Questions about Last Week?
Questions about Assignment?

- Comments about submission
  - Deadline is Friday, 2/26 at 5:30 pm
    Standard rules about penalties etc. apply
  - Impress us with clean code: Comments, no dead code, ...
  - Put a header with your name in every file you submit
  - Look for mechanics of submission on the webpage

Overview of Week 6

- Scan Conversion
  - lines, triangles, polygons
  - attribute interpolation and perspective correction

- Second Assignment
Scan Conversion: Overview

- Scan Conversion of an object
  - a.k.a Rasterization
  - Determine which pixels are affected by the object, i.e. which pixels must be set to display the object
  - Determine pixel value at those pixels
  - Pixel values a.k.a. pixel attributes
    - Color (RGB), Depth (Z), Alpha (A), e.g. transparency, Texture (u,v), Fog, Stencil, etc.

Scan Conversion: Primitives

- Each primitive type requires a special scan conversion procedure
  - For instance, polygons are rasterized different from circle or lines

- We will discuss
  - Points
  - Lines
  - Triangles
  - Polygons

- We will not discuss (see textbook)
  - Circles and ellipses
  - Freeform surfaces
  - Text
Scan Conversion in the Rendering Pipeline

- Scan conversion follows the geometric operations
  - Converts geometric primitives to screen primitives (pixels)
  - Scan conversion falls into 2 parts: setup calculations + rasterization

Scan Conversion: Steps

- Setup
  - Scan conversion is typically performed as an iteration
  - Setup calculates the parameters of the iteration, e.g. increments

- Determine covered pixels
  - Calculate the pixel coordinates of pixels belonging to the object
  - This is done either by testing candidate pixels or by enumerating covered pixels

- Determine pixel value
  - Given a pixel belonging to an object, calculate and assign the value of the pixel attributes

- We will discuss these steps for each primitive type
What is a Pixel? (1)

- Pixel means "Picture Element"
- A pixel is the smallest addressable unit on the screen
  - On a CRT the beam can be modulated at the granularity of a pixel
- The actual shape of a pixel depends on the device
  - CRT: Pixels are approximately round
  - LCD: Pixels are square

What is a Pixel? (2)

- Pixels are organized as rows and columns
- Screen coordinates describe pixel location
  - We will use a right-handed screen coordinate system
- Pixel addresses are integer values, denoting the location of the pixel center
  - A pixel covers a 1x1 area
  - Pixel corners lie on [x +/- 0.5, y +/- 0.5]
What is a Pixel? (3)

- There are other ways to define a pixel
  - For instance: lower-left corner is on integer coordinates
  - Then, pixel centers lie on half-integer coordinates

- Also, pixel coordinates may be defined differently
  - Left-handed coordinate systems

Scan Conversion of Points

- Points are described as a single coordinate in screen coordinates
- Set the pixel whose pixel center is closest to the point
  - Round the points coordinates to the next pixel coordinates
- Points have one a single value for each attribute
  - Constant color, depth etc. for each point
**Scan Conversion of Lines (1)**

- **Rasterization of a line** computes pixels on or near the ideal, infinitely thin, straight line:
  - Most pixels lie off the line
  - Raster lines are not straight
  - Lines are not infinitely thin

- **Requirements**
  - Centered around ideal line
  - For thin lines, one pixel thick
  - For slopes of -1 to +1, set exactly one pixel per column
  - Monotonous behavior

**Scan Conversion of Lines (2)**

- **Requirement:** Only one pixel per column (row)

- **X-major lines**
  - Slope between -1.0 and +1.0
  - Advances faster in X than Y

- **Y-major line**
  - Slope between +1.0 (through infinity) and -1.0
  - Advances faster in Y than X
Scan Conversion of Lines (3)

Simple Approach:
- Determine whether line is of X or Y major form
- For each X coordinate along the line compute the Y coordinate

\[ \Delta = x_E - x_S \quad ; \quad \Delta y = y_E - y_S \]

