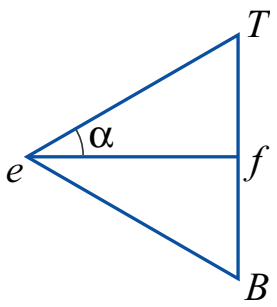
7 Visibility

We have seen so far how to determine how 3D points project to the camera's image plane. Additionally, we can render a triangle by projecting each vertex to 2D, and then filling in the pixels of the 2D triangle. However, what happens if two triangles project to the same pixels, or, more generally, if they overlap? Determining which polygon to render at each pixel is **visibility.** An object is visible if there exists a direct line-of-sight to that point, unobstructed by any other objects. Moreover, some objects may be invisible because they are behind the camera, outside of the field-of-view, or too far away.

7.1 The View Volume and Clipping

The **view volume** is made up of the space between the near plane, f, and far plane, F. It is bounded by B, T, L, and R on the bottom, top, left, and right, respectively.

The angular field of view is determined by f, B, T, L, and R:



From this figure, we can find that $\tan(\alpha) = \frac{1}{2}\frac{T-B}{|f|}$.

Clipping is the process of removing points and parts of objects that are outside the view volume.

We would like to modify our homogeneous perspective transformation matrix to simplify clipping. We have

$$\hat{M}_p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -\frac{1}{f}\left(\frac{f+F}{f-F}\right) & \frac{2F}{f-F} \\ 0 & 0 & -1/f & 0 \end{bmatrix}.$$

Since this is a homogeneous transformation, it may be multiplied by a constant without changing

its effect. Multiplying $\hat{M}_p$ by f gives us

$$\begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & -\left(\frac{f+F}{f-F}\right) & \frac{2fF}{f-F} \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

If we alter the transform in the x and y coordinates to be

$$\hat{x}^* = \begin{bmatrix} \frac{2f}{R-L} & 0 & \frac{R+L}{R-L} & 0 \\ 0 & \frac{2f}{T-B} & \frac{T+B}{T-B} & 0 \\ 0 & 0 & -\left(\frac{f+F}{f-F}\right) & \frac{2fF}{f-F} \\ 0 & 0 & 1 & 0 \end{bmatrix} \hat{p}^c,$$

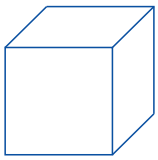
then, after projection, the view volume becomes a cube with sides at -1 and $+1$. This is called the **canonical view volume** and has the advantage of being easy to clip against.

Note:
The OpenGL command glFrustum(l, r, b, t, n, f) takes the distance to the near and far planes rather than the position on the z-axis of the planes. Hence, the n used by glFrustum is our $-f$ and the f used by glFrustum is $-F$. Substituting these values into our matrix gives exactly the perspective transformation matrix used by OpenGL.

7.2 Backface Removal

Consider a closed polyhedral object. Because it is closed, far side of the object will always be invisible, blocked by the near side. This observation can be used to accelerate rendering, by removing **back-faces.**

Example:
For this simple view of a cube, we have three backfacing polygons, the left side, back, and bottom:



Only the near faces are visible.

We can determine if a face is back-facing as follows. Suppose we compute a normals $\vec{n}$ for a mesh face, with the normal chosen so that it points outside the object For a surface point $\bar{p}$ on a planar

patch and eye point $\bar{e}$, if $(\bar{p} - \bar{e}) \cdot \vec{n} > 0$, then the angle between the view direction and normal is less than $90°$, so the surface normal points away from $\bar{e}$. The result will be the same no matter which face point $\bar{p}$ we use.

Hence, if $(\bar{p} - \bar{e}) \cdot \vec{n} > 0$, the patch is backfacing and should be removed. Otherwise, it *might* be visible. This should be calculated in world coordinates so the patch can be removed as early as possible.

Note:
To compute $\vec{n}$, we need three vertices on the patch, in counterclockwise order, as seen from the outside of the object, $\bar{p}_1$, $\bar{p}_1$, and $\bar{p}_3$. Then the unit normal is

$$\frac{(\bar{p}_2 - \bar{p}_1) \times (\bar{p}_3 - \bar{p}_1)}{\|(\bar{p}_2 - \bar{p}_1) \times (\bar{p}_3 - \bar{p}_1)\|}.$$

Backface removal is a "quick reject" used to accelerate rendering. It must still be used together with another visibility method. The other methods are more expensive, and removing backfaces just reduces the number of faces that must be considered by a more expensive method.

7.3 The Depth Buffer

Normally when rendering, we compute an image buffer `I(i,j)` that stores the color of the object that projects to pixel (i, j). The depth d of a pixel is the distance from the eye point to the object. The **depth buffer** is an array `zbuf(i, j)` which stores, for each pixel (i, j), the depth of the nearest point drawn so far. It is initialized by setting all depth buffer values to infinite depth: `zbuf(i,j)`$= \infty$.

To draw color c at pixel (i, j) with depth d:

```
if d < zbuf(i, j) then
  putpixel(i, j, c)
  zbuf(i, j) = d
end
```

When drawing a pixel, if the new pixel's depth is greater than the current value of the depth buffer at that pixel, then there must be some object blocking the new pixel, and it is not drawn.

