Rendering

- **Rendering**: computing how each pixel of the picture of a scene should look.
- **Shading model (illumination model)**: method of modeling how lights interact with objects in a scene.
- **Rendering hierarchy**: rendering methods using different shading models that approximate physical lighting process at different levels, leading to different levels of realism.
  - **Wireframe**: draw edges only, hidden lines may and may not be removed.
  - **Flat shading**: each face is shaded uniformly, i.e. drawn with the same color.
  - **Smooth shading**: different points are shaded differently using an interpolation technique.
  - **Special effects**: shadowing, texture mapping.
Lights

- Lights (from either point light source or ambient light source) that reach the surface of an object can be:
  - absorbed by the object (converted into heat)
  - reflected and scattered from the surface (surface is visible through reflected lights)
  - transmitted into and through the surface (translucent and transparent objects)

- The surface material determines how much (fraction) light (of different wavelengths) is reflected and transmitted.
Reflections

✓ Important vectors:
  s: light direction; m: surface normal;
  v: viewing direction (sight vector).

✓ Light reflection: combination (sum) of diffuse and specular reflections. The reflected lights that reach the “eye” determine the intensity/color of the pixel.
  - Diffuse reflection: scattering of reflected lights in all directions equally, generating dull surfaces and more uniform lighting.
  - Specular reflection: highly directional mirror-like reflection, generating highlights and shiny look.

Diffuse reflection

✓ The diffuse component of the light reflection:

\[ I_d = I_s \rho_d \cos(\theta) = I_s \rho_d \frac{\bar{s} \cdot \bar{m}}{s \cdot m} \]

where:

- \( I_s \): light intensity (or one component of the light color)
- \( \rho_d \): diffuse reflection coefficient. constant for each face or object, but can be much more complex in reality.

✓ A more general form:

\[ I_d = I_s \rho_d f(d) \max\left(\frac{\bar{s} \cdot \bar{m}}{s \cdot m}, 0\right) \]

i.e.

\[ I_d = 0 \quad \text{if } \bar{s} \cdot \bar{m} < 0 \]

f(d): distance attenuation. e.g. \( f(d) = \frac{1}{\sqrt{ad^2 + bd + c}} \)
Specular reflection

- Simple specular mode: Phong model (Supported by OpenGL), Phong model provides simple highlights and plastic-like appearance.
- More complex specular effects such as shiny metallic appearance can only be modeled by more sophisticated global illumination model, e.g. ray tracing.
- Phong model: The reflected light is the strongest in the perfect reflection direction (mirror reflection). Its strength diminishes as the reflection angle deviates from the perfect reflection angle.
Phong reflection model

- Specular reflection intensity:
  \[ I_{sp} = I_s \rho_s \cos^f(\phi) = I_s \rho_d \left( \frac{\bar{r} \cdot \bar{v}}{||\bar{r}||} \right)^f \]
  where \( r = -s + 2 \frac{\bar{s} \cdot \bar{m}}{||m||} \bar{m} \)
  if \( \bar{r} \cdot \bar{v} < 0 \), \( I_{sp} = 0 \)

\( \rho_s \): specular reflection coefficient, assumed to be constant for each face or object, and determined experimentally.

\( f \): shininess coefficient, larger \( f \) represents more shiny material (Fig 8.13). Normally,
\[ 1 \leq f \leq 200, \text{ and } f_{mirror} = \infty \]

- Specular reflection is directional, i.e. it depends on the viewing vector \( \bar{v} \).
Halfway vector

- Phong model is expensive.
- Halfway vector: a cheaper alternative (by Jim Blinn) that avoids computing the reflection vector.

\[
\vec{h} = \vec{s} + \vec{v}
\]

\[
I_{sp} = I_s \rho_s \cos(\beta) = I_s \rho_s \left( \frac{\vec{h} \cdot \vec{m}}{|\vec{h}|} \right)
\]

- When the light source and the viewer are both far away, the halfway vector is nearly a constant.
- Shading results with different reflection coefficients and shininess coefficients:
Ambient light and global illumination

