Graphics 2011/2012, 4th quarter

Lecture 8

Graphics pipeline (clipping and culling)
Perspective projection by matrix multiplication:

\[
\begin{pmatrix}
  x_{\text{pixel}} \\
  y_{\text{pixel}} \\
  z_{\text{canonical}} \\
  1
\end{pmatrix} = M_{vp}M_{per}M_{cam}
\begin{pmatrix}
  x \\
  y \\
  z \\
  1
\end{pmatrix}
\]

This is an important part of the graphics pipeline.
Graphics pipeline - part 1 (recap)

- **View frustum**
  Specify camera position and view frustum

- **Camera transformation**
  Move from world space to camera space

- **Orthographic view volume**
  Transform view frustum to axis-parallel box

- **Canonical view volume**
  Transform orthographic view volume to 2x2x2 cube around the origin

- **Windowing transformation**
  Orthographic projection and transformation to screen space
Extending homogeneous coordinates

Linear transformations: Points represented by vectors \((x, y, z)\)

\[ x \mapsto x' = ax + by + cz \]
Extending homogeneous coordinates

**Linear transformations:** Points represented by vectors \((x, y, z)\)

\[
x \mapsto x' = ax + by + cz
\]

**Affine transformations:** \((x, y, z, 1)\) represents point \((x, y, z)\)

\[
x \mapsto x' = ax + by + cz + d
\]
Geometric interpretation (in 2D)
Extending homogeneous coordinates

**Linear transformations:** Points represented by vectors \((x, y, z)\)

\[ x \mapsto x' = ax + by + cz \]

**Affine transformations:** \((x, y, z, 1)\) represents point \((x, y, z)\)

\[ x \mapsto x' = ax + by + cz + d \]

**Projective transformations:** \((\tilde{x}, \tilde{y}, \tilde{z}, w)\) represents \((\tilde{x}/w, \tilde{y}/w, \tilde{z}/w, 1)\)
which in turn represents \((\tilde{x}/w, \tilde{y}/w, \tilde{z}/w)\)
which in turn represents point \((x, y, z)\).

\[ x \mapsto x' = \frac{ax + by + cz + d}{ex + fy + gz + h} \]
Geometric interpretation (in 1D)
Extending homogeneous coordinates

Matrix multiplication:

\[
\begin{pmatrix}
\tilde{x} \\
\tilde{y} \\
\tilde{z} \\
\tilde{w}
\end{pmatrix} =
\begin{pmatrix}
a_1 & b_1 & c_1 & d_1 \\
a_2 & b_2 & c_2 & d_2 \\
a_3 & b_3 & c_3 & d_3 \\
e & f & g & h
\end{pmatrix}
\begin{pmatrix}
x \\
y \\
z \\
1
\end{pmatrix} =
\begin{pmatrix}
a_1x + b_1y + c_1z + d_1 \\
a_2x + b_2y + c_2z + d_2 \\
a_3x + b_3y + c_3z + d_3 \\
ex + fy + gz + h
\end{pmatrix}
\]

Homogenization:

\[
\begin{pmatrix}
\frac{a_1x + b_1y + c_1z + d_1}{ex + fy + gz + h} \\
\frac{a_2x + b_2y + c_2z + d_2}{ex + fy + gz + h} \\
\frac{a_3x + b_3y + c_3z + d_3}{ex + fy + gz + h} \\
\frac{ex + fy + gz + h}{1}
\end{pmatrix}
\]
Extending homogeneous coordinates

Matrix multiplication:

\[
\begin{pmatrix}
\tilde{x} \\
\tilde{y} \\
\tilde{z} \\
\tilde{w}
\end{pmatrix} = c
\begin{pmatrix}
a_1 & b_1 & c_1 & d_1 \\
a_2 & b_2 & c_2 & d_2 \\
a_3 & b_3 & c_3 & d_3 \\
e & f & g & h
\end{pmatrix}
\begin{pmatrix}
x \\
y \\
z \\
1
\end{pmatrix} =
\begin{pmatrix}
c(a_1 x + b_1 y + c_1 z + d_1) \\
c(a_2 x + b_2 y + c_2 z + d_2) \\
c(a_3 x + b_3 y + c_3 z + d_3) \\
c(ex + fy + gz + h)
\end{pmatrix}
\]

