Colour in OpenGL

Colour of a vertex can be set explicitly as an RGB triple with either:

\[ \text{glColor3f}(r, g, b) \]

or

\[ \text{glColor4f}(r, g, b, a) \]

where \( r = \text{red}, \ g = \text{green}, \ b = \text{blue}, \ a = \text{alpha}, \) and \( 0 \leq r, g, b, a \leq 1. \)

Alpha is used to define an object's level of transparency. If \( a = 1, \) object is fully opaque. If \( a = 0, \) object is fully transparent.

In order for transparency to implement properly in OpenGL, two requirements must be satisfied:

1. Blending must be enabled and a blending function must be selected.
2. Objects must be drawn back-to-front from the perspective of the view direction.
Blending is enabled in OpenGL with:

```gl
glEnable (GL_BLEND)
```

Without this, all alpha values are ignored by OpenGL.

Once blending is enabled, a "blend function" must be chosen to describe how to blend colours already in the frame buffer with the colours of a new polygon overlapping part of the frame buffer. Suppose the values in the frame buffer (called the "destination") are \((R_d, G_d, B_d, A_d)\) and the values for the polygon (called the "source") are \((R_s, G_s, B_s, A_s)\). Mathematically, the final values written back to the frame buffer are:

\[
(R_{s+}, G_{s+}, B_{s+}, A_{s+}) = \frac{(R_s S_s + R_d D_s, G_s S_s + G_d D_s, B_s S_s + B_d D_s, A_s S_s + A_d D_s)}{S_s + D_s}
\]

where \((S_s, S_a, S_b, S_a)\) is the source blending factor, and \((D_s, D_a, D_b, D_a)\) is the destination blending factor. Source and destination blending factors are set with:

```gl
glBlendFunc (factor, dFactor)
```
OpenGL offers a variety of available blending factors. Some common ones include:

- **GL_ZERO**
  - \((R, G, B, A)\)
- **GL_ONE**
  - \((1, 1, 1, 1)\)
- **GL_DST_COLOR**
  - \((R_g, G_g, B_g, A)\)
- **GL_SRC_COLOR**
  - \((R_s, G_s, B_s, A_s)\)
- **GL_ONE_MINUS_DST_COLOR**
  - \((1, 1, 1) - (R_g, G_g, B_g, A)\)
- **GL_ONE_MINUS_SRC_COLOR**
  - \((1, 1, 1) - (R_s, G_s, B_s, A_s)\)
- **GL_DST_ALPHA**
  - \((A_g, A_p, A_p, A)\)
- **GL_SRC_ALPHA**
  - \((A_s, A_s, A_s, A_s)\)
- **GL_ONE_MINUS_DST_ALPHA**
  - \((1, 1, 1) - (A_g, A_p, A_p, A)\)
- **GL_ONE_MINUS_SRC_ALPHA**
  - \((1, 1, 1) - (A_s, A_s, A_s, A_s)\)

Some examples:

1. `glBlendFunc (GL_ONE, GL_ZERO)`

   Identical to the situation where blending is disabled (i.e., \((R_s + R_g, G_s + G_g, B_s + B_g, A_s + A_g) = (R, G, B, A)\)).

2. Draw a picture composed of half of one image and half of another, equally blended:

   - `glBlend (GL_ONE, GL_ZERO)`
   - draw first image w/alpha = 1
3. Drawing a brush stroke in a paint program that gradually adds more color to what exists on the canvas:

```g直接影响(GL-SRC-ALPHA, GL-ONE-MINUS-SRC-ALPHA);
draw stroke with low alpha (e.g., 0.1)`
```

4. Compose 3 translucent surfaces, background solid black, farthest surface is red and 80% transparent, next surface is green and 40% transparent, closest surface is blue and 90% transparent.

```g影响Color(0, 0, 0, 1)
g影响Color(GL-Color-BUFFER-BIT)
```

```g直接影响(GL-SRC-ALPHA, GL-ONE-MINUS-SRC-ALPHA)
g影响4 (*1, 0.8, 0.8, 0.2)
g影响4 (*0.8, 1, 0.8, 0.6)
g影响4 (*0.8, 1, 0.8, 0.1)
```

Order of drawing matters. Consider above example in original and reversed order.
Recall \( S = (A_1, A_2, A_3, A_4) \) and 
\( P = (1 - A_1, 1 - A_2, 1 - A_3, 1 - A_4) \):