- **Local illumination** (Phong model): the shading of a point only depends on lights that directly reach the point, and the local geometry and material properties. Local illumination can be directly applied in a graphics pipeline (as in OpenGL).
- **Global illumination**: the shading of a point is affected by the light reflections of all other points (objects) in the scene. It involves solving a global rendering equation, and therefore cannot be directly applied in a graphics pipeline.
- In local illumination, points that are not directly reachable by a light source is not shaded (i.e. black). Ambient light is designed to provide a uniform light for all points in the scene, regardless of their physical locations.
- Ambient reflection = \( I_a \rho_a \)
  
  \( \rho_a \): ambient reflection coefficient; \( I_a \): ambient intensity
Combining light contributions

- The total amount of light that reaches the eye from a point:
  \[ I = I_a \rho_a + I_d \rho_d \cdot L + I_s \rho_s \cdot P^f \]
  where \( L = \max(\frac{\mathbf{s} \cdot \mathbf{m}}{\mathbf{h} \cdot \mathbf{m}}, 0) \), \( P = \max(\frac{\mathbf{h} \cdot \mathbf{m}}{\mathbf{h} \cdot \mathbf{m}}, 0) \)

- In practice, \( I_d \) and \( I_s \) are normally the same.

- Multiple light sources:
  \[ I = I_a \rho_a + \sum_{j=1}^{n} (I_{c_j} \rho_{c_j} L + I_{s_j} \rho_{s_j} P^f) \]

- Color components:
  \[ I_r = I_{w_r} \rho_{w_r} + I_{d_r} \rho_{d_r} \cdot L + I_{s_r} \rho_{s_r} \cdot P^f \]
  \[ I_g = I_{w_g} \rho_{w_g} + I_{d_g} \rho_{d_g} \cdot L + I_{s_g} \rho_{s_g} \cdot P^f \]
  \[ I_b = I_{w_b} \rho_{w_b} + I_{d_b} \rho_{d_b} \cdot L + I_{s_b} \rho_{s_b} \cdot P^f \]

- Material parameters: Plate 26
Shading and OpenGL pipeline

Shading is applied when objects are projected from 3D to 2D, carrying the shaded colors onto the 2D vertices.

The normal of each vertex is supplied by the user and will be transformed along with the vertex for shading computation.

```c
glBegin (GL_POLYGON)
  for (i=0; i < 3; i++) {
    glNormal3f (norm[i].x, norm[i].y, norm[i].z);
    glVertex3f (pt[i].x, pt[i].y, pt[i].z);
  }
glEnd ();
```

Light sources in OpenGL

Creating a light source: up to 8 light sources (GL_LIGHT0, ..., GL_LIGHT7) can be created.

```c
glLightf (GL_LIGHT0, para_name, parameter);
glEnable (GL_LIGHTING);
glEnable (GL_LIGHT0);
```

- para_name = GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR:
  parameter = light color (RGBA), e.g. \((I_d, I_g, I_a, 1)\)
- When para_name = GL_POSITION, parameter is the location of a point light: \((x, y, z, w)\). If \(w = 0\), it is a parallel light.
- Light sources have default values.

Distance attenuation: \(f(d) = 1/(k_a + k_i D + k_q D^3)\)

the coefficients may be defined in glLightf ()
OpenGL Lighting Model

- Lighting model: general rule in applying lighting.
- Ambient light
  
  ```
  GLfloat amb[] = {0.2, 0.3, 0.1, 1.0};
  glLightModelfv(GL_LIGHT_MODEL_AMBIENT, amb);
  ```
- Local viewpoint: specify whether to compute shading using the true viewing direction vector (v) or a constant (0,0,1).
  
  ```
  glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
  ```

Polygon sides

- Front face is defined by counter clockwise (CCW) vertex order while looking from the camera. The other side is the back face.
- `glLightModeli(GL_LIGHT_MODEL_TWO_SIDES, GL_TRUE)`; (normals will be reversed when draw back faces.)

