OpenGL Stages

- After projection, several stages before objects drawn to screen
- These stages are non-programmable

Vertex shader: programmable

In hardware: **NOT** programmable
Hardware Stage: Primitive Assembly

- Up until now: Transformations and projections applied to vertices individually
- **Primitive assembly**: After transforms, projections, individual vertices grouped back into primitives
- **E.g. v6, v7 and v8** grouped back into triangle
Hardware Stage: Clipping

- After primitive assembly, subsequent operations are per-primitive
- **Clipping**: Remove primitives (lines, polygons, text, curves) outside view frustum (canonical view volume)
Rasterization

- Determine which pixels that primitives map to
  - Fragment generation
  - Rasterization or scan conversion
Fragment Processing

● Some tasks deferred until fragment processing

Hidden Surface Removal

Antialiasing

Modeling → Geometric processing → Rasterization → Fragment processing → Frame buffer

Transformation

Projection

Hidden surface Removal

Antialiasing
Clipping

- 2D and 3D clipping algorithms
  - 2D against clipping window
  - 3D against clipping volume
- 2D clipping
  - Lines (e.g. dino.dat)
  - Polygons
  - Curves
  - Text
Clipping 2D Line Segments

- **Brute force approach**: compute intersections with all sides of clipping window
  - Inefficient: one division per intersection
2D Clipping

- **Better Idea:** eliminate as many cases as possible without computing intersections
- Cohen-Sutherland Clipping algorithm

```
<table>
<thead>
<tr>
<th></th>
<th>y = y_{max}</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = x_{min}</td>
<td></td>
</tr>
</tbody>
</table>

x = x_{max}

<table>
<thead>
<tr>
<th></th>
<th>y = y_{min}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Determine whether a point \((x,y)\) is inside or outside of the world window?

If \((x_{\text{min}} \leq x \leq x_{\text{max}})\)

\textbf{AND} \((y_{\text{min}} \leq y \leq y_{\text{max}})\)

then the point \((x,y)\) is inside

else the point is outside
Clipping Lines

3 cases:
- **Case 1:** All of line in
- **Case 2:** All of line out
- **Case 3:** Part in, part out
Clipping Lines: Trivial Accept

Case 1: All of line in
Test line endpoints:

\[ X_{min} \leq P1.x, \quad P2.x \leq X_{max} \quad \text{AND} \quad Y_{min} \leq P1.y, \quad P2.y \leq Y_{max} \]

Note: simply comparing x,y values of endpoints to x,y values of rectangle

Result: trivially accept.
Draw line in completely
Clipping Lines: Trivial Reject

Case 2: All of line out
Test line endpoints:

- \( p1.x, p2.x \leq X_{\text{min}} \) OR
- \( p1.x, p2.x \geq X_{\text{max}} \) OR
- \( p1.y, p2.y \leq Y_{\text{min}} \) OR
- \( p1.y, p2.y \geq Y_{\text{max}} \)

Note: simply comparing x,y values of endpoints to x,y values of rectangle

Result: trivially reject.
Don’t draw line in
Clipping Lines: Non-Trivial Cases

Case 3: Part in, part out

Two variations:
  - One point in, other out
  - Both points out, but part of line cuts through viewport

Need to find inside segments

Use similar triangles to figure out length of inside segments

\[
\frac{d}{dely} = \frac{e}{delx}
\]
Clipping Lines: Calculation example

If chopping window has 
(left, right, bottom, top) = (30, 220, 50, 240), 
what happens when the following lines are 
chopped?

(a) p1 = (40, 140), p2 = (100, 200)

(b) p1 = (20, 10), p2 = (20, 200)

(c) p1 = (100, 180), p2 = (200, 250)
Cohen-Sutherland Pseudocode

```c
int clipSegment( Point2& p1, Point2& p2, RealRect W ) {
    do {
        if( trivial accept ) return 1; // whole line survives
        if( trivial reject ) return 0;  // no portion survives
        // Otherwise chop
        if( p1 is outside )
            // find surviving segment
            {
                if( p1 is to the left ) chop against left edge
                else if( p1 is to the right) chop against right edge
                else if( p1 is below ) chop against the bottom edge
                else if( p1 is above ) chop against the top edge
            }
    }
}
```
else // p2 is outside
    // find surviving segment
    {
        if( p2 is to the left ) chop against left edge
        else if( p2 is to right ) chop against right edge
        else if( p2 is below ) chop against the bottom edge
        else if( p2 is above ) chop against the top edge
    }
} while( 1 );
}
Using Outcodes to Speed Up Comparisons

- Encode each endpoint into outcode (what quadrant)

\[
\begin{align*}
  b_0 &= 1 \text{ if } y > y_{\text{max}}, \ 0 \text{ otherwise} \\
  b_1 &= 1 \text{ if } y < y_{\text{min}}, \ 0 \text{ otherwise} \\
  b_2 &= 1 \text{ if } x > x_{\text{max}}, \ 0 \text{ otherwise} \\
  b_3 &= 1 \text{ if } x < x_{\text{min}}, \ 0 \text{ otherwise}
\end{align*}
\]

\[
\begin{array}{c|c|c}
  \text{b}_0\text{b}_1\text{b}_2\text{b}_3 & 1001 & 1000 & 1010 \\
\hline
  y = y_{\text{max}} & 0001 & 0000 & 0010 \\
  y = y_{\text{min}} & 0101 & 0100 & 0110 \\
\end{array}
\]

- Outcodes divide space into 9 regions
- Trivial accept/reject becomes bit-wise comparison
References