If \( \Delta x \geq \Delta y \)
then // X - major
\[ m = \frac{\Delta y}{\Delta x} \]
for \( (x = x_S ; x \leq x_E ; x + +) \)
\[ y = y_S + m(x - x_S) ; \]
setpixel(round(x), round(y)) ;
else // Y - major
\[ m = \frac{\Delta x}{\Delta y} \]
for \( (y = y_S ; y \leq y_E ; y + +) \)
\[ x = x_S + m(y - y_S) ; \]
setpixel(round(x), round(y)) ;

Scan Conversion of Lines (4)

Simple Approach requires ...
- floating point calculations
- one multiplication per pixel (\( \Delta y/\Delta x \) or \( \Delta x/\Delta y \) can be precomputed !)
- one float-to-int conversion

This is expensive and slow!

We would like to produce the same result and ...
- avoid multiplications by using iterative additions,
- use only integer calculations (Bresenham algorithm)
Scan Conversion of Lines (5)

- **Digital Differential Analyzer (DDA)**
  - Incremental computation of pixel coordinates
  - Relies on
    \[ y_{x+1} = m(x+1) + b = mx + b + m = y_x + m \]
  - 1 floating point addition and 2 float-to-int conversions per pixel
  - Next we will eliminate the floating point computations

\[
\Delta x = x_E - x_S \quad ; \quad \Delta y = y_E - y_S
\]

If \( \Delta x \geq \Delta y \)

then // X - major

\[
m = \frac{\Delta y}{\Delta x} ;
\]

for \( (x = x_S, \ y = y_S \ ; \ x \leq x_E ; \ x++) \)

\[
y = y + m ;
\]

setpixel(round(x), round(y)) ;

else // Y - major

\[
m = \frac{\Delta x}{\Delta y} ;
\]

for \( (x = x_S, \ y = y_S \ ; \ y \leq y_E ; \ y++) \)

\[
x = x + m ;
\]

setpixel(round(x), round(y)) ;

Scan Conversion of Lines (6)

- **Midpoint Algorithm (Bresenham algorithm)**
  - Basic idea: Incrementally compute an error term \( d \)
  - Chose next pixel \( H \) if the line is above \( M \) and otherwise pixel \( L \)
  - Works only for lines with slopes of 0 ... +1 (first octant)
Scan Conversion of Lines (7)

- Line can be represented in two forms:
  \[ \ell: \ ax + by + c = 0 \] (Implicit form)
  \[ \ell: \ y = mx + B \] (Slope - intercept form)

- Comparing coefficients:
  - Using \( m = \Delta y / \Delta x \)

- Distance of a point from the line:
  - \( F = 0 \) for points on the line
  - \( F > 0 \) for points below the line
  - \( F < 0 \) for points above the line

\[
F(x, y) = ax + by + c \\
= \Delta y \cdot x - \Delta x \cdot x + B \cdot \Delta x
\]

Scan Conversion of Lines (8)

- Midpoint criterion:
  - Determine position of \( M \) with respect to the line
    - Compute sign of error \( d \)
    - Determine pixel
      - If \( d < 0 \) choose H
      - If \( d > 0 \) choose L

- Next, compute the value of \( d \) for the next pixel \( x_{i+2} \)
\[
d_{i+1} = F(M) = F(x_M, y_M) \\
= F(x_p + 1, y_p + 1/2) \\
= a \cdot (x_p + 1) + b \cdot (y_p + 1/2) + c
\]
Next, compute the value of \( d \) for the next pixel \( x_{i+2} \)
- This depends on which pixel (L or H) was chosen
- If L was chosen ....................
  \[
  d_{i+2} = F(x_p + 2, y_p + 1/2) \\
  = a \cdot (x_p + 1) + b \cdot (y_p + 1/2) + c + a
  \]
- If H was chosen ....................
  \[
  d_{i+2} = F(x_p + 2, y_p + 3/2) \\
  = a \cdot (x_p + 2) + b \cdot (y_p + 3/2) + c + a + b
  \]

