Graphics 2012/2013, 4th quarter

Lecture 8

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

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

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
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.
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)
Implicit equations revisited

Remember the implicit equation of a plane:

$$f(\vec{p}) = \vec{n}(\vec{p} - \vec{p}_0) = 0 \quad \text{or} \quad f(\vec{p}) = \vec{n} \cdot \vec{p} + d = 0$$
Implicit equations revisited

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.

(cf. tutorial 1, exercise 16 for 2D and tutorial 2, exercise 3, for 3D)
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

Notice the phrasing "inside the plane", i.e. we test against the planes that define the sides of the view frustum.
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.
Clipping against a hyperplane

But what do we do if a triangle is intersected by two clipping planes?
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...
Clipping against a hyperplane

...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.
Finding intersection points

And the case where the triangle intersects with the view frustum but none of the vertices is inside of it.
Where should we do the clipping?

1. Multiply vertex positions by transform matrix.
2. Homogeneous divide.
3. Rasterize.

Recap
Clipping
Culling
Hidden surface removal

How to clip triangles
Where to clip?
Clipping arbitrary polygons

Graphics 2012/2013, 4th quarter
Lecture 08: graphics pipeline (clipping and culling)
Clipping after the perspective divide

Advantage: equations for these hyperplanes are quite simple

\[-x + l = 0\]
\[x - r = 0\]
\[-y + b = 0\]
\[y - t = 0\]
\[-z + n = 0\]
\[z - f = 0\]
Clipping after the perspective divide

Example: left side of the view frustum

\[-x + l = 0\]
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:

$$z' = n + f - \frac{fn}{z}$$
Clipping after the perspective divide

\[ z' = n + f - \frac{fn}{z} \]

Because of the discontinuity, objects behind the eye can move in front of it.

Because of the change of signs at \( f(z) = 0 \), objects in front of the eye can move behind it.
Clipping before the perspective divide

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 $M_{per}^{-1}$.

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:

\[
\begin{align*}
-x' + lw' &= 0 \\
x' - rw' &= 0 \\
-y' + bw' &= 0 \\
y' - tw' &= 0 \\
-z' + nw' &= 0 \\
z' - fw' &= 0
\end{align*}
\]
Clipping in homogeneous coordinates

Hyperplanes in homogeneous coordinates:

\[-x' + lw' = 0\]

\[x' = p_x'\]

Hyperplanes after the homogeneous divide:

\[-x + l = 0\]

\[x = \frac{p_x'}{w'}\]
Some rendering engines deal with arbitrary polygons rather than with triangles.

Also, in drawing programs we often need to clip arbitrary polygons.
The Sutherland-Hodgman clipping algorithm clips 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

Unfortunately, the Sutherland-Hodgman algorithm can result 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.
Building the graph:

- Make a graph with three groups of vertices:
  - polygon vertices
  - clipping region vertices
  - intersection vertices

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

- Insert directed edges by walking along the boundary of the clipping region, including the intersection vertices.
The Weiler-Atherton algorithm
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 graph), reporting every vertex along the way.
- Upon encountering an **incoming intersection vertex**, continue along the **boundary of the polygon** (red edges in graph).
- 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

\[ p_0 \rightarrow p_1 \rightarrow p_2 \rightarrow p_3 \rightarrow i_0 \rightarrow r_0 \rightarrow i_1 \rightarrow r_1 \rightarrow r_2 \rightarrow r_3 \rightarrow p_0 \]
The Weiler-Atherton algorithm

Clipped polygons:

\[ i_0 \rightarrow i_1 \rightarrow p_0 \rightarrow i_0 \]
\[ i_2 \rightarrow i_3 \rightarrow p_4 \rightarrow p_5 \rightarrow i_2 \]
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?)
Culling bounding volumes

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

Using the scalar product
\[ \vec{a} \cdot \vec{b} = \cos \theta \|\vec{a}\| \|\vec{b}\| \]

we can rewrite the condition as
\[ \frac{(\vec{c} - \vec{a}) \cdot \vec{n}}{\|\vec{n}\|} = \frac{\cos \theta \|\vec{c} - \vec{a}\|}{\|\vec{n}\|} = \cos \theta \|\vec{c} - \vec{a}\| \]

Remember that the length \( p \) of the projection of a vector \( \vec{v} \) onto another vector is \( \cos \theta \|\vec{v}\| \).

Hence, the length of the projection of \( \vec{c} - \vec{a} \) onto \( \vec{n} \) is \( \cos \theta = \frac{p(\vec{c} - \vec{a})}{\|\vec{c} - \vec{a}\|} \)
and thus \( \cos \theta \|\vec{c} - \vec{a}\| = p(\vec{c} - \vec{a}) \)
Culling bounding volumes

Hence, our condition becomes

\[
\frac{(\vec{c} - \vec{a}) \cdot \vec{n}}{\|\vec{n}\|} = \cos \theta \|\vec{c} - \vec{a}\| = p \vec{c} - \vec{a} > r
\]
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 within the frustum that are occluded by others

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

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 $f(\vec{e}) > 0$ then we draw the triangle
- If $f(\vec{e}) < 0$ then 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:
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 $z_{max}$
- 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
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.