Clipping & Scan Conversion

CSE167: Computer Graphics
Instructor: Steve Rotenberg
UCSD, Fall 2005
Project 2

- Render a 3D hand (made up of individual boxes) using hierarchical transformations (push/pop)
- The hand should perform some simple motion, such as opening and closing the fingers
- Enable some basic lighting
- Use object oriented classes for:
  - Model (like project 1)
  - Hand (& Finger if you want)
  - Camera
  - Light
Example: Yaw

A spaceship is floating out in space, with a matrix $W$. The pilot wants to turn the ship 10 degrees to the left (yaw). Show how to modify $W$ to achieve this.
Example: Yaw

- We rotate $W$ around its own $b$ vector, using the ‘arbitrary axis rotation’ matrix. In addition, we pivot the rotation about the object’s position ($d$ vector):

$$M = T(W,d) \cdot R_a(W,b,10^\circ) \cdot T(-W,d)$$

$$W' = M \cdot W$$

where $R_a(a,\theta) =$

$$
\begin{bmatrix}
  a_x^2 + c_\theta(1 - a_x^2) & a_x a_y (1 - c_\theta) - a_z s_\theta & a_x a_z (1 - c_\theta) + a_y s_\theta & 0 \\
  a_x a_y (1 - c_\theta) + a_z s_\theta & a_y^2 + c_\theta(1 - a_y^2) & a_y a_z (1 - c_\theta) - a_x s_\theta & 0 \\
  a_x a_z (1 - c_\theta) - a_y s_\theta & a_y a_z (1 - c_\theta) + a_x s_\theta & a_z^2 + c_\theta(1 - a_z^2) & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix}
$$
Triangle Rendering

- The main stages in the *traditional graphics pipeline* are:
  - Transform
  - Lighting
  - Clipping / Culling
  - Scan Conversion
  - Pixel Rendering
Transformation

- In the transformation stage, vertices are transformed from their original defining object space through a series of steps into a final ‘2.5D’ device space of actual pixels.

\[
\mathbf{v}_{(4D)}' = \mathbf{P} \cdot \mathbf{C}^{-1} \cdot \mathbf{W} \cdot \mathbf{v}
\]

\[
\mathbf{v}'' = \begin{bmatrix}
\frac{v'}{x} & \frac{v'}{y} & \frac{v'}{z} \\
\frac{v'}{w} & \frac{v'}{w} & \frac{v'}{w}
\end{bmatrix}
\]

\[
\mathbf{v}''' = \mathbf{D} \cdot \mathbf{v}''
\]
Transformation: Step 1

\[ v'_{(4D)} = P \cdot C^{-1} \cdot W \cdot v \]

- **v**: The original vertex in object space
- **W**: Matrix that transforms object into world space
- **C**: Matrix that transforms camera into world space (\(C^{-1}\) will transform from world space to camera space)
- **P**: Non-affine perspective projection matrix
- **v'**: Transformed vertex in 4D un-normalized viewing space

Note: sometimes, this step is broken into two (or more) steps. This is often done so that lighting and clipping computations can be done in camera space (before applying the non-affine transformation)
In the next step, we map points from 4D space into our normalized viewing space, called *image space*, which ranges from -1 to 1 in x, y, and z.

From this point on, we will mainly think of the point as being 2D (x & y) with additional depth information (z). This is sometimes called 2.5D.
In the final step of the transformation stage, vertices are transformed from the normalized -1…1 image space and mapped into an actual rectangular *viewport* of pixels.

\[ v''' = D \cdot v'' \]
Transformation

\[ \mathbf{v}'_{(4D)} = P \cdot C^{-1} \cdot W \cdot \mathbf{v} \]

\[ \mathbf{v}'' = \begin{bmatrix} \dot{v}'_x & \dot{v}'_y & \dot{v}'_z \\ \dot{v}'_w & \dot{v}'_w & \dot{v}'_w \end{bmatrix} \]

\[ \mathbf{v}''' = D \cdot \mathbf{v}'' \]
Clipping & Culling
Clipping

- Some triangles will be completely visible on the screen, while others may be completely out of view.
- Some may intersect the side of the screen and require special handling.
- The camera’s viewable space forms a volume called the *view volume*. Triangles that intersect the boundary of the view volume must be *clipped*.
- The related process of *culling* refers to the determination of which primitives are completely invisible.
- The output of the clipping/culling process is a set of visible triangles that lie within the dimensions of the display device.
Clipping

- Triangles are generally clipped one at a time, although more complex implementations might do them in groups.
- We will just consider the clipping of a single triangle.
- Both perspective and orthographic view volumes are bounded by 6 planes, and so can be treated very similarly.
- A single triangle could potentially intersect all 6 planes (although it would be pretty uncommon for a triangle to intersect more than 2 in practice).
- The number of planes isn’t actually that important, since the basic algorithm clips to each plane one at a time.
If a triangle intersects a particular clipping plane, it will become either one or two new triangles.