The new \( d \) is computed incrementally as:
\[
\begin{align*}
  d_{i+2} &= \begin{cases} 
  d_{i+1} + a & \text{if } y_{i+2} = y_L \\
  d_{i+1} + a + b & \text{if } y_{i+2} = y_R 
  \end{cases}
\end{align*}
\]

How is \( d \) initialized?
- \((x_S, y_S)\) is on the line
- Therefore: \( F(x_S, y_S) = 0 \).
- In practice we only care about the sign of \( d \)
- Therefore, we use \( d' = 2d \) to avoid the division by 2.

\[
\begin{align*}
  d_1 &= F(x_S + 1, y_S + 1/2) \\
  &= a \cdot (x_S + 1) + b \cdot (y_S + 1/2) + c \\
  &= a \cdot x_S + b \cdot y_S + c + (a + b/2) \\
  &= F(x_S, y_S) + a + b/2 \\
  &= a + b/2
\end{align*}
\]
**Scan Conversion of Lines (11)**

**Putting it all together:**

- **Initialization:**
  
  ```plaintext
dx = x_end - x_start;
dy = y_end - y_start;
d = 2*dy - dx;
incr_L = 2*dy;
incr_H = 2*(dy-dx);
  ```

- **Iterate over all columns:**
  
  ```plaintext
for (x = x_start ; y = y_start ;
x < x_end ; x++)
{
  WritePixel (x,y);
  if (d <= 0)
    d += incr_L ;
  else
    { d += incr_H ;
      y++ ;
    }
}
  ```

**Scan Conversion of Lines (12)**

**Endpoint Order**

- If all pixels along the line are set, adjacent line segments will touch first/last pixel multiple times

- Creates problems with some pixel algorithms, e.g. transparency

- Therefore, typically the first/last is not drawn
Scan Conversion of Triangles (1)

- Determine all pixels that belong to a triangle
  - Point sampling: Generate only pixels with center inside the triangle
  - Pixels exactly on the triangle border require special tie-breaker rule
  - This ensures that a pixel is not touched twice by adjacent triangles
  - Rasterization rules for triangles and lines are different

Scan Conversion of Triangles (2)

- Triangles are scan converted by finding the limits of covered spans
  - A span is a continuous, horizontal range of pixels
  - Starting and ending pixels are computed incrementally
  - Edge slopes are represented with floating-point or fixed-point numbers

- Pixels in between the edge pixels are filled in
Scan Conversion for Triangles (3)

- Computing the edge pixels:
  - Compute X-coordinates of edge in current scanline
    - Incrementally computed using edge slopes $dx/dy$
  - Adjust this coordinate to get the pixel inside the triangle
    - Left edge: nudge to the right --- Right edge: nudge to the left

\[
\begin{align*}
  x_S &= \left\lfloor x_L \right\rfloor \\
  x_E &= \left\lfloor x_R \right\rfloor
\end{align*}
\]

Scan Conversion of Triangles (4)

- Non-integer coordinates and slopes
  - The application or clipping may produce vertices that do not fall onto pixel centers
  - However, in screen coordinates all quantities are represented on a fixed grid
  - Slopes are therefore always rational numbers, i.e. ratio of numbers on that fixed grid
  - Numbers are represented using a fixed-point representation
    - Vertex positions and slopes
    - Number of integer bits determined by the number of grid positions
    - Number of fractional bits is determined by smallest slope
Scan Conversion of Triangles (5)

- Tie-breaker rule
  - Include pixels on a top edge
  - Include pixel on a triangle vertex, if the vertex is the right vertex of the edge and does not lie below the left vertex (top-right vertex)
  - Still does not catch all possible (pathological) cases

Scan Conversion of Triangles (6)