Original:

1. \( (\emptyset, \emptyset, \emptyset, 1) \)
2. \( g|_{C_0}^4 \phi (1, \emptyset, \emptyset, \emptyset, 2) \) 
   \( (1, \emptyset, \emptyset, \emptyset, 0.2 \cdot 0.2 + 0.8 \cdot 1) \) 
   \( = (\emptyset, \emptyset, \emptyset, \emptyset, 0.84) \)
3. \( g|_{C_0}^4 \phi (\emptyset, 1, \emptyset, \emptyset, 4) \) 
   \( (0.2 \cdot 0.4, 1, \emptyset, \emptyset, 0.6 \cdot 0.6 + 0.8 \cdot 0.4) \) 
   \( = (\emptyset, \emptyset, 0.6, 0.6, 0.696) \)
4. \( g|_{C_0}^4 \phi (\emptyset, \emptyset, 1, \emptyset, 1) \) 
   \( (\emptyset, \emptyset, \emptyset, 0.9, \emptyset, 0.7, \emptyset, 1 + 0.696 \cdot 0.9) \) 
   \( = (\emptyset, \emptyset, 0.72, 0.54, 0.6, 0.634) \)

Reverse:

1. \( (\emptyset, \emptyset, \emptyset, 1) \)
2. \( g|_{C_0}^4 \phi (\emptyset, \emptyset, 1, \emptyset, 1) \) 
   \( (\emptyset, \emptyset, 1, \emptyset, 1, \emptyset, 0.8 + 0.2 \cdot 0) \) 
   \( = (\emptyset, \emptyset, 0.8, 0.1, 0.9) \)
3. \( g|_{C_0}^4 \phi (\emptyset, 1, \emptyset, \emptyset, 6) \) 
   \( (\emptyset, 1, \emptyset, 0.8, 0.6 \cdot 0.4, 0.6 \cdot 0.6 + 0.91 \cdot 2) \) 
   \( = (\emptyset, \emptyset, 0.84, 0.724) \)
4. \( g|_{C_0}^4 \phi (1, \emptyset, \emptyset, \emptyset, 2) \) 
   \( (1, \emptyset, \emptyset, \emptyset, 0.8, 0.8 \cdot 0.4, 0.8 \cdot 0.2 + 0.724 \cdot 0.8) \) 
   \( = (\emptyset, 0.48, 0.832, 0.6192) \)
Illumination Models

Methods that attempt to simulate physical rules of optics and thermal radiation due to computational and other complexities, initial models are simplified estimates with no foundation to underlying principles of light. More advanced techniques ever much more from real laws of physics.

Illumination in OpenGL comes from three sources: ambient reflection, diffuse reflection, and specular reflection.

Ambient Light

Meant to estimate background light in a scene, due to the scattering of emitted light over multiple surfaces. In OpenGL, this is modelled in three places: the amount of ambient light an emitter (light source) contributes to the scene, the amount of ambient light present in a scene independent of any light source, and the percentage of ambient light each polygon reflects (ambient reflection coefficient).

Ambient reflection intensity is independent of both direction to light source and
Diffuse Light

A diffuse surface, when lit by a point light source, reflects the incoming light equally in all directions. Brightness of reflected light depends on the angle $\Theta$ between the direction to the light source $I$ and the surface normal $\mathbf{N}$. Specifically, the intensity is proportional to the cosine of this angle, $\cos \Theta$.

\[ \mathbf{I} \perp \mathbf{N} \]

If $\mathbf{N}, \mathbf{T}$ normalized, then $\cos \Theta = \mathbf{N} \cdot \mathbf{T}$

Also, if light source is at infinity, $I$ is constant (otherwise $I$ must be computed for each surface position where we want to perform lighting calculations).

In OpenGL, diffuse light is modelled as an amount of diffuse light each emitter contributes to the scene, plus the percentage of diffuse light each polygon reflects (diffuse reflection coefficient).
So diffuse reflection intensity for a light source \( p \) and polygon \( d \) is:

\[
I = Ip k_d (\vec{N} \cdot \vec{I})
\]

where \( Ip \) is light source's diffuse intensity, \( k_d \) is polygon's diffuse reflection coefficient, and \( \vec{N} \) and \( \vec{I} \) are polygon normal and direction to light source, both normalized.