![Diagram of front and back faces in OpenGL lighting model](image)
Moving light sources

- Light sources are treated the same as vertices, i.e. are subject to ModelView transformations.
- Lights moving with camera:
  - `glMatrixMode(GL_MODELVIEW);` `glLoadIdentity();`
  - `glLightfv(GL_LIGHT0, GL_POSITION, pos);`
  - `gluLookAt(...);` `draw object;`
- Lights moving independent of camera:
  - `glMatrixMode(GL_MODELVIEW);` `glLoadIdentity();`
  - `glPushMatrix();`
  - `glRotated(...);` `glTranslated(...);`
  - `glLightfv(GL_LIGHT0, GL_POSITION, pos);`
  - `glPopMatrix();`
  - `gluLookAt(...);` `draw object;`

Material properties in OpenGL

- Reflection coefficients can be separately defined for each color component in OpenGL.
  - `glMaterialf(face, para_name, parameter);`
    - `face:` `GL_FRONT, GL_BACK, GL_FRONT_AND_BACK`
    - `para_name:` `GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_SHININESS, GL_EMISSION`
- OpenGL illumination model (green and blue are similar):
  \[
  I_r = I_{ar} \rho_a + \sum_{j=1}^{4} f(d)(I_{ar} \rho_{ar} + I_{mr} \rho_{mr} L + I_{cr} \rho_{cr} P^f)
  \]
Polygon filling

Polygon filling: filling pixels within a polygon. Pixels are normally filled scanline by scanline, and left to right.

```c
for (y = ymin; y <= ymax; y++) {
    xmin and xmax for this scanline;
    for (x = xmin; x <= xmax; x++) find and fill the color for this pixel (x,y);
}
```

Pixel colors are determined by the polygon vertices and their normals.

Flat shading uses the same normal for all vertices of a polygon.

Smooth shading uses normals computed from the actual surface that polygon is approximating.

Surface normals

- Planar surface:
  - Ax+By+Cz+D=0 : \( \vec{n} = (A, B, C) \)
  - Polygon:
    \( \vec{n} = (p_2 - p_1) \times (p_3 - p_2) \)

- General implicit surface \( f(x,y,z)=0 \) :
  \( \vec{n} = \text{grad}(f) = \left( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right) \)

- Parametric surface \( S(u,v) = (x(u,v), y(u,v), z(u,v)) \)
  \( \vec{n} = \frac{\partial S}{\partial u} \times \frac{\partial S}{\partial v} \)

  example: a sphere surface
  \( S(u, v) = (r \cos(u) \cos(v), r \cos(u) \sin(v), r \sin(u)) \)
Flat shading and smooth shading

- Shading model:
  - `glShadeModel (GL_FLAT)` or `glShadeModel (GL_SMOOTH)`
- Flat shading: Uses the same color for all pixels in a polygon. The color is pre-computed (i.e., outside the polygon filling loop) using the normal of the first vertex of the polygon.
  - Edges are visible;
  - Specular highlights are not well rendered.
- Smooth shading: Uses different colors for the pixels in a polygon, so that edges can be "smoothed out". There are two typical smoothing shading methods: Gourand shading and Phong shading.
Gourand shading

- Compute the color for each vertex of the polygon, using an illumination model, and then interpolate the colors at the vertices to generate colors for all filling pixels.

**Bilinear interpolation of colors:**

\[ I_u = (1-u) \cdot I_A + u \cdot I_B, \quad u = \frac{x - x_u}{x_v - x_u} \]
\[ I_v = (1-v) \cdot I_C + v \cdot I_D, \quad v = \frac{y - y_v}{y_u - y_v} \]
\[ I_p = (1-t) \cdot I_Q + t \cdot I_R, \quad t = \frac{z - z_u}{z_v - z_u} \]
Phong shading

- Compute the normal vector at each pixel within the polygon by bilinear interpolation of normal vectors at polygon vertices. And then compute the color the pixel using an illumination model.

Bilinear interpolation of normals:
\[
\begin{align*}
\vec{m}_Q &= (1-u) \cdot \vec{m}_A + u \cdot \vec{m}_B, \quad u = \frac{(v - v_A)}{(v_B - v_A)} \\
\vec{m}_R &= (1-v) \cdot \vec{m}_C + v \cdot \vec{m}_D, \quad v = \frac{(u - u_C)}{(u_D - u_C)} \\
\vec{m}_P &= (1-t) \cdot \vec{m}_Q + t \cdot \vec{m}_R, \quad t = \frac{(s - s_Q)}{(s_R - s_Q)}
\end{align*}
\]

Gouraud shading vs Phong shading

- Gouraud shading is faster than Phong shading since shading computation is only performed at the vertices.
- Phong shading presents higher image quality since shading computation is done at each pixel.
- Highlights are not as well rendered in Gouraud shading as in Phong shading since color interpolation removes some highlights within the polygons.
- OpenGL implements Gouraud shading.
- Problems with concave polygons (both methods):
Hidden line and hidden surface removal