- Computing interior pixels
  - Iterate from starting pixel to ending pixel:
  - for (x=x_s; x<=x_e; x++) setpixel (x, y_i);
Scan Conversion of Triangles (7)

Triangle Terminology

- Top Vertex
- Trailing Edge
- Top Half
- Leading Edge
- Bottom Half
- Trailing Edge
- Bottom Vertex
- Middle Vertex
- Top Vertex
- Bottom Vertex

Scan Conversion of Triangles (8)

Triangle Types

- Left Triangle
- Right Triangle
- Top Triangle
- Bottom Triangle
Scan Conversion of Triangles (9)

Putting it all together (for left-triangle):

- Initialization:

```c
// Determine leading edge
// & midpoint of trailing edge
x_top = ... ; y_top = ... ;
x_bot = ... ; y_bot = ... ;
x_mid = ... ; y_mid = ... ;
slope_lead = ... ;
slope_trail1 = ... ;
slope_trail2 = ... ;
```

- Iterate (only top half):

```c
x_lead = x_trail = x_top ;
slope_trail = slope_trail1 ;
for (y=y_top, y < y_mid ; y++)
{  
  xs = ceil(x_lead) ;
  xe = floor(x_trail) ;
  setpixel (xs, y) ;
  for (x=xs ; x < xe ; x++)
  {  
    setpixel (x,y) ;
    x_lead += slope_lead ;
    x_trail += slope_trail ;
  }
}
```

Scan Conversion of Triangles (10)

- So far, we have answered the question: Which pixels are in the triangle ?
  - Direct computation of the rasterization

- Conversely, we could have also asked: Is this pixel inside the triangle ?
  - Point-in-triangle Test
Scan Conversion of Triangles (11)

A triangle is the intersection of 3 half-planes
- Each half-plane is limited by an oriented line
  \( ax + by + c = 0 \)

Same algorithm works for all convex polygons

Scan Conversion of Polygons
Scan Conversion of Polygons

- General polygons differ from triangles
  - Not always convex
  - May have holes
  - Can be self-intersecting
  - Generally not planar

- Approaches
  - Divide & Conquer: Triangulate the polygon before scan-conversion
    - Triangulation is a difficult problem also
  - Direct rasterization

Scan Conversion of Polygons: Overview

- Extension of the triangle scan conversion algorithm
  - Works for many classes of polygons, including concave, self-intersecting polygons as well as polygons with interior holes

- Determine spans of pixels that are inside the polygon
  - Similar to triangle rasterization, we use tie-breaker rules to avoid writing a pixel twice by adjacent polygons

- Calculate the spans' start and end points incrementally
  - Calculate the intersection of polygon edges with scan lines
Scan Conversion of Polygons: Example

Scan Conversion of Polygons: Terminology

- **Active edge (AE)**
  - An edge that intersects the current scanline

- **Span**
  - Continuous set along a scanline

- **Leading/trailing edge**
  - Edge defining the left/right end of a span
Scan Conversion of Polygons:
Basic Algorithm

- Intersect all edges with the current scanline
  - This will determine the set of active edges

- Sort all intersection by increasing x and mark as leading or trailing edge

- Compute starting and ending pixel coordinates
  - Round X up/down for leading/trailing edge

- Fill pixels for all spans defined by a pair of leading/trailing edges

Scan Conversion of Polygons:
Special Cases (1)

- Horizontal Edges
  - No (unique) intersection with the scanline
  - Ignore. Let adjacent, non-horizontal edges take care of pixels on horizontal edges. Use tie-breaker rules.
Scan Conversion of Polygons: Special Cases (2)

- Clipped Polygons
  - Clipping can generate half-open spans
  - Leftmost edge-scanline intersection may be a trailing edge and/or rightmost may be a leading edge
  - If leftmost pixel is outside (inside) the first intersection is a leading (trailing) edge
  - Then alternate between leading and trailing edge