**Specular Light**

Specular light is light reflected by shiny objects to form highlights. The location of specular light depends on both the direction to the light source and the direction to the viewer.

For perfect reflectors (e.g., a mirror) specular light is reflected only in the perfect reflection direction \( \vec{R} \), which is \( \vec{I} \) mirrored about \( \vec{N} \), the surface normal.
For nonperfect reflectors, Specular light is reflected in a cone about \( R \). \( \alpha \) represents the angular difference between \( R \) and the direction to the viewer \( V \). If \( \alpha \) is small enough, the viewer will see some specular light reflected from the given surface position.

Again, OpenGL allows you to specify a specular light intensity for each emitter, and a specular reflection coefficient for each polygon.

A Phong model is used to compute specular intensity. Proposed by Phong Bui-Tuong, it assumes maximum specular reflection at \( \alpha = 0 \), and a rapid falloff as \( \alpha \) increases. The falloff is modelled with an exponential cosine function 
\[ \cos^N \alpha \] 
where \( N \) is specular reflection exponent (or shininess factor). As \( N \) increases from 1, sharpness of falloff increases. This means smaller and smaller \( \alpha \) are needed to produce any specular
reflection, and therefore the size of
the highlight on the object shrinks or
"tightens up" around locations with a
perfect reflection direction that points
back to the viewer (i.e., locations where
$\mathbf{E} = \nabla$)
Lighting in OpenGL

Lighting determines the colour of vertices in the scene based on light properties and materials properties assigned to different surfaces in the scene.

There are four steps involved in using lighting in OpenGL:

1. Define normals for vertex of each object
2. Create, position one or more light sources
3. Create, initialize lighting model
4. Define materials properties for objects in scene

In order for diffuse and specular lighting calculations to work properly, normals must be specified for each vertex in the scene with:

```
glNormal3f (x, y, z)
```

where \((x, y, z)\) is the direction of the normal.

As previously mentioned, normals to a polygon can be computed as cross-product of any two adjacent, non-colinear edges of the polygon.
If Gouraud shading is required (i.e., smooth shading with no seams at polygon edges), each vertex normal must be computed as the average of the normals of the faces incident to the vertex.

Normals are specified prior to the vertex they correspond to:

\[
g1Normal3f (ux, uy, uz)
\]
\[
g1Vertex3f (vx, vy, vz)
\]


light Sources

A light source is defined by its position, and by the intensity of the ambient, diffuse, and specular light that it emits into the scene.

All light properties are set up with:

\[
g1Light.fv (light, pa-am, value)
\]

where light defines which light's properties are being set (ranks from GL_LIGHT0 thru GL_LIGHT7), pa-am specifies which parameter of the light is being set, and values holds the values of the parameter.
Common parameters include:

**GL-AMBIENT** \[ R, G, B, A \text{ intensity of ambient contribution (default = (0,0,0,1))} \]

**GL-DIFFUSE** \[ R, G, B, A \text{ intensity of diffuse contribution (default = (1,1,1,1))} \]

**GL-SPECULAR** \[ R, G, B, A \text{ intensity of specular contribution (default = (1,1,1,1))} \]

**GL-POSITION** \[ (x,y,z,w) \text{ position of light (default = (0,0,0,1))} \]

---

Note for **GL-POSITION**: if \( w \neq 0 \) then light source assumed to be at infinity. This simplifies lighting calculations since direction from any vertex in scene to an infinite light source is constant. If \( w \neq 0 \), light source is "local" to scene, and direction to light source must be computed at each vertex.

Also, light source position is treated as a geometric primitive, so it is subjected to scene matrix transformations as any other primitive. This means you can manipulate the light's position via the MODELVIEW stack just as you would for any other vertex.
Once a light source's properties are set, you must "turn it on" with:

```gl
glEnable (GL_LIGHT0)
```

**Lighting Model**

The lighting model allows you to set the following properties of the scene:

- a global ambient light intensity with `glLightModeli (GL_LIGHT_MODEL_AMBIENT, amb)`, where `amb` is a `(r, g, b)` float array defining the ambient intensity.

- whether viewpoint is local or infinite (default is infinite to provide a constant direction from any vertex to viewer). Can be changed with `glLightModeli (GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE)`.

