Computer Graphics - Chapter 7
From Vertices to Fragments

Objectives are:
• How your program are processed by the system that you are using,
• Learning to efficiently use a graphics system by learning the implementation process, and
• Learn about capabilities that utilizes the frame buffer.

Learning implementation requires studying algorithms. We must be careful to consider issues such as theoretical versus practical performance, hardware versus software implementations.

Of course, we can test whether an OpenGL implementation is working correctly. How? By looking at the outcome. But there are many different algorithms that can be employed, are we testing all of them to find the best?

Major Tasks
There are four major tasks that any implementation must perform to render a geometric entity.
1. Modeling
   Set of vertices that specifies a set of geometric objects.

2. Geometric processing
   View volume, hidden-surface removal (visible surface determination), also called front-end-processing.

3. Rasterization
   Rasterization or scan conversion.

4. Display
   aliasing
Basic Implementation Strategies

There are two major approaches to implementation. They differ in the order that process the required processing steps. A program takes a set of vertices specifying a geometric object, and produces as output pixels in the frame buffer. If the program produces an image, it must assign every pixel. If it produces a correct image, it must process every geometric primitive and every light source. Thus, a program has some loops that iterate over these basic variables. The question then is; which variable controls the outer loop?

1. **Image Oriented (sort first)**
   The loop is over pixels, or rows of pixels called **scan lines**, that constitute the frame buffer.
   ```
   for(each_pixel) assign_a_color(pixel);
   ```

2. **Object Oriented (sort last)**
   The outer loop is over the objects. We can think of the program as controlled by a loop of this form:
   ```
   for(each_object) render(object);
   ```
   A pipeline renderer fits this description.

Implementation of Transformations

In the homogeneous coordinate, the affine transformations are achieved by concatenating a sequence of matrices.

1. **Object (world) coordinates**
2. **Eye (camera) coordinates**
3. **Clip coordinates**
4. **Normalized device coordinates**
5. **Window (screen) coordinates**.

![Diagram of transformation stages](image)

**Clipping** is the process of determining which primitives, and parts of primitives, fit within the clipping volume defined by the application program. When a component of a clipped vertex is not equal to 1, we have to do a perspective division to convert back from homogeneous coordinates to normal 3-D coordinates.
Implementation of Transformations – cont.

We do this, because, all primitives that can possibly be displayed lie within the cube: \( x = \pm 1, \quad y = \pm 1, \quad z = \pm 1. \)

This coordinate system is called normalized device coordinates, because it depends on neither the original application units nor the particulars of the display device.

\[
\begin{align*}
X_p &= x, \\
y_p &= y, \\
z_p &= 0.
\end{align*}
\]

What if we had a 3-D viewports?

Line Segment Clipping

A clipper decides which primitives, or parts of primitives, appear on the display. Primitives that fit within the specified view volume pass through the clipper, or are accepted. Primitives that cannot appear on the display are eliminated, or rejected or culled.

Primitives that are only partially within the view volume must be clipped such that any part lying outside the volume is removed.

Primitives may be clipped after they have been projected from three to two dimensional objects.
Cohen-Sutherland Clipping

The sides of the window is extended to infinity, thus the space is broken to 9 regions.

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

Can you write the \( b_1, b_2 \) and \( b_3 \)?

Consider a line segment whose outcodes are given by \( o_1 = \text{outcode}(x_1, y_1) \) and \( o_2 = \text{outcode}(x_2, y_2) \). What are the four cases?

Cohen-Sutherland Clipping – cont.

Case (1): \( o_1 = o_2 = 0 \)

Case (2): \( o_1 \neq 0, o_2 = 0 \)
Or vise versa

Case (3): \( o_1 \& o_2 \neq 0 \)

Case (4): \( o_1 \& o_2 = 0 \)
Cohen-Sutherland Clipping – cont.

This algorithm works best when there are many line segments, but few are actually displayed. In this case, most of the line segments lie fully outside one or two of the extended sides of the clipping rectangle, and thus can be eliminated on the basis of their outcodes.

What are the possible problems you may face, when you use this method?

Let’s solve this example:

**Liang-Barsky Clipping**

If we use the parametric form for lines, we can approach the clipping of line segments in a different and more difficult manner.

Suppose the two endpoints are given as \( p_1 = [x_1, y_1]^T \) and \( p_2 = [x_2, y_2]^T \), the unique line that crosses these two points is:

\[
p(\alpha) = (1 - \alpha)p_1 + \alpha p_2
\]

Or as two scalar equations:

\[
x(t) = (1 - \alpha)x_1 + \alpha x_2 \\
y(t) = (1 - \alpha)y_1 + \alpha y_2
\]

Advantages?

Possible values for \( \alpha \)?

How do we distinguish between (a) and (b)?
Polygon Clipping

So are we going to clip polygons against rectangular windows for display?

We can generate polygon-clipping algorithms directly from line-clipping algorithms by clipping the edges of the polygon successively. However, there is a problem with this algorithm. A polygon is an object, and, Depending on the form of the polygon, we can generate more than one polygon object by clipping.

A Solution?

Polygon Clipping - cont

So are we going to clip polygons against rectangular windows for display?