These are then tested against the remaining clipping planes.
Clipping

To clip a triangle to a particular clipping plane, we first have to determine which of the triangle’s 3 vertices are on which side of the plane.

- If all 3 vertices are on the ‘inside’, then the triangle doesn’t need to be clipped.
- If all 3 vertices are on the ‘outside’, then the triangle is completely invisible and gets thrown away (culled).
- If only 1 vertex is inside, then two new vertices must be (temporarily) created and a single new triangle is created.
- If 2 verts are inside, then two new vertices must be created as well as two new triangles.
- Both of the clipping cases involve finding where 2 edges of the triangle intersect the plane, and so can be treated very similarly.
A clipping plane can be defined by a point \( p \) on the plane and a unit length normal \( n \), (we will choose \( n \) so that it points towards the inside of the view volume).

To test if a triangle’s verts are inside or outside of the clipping plane, we compute a signed distance to the plane for each of the 3 vertices, \( v_0 \), \( v_1 \), and \( v_2 \).

\[
\begin{align*}
d_0 &= (v_0 - p) \cdot n \\
d_1 &= (v_1 - p) \cdot n \\
d_2 &= (v_2 - p) \cdot n
\end{align*}
\]

A positive (or 0) distance indicates that the vertex is inside the view volume, and negative distance indicates that it is outside.
To find point $\mathbf{x}$ where an edge intersects a plane, we need the two verts $\mathbf{v}_a$ and $\mathbf{v}_b$ that make up the edge, as well as their signed distances $d_a$ and $d_b$.

$$t = \frac{d_b}{d_b - d_a}$$

$$\mathbf{x} = t\mathbf{v}_a + (1 - t)\mathbf{v}_b$$
Clipping: Spaces

- Clipping can be done in just about any space desired.
- It is common to perform clipping in camera space, as it is a regular 3D space and the clipping planes are conveniently described in this space.
- This necessitates transforming verts into camera space, then clipping, then transforming them into 4D un-normalized view space.
- If one is clever, this can be sped up and clipping can be performed in 4D un-normalized space, eliminating the need to transform the vertex through two separate matrices.
Clipping: Setup

- In preparation for the clipping process, it is useful to pre-compute the normals for the 6 planes in the view volume.
- These can be done once per frame when the projection matrix is specified (before the actual rendering begins).
In computer graphics, *culling* refers to the process of determining what is *not* visible.

The sooner we can detect that a triangle is not going to be visible, the less time has to be spent processing it.

Culling happens at the level of individual triangles, but can also happen at the object level as well.

If we can quickly determine that an entire object lies off the screen, then we don’t have to process any of the vertices or triangles at all.

For now, we will not concentrate on object culling, but just look at culling of individual triangles.
Culling

- There are three common reasons to cull a particular triangle
  - If it doesn’t lie within the view volume (view frustum culling)
  - If it is facing ‘away’ from the viewer (backface culling)
  - If it is degenerate (area=0)
- The first case is built automatically into the clipping algorithm which we already covered
Backface Culling

- It is very common to use triangles to model the *surface* of an object which is inherently volumetric in nature.
- In many cases, most or all of these triangles will only be visible from the outside of the object.
- In a sense, these triangles are one-sided and only visible from the ‘front’, as we will never have to see them from the ‘back’.
- Any back facing triangles should be culled as early as possible, as it would be expected that up to 50% of the triangles in a scene would be back facing.
- Usually, backface culling is done before clipping, as it is a very quick operation and will affect a much larger percentage of triangles than clipping.
Backface Culling

- By convention, the front side of the triangle is defined as the side where the vertices are arranged in a counterclockwise fashion.

- Most renderers allow triangles to be defined as one or two sided. Only one-sided triangles need to be backface culled.