Homogenization:

\[
\begin{pmatrix}
c(a_1 x + b_1 y + c_1 z + d_1) \\
c(a_2 x + b_2 y + c_2 z + d_2) \\
c(a_3 x + b_3 y + c_3 z + d_3) \\
c(ex + fy + gz + h)
\end{pmatrix}
\xrightarrow{\text{homogenize}}
\begin{pmatrix}
c(a_1 x + b_1 y + c_1 z + d_1) \\
c(ex + fy + gz + h) \\
c(a_2 x + b_2 y + c_2 z + d_2) \\
c(a_3 x + b_3 y + c_3 z + d_3) \\
c(ex + fy + gz + h)
\end{pmatrix}
\]
Extending homogeneous coordinates

Hence, both of these two matrices can be used as perspective matrix:

\[
\begin{pmatrix}
n & 0 & 0 & 0 \\
0 & n & 0 & 0 \\
0 & 0 & n + f & -fn \\
0 & 0 & 1 & 0 \\
\end{pmatrix}
\quad \text{and} \quad
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & \frac{n+f}{n} & -f \\
0 & 0 & \frac{1}{n} & 0 \\
\end{pmatrix}
\quad \text{and} \ldots
\]
Other stages in the graphics pipeline (recap)

- Triangles that lie (partly) outside the view frustum need not be projected, and are clipped.
- The remaining triangles are projected if they are front facing.
- Projected triangles have to be shaded and/or textured.
Clipping

In general, we cannot expect all triangles to lie within the view frustum. Triangles that lie partly outside the view frustum must be clipped.

We must . . .

1. verify if a triangle intersects with a hyperplane
2. create new triangle(s)
Remember the implicit equation of a plane:

\[ f(\vec{p}) = \vec{n}(\vec{p} - \vec{p}_0) = 0 \]

or

\[ f(\vec{p}) = \vec{n} \cdot \vec{p} + d = 0 \]

We already learned that \( f(\vec{p}) \) splits our space into a positive and a negative subspace for which \( f(\vec{p}) > 0 \) and \( f(\vec{p}) < 0 \), respectively.
Finding candidates for clipping

In tutorial 1 and 2, we have proven that the normal always points to the positive side.

Hence, if the normals of the clipping planes point outward:

- $\vec{p}$ is inside the plane if $f(\vec{p}) < 0$
- $\vec{p}$ is outside the plane if $f(\vec{p}) > 0$
Finding intersection points

If two points \( \vec{a} \) and \( \vec{b} \) are on different sides of a hyperplane, we first determine the parametric equation of the line through the points:

\[
\vec{p}(t) = \vec{a} + t(\vec{b} - \vec{a})
\]

Substituting this into the hyperplane equation yields

\[
t' = \frac{\vec{n} \cdot \vec{a} + d}{\vec{n} \cdot (\vec{a} - \vec{b})}
\]
Creating clipped triangles

Given the intersection points, we can clip the triangle against the hyperplane as follows:

- If two vertices are on the positive side, we get one new triangle.
- If one vertex is on the positive side of the hyperplane, we get two new triangles.
But what do we do if a triangle is intersected by two clipping planes?
Clipping against a hyperplane

We have to deal with such situations **one clipping plane at a time**.

We first clip the initial triangle against one of the clipping planes...
...and clip the remaining triangle(s) against the other clipping plane.
Similarly, we can deal with situations where a triangle is intersected by three clipping planes.
Where should we do the clipping?

1. Multiply vertex positions by transform matrix
2. Homogeneous divide
3. Rasterize
Clipping after the perspective divide

Advantage: equations for these hyperplanes are quite simple:

\[-x + l = 0\]
\[x - r = 0\]
\[-y + b = 0\]
\[y - t = 0\]
\[-z + n = 0\]
\[z - f = 0\]

But: there’s a problem at the $XY$-plane . . .
Clipping after the perspective divide may lead to incorrect results if line segments cross the $XY$-plane, since a division by $z$ is involved:

$$z' = n + f - \frac{fn}{z}$$
It is possible to clip against the six clipping planes right before the perspective divide.

