Computer Graphics - Week 7

Bengt-Olaf Schneider
IBM T.J. Watson Research Center

Questions about Last Week?
Comments about the Assignment

► Specific comments
  - The clip volume does not need to be closed
  - Rotate the polygon around the origin, not the center of the clip volume or the center of the polygon
  - Polygons are 2-sided, i.e. they are visible from both sides
  - Rasterization is done using integer arithmetic
  - If you encounter coincident edges, they should not appear

► What are star-shaped polygons?
  - Star-shaped polygons have one interior star-point from where all edges are visible
  - In the assignment all polygons are star-shaped with the first vertex a star point.
  - This is helpful for triangulation.

's' command
  - Displays rasterized clipped polygon on the screen rectangle
  - Rasterized polygon interpolate the colors based on the colors specified for the input polygon after clipping
  - Make sure shading results for interior pixels do not depend on clipping or rotation

'i' command
  - Displays outline of the clipped polygon using OpenGL lines
  - Don't rasterize the lines onto the screen rectangle
Overview of Week 7

- **Fragment Processing and Pixel Manipulation Methods**
  - Texture Mapping
  - Alpha Blending
  - Z-Buffering

- **Hidden-Surface Removal Algorithms**
  - Z-Buffering
  - Scanline algorithm
  - Depth-sorting algorithm

---

Fragment Processing (a.k.a. pixel processing) in the Rendering Pipeline

- **Fragment processing follows the scan conversion**
  - Processes the pixels generated by the scan conversion process
  - Forms the interface between scan conversion and the pixel buffers
Fragment Processing Concepts

- A fragment is pixel data generated during scan conversion
  - Pixel coordinates
  - Associated attributes, e.g. color, depth, texture, etc.

- Fragment Processing manipulates the fragment data
  - Look-up operations, e.g. texturing
  - Modification of pixel value based on global parameters, e.g. fogging
  - Test of fragment data against frame buffer data, depth test
  - Pixel arithmetic, e.g. alpha blending
  - Anti-aliasing and dithering

- We will call pixel the data stored in the frame buffer
  - To be distinguished from fragment data.

Fragment Processing Pipeline (1)

- OpenGL fragment processing pipeline
  - Texturing, Fogging and Anti-aliasing
  - Pixel Tests
  - Frame Buffer Operations
**Fragment Processing Pipeline (2)**

- **Texturing** (... more in a few minutes)
  - Apply image information to a polygon
  - "Paste an image onto the polygon"

- **Fogging**
  - See lecture on lighting and shading models

- **Anti-aliasing**
  - Eliminate jaggies and broken-up polygons by accounting for partial pixel coverage by primitives

**Fragment Processing Pipeline (3)**

- **Pixel Tests**
  - Test various conditions about the fragment and the current frame buffer content
  - Determines whether the fragment is processed any further or discarded

- **Frame Buffer Operations**
  - Read - Modify - Write processing of frame buffer content
  - Combines fragment with current frame buffer contents
Fragment Processing Pipeline (4)

- We will not go into all details of these operations
  - See OpenGL programming guide for more details

- We will discuss
  - Texture Mapping
  - Blending
  - Z-Buffer

Texture Mapping
Texture Mapping: Example

Texture mapping:
- Pasting of an image to the interior of an object
- If necessary, repeat the image to fill the entire interior
- Texture coordinates defined across the object, define where the image pixels appear in the object.

Texture Mapping Concepts

- **Texture Map** is a 1/2/3-dimensional array of color values
  - 2D texture maps are images
  - 3D texture maps are volume data

- **Texels** are the individual pixels in the texture map
  - Texels can be represented in a variety of formats, e.g. 1/4/8 bit color index, 16/24/32 bit true color

- **Texture Coordinates** are indices (addresses) into the texture map
  - Interpolated across primitives
  - Often denoted \((u,v)\) coordinates
**Texture Mapping Principle**

- Texture coordinates are computed/assigned at vertices and interpolated across the triangle
- The texture coordinates are used to look up the texel
- Texel value is assigned to associated pixel

**Texture Mapping: Sampling**

- **Point Sampling**
  - If only the computed texture coordinates at the pixel are used
  - Will yield exactly one texel for every pixel
    (but generally not vice versa)

- **Area Sampling**
  - Determine all texels affecting the pixel
  - For instance map the 4 corners of the pixel and interpolate within the texture
Texture Mapping: Forward vs Inverse Mapping

> Texture mapping can be defined in 2 directions

> **Forward mapping** maps a texel into screen space
  * Given a texture coordinate \((u,v)\), determine the corresponding pixel coordinate \((x,y)\)
  * May leave holes in the pixel image, i.e. there may be pixels that no texel was mapped to
  * Often used in image processing

> **Inverse mapping** maps pixels into texture space
  * Given a pixel coordinate \((x,y)\), determine the corresponding texture coordinate \((u,v)\)
  * May miss texels, i.e. there may be texels that no pixel maps to
  * Frequently used during texture mapping

---

Texture Filtering (1)

> Both directions of mapping may create problems, if there is no 1:1 mapping between pixels and texels
  * Will create artifacts
  * Texture breaks up if texels are missed (Minification)
  * Texture appears blocky if several pixels map into the same texel (Magnification)
  * Mixed cases are possible if magnified along one axis and minified along the other axis
Texture Filtering (2)

- The fundamental problem is a sampling problem
  - For *minification* the texture is undersampled
  - The spatial sampling frequency is smaller than the spatial frequency of the texture map (less than one sample per texel)
  - For *magnification* the texture is oversampled
  - The spatial sampling frequency is higher than the spatial frequency of the texture map (more than one sample per texel)

- Proper treatment of such problem requires filtering
  - Without going into too many details ...

Texture Filtering (3)

- Magnification
  - Texture map can be considered sampling of a continuous function
  - Texture mapping resamples (reconstructs) that function
  - Neighboring texels must be taken into account

- Consider a neighborhood around the sample point
- Compute a weighted average
Texture Filtering (4)

Minification
- All texels contributing to the pixel must be taken into account
- Integrate over the area a pixel covers in the texture map
- This will take into account all texels contributing to the pixel
- Fairly expensive to compute
- There are several approximations to deal with this problem:

Mip-maps
- Down-sampled copies of the texture

Summed-area tables
- Precomputed integrals over axis-aligned rectangular areas in the texture

Texture Filtering (5): Mip-Mapping (i)

Hierarchy of texture images
- Original texture (base texture) is down-sampled repeatedly
- Average 2x2 texel blocks into a texel of the next higher texture level

Select mip-map level based on minification
- Scale factor \( \rho \) (to be defined)
- Level-of-Detail (LOD)
  \[ \lambda = \log_2(\rho) \]
Texture Filtering (5): Mip-Mapping (ii)

- Selection of mip-map level
  - Ideally, unit step in pixel space result in a unit step in texture space
  - Look at derivatives of texture coordinates in pixel space
  - Use biggest step size to compute scale factor $\rho$

$$
\rho = \max \left( \sqrt{\left( \frac{\partial u}{\partial x} \right)^2 + \left( \frac{\partial v}{\partial x} \right)^2}, \sqrt{\left( \frac{\partial u}{\partial y} \right)^2 + \left( \frac{\partial v}{\partial y} \right)^2} \right)
$$

- This expression is often simplified with slight loss of quality:

$$
\rho = \max \left( \frac{|\partial u|}{\partial x}, \frac{|\partial u|}{\partial y}, \frac{|\partial v|}{\partial x}, \frac{|\partial v|}{\partial y} \right)
$$

Texture Mapping (6)

- