutSolidTorus`. These functions generate the appropriate surface normals for these objects. However, you often have to specify normal vectors yourself.

For proper lighting, these normal vectors must have unit length. Even if you specify unit length normals, their lengths can be changed by subsequent transformations (scaling and shearing). Therefore you may have to renormalize them. OpenGL can do this for you automatically (at some computational expense). To set this up call `glEnable` with `GL_NORMALIZE` or `GL_RESCALE_NORMAL` as a parameter (see red book for details).

The function `glNormal*(x,y,z)` sets the normal vector for a vertex.

Example: `tent.c`

Example: `tent.c`

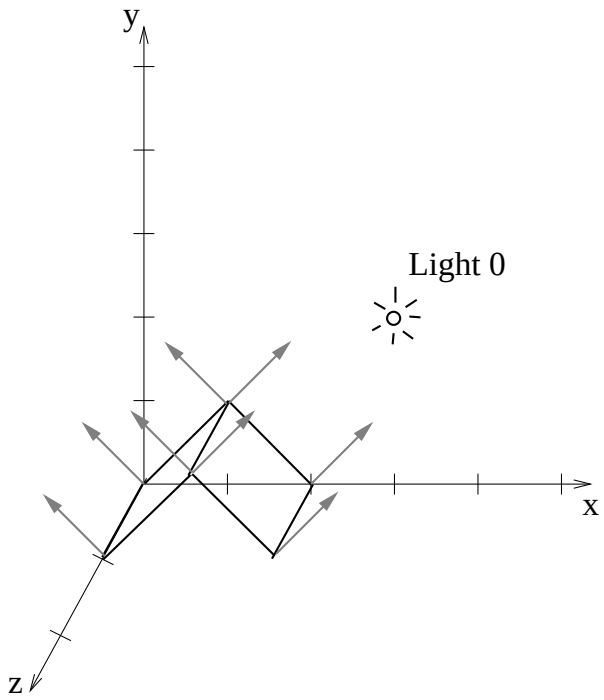
The program `tent.c` shows a 'tent' composed of two polygons.

Example: `tent.c`

The program `tent.c` shows a 'tent' composed of two polygons. Initially, the normal vectors are specified as follows:

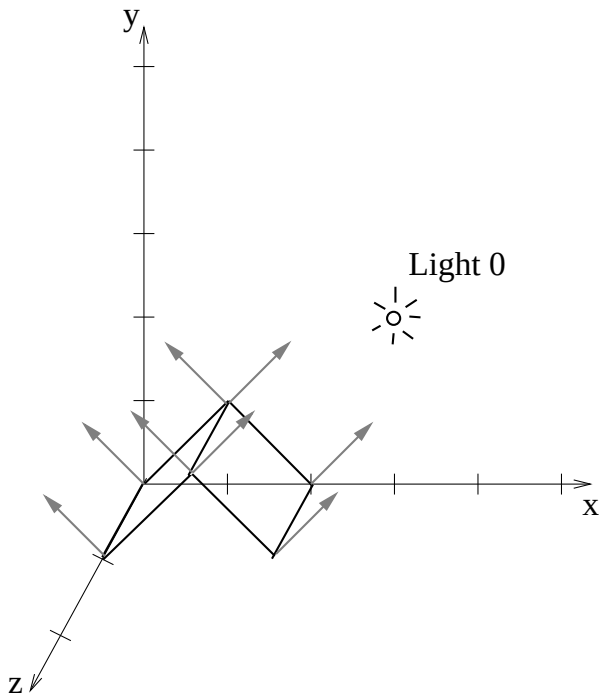
Example: `tent.c`

The program `tent.c` shows a 'tent' composed of two polygons. Initially, the normal vectors are specified as follows:



Example: `tent.c`

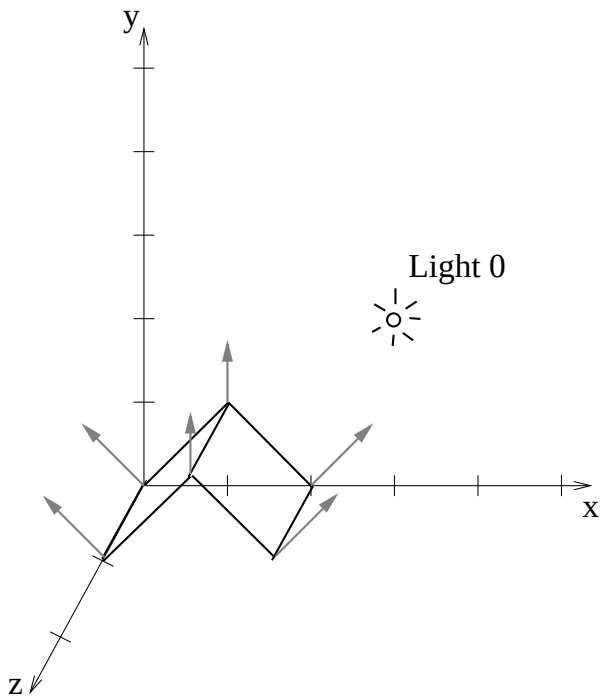
The program `tent.c` shows a 'tent' composed of two polygons. Initially, the normal vectors are specified as follows:



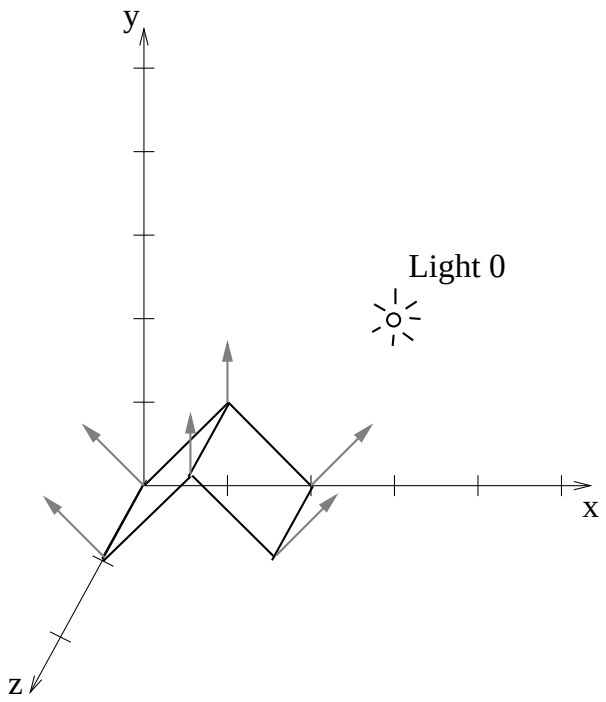
The diffuse light source illuminates the right polygon much more than the left, and the transition in illumination is abrupt.

A more gentle transition in illumination can be enabled by specifying normal vectors as follows:

A more gentle transition in illumination can be enabled by specifying normal vectors as follows:

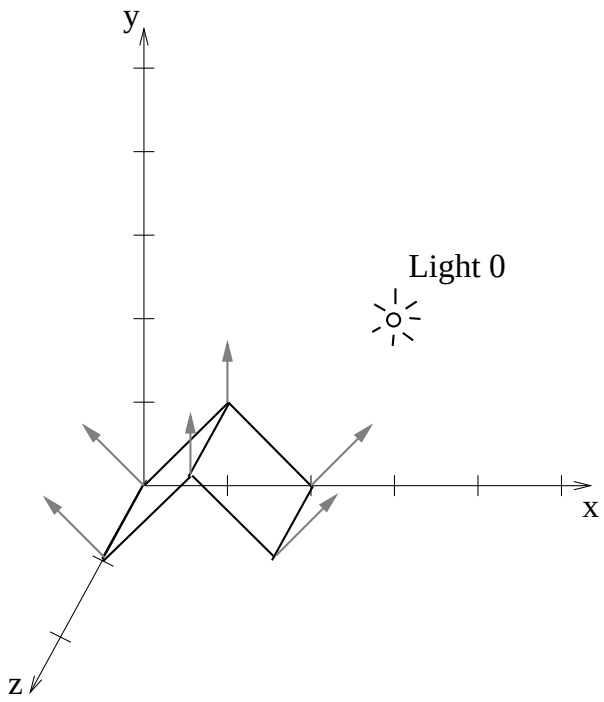


A more gentle transition in illumination can be enabled by specifying normal vectors as follows:



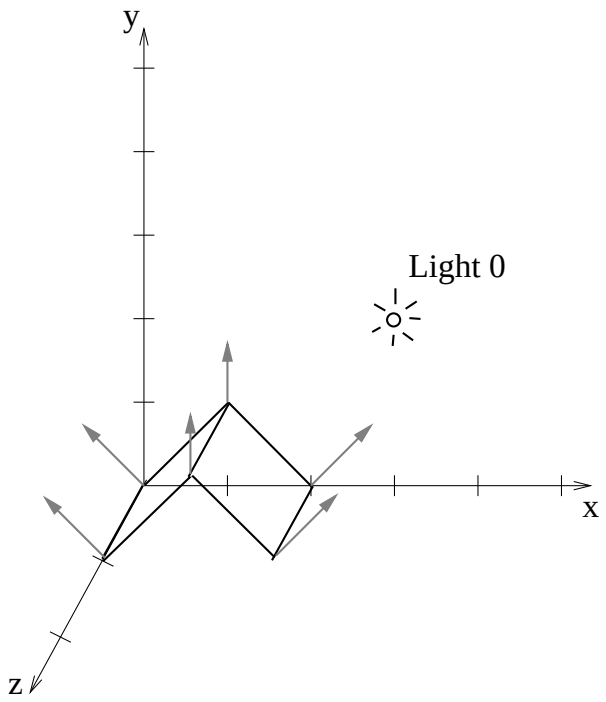
These two modes can be toggled by pressing ‘t’.

A more gentle transition in illumination can be enabled by specifying normal vectors as follows:



These two modes can be toggled by pressing ‘t’. Additionally, a white specular component can be added to the light source by pressing ‘s’.

A more gentle transition in illumination can be enabled by specifying normal vectors as follows:



These two modes can be toggled by pressing ‘t’. Additionally, a white specular component can be added to the light source by pressing ‘s’.

The following is the most relevant code from `tent.c`...

```
void display(void) {
```

```
void display(void) {
      .
      .
      .
 /* We require unit length normal vectors.  For a vector with 2 non-zero
  * elements of equal magnitude, the magnitude of both elements should be equal
  * to 1 / sqrt(2) = 0.7071 for the vector to be of unit length. */
   double l = 0.7071;
```

```
void display(void) {
      .
      .
      .
 /* We require unit length normal vectors.  For a vector with 2 non-zero
  * elements of equal magnitude, the magnitude of both elements should be equal
  * to 1 / sqrt(2) = 0.7071 for the vector to be of unit length. */
   double l = 0.7071;

 /* Set up the light in a position perpendicular to the right polygon. */
   GLfloat light_position[] = { 3.0, 2.0, 0.0, 1.0 };
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

```
void display(void) {
      .
      .
      .
 /* We require unit length normal vectors.  For a vector with 2 non-zero
  * elements of equal magnitude, the magnitude of both elements should be equal
  * to 1 / sqrt(2) = 0.7071 for the vector to be of unit length. */
   double l = 0.7071;

 /* Set up the light in a position perpendicular to the right polygon. */
   GLfloat light_position[] = { 3.0, 2.0, 0.0, 1.0 };
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
      .
      .
      .
 /* Draw the tent. */
   glBegin(GL_QUADS);
```

```
void display(void) {
      .
      .
      .
 /* We require unit length normal vectors.  For a vector with 2 non-zero
  * elements of equal magnitude, the magnitude of both elements should be equal
  * to 1 / sqrt(2) = 0.7071 for the vector to be of unit length. */
   double l = 0.7071;

 /* Set up the light in a position perpendicular to the right polygon. */
   GLfloat light_position[] = { 3.0, 2.0, 0.0, 1.0 };
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
      .
      .
      .
 /* Draw the tent. */
   glBegin(GL_QUADS);
    /* Left side. */
```

```
void display(void) {
      .
      .
      .
 /* We require unit length normal vectors.  For a vector with 2 non-zero
  * elements of equal magnitude, the magnitude of both elements should be equal
  * to 1 / sqrt(2) = 0.7071 for the vector to be of unit length. */
   double l = 0.7071;

 /* Set up the light in a position perpendicular to the right polygon. */
   GLfloat light_position[] = { 3.0, 2.0, 0.0, 1.0 };
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
      .
      .
      .
 /* Draw the tent. */
   glBegin(GL_QUADS);
    /* Left side. */
      glNormal3f(-l, l, 0);
```

```
void display(void) {
      .
      .
      .
 /* We require unit length normal vectors.  For a vector with 2 non-zero
  * elements of equal magnitude, the magnitude of both elements should be equal
  * to 1 / sqrt(2) = 0.7071 for the vector to be of unit length. */
   double l = 0.7071;

 /* Set up the light in a position perpendicular to the right polygon. */
   GLfloat light_position[] = { 3.0, 2.0, 0.0, 1.0 };
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
      .
      .
      .
 /* Draw the tent. */
   glBegin(GL_QUADS);
    /* Left side. */
      glNormal3f(-l, l, 0);
      glVertex3f(0, 0, 0);
```

```
void display(void) {
      .
      .
      .
 /* We require unit length normal vectors.  For a vector with 2 non-zero
  * elements of equal magnitude, the magnitude of both elements should be equal
  * to 1 / sqrt(2) = 0.7071 for the vector to be of unit length. */
   double l = 0.7071;

 /* Set up the light in a position perpendicular to the right polygon. */
   GLfloat light_position[] = { 3.0, 2.0, 0.0, 1.0 };
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
      .
      .
      .
 /* Draw the tent. */
   glBegin(GL_QUADS);
    /* Left side. */
      glNormal3f(-l, l, 0);
      glVertex3f(0, 0, 0);
      glVertex3f(0, 0, 1);
```

```
void display(void) {
      .
      .
      .
 /* We require unit length normal vectors.  For a vector with 2 non-zero
  * elements of equal magnitude, the magnitude of both elements should be equal
  * to 1 / sqrt(2) = 0.7071 for the vector to be of unit length. */
   double l = 0.7071;

 /* Set up the light in a position perpendicular to the right polygon. */
   GLfloat light_position[] = { 3.0, 2.0, 0.0, 1.0 };
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
      .
      .
      .
 /* Draw the tent. */
   glBegin(GL_QUADS);
    /* Left side. */
      glNormal3f(-l, l, 0);
      glVertex3f(0, 0, 0);
      glVertex3f(0, 0, 1);
      if ( topUp ) glNormal3f(0, 1, 0);
```

```
void display(void) {
      .
      .
      .
 /* We require unit length normal vectors.  For a vector with 2 non-zero
  * elements of equal magnitude, the magnitude of both elements should be equal
  * to 1 / sqrt(2) = 0.7071 for the vector to be of unit length. */
   double l = 0.7071;

 /* Set up the light in a position perpendicular to the right polygon. */
   GLfloat light_position[] = { 3.0, 2.0, 0.0, 1.0 };
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
      .
      .
      .
 /* Draw the tent. */
   glBegin(GL_QUADS);
    /* Left side. */
      glNormal3f(-l, l, 0);
      glVertex3f(0, 0, 0);
      glVertex3f(0, 0, 1);
      if ( topUp ) glNormal3f(0, 1, 0);
      glVertex3f(1, 1, 1);
```

```
void display(void) {
      .
      .
      .
 /* We require unit length normal vectors.  For a vector with 2 non-zero
  * elements of equal magnitude, the magnitude of both elements should be equal
  * to 1 / sqrt(2) = 0.7071 for the vector to be of unit length. */
   double l = 0.7071;

 /* Set up the light in a position perpendicular to the right polygon. */
   GLfloat light_position[] = { 3.0, 2.0, 0.0, 1.0 };
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
      .
      .
      .
 /* Draw the tent. */
   glBegin(GL_QUADS);
    /* Left side. */
      glNormal3f(-l, l, 0);
      glVertex3f(0, 0, 0);
      glVertex3f(0, 0, 1);
      if ( topUp ) glNormal3f(0, 1, 0);
      glVertex3f(1, 1, 1);
      glVertex3f(1, 1, 0);
```

```
void display(void) {
      .
      .
      .
 /* We require unit length normal vectors.  For a vector with 2 non-zero
  * elements of equal magnitude, the magnitude of both elements should be equal
  * to 1 / sqrt(2) = 0.7071 for the vector to be of unit length. */
   double l = 0.7071;

 /* Set up the light in a position perpendicular to the right polygon. */
   GLfloat light_position[] = { 3.0, 2.0, 0.0, 1.0 };
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
      .
      .
      .
 /* Draw the tent. */
   glBegin(GL_QUADS);
    /* Left side. */
      glNormal3f(-l, l, 0);
      glVertex3f(0, 0, 0);
      glVertex3f(0, 0, 1);
      if ( topUp ) glNormal3f(0, 1, 0);
      glVertex3f(1, 1, 1);
      glVertex3f(1, 1, 0);

    /* Right side. */
```

```
void display(void) {
      .
      .
      .
 /* We require unit length normal vectors.  For a vector with 2 non-zero
  * elements of equal magnitude, the magnitude of both elements should be equal
  * to 1 / sqrt(2) = 0.7071 for the vector to be of unit length. */
   double l = 0.7071;