- Hidden line removal: wireframe display
- Hidden surface removal: depth order in surface rendering.
- Algorithms: depth sort (painter’s), raycasting, space subdivision (Warnock, BSP, Octree), Z-buffer.
- OpenGL solution: Z-buffer (depth buffer)
Depth buffer (Z-buffer) approach

- Depth buffer (Z-buffer): a framebuffer-like memory storing the depth value of the closest point to each pixel. During display, the (i,j) location of Z-buffer stores a value d(i,j) representing the closest depth value so far at pixel location (i,j). d(i,j) will be replaced if a closer point is drawn later to the same pixel.

```
for (each point P to be drawn)
    compute the projected pixel location (i,j);
    if (d(i,j) > depth (P))
        d(i,j) = depth (P); draw P;
```

Depth computation

- The z value of a point after viewport transformation is scaled into [0,1], i.e. 0 on the near plane, and 1 on the far plane. The d(i,j) values are initialized to 1.
- Planar polygon with equation: ax+by+cz+q=0

```
if (c==0) ignore the polygon:
else
    for (each scanline QR)
        compute depth of Q: d = -(ax+by+q)/c;
        for (each subsequent pixel on the scanline)
            d = d - a/c;
```

- Non-planar polygon: bilinear interpolation:

```
d_{Q} = (1-u) \cdot d_{A} + u \cdot d_{B}, \quad u = \frac{(y_{x}-y)}{(y_{x}-y_{A})}
d_{R} = (1-v) \cdot d_{C} + v \cdot d_{D}, \quad v = \frac{(y_{x}-y)}{(y_{x}-y_{D})}
d_{P} = (1-t) \cdot d_{Q} + t \cdot d_{R}, \quad t = \frac{(x_{y}-x)}{(x_{y}-x_{Q})}
```
Depth buffer in OpenGL

- **OpenGL implementation**
  
  ```
  glutInitDisplayMode (GLUT_DEPTH);
  glEnable (GL_DEPTH_TEST);
  glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  ```

- **Pros & Cons:**
  - Simple to implement
  - Can handle any type of object
  - Less sensitive to scene complexity
  - Requires large (high-speed) memory
  - Precision in depth is limited by Z-buffer bit depth

Depth sorting: painter’s algorithm

- **Priority list**
  - An object space algorithm
  - It builds a depth priority list of all objects (polygons) based on their distances to the viewpoint; and then draws the objects in a reverse priority order.
  - Often implemented in parallel projection environment (i.e. after objects are transformed into an orthogonal view volume.

```
Projection plane

(1) (2) (3) (4) (5)

Drawing order: (5), (4), (3), (2), (1)
```
Painter's algorithm

1. Compute bounding box for each polygon
2. Build a preliminary depth priority list using the maximum depth, \(d_{\text{max}}\), of each polygon. (The first in the list has the \(d_{\text{max}}\) farthest from the viewer)
3. For (each polygon \(P\) in the list)
   - let \(Q\) be the next polygon in the list;
   - if \((d_{\text{min}}(P) \geq d_{\text{max}}(Q))\) draw \(P\);
   - else
     - let \(S = \{Q: d_{\text{min}}(P) > d_{\text{max}}(Q)\}\);
     - apply a priority test sequence on all polygons in \(S\);
     - if (All polygons in \(S\) pass the test) draw \(P\);
     - else interchange \(P\) and \(Q\), go back to (3);
   
   If (the same \(P\) and \(Q\) need interchange again later)
   - cyclical overlapping or intersection exists, split \(P\) along
     the plane of \(Q\), remove original \(P\) from the list, insert the
     two new polygons into the list, start the algorithm again.

Priority test sequence

- Are bounding boxes of \(P\) and \(Q\) disjoint in \(X\) or \(Y\)?
  \[P_{x_{\text{max}}} < Q_{x_{\text{min}}} \text{ or } P_{y_{\text{min}}} < Q_{y_{\text{max}}} \]

- Is \(P\) wholly on the far side of the plane of \(Q\)?
  Let \(p(x,y,z)=Ax+By+Cz+D=0\) be the plane equation of \(Q\).
  If \(p()\) have the same sign as \(\text{sign}(-C)\) for all vertices of \(P\),
  then \(P\) is wolly on the far side of \(Q\)

- Is \(Q\) wholly on the near side of the plane of \(P\)?
  test is similar.

- \(Q\) passes the test if any of the tests answers yes.
Cyclical overlap

- Cyclical priority order:

- How to split a polygon by a plane?

Raycasting Algorithm

- Image space algorithm. It determines the visibility for each pixel by casting a ray, connecting the viewpoint and the center of the pixel, into the scene. The first intersection the ray has is the visible point for the pixel.
- A simple algorithm:

```cpp
for (each pixel) {
    construct a ray;
    for (each object in the scene)
        compute and store the ray-object intersection points, on a point list;
    if (the point list is not empty)
        determine the closest point and its shading color, and set the pixel color;
    else set background color;
}
```
Optimization for raycasting

- **Bounding box**: Minimum and maximum coordinates of the object in X, Y, and Z directions.
- Testing ray intersection with bounding box may not be efficient: need transformation of ray or object.
- **Bounding sphere**: The minimum sphere enclosing the object. It can be directly computed from bounding box (not a tight bound), or by numerical minimization.
- Testing ray intersection with a sphere is easy: computing distance from the center of the sphere to the ray.
  \[ d = \sqrt{(P_0 - C)^2 - ((C - P_0) \cdot \hat{V} / |\hat{V}|)^2} \]

Optimization for raycasting (2)

- **Clustering**: Building a hierarchy of bounding spheres for groups of objects. Each ray will traverse the hierarchy tree in a depth-first order, but skip subtrees whose bounding boxes or spheres do not intersect the ray.
- **Priority sorting**: Pre-sort objects in priority order, and compute the ray-object intersection according to priority order - avoid unnecessary intersection computation.
- **Spatial subdivision**: Hierarchical space subdivision. Only the objects that intersect the subspaces that the ray passes through need to be computed.
- **Ray-polygon intersection**
  1. Ray-plane intersection
  2. Inside test
Raycasting algorithm using a clustering hierarchy

for (each ray) {
    for (each cluster sphere on the hierarchy tree) perform a bounding sphere test;
    if (the ray intersects the sphere)
        if (the sphere is an object sphere, i.e. leaf node) place the object on the active object list;
        else check child clusters;
        else skip this subtree;
    if (active object list is empty) display background color;
    else
        for (each active object)
            compute ray-object intersection, if any, and place the intersection point(s) on the intersection list;
        if (intersection list is empty) display background color;
        else determine the nearest intersection point, and display the shaded color at this point;
}

BSP (Binary Space Partitioning)

- Basic ideas:
  - Suitable for highly complex but static scenes, with high rendering rate and rapidly changing view point (e.g. flight simulation).
  - Preprocess of the scene in the WCS such that the depth order for any view point can be easily obtained.
  - Based on the fact that object space can be divided into two half spaces by each polygon.

![Diagram of BSP tree with nodes and halfspaces]
BSP tree

A binary tree can be constructed by recursively subdivide the object space into halfspaces. Each halfspace is recursively subdivided by one of the polygons within the halfspace, until there is only one (or zero) polygon left in the halfspace. This BSP tree is not unique for a given scene.

- When the dividing plane intersects a polygon, the polygon will be splitted into two, one in each halfspace.
- To check which halfspace a polygon belongs to, a simple substitution with each polygon vertex to the dividing plane equation, $Ax+By+Cz+D=0$, is sufficient. If the results are all positive or 0, the polygon is in “+halfspace”, if all negative, it’s in “-halfspace”, otherwise splitting is necessary.
- A good BSP tree will have minimal splits.

BSP construction algorithm

```c
BSPtree BSPConstruct (polylist) {
  if (polylist is empty) return null;
  else
    SelectPolygon (polylist, root);
    backlist=null; frontlist=null;
    for (each remaining polygon, poly, on polylist)
      if (poly is in front of root) AddtoBSTree (polygon, frontlist);
      else if (poly is behind root) AddtoBSTree (polygon, backlist);
      else
        SplitPolygon (polygon, root, frontpart, backpart);
        AddtoBSTree (frontpart, frontlist);
        AddtoBSTree (backpart, backlist);
    lefttree = BSFConstruct (frontlist);
    righttree = BSFConstruct (backlist);
  return combine (lefttree, root, righttree);
}
```
**BSP tree display**

- The display algorithm