Scan Conversion of Polygons: Implementation (1)

- Simple-minded computation of intersection between edges and scanlines can be inefficient (slow)
  - Often, only a few edges intersect the active scanline
  - Neighboring scanlines tend to be intersected by the same edges (edge coherence)

- Determination of active edges can be optimized
  - Incrementally compute the intersection with the next scanline from the intersection point with the current scanline (see textbook Fig. 3.26)
  - Maintain a global edge table (ET) containing all edges
  - Maintain an active edge table (AET) containing all edges intersecting the active scanline
  - Update the ET and AET for every new scanline
Scan Conversion of Polygons: Implementation (2)

- **Edge Table (ET)**
  - Bucket sorted list of all edges, with a bucket for each scanline
  - Edges are sorted by their minimum (maximum) Y-coordinate

- **Active Edge Table (AET)**
  - List of edges intersecting the current scanline
  - Sorted by increasing X-coordinate of the intersection
  - For each new scanline Y
    - Update X coordinate of intersection for active edges
    - Insert edges from the ET into the AET that become active, i.e. for which \( Y_{\text{min}} = Y \)
    - Remove edges from the AET that are no longer active, i.e. for which \( Y_{\text{max}} = Y \)
    - Resort AET
    - Compute starting and ending coordinates for spans defined by the active edges
    - Fill in pixel spans

Attribute Interpolation
Attribute Interpolation

- So far, we have only determined which pixels are covered by a primitive

- Pixel values are determined by primitive attributes
  - Attributes can be computed different ways, but most common is linear or bilinear interpolation based on values at the vertices
  - In the following, we will treat a generic attribute $A$
  - Color, Texture, Normal vector, Transparency, ...
  - Each component is interpolated individually
  - Can be applied to lines, triangles and polygons

Linear Interpolation (1)

- Attribute defined at the vertices
- $A(x,y)$ defines a plane in x-y-A space
Linear Interpolation (2)

Computing linear interpolation
- Attribute value is a linear function in x, y
- \( A(x,y) = ax + by + c \)
- \( a \) describes the change from \( x \) to \( x+1 \), a.k.a. x gradient or x slope
- \( b \) describes the change from \( y \) to \( y+1 \), a.k.a. y gradient or y slope
- \( c \) is the value of \( A \) at the origin

Computing the parameters
- System of linear equations:
- Solve using preferred method
- E.g. Cramer's rule

\[
\begin{pmatrix}
    x_1 & y_1 & 1 \\
    x_2 & y_2 & 1 \\
    x_3 & y_3 & 1
\end{pmatrix}
\begin{pmatrix}
    a \\
    b \\
    c
\end{pmatrix}
= \begin{pmatrix}
    A_1 \\
    A_2 \\
    A_3
\end{pmatrix}
\]

Linear Interpolation (3)

\[
a = \frac{A_1(y_2 - y_3) - A_2(y_1 - y_3) + A_3(y_1 - y_2)}{x_1(y_2 - y_3) - x_2(y_1 - y_3) + x_3(y_1 - y_2)}
\]

\[
b = \frac{-A_1(x_2 - x_1) + A_2(x_1 - x_3) - A_3(x_1 - x_2)}{x_1(y_2 - y_3) - x_2(y_1 - y_3) + x_3(y_1 - y_2)}
\]

\[
c = \frac{x_1(A_3y_2 - A_2y_3) - x_2(A_3y_1 - A_1y_3) + x_3(A_2y_1 - A_1y_2)}{x_1(y_2 - y_3) - x_2(y_1 - y_3) + x_3(y_1 - y_2)}
\]
**Bilinear Interpolation (1)**

- Attribute defined at vertices
- \( A(x,y) \) defines piecewise linear in x-y-A space
- Generally, not rotation-invariant !!