- Renderers also usually allow the user to specify whether the front or back should be culled, as there are various cases where one may want to one side or the other (certain special effects, cases where reflection transformations are used...).
Backface Culling

- Backface culling is usually done in 3D camera space, where the camera itself is located at (0,0,0), however, it can also be done in object space (even before transformation) by transforming the camera position into object space.
- We must first compute the triangle normal (it does not need to be unit length).
- Then, we check if the normal is more than 90 degrees from the vector from the triangle to the camera position \((\mathbf{e})\):

\[
\mathbf{n} = (\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_2 - \mathbf{p}_0) \\
\text{if} ((\mathbf{e} - \mathbf{p}_0) \cdot \mathbf{n} \leq 0) \text{ then triangle is invisible}
\]
Degenerate Culling

- It would also be nice if we could throw away any degenerate triangles at this point also.
- A degenerate triangle has its three vertices arranged in such a way as to cause the area of the triangle to be 0.
- This might happen if all 3 verts lie in a straight line.
- It might also happen if 2 of the verts (or all 3) are located at the exact same place.
- Fortunately, the backface cull test will automatically reject these, as $\mathbf{n}$ will be $[0 \ 0 \ 0]$ in these cases.

\[
\mathbf{n} = (\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_2 - \mathbf{p}_0)
\]

\[
\text{if } ((\mathbf{e} - \mathbf{p}_0) \cdot \mathbf{n} \leq 0) \text{ then triangle is invisible}
\]
Transform, Projection, Clipping

- At this point we have seen how to take a set of triangles defined in object space and transform/project/clip/cull them into a new set of triangles in device space that are guaranteed to lie within the boundaries of the viewport.
- In the clipping/culling process, some triangles will be removed, while other triangles may be created.
- The most straightforward way to do all this is to simply process one triangle at a time.
- More complex implementations may try to operate on a set of triangles all at the same time to share some work, but we will not focus on this issue now.
Scan Conversion
Scan Conversion

- Now that we have 2D triangles in device coordinates, we need to determine exactly which pixels are covered by that triangle.
- This process is known as *scan conversion* or *rasterization*.
- Scan conversion is essentially a 2D process, operating in xy device space.
- In scan conversion, per-vertex data, such as color and depth, are interpolated across the polygon so that a unique color and depth are computed per pixel.
- It is also common to interpolate other per-vertex data such as texture coordinates, normals, or arbitrary application-specific properties.
Device Coordinates

- If we have a viewport with 800 x 600 pixels, our device coordinates range from 0.0 to 800.0 in x and 0.0 to 600.0 in y
- The center of the lower left pixel is 0.5, 0.5 and the center of the upper right pixel is 799.5, 599.5
Our main focus will be on triangle rasterization, but it is important to remember that other rendering primitives such as lines and points can be rasterized.
Rasterization Rules

- Our basic rule for rasterizing a triangle is that we want a pixel to be filled in if the center of the pixel lies within the triangle.

- If the center of the pixel exactly touches the edge or vertex of the triangle, we will render the triangle only if the triangle covers the point (not pixel) immediately to the right of the center of the pixel (like 0.000001 to the right).

- By defining precise rules like these, that the exact same pixels are rendered regardless of the order the triangles are rendered (and that no pixels get rendered twice if they are on the border of two or more triangles).

- It is also important to remember that the vertex coordinates in device space are still floating point values and should *not* be rounded to integers.
Triangle Rasterization
There are several different approaches to triangle rasterization that have been developed over the years to suit different systems. I will broadly classify them as sequential and parallel approaches. The sequential approaches fill pixels in order and take advantage of fast incremental computations. The parallel approaches require more total computation, but can be distributed to several different computational units (within a single chip). The parallel approaches also usually have the advantage of less time in set-up, and so can be more efficient for very small triangles (like ones that cover 3 pixels or less). This is actually very useful for rendering highly tessellated surfaces. We will mainly focus on the sequential methods, as they are more common in modern systems and can still be parallelized to some extent.
Sequential Triangle Rasterization

- We will start at the top and work our way down one scanline at a time.
- For each scanline, we will fill from left to right.
- This is the most common way people think of it, but different renderers may do it in which ever order they want.
- In a hardware renderer, the order might be arranged in such a way as to optimize memory access performance.
Triangle Rasterization

- The input to the triangle rasterization process is three 2D vertices, each with a depth value and possible other properties.
- The first step is to sort these vertices into top-down order.
- We will say that \( v_0 \) is now the top (highest y value), with \( v_1 \) in the middle, and \( v_2 \) at the bottom.
- This may break our counterclockwise ordering, but is not important at this point, as that information is used in backface culling, which should have happened already by this point.
- Our process will be to first render the top half of the triangle (the part between \( v_0 \) and \( v_1 \)) and then the bottom half (from \( v_1 \) to \( v_2 \)).
Triangle Rasterization

\[ \text{V}_0 \]

\[ \text{V}_1 \]

\[ \text{V}_2 \]
To set up for our rasterization, we must compute the slope of each edge.