The eight corners of the view frustum are easily found by the inverse transformation $M^{-1}_{per}$.

From these, we can derive the plane equations for the view frustum.
Clipping in homogeneous coordinates

Surprisingly, it turns out to be easiest to clip in homogeneous coordinates, which means that we clip in four dimensions against three-dimensional clipping hyperplanes.

The equations for these hyperplanes are quite simple:

\[-x' + lw' = 0\]
\[x' - rw' = 0\]
\[-y' + bw' = 0\]
\[y' - tw' = 0\]
\[-z' + nw' = 0\]
\[z' - fw' = 0\]
Clipping in homogeneous coordinates

Hyperplanes in homogeneous coordinates:

\[-x' + lw' = 0\]
\[x' - rw' = 0\]
\[-y' + bw' = 0\]
\[y' - tw' = 0\]
\[-z' + nw' = 0\]
\[z' - fw' = 0\]
Some rendering engines deal with \textit{arbitrary polygons} rather than with \textit{triangles}.

Also, in \textit{drawing programs} we often need to clip arbitrary polygons.
The Sutherland-Hodgman clipping algorithm is straightforward:

Clip the polygon subsequently against every clipping hyperplane.
The Sutherland-Hodgman algorithm

- Extend upper side of rectangle across 2D space
- Start at one vertex of the polygone and follow its path
- Create new vertex where path crosses clipping line
- Repeat till we are back at starting vertex
- Create new polygon from sets over vertices on or beneath clipping line
- Repeat for every clipping line
The Sutherland-Hodgman algorithm

However, sometimes the Sutherland-Hodgman algorithm results in degenerate polygons.

For example, the resulting polygon on the right has vertices $p_0, i_0, r_2, i_3, p_4, p_5, i_2, r_2,$ and $i_1,$ consecutively.
The Weiler-Atherton algorithm

Basic idea: create a graph with nodes for all vertices of the polygon, corners of the view frustum, and intersection points and edges that allow us to easily extract the clipped polygons.
The Weiler-Atherton algorithm

Building the graph:

- Make a **graph** with three groups of vertices: polygon vertices, clipping region vertices, and intersection vertices.

- Insert edges by walking along the **boundary of the polygon**, including the intersection vertices. Distinguish **outgoing** intersections (colored red in the image) and **incoming** intersections (colored pink).

- Insert edges by walking along the **boundary of the clipping region**, including the intersection vertices.
The Weiler-Atherton algorithm

$p_0$
$p_1$
$p_2$
$p_3$
$p_4$
$p_5$
$p_6$
$p_7$
$p_8$
$i_0$
$i_1$
$i_2$
$i_3$
$r_0$
$r_1$
$r_2$
$r_3$
The Weiler-Atherton algorithm

Using the graph:

- Start at an **outgoing intersection vertex**, and walk along the **boundary of the clipping region** (black edges in image), reporting every vertex along the way.

- Upon encountering an **incoming intersection vertex**, continue along the **boundary of the polygon** (red edges in image).

- Continue, changing from polygon boundary to clipping region boundary and the other way around at outgoing and incoming intersection vertices, respectively, until we reach the starting vertex.

- Start at the next unvisited outgoing intersection vertex to output the next clipped polygon, etc, until no unvisited outgoing intersection vertices are left.
The Weiler-Atherton algorithm
Culling triangles

When a triangle lies entirely outside the view frustum, it can be **culled**, i.e., removed from the graphics pipeline.

Testing **individual triangles** is expensive, however.
Culling bounding volumes of complicated objects (consisting of many triangles) will generally improve the performance of the graphics pipeline.

If a bounding volume is outside of the frustum, then so are the enclosed triangles.

This is called a conservative test. (Why?)
Often spheres are used as bounding volumes.