There is no problem with convex polygons, so the options for treating concave polygons are:
1) forbid it
2) divide (tessellate) a given polygon into a set of convex polygons.
A line-segment clipper can be viewed as a black box whose input is the pair of vertices from the segment to be tested and clipped, and whose output is either either a pair of vertices corresponding to the clipped line segment, or nothing if the line lies outside.
Polygon Clipping - cont

We can clip only against one of the sides. Suppose, we clip against the top. We can think of this operation as a black box in which both the input and output are pairs of vertices, with the value of $y_{\text{max}}$ as a parameter known to the clipper.

Thus, the clipper returns one of the three pairs:
1) $\{(x_1, y_1), (x_2, y_2)\}$, 2) $\{(x_1, y_1), (x_p, y_{\text{max}})\}$, or 3) $\{(x_p, y_{\text{max}}), (x_2, y_2)\}$.

We can clip against the bottom, right, and left lines independently, using the same equations with the roles of $x$ and $y$ exchanged as necessary, and the values for sides of the window inserted.
Clipping of Other Primitives

Sometimes, we may want to clip objects before they reach the hardware or to employ algorithms optimized for other primitives.

Bounding Boxes

If we try to use clipping to clip a polygon with many sides, we may end up with a situation where the polygon lie outside the clipping window. We can exploit this observation through the use of bounding box or extent of polygon:

The smallest rectangle, aligned with the window, that contains the polygon.

How can we calculate such a window?
Bounding Boxes – cont.

What is next?

Three situations may appear:
1) Polygon outside the window; no clipping is necessary. Why?

2) The polygon inside the window, we can determine that it is inside by comparing the bounding box with the window.

3) This case is more complicated. When a polygon partially inside. In such a case, we have to carry detail clipping, using all the edges of the polygon.

Curves, Surfaces, and Text

The variety of curves and surfaces that we can define mathematically makes it difficult to find general algorithms for processing these objects.

The potential difficulties can be seen from the two-dimensional curves such as those shown here.

If we have a simple curve, such as a quadratic, we can compute intersections. This cost more than that of a line.

For more complex curves, we need to calculate intersections. That is a difficult task. We can avoid such problems by approximating curves with line segments and surfaces with planar polygons.

The use of bounding boxes can prove helpful.
Curves, Surfaces, and Text

The handling of text differs from API to API, with many APIs allowing the user to specify how detailed a rendering of text is required.

There are two extremes:

1) Text is stored as bit patterns and is rendered directly by the hardware without any geometric processing. Any required clipping is done in the frame buffer.

2) Text is defined like any other geometric object, and is then processed through the standard viewing pipeline.

OpenGL allows both types. The user can decide between the bitmap or the stroke characters.

We can do the clipping until after objects have been projected and converted into screen coordinates. So clipping can be done in the frame buffer through a technique called, **scissoring**. However, it is better to to clip geometric entities before the vertices reach the frame buffer.

---

Clipping in Three Dimensions

In three dimension, we clip against a bounded volume, rather than against a bounded region in the plane.

\[
\begin{align*}
x_{\min} & \leq x \leq x_{\max} \\
y'_{\min} & \leq y' \leq y'_{\max} \\
z_{\min} & \leq z \leq z_{\max}
\end{align*}
\]
Clipping in Three Dimensions – cont.

In three dimension, we clip against a bounded volume, rather than against a bounded region in the plane.

The concepts of all three clipping algorithms can be extended to 3-D case.

For example, for the Cohen-Sutherland algorithm, we replace the 4-bit outcode with a 6-bit outcode.

What do we use the extra 2 bits for?

For the Liang-Barsky algorithm, we add

\[ z(\alpha) = (1 - \alpha)z_1 + \alpha z_2 \]

The difference between 2-D and 3-D clippers is that instead of clipping lines against lines, as we do in 2-D, we are clipping either lines against surfaces or surface against surfaces.

Clipping in Three Dimensions – cont.

If we write the line and plane equations in matrix form (where \( \mathbf{n} \) is the normal to the plane and \( \mathbf{p}_0 \) is a point on the plane), we must solve the equation:

\[
\mathbf{p}(\alpha) = (1 - \alpha)\mathbf{p}_1 + \alpha \mathbf{p}_2 \\
\mathbf{n} \cdot (\mathbf{p}(\alpha) - \mathbf{p}_0) = 0
\]

For the \( \alpha \) corresponding to the point of intersection:

\[
\alpha = \frac{\mathbf{n} \cdot (\mathbf{p}_0 - \mathbf{p}_1)}{\mathbf{n} \cdot (\mathbf{p}_2 - \mathbf{p}_1)}
\]
Hidden-Surface Removal (All very Important)

Object-Space and Image-Space Approaches

Back face Removal

The z-Buffer Algorithm

Depth Sort and the Painter’s Algorithm

The Scan-line Algorithm

Scan Conversion

Pixels can be set to a given color using a call like:

\[ \text{write\_pixel}(\text{int } \text{ix}, \text{int } \text{iy}, \text{int } \text{value}); \]

The argument value can be either an index, in color-index mode, or a pixel value, such as a 32-bit number in RGBA-color mode.

The simplest scan conversion algorithm is DDA algorithm stands for the Digital Differential Analyzer.