We will compute \( \frac{dx}{dy} \), so we know how much the \( x \)-value changes with each full pixel step in \( y \).
Finding The First Scanline

- Assuming we are filling the top half of the triangle (from $v_0$ to $v_1$), we must find the first actual scanline to start on.
- We also find the $x$ values of where edges $v_0v_1$ and $v_0v_2$ intersect that scanline.
- We will call these $x_0$ (left) and $x_1$ (right).
- We can use the computed slopes to do this quickly.

$$ y = \text{int}(v_{0y} + 0.5) - 0.5 $$

$$ \Delta y = \text{frac}(v_{0y} + 0.5) $$

$$ x_0 = v_{0x} - \Delta y \frac{dx_0}{dy} $$
Filling The Span

- Now we can loop from $x_0$ to $x_1$ to fill in the actual pixels, making sure to round things so that we only fill pixels whose centers lie between $x_0$ and $x_1$. 
Looping in Y

- We loop through all scanlines from $v_0$ down to $v_1$
- With each scanline, we compute new values of $x_0$ and $x_1$ incrementally with the slopes

\[ x_0 = x_0 - \frac{dx_0}{dy} \]
\[ x_1 = x_1 - \frac{dx_1}{dy} \]
Looping in Y

- We loop from \( v_0 \) down to \( v_1 \)
- Then, we recompute some slopes and set up to render the bottom half from \( v_1 \) down to \( v_2 \), which proceeds just like the top half
Sequential Rasterization

- As we can see, the actual cost per pixel is very low, and the cost per scanline is also very low.
- However, as we can see, there is some cost in the set-up (several divisions, and some other stuff).
- The algorithm gets its speed by taking advantage of incrementally computing most of the information it needs with simple addition operations.
- In hardware, the entire rasterization process is usually done with fixed point math, and much of the work can be done with pretty low precision fixed point (16 bits is usually fine for most operations).
The term *hidden surface removal* refers to the process of making sure that pixels of triangles that are blocked by others don’t get drawn.

Historically, there have been several approaches to this, but these days, the most popular approach (within the ‘traditional graphics pipeline’) is to use a z-buffer.
Z-Buffer

- With the Z-Buffer technique, every pixel stores a depth (or z) value, often in 32 bit fixed point format.
- When the screen is cleared at the start of rendering a frame, the z-buffer is cleared as well, and every value is set to the furthest value.
- When a triangle is rasterized, the z value is interpolated across the triangle to compute a z value for each pixel.
- Before the pixel gets rendered, the interpolated z value is compared to the value stored in the z-buffer for that pixel.
- The pixel is only rendered if it is closer than the value already in the z-buffer.
- When the pixel is rendered, the new (near) z value is written into the framebuffer.
- This process ensures pixel accurate hidden surface removal so that triangles will render correctly even when they intersect other triangles.
Z Interpolation

- To implement z-buffering, we can make some relatively minor additions to our scan conversion process
- Because the triangle is flat, we can compute a $dz/dy$ and $dz/dx$ value for the entire triangle at the start of rasterization
- The z value gets incremented in much the same way as the x values were previously
- This works largely because we chose a perspective mapping that preserves straight lines
Z-Buffer Limitations

- Z-buffering is a great, simple, fast technique, but can suffer from occasional visual artifacts if not treated carefully.
- For one thing, the perspective mapping causes the z-buffer to have greater resolution for close objects than far objects.
- This is actually a good thing, as we usually render objects with more detail when they are close and with less detail when they are far. The more detailed objects require more precision in the z-buffer.
- A common problem with z-buffering the z-fighting.
- When two triangles are co-planar or nearly co-planar, the fixed point z-interpolation can cause errors and we see flickering patterns where the two triangles overlap.
- There are some ways to fix this, but one way to improve precision is to keep the ratio of far/near clip distances as low as possible.
- For example, a near clip distance of 0.1 and a far clip distance of 1000 (ratio of 10000) works reasonably well with a 32 bit z-buffer. Higher ratios can run into more problems...
In addition to interpolating a z value across the triangle, we can interpolate other properties as well.

Examples include color, normals, texture coordinates, or other things.

To interpolate a color for example, the red, green, and blue components would each be interpolated in much the same way as the z value.

When we discuss texture mapping, we will see that some additional work needs to be done to make sure that properties are interpolated correctly.