![Bilinear Interpolation Diagram](image)

**Bilinear Interpolation (2)**

- Interpolate attribute along starting and trailing edge
  - Use algorithms developed earlier for scan conversion
  - Use constant \( \frac{dA}{dy} \) for each edge (a.k.a. edge slope)
  - Adjust attribute to pixel centers

- Interpolate attribute along scanline between starting and trailing edge pixels
  - Compute \( \frac{dA}{dx} \) for each span (a.k.a. x-gradient or x-slope)

- Integrates nicely with scanline rasterization algorithms
Linear vs. Bilinear Interpolation

- Linear interpolation only defined over triangles
- 4+ vertices need higher-order polynomial interpolation
- Bilinear interpolation provides a piece-wise linear approximation to this polynomial
- Bilinear interpolation fits with scanline rasterization
- Bilinear interpolation is not rotation-invariant
- Rotation invariance can be ensured by triangulation

Perspective Projection (1)

- Perspective projection plays havoc with linearly interpolated parameter values
  - Non-linear mapping of depth values also affects attributes
  - Attribute ranges is compressed with increasing distance
Perspective Projection (2)

- This effect has been overlooked a long time because it was not very apparent with Gouraud / Phong shading.
- Texture mapping makes these errors very noticeable.
  - Linear interpolation after perspective projection generates the wrong attribute values.

Perspective Projection (3)

- Attributes are defined at the vertices in object coordinates (eye or world coordinates).
- Attribute interpolation occurs in screen coordinates, i.e. after the perspective division.
  - The attribute values at the vertices are correct by construction.
  - Linear interpolation of attributes produces wrong values.

Problem:
Given a point in screen coordinates, what is the attribute value corresponding to this point?
For this we need to find out which attribute value belongs to this point in object space.
**Perspective Projection (4)**

- Points in object space:
  \[ P_o = (x_o, y_o, z_o, 1)^T \]

- Points in clip space:
  \[ P_c = (x_c, y_c, z_c, w_c)^T \]

- Points in screen space:
  \[ P_s = (x_s, y_s, z_s, 1)^T \]

- Determine perspective transformation and division affect a point’s representation.

**Perspective Projection (5)**

- Going from object space to screen space:
  \[
  P_o = (x_o, y_o, z_o, 1)^T \xrightarrow{M} P_c = (x_c, y_c, z_c, w_c)^T \xrightarrow{\frac{1}{w}} P_s = \left(\frac{x_c}{w_c}, \frac{y_c}{w_c}, \frac{z_c}{w_c}\right)^T
  \]

- Going back from screen space to object space:
  \[
  P_o' = (x_o'w_o, y_o'w_o, z_o'w_o, w_o)^T \xleftarrow{M^{-1}} P_c = (x_s, y_s, z_s, 1)^T \xleftarrow{} P_s = (x_s, y_s, z_s)^T
  \]

- Note: \( M \) and \( M^{-1} \) are perspective matrices, i.e. they both change the \( w \) component
- Tilde notation indicates that \( w \neq 1 \).
Perspective Projection (6)

- To determine the objects space point \( P_0 \) corresponding to the point \( P_s \) in screen space, \( P_s \) must be transformed by the inverse of \( M \).
  - A full matrix-point multiplication for every pixel is very expensive
  - There is a better way ...

\[
\vec{P}_o = M^{-1} \cdot P_s = M^{-1} \cdot \frac{P_c}{w_c} = M^{-1} \cdot \frac{M \cdot P_o}{w_c} = \frac{P_o}{w_c}
\]

\[
\vec{P}_o = (x_o w_o, y_o w_o, z_o w_o, w_o)^T = \left( \frac{x_o}{w_c}, \frac{y_o}{w_c}, \frac{z_o}{w_c}, \frac{1}{w_c} \right)^T
\]

\[
\Rightarrow w_o = 1 / w_c
\]

Perspective Projection (7)

- Now, let's look what happens to the attributes:

- Object coordinates are mapped linearly to attributes

\[
A_o = ax_o + by_o + cz_o + d = (a, b, c, d)^T \cdot (x_o, y_o, z_o, 1)^T = N \cdot P_o
\]

- Attributes are interpolated in screen space:

\[
A_s = A_1 + t \cdot (A_2 - A_1)
\]

- Similarly, points are generated by linear interpolation in screen space:

\[
P_s = P_1 + t \cdot (P_2 - P_1)
\]
Perspective Projection (8)

- Object-space attributes and screen-space points are related:
  \[ A_O = N \cdot P_O = N \cdot \frac{\tilde{P}_O}{w_O} = \frac{1}{w_O} \cdot N \cdot \tilde{P}_O \]

- We define screen-space attributes as
  - Screen-space attributes are proportional to screen points \( P_s \), because \( M \) is affine

- Therefore:
  \[ A_O = \frac{1}{w_O} A_S \]

Perspective Projection (9)

- Recall
  \[ P_o = \frac{\tilde{P}_o}{w_o} \quad w_o = 1 / w_c \]

- For a given point in screen space, the attribute values are therefore:
  \[ A_s = N \cdot \tilde{P}_o = \frac{N \cdot P_o}{w_c} = \frac{A_o}{w_c} \]
  \[ A_o = \frac{A_s}{w_o} = \frac{A_s}{1 / w_c} \]

- What does that mean?
  - Initialize \( A_s \) to \( A_o / w_c \) at the vertices
  - Interpolate \( A_s \) and \( 1/w_c \), then divide by \( 1/w_c \) at every pixel
Perspective Projection (10)

\[ A_o = A_s / 1/w_c \]

- To obtain the proper attribute value, the value obtained by linear interpolation in screen space must be divided by the value of the interpolate 1/w-component.
- The 1/w-component itself, is also obtained by linear interpolation in screen space

\[ A_o = \frac{A_s}{1/w_c} = \frac{ax_s + by_s + c}{dx_s + ey_s + f} \]

- Numerator and denominator are linear function in screen X and Y
- \( A_o \) is rational function
- The computation is a hyperbolic interpolation

Perspective Projection (11)

- Summary
  - Both attributes and 1/W component are interpolated separately
  - Proper interpolation of parameters requires division at every pixel of interpolated attribute value and interpolated 1/W
  - This a very costly operation that can be avoided for shading but must be performed for texture mapping
  - Alternative:
    Subdivision of the surface to introduce more points over the surface that have the correct attribute value. Then the maximum error is reduced and linear interpolation becomes acceptable.
Summary

- Scan Conversion determines which pixels are covered by a primitive
- Primitive scan conversion must determine integer pixel coordinates
- Scan Conversion is usually done incrementally, exploiting coherence between pixels, scanlines, edges

- Various algorithms for specific primitive types:
  - Point, lines, triangles, polygons

- Attribute interpolation assigns pixel values
  - Typically also done incrementally using (bi)linear expressions
  - Attention: perspective projection requires special treatment

Short Overview of 2nd Assignment

- Apply your knowledge about viewing, clipping and scan-conversion
  - Read polygons descriptions from file
  - Display them unclipped as wireframe
  - Display them clipped against a clip volume as filled polys
  - Scan-convert them to a virtual raster screen and display this raster in front of the clip volume
  - Provide view manipulation
**Homework**

- Review primitive scan conversion in textbook
  - Also look at circle and ellipsis scan conversion

- Prepare for fragment processing discussion next week
  - Read OpenGL Programming Manual re Anti-aliasing and Fragment Processing (chapters 6, 9 and 10)
  - Foley et al., chapters 3.8, 14.10, 15.4, 19.3

- Study hidden-surface algorithms
  - Z-buffering, Depth-sorting algorithm, Scanline algorithms
  - Foley et al., chapters 15.2, 15.4, 15.5.1, 15.6

**Next Week ...**

- Fragment Processing

- Hidden-Surface Removal