  ```cpp
  DisplayBSP(BSPtree) {
      if (BSPtree is not empty)
          if (viewpoint is in front of root)
              DisplayBSP(BSPtree.right);
              display root.polygon;
              DisplayBSP(BSPtree.left);
          else
              DisplayBSP(BSPtree.left);
              display root.polygon;
              DisplayBSP(BSPtree.right);
  }
  ```

- SelectPolygon() : testing with different polygons as roots. Experiments show that a few (5-6) tests are normally sufficient.

**Octree**

- Object space subdivision: recursively subdivide a cubic domain of the object space into 8 equal sized cubes (octants or voxels). Octants are numbered from 0 to 7 in a left-right, down-up and front-back order.

- Octree nodes: root represents the original object space, each level of the octree represents the corresponding level of subdivision, and leaf nodes represents octants that do not require further subdivision (based on certain criteria).

- Nodes are labels: F (full), E (empty), P (partially full, gray).

- A simple subdivision criterion

  ```cpp
  if ((node.type == 'P') && (node.size > minimal_size) &&
      (node is above the minimal complexity level))
      then subdivide;
  ```

- Minimal complexity level: the level of object complexity that is considered easy to solve. (e.g. only one polygon remains)
Octree display

- Octree is constructed in WCS. The display order is determined based on the viewing direction.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>Display order</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;=0</td>
<td>&gt;=0</td>
<td>&gt;=0</td>
<td>7,6,5,4,3,2,1,0</td>
</tr>
<tr>
<td>&lt;=0</td>
<td>&gt;=0</td>
<td>&gt;=0</td>
<td>6,7,4,5,2,3,0,1</td>
</tr>
<tr>
<td>&gt;=0</td>
<td>&lt; 0</td>
<td>&gt;=0</td>
<td>5,4,7,6,1,0,3,2</td>
</tr>
<tr>
<td>&gt;=0</td>
<td>&gt;=0</td>
<td>&lt; 0</td>
<td>3,2,1,0,7,6,5,4</td>
</tr>
<tr>
<td>&lt;=0</td>
<td>&lt; 0</td>
<td>&gt;=0</td>
<td>4,5,6,7,0,1,2,3</td>
</tr>
<tr>
<td>&lt;=0</td>
<td>&gt;=0</td>
<td>&lt; 0</td>
<td>2,3,0,1,6,7,4,5</td>
</tr>
<tr>
<td>&gt;=0</td>
<td>&lt; 0</td>
<td>&lt; 0</td>
<td>1,0,3,2,5,4,7,6</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>&lt; 0</td>
<td>&lt; 0</td>
<td>0,1,2,3,4,5,6,7</td>
</tr>
</tbody>
</table>

Octree display (2)

- Display algorithm
  
  for (each given viewing direction)
  
  build an index for octree display order: index[0...7];

  displayOctree (root):

  PROCEDURE displayOctree (octNode)
  {
    if (octNode is a leaf node)
      if (octNode is not empty)
        display octNode;
    else
      for (i = 0 ... 7)
        displayOctree (octNode.children[index[i]]):
  }
Blending

- When blending is enabled, a color drawn into a pixel (source color) will be blended with the color that is already in that pixel (destination color): `glEnable(GL_BLEND)`.
- Alpha value of a color is often used as the opacity of the color, i.e. low alpha value means high transparency.
- Blending is carried out through a blending function:
  \[ \text{dstColor} = \text{srcFactor} \times \text{srcColor} + \text{dstFactor} \times \text{dstColor} \]
- `glBlendFunc(srcFactor, dstFactor)`;
- `srcFactor, dstFactor`: `GL_ZERO, GL_ONE, GL_SRC_ALPHA, GL_DST_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_DST_COLOR, GL_ONE_MINUS_DST_ALPHA, ...`
- Linear blending: `dstFactor=(1-srcAlpha), srcFactor=srcAlpha`
  \[ C = (1-\alpha) \times C + \alpha \times C_{in} \]

Transparency

- Simple transparency:
  `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`: 
- Using transparency with a depth buffer (for opaque objects):
  First draw all opaque objects with depth buffer enabled; then use `glDepthMask` to set the depth buffer to read-only. It allows to transparent objects to be drawn without wrongly set the depth buffer, but still detect transparent objects that are behind opaque objects (thus should not draw).