 /* Set up the light in a position perpendicular to the right polygon. */
   GLfloat light_position[] = { 3.0, 2.0, 0.0, 1.0 };
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
      .
      .
      .
 /* Draw the tent. */
   glBegin(GL_QUADS);
    /* Left side. */
      glNormal3f(-l, l, 0);
      glVertex3f(0, 0, 0);
      glVertex3f(0, 0, 1);
      if ( topUp ) glNormal3f(0, 1, 0);
      glVertex3f(1, 1, 1);
      glVertex3f(1, 1, 0);

    /* Right side. */
      glNormal3f(l, l, 0);
```

```
void display(void) {
      .
      .
      .
 /* We require unit length normal vectors.  For a vector with 2 non-zero
  * elements of equal magnitude, the magnitude of both elements should be equal
  * to 1 / sqrt(2) = 0.7071 for the vector to be of unit length. */
   double l = 0.7071;

 /* Set up the light in a position perpendicular to the right polygon. */
   GLfloat light_position[] = { 3.0, 2.0, 0.0, 1.0 };
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
      .
      .
      .
 /* Draw the tent. */
   glBegin(GL_QUADS);
    /* Left side. */
      glNormal3f(-l, l, 0);
      glVertex3f(0, 0, 0);
      glVertex3f(0, 0, 1);
      if ( topUp ) glNormal3f(0, 1, 0);
      glVertex3f(1, 1, 1);
      glVertex3f(1, 1, 0);

    /* Right side. */
      glNormal3f(l, l, 0);
      glVertex3f(2, 0, 1);
```

```
void display(void) {
      .
      .
      .
 /* We require unit length normal vectors.  For a vector with 2 non-zero
  * elements of equal magnitude, the magnitude of both elements should be equal
  * to 1 / sqrt(2) = 0.7071 for the vector to be of unit length. */
   double l = 0.7071;

 /* Set up the light in a position perpendicular to the right polygon. */
   GLfloat light_position[] = { 3.0, 2.0, 0.0, 1.0 };
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
      .
      .
      .
 /* Draw the tent. */
   glBegin(GL_QUADS);
    /* Left side. */
      glNormal3f(-l, l, 0);
      glVertex3f(0, 0, 0);
      glVertex3f(0, 0, 1);
      if ( topUp ) glNormal3f(0, 1, 0);
      glVertex3f(1, 1, 1);
      glVertex3f(1, 1, 0);

    /* Right side. */
      glNormal3f(l, l, 0);
      glVertex3f(2, 0, 1);
      glVertex3f(2, 0, 0);
```

```
void display(void) {
      .
      .
      .
 /* We require unit length normal vectors.  For a vector with 2 non-zero
  * elements of equal magnitude, the magnitude of both elements should be equal
  * to 1 / sqrt(2) = 0.7071 for the vector to be of unit length. */
   double l = 0.7071;

 /* Set up the light in a position perpendicular to the right polygon. */
   GLfloat light_position[] = { 3.0, 2.0, 0.0, 1.0 };
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
      .
      .
      .
 /* Draw the tent. */
   glBegin(GL_QUADS);
    /* Left side. */
      glNormal3f(-l, l, 0);
      glVertex3f(0, 0, 0);
      glVertex3f(0, 0, 1);
      if ( topUp ) glNormal3f(0, 1, 0);
      glVertex3f(1, 1, 1);
      glVertex3f(1, 1, 0);

    /* Right side. */
      glNormal3f(l, l, 0);
      glVertex3f(2, 0, 1);
      glVertex3f(2, 0, 0);
      if ( topUp ) glNormal3f(0, 1, 0);
```

```
void display(void) {
      .
      .
      .
 /* We require unit length normal vectors.  For a vector with 2 non-zero
  * elements of equal magnitude, the magnitude of both elements should be equal
  * to 1 / sqrt(2) = 0.7071 for the vector to be of unit length. */
   double l = 0.7071;

 /* Set up the light in a position perpendicular to the right polygon. */
   GLfloat light_position[] = { 3.0, 2.0, 0.0, 1.0 };
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
      .
      .
      .
 /* Draw the tent. */
   glBegin(GL_QUADS);
    /* Left side. */
      glNormal3f(-l, l, 0);
      glVertex3f(0, 0, 0);
      glVertex3f(0, 0, 1);
      if ( topUp ) glNormal3f(0, 1, 0);
      glVertex3f(1, 1, 1);
      glVertex3f(1, 1, 0);

    /* Right side. */
      glNormal3f(l, l, 0);
      glVertex3f(2, 0, 1);
      glVertex3f(2, 0, 0);
      if ( topUp ) glNormal3f(0, 1, 0);
      glVertex3f(1, 1, 0);
```