Advantages

- Simple and accurate
- Independent of order of polygons drawn

Disadvantages

- Memory required for depth buffer
- Wasted computation on drawing distant points that are drawn over with closer points that occupy the same pixel

To represent the depth at each pixel, we can use pseudodepth, which is available after the homogeneous perspective transformation.[1] Then the depth buffer should be initialized to 1, since the pseudodepth values are between -1 and 1. Pseudodepth gives a number of numerical advantages over true depth.

To scan convert a triangular polygon with vertices $\bar{x}_1$, $\bar{x}_2$, and $\bar{x}_3$, pseudodepth values d_1, d_2, and d_3, and fill color c, we calculate the x values and pseudodepths for each edge at each scanline. Then for each scanline, interpolate pseudodepth between edges and compare the value at each pixel to the value stored in the depth buffer.

7.4 Painter's Algorithm

The **painter's algorithm** is an alternative to depth buffering to attempt to ensure that the closest points to a viewer occlude points behind them. The idea is to draw the most distant patches of a surface first, allowing nearer surfaces to be drawn over them.

In the heedless painter's algorithm, we first sort faces according to depth of the vertex furthest from the viewer. Then faces are rendered from furthest to nearest.

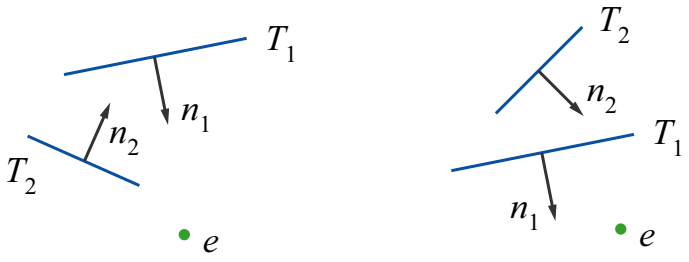
There are problems with this approach, however. In some cases, a face that occludes part of another face can still have its furthest vertex further from the viewer than any vertex of the face it occludes. In this situation, the faces will be rendered out of order. Also, polygons cannot intersect at all as they can when depth buffering is used instead. One solution is to split triangles, but doing this correctly is very complex and slow. Painter's algorithm is rarely used directly in practice; however, a data-structure called BSP trees can be used to make painter's algorithm much more appealing.

7.5 BSP Trees

The idea of **binary space partitioning trees** (BSP trees) is to extend the painter's algorithm to make back-to-front ordering of polygons fast for any eye location and to divide polygons to avoid overlaps.

Imagine two patches, T_1 and T_2, with outward-facing normals $\vec{n}_1$ and $\vec{n}_2$.

[1] The OpenGL documentation is confusing in a few places — "depth" is used to mean pseudodepth, in commands like `glReadPixels` and `gluUnProject`.



If the eye point, $\bar{e}$, and T_2 are on the same side of T_1, then we draw T_1 before T_2. Otherwise, T_2 should be drawn before T_1.

We know if two points are on the same side of a plane containing T_1 by using the implicit equation for T_1,

$$f_1(\bar{x}) \quad = \quad (\bar{x} - \bar{p}_1) \cdot \vec{n}. \tag{1}$$

If $\bar{x}$ is on the plane, $f_1(\bar{x}) = 0$. Otherwise, if $f_1(\bar{x}) > 0$, $\bar{x}$ is on the "outside" of T_1, and if $f_1(\bar{x}) < 0$, $\bar{x}$ is "inside."

Before any rendering can occur, the scene geometry must be processed to build a BSP tree to represent the relative positions of all the facets with respect to their inside/outside half-planes. The same BSP tree can be used for any eye position, so the tree only has to be constructed once if everything other than the eye is static. For a single scene, there are many different BSP trees that can be used to represent it — it's best to try to construct balanced trees.

The tree traversal algorithm to draw a tree with root F is as follows:

```
if eye is in the outside half-space of F
  draw faces on the inside subtree of F
  draw F
  draw faces on the outside subtree of F
else
  draw faces on the outside subtree of F
  draw F (if backfaces are drawn)
  draw faces on the inside subtree of F
end
```

7.6 Visibility in OpenGL

OpenGL directly supports depth buffering, but it is often used in addition to other visibility techniques in interactive applications. For example, many games use a BSP tree to prune the amount of static map geometry that is processed that would otherwise not be visible anyway. Also, when

dealing with blended, translucent materials, these objects often must be drawn from back to front without writing to the depth buffer to get the correct appearance. For simple scenes, however, the depth buffer alone is sufficient.

To use depth buffering in OpenGL with GLUT, the OpenGL context must be initialized with memory allocated for a depth buffer, with a command such as

```
glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
```

Next, depth writing and testing must be enabled in OpenGL:

```
glEnable(GL_DEPTH_TEST);
```

OpenGL will automatically write pseudodepth values to the depth buffer when a primitive is rendered as long as the depth test is enabled. The `glDepthMask` function can be used to disable depth writes, so depth testing will occur without writing to the depth buffer when rendering a primitive.

When clearing the display to render a new frame, the depth buffer should also be cleared:

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