- whether only front or both front and back faces of polygons should have lighting performed (default is front faces only). Can be changed with `glLightModeli (GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE)`.
Once lighting model is set, lighting must be "turned on" with:

```
glEnable (GL_LIGHTING)
```

Materials Properties

In lighting mode, colors for vertices are not specified directly. Rather, materials properties are defined for a surface or vertex, and these are combined with the vertex normal, direction to light and viewer, and light properties to determine the color of the vertex.

Materials properties are set with:

```
glMaterialfv (face, param, values)
```

where face defines the face to set (GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK), param specifies which parameter of the surface is being set, and values holds the values of the parameter.

Common parameters includes
GL-AMBIENT  R, G, B, A of ambient reflection  
  (default = (.2, .2, .2, 1))

GL-DIFFUSE  R, G, B, A of diffuse reflection  
  (default = (.8, .8, .8, 1))

GL-SPECULAR  R, G, B, A of specular reflection  
  (default = (1, 1, 1, 1))

GL-SHININESS  specular exponent (default = 0)

As with glColor, glMaterial is used to set materials properties for all vertices following call, up to next glMaterial call.
Shading Models

OpenGL offers two shading models, GL_FLAT and GL_SMOOTH, set with glShadeModel().
This, together with lighting and materials values, and the types of surface normals specified, dictate how colours computed at each polygon vertex are used to shade the polygon's surface.

Constant (Flat) Shading

Here, a single colour (intensity) for the polygon is used to shade the entire polygon, a single, constant value.

If the light source is assumed to be at infinity, if there is no specular reflection, and if the polygon is not approximating a smooth surface, this is the correct shading solution (since NT constant across entire polygon so ambient and diffuse reflections constant also).

OpenGL normally uses the colour of the last vertex specified for a polygon to colour the polygon during flat shading.
Interpolated (smooth) Shading

The next step is to compute a separate colour at each polygon vertex, then interpolate between them to produce a smooth variation of colour across the polygon's surface. This is how OpenGL performs smooth shading.

Note that if you have a smooth surface approximated by flat polygons, you must ensure a common surface normal at each vertex in the object. Otherwise, common edges where polygons meet will use different normals to shade either side of the edge, producing a visible seam in the resulting render.

Normals in Shading

As noted above, if you use a normal target to a polygon's surface for each polygon rendered, you will see visible seams along edges between polygons. This is because the edge is being shaded "twice", on one side with one set of normals and on the other side with a different set of normals, e.g.
So on one side of \( V, V_2 \) we use \( N_A \) to do lighting calculations to pick a colour for \( V_1 \) and \( V_2 \), then interpolate to shade the edge. On the other side we use \( N_B \), giving different lighting results and therefore different colours to \( V_1 \) and \( V_2 \), producing different shading.

This is fine if your model is meant to look faceted, however, if you have a polygon model meant to approximate a smooth surface, the effect is undesirable.

To avoid this, we switch to a Gouraud shading model. Rather than specifying a vertex's normal based on the polygon currently being shaded, we precompute a single normal for each vertex. These normals are then used during rendering.

For vertices adjacent to only one polygon, the vertex normal is equal to the polygon normal.
For vertices shared by multiple polygons, polygon normals can be averaged, or a normal can be computed analytically if the underlying surface's functional form is known, e.g.

\[ \mathbf{N_v} = \frac{\sum_i \mathbf{N}_i}{\sum_i |\mathbf{N}_i|} \]

Or \[ \mathbf{N_v} = \frac{\sum_i \mathbf{N}_i}{n} \] if all \( \mathbf{N}_i \) normalized.

New edges will appear smooth, since the same normals used each time a polygon edge \( \mathbf{N_v} \) is shaded.

**Vertex-Based Phong Shading**

Specular highlights can be computed by OpenGL at each vertex, just like ambient and diffuse reflection are computed. A Phong model is used to estimate the specular contribution.
Unfortunately, when the vertex colours are interpolated to shade the surface of the polygon, the resulting specular highlights can contain large visual errors.

Intuitively, this makes sense, since linear interpolation is a very poor approximation of the cosine function used by Phong to model the falloff of the specular contribution.

Two problems are common: smeared highlights and missing highlights.

**Interpolated:**
- no spec
- no spec
- small $\alpha \Rightarrow$ large spec
- keep $\alpha$ rapid falloff

**Proper:**

Missing highlights occur when the highlight is interior to a polygon and strikes no vertex.

- no spec
- no spec
- no spec