```
void display(void) {
      .
      .
      .
 /* We require unit length normal vectors.  For a vector with 2 non-zero
  * elements of equal magnitude, the magnitude of both elements should be equal
  * to 1 / sqrt(2) = 0.7071 for the vector to be of unit length. */
   double l = 0.7071;

 /* Set up the light in a position perpendicular to the right polygon. */
   GLfloat light_position[] = { 3.0, 2.0, 0.0, 1.0 };
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
      .
      .
      .
 /* Draw the tent. */
   glBegin(GL_QUADS);
    /* Left side. */
      glNormal3f(-l, l, 0);
      glVertex3f(0, 0, 0);
      glVertex3f(0, 0, 1);
      if ( topUp ) glNormal3f(0, 1, 0);
      glVertex3f(1, 1, 1);
      glVertex3f(1, 1, 0);

    /* Right side. */
      glNormal3f(l, l, 0);
      glVertex3f(2, 0, 1);
      glVertex3f(2, 0, 0);
      if ( topUp ) glNormal3f(0, 1, 0);
      glVertex3f(1, 1, 0);
      glVertex3f(1, 1, 1);
```

```
void display(void) {
      .
      .
      .
 /* We require unit length normal vectors.  For a vector with 2 non-zero
  * elements of equal magnitude, the magnitude of both elements should be equal
  * to 1 / sqrt(2) = 0.7071 for the vector to be of unit length. */
   double l = 0.7071;

 /* Set up the light in a position perpendicular to the right polygon. */
   GLfloat light_position[] = { 3.0, 2.0, 0.0, 1.0 };
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
      .
      .
      .
 /* Draw the tent. */
   glBegin(GL_QUADS);
    /* Left side. */
      glNormal3f(-l, l, 0);
      glVertex3f(0, 0, 0);
      glVertex3f(0, 0, 1);
      if ( topUp ) glNormal3f(0, 1, 0);
      glVertex3f(1, 1, 1);
      glVertex3f(1, 1, 0);

    /* Right side. */
      glNormal3f(l, l, 0);
      glVertex3f(2, 0, 1);
      glVertex3f(2, 0, 0);
      if ( topUp ) glNormal3f(0, 1, 0);
      glVertex3f(1, 1, 0);
      glVertex3f(1, 1, 1);
   glEnd();
```

```
void display(void) {
      .
      .
      .
 /* We require unit length normal vectors.  For a vector with 2 non-zero
  * elements of equal magnitude, the magnitude of both elements should be equal
  * to 1 / sqrt(2) = 0.7071 for the vector to be of unit length. */
   double l = 0.7071;

 /* Set up the light in a position perpendicular to the right polygon. */
   GLfloat light_position[] = { 3.0, 2.0, 0.0, 1.0 };
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
      .
      .
      .
 /* Draw the tent. */
   glBegin(GL_QUADS);
    /* Left side. */
      glNormal3f(-l, l, 0);
      glVertex3f(0, 0, 0);
      glVertex3f(0, 0, 1);
      if ( topUp ) glNormal3f(0, 1, 0);
      glVertex3f(1, 1, 1);
      glVertex3f(1, 1, 0);

    /* Right side. */
      glNormal3f(l, l, 0);
      glVertex3f(2, 0, 1);
      glVertex3f(2, 0, 0);
      if ( topUp ) glNormal3f(0, 1, 0);
      glVertex3f(1, 1, 0);
      glVertex3f(1, 1, 1);
   glEnd();
      .
      .
      .
}
```

```
void init (void)
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL_SMOOTH);

 /* Set up material properties. */
   GLfloat mat_ambient[] = { 1.0, 0.0, 0.0, 1.0 };
   GLfloat mat_diffuse[] = { 0.0, 1.0, 0.0, 1.0 };
   glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
   glMaterialf(GL_FRONT, GL_SHININESS, 50.0 ); /* No effect if specular == 0 */
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
}
```

```
void key( unsigned char k, int x, int y )
{
   switch (k) {
   case 27:  /* Escape */
     exit(0);
     break;
   case 's':
     specular = !specular;
     if ( specular ) {
        GLfloat mat_specular[] = { 0.0, 0.0, 1.0, 1.0 };
        glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
     } else {
        GLfloat mat_specular[] = { 0.0, 0.0, 0.0, 1.0 };
        glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
     }
     break;
   case 't':
     topUp = !topUp;
     break;
   default:
     return;
   }
   glutPostRedisplay();
}
```