If the plane is defined by

$$(\vec{p} - \vec{a}) \cdot \vec{n} = 0$$

and the bounding sphere has center $\vec{c}$ and radius $r$, then we have to check if

$$\frac{(\vec{c} - \vec{a}) \cdot \vec{n}}{||\vec{n}||} > r$$
Culling bounding volumes

\[
\frac{(\vec{c} - \vec{a}) \cdot \vec{n}}{\|\vec{n}\|} = \frac{\cos \theta \|\vec{c} - \vec{a}\|}{\|\vec{n}\|} = \cos \theta \|\vec{c} - \vec{a}\|
\]

Remember that the projection \(p_{\vec{v}}\) of a vector \(\vec{v}\) onto another vector is \(\cos \theta \|\vec{v}\|\) with \(\theta\) being the angle between both vectors.

Hence, \(\cos \theta = \frac{p_{\vec{c} - \vec{a}}}{\|\vec{c} - \vec{a}\|}\)

and thus

\[
\cos \theta \|\vec{c} - \vec{a}\| = p_{\vec{c} - \vec{a}}
\]
Other forms of culling

- **frustum culling**
  culling of triangles outside of the view frustum

- **occlusion culling**
  culling of triangles within the frustum that are occluded by others

- **backface culling**
  culling of triangles facing away from the camera
Backface culling

- **frustum culling**
  culling of triangles outside of the view frustum

- **occlusion culling**
  culling of triangles withing the frustum that are occluded by others

- **backface culling**
  culling of triangles facing away from the camera
Surfaces of polyhedral objects are often modeled with connected, outward facing triangles.

How many triangles do we need to model a cube?

What is the maximum number of these triangles that is visible from any given viewing direction?
Backface culling

Obviously, if a camera faces the backside of a triangle, there is no need to draw it.

But how do we know which side we are looking at?
Backface culling

If our models have outward facing normals, we can again use the implicit representation to verify if the camera $\vec{e}$ is on the positive or negative side of the plane / triangle:

- If $f(\vec{e}) > 0$, we draw the triangle
- If $f(\vec{e}) < 0$, we apply backface culling
If we draw triangles that are further away before triangles that are closer to the viewing point, we end up with a correct image:
If we draw triangles that are further away before triangles that are closer to the viewing point, we end up with a correct image:
If we draw triangles that are further away before triangles that are closer to the viewing point, we end up with a correct image:
Cyclic overlap

However, not every arrangement of triangles admits a back-to-front ordering:

And how about intersecting triangles?
Most common approach for hidden surface removal: **Z-buffer** (or *depth buffer*)

Uses additional storage for depth information (mostly hardware, but implementations in software exist as well)
Z-buffer algorithm: basic idea

Apart from the frame buffer, which contains the pixels of the image, also maintain a Z-buffer of the same width and height, to store depth information for the projected triangles.
Z-buffer algorithm: basic idea

- Initialize all Z-buffer entries to $z_{\text{max}}$
- If a pixel is to be drawn at position $[i, j]$, first test if its corresponding $z$-value $p_z$ is smaller than $Z$-buffer$[i, j]$
- If this is the case, draw the pixel, and update $Z$-buffer$[i, j]$ to $p_z$
- Otherwise, do not draw the pixel
- $z$-values for projected vertices: calculated; for remaining pixels: interpolated
Precision issues

For speed: \( z \)-values stored as non-negative integers, i.e.

\[
B \text{ values } \{0, 1, \ldots, B - 1\}
\]

Hence, the \( z \)-values are mapped to intervals of size

\[
\Delta z = (f - n)/B
\]

Points on near plane \( z = n \) → mapped to 0
Points on far plane \( z = f \) → mapped to \( B - 1 \)

Remember: \( z' \) is reciprocal to \( z \)!
Precision issues

This results in

- higher precision close to the eye
- lower precision in the distance

If we have $b$ bits for our $z$-buffer, then $B = 2^b$ and the size of our intervals is

$$\Delta z = (f - n)/2^b$$

Hence, we can influence the precision by

- increasing $b$
- pushing $n$ further away
- moving $f$ closer
Triangles that lie (partly) outside the view frustum need not be projected, and are clipped.

The remaining triangles are projected if they are front facing.

Projected triangles have to be shaded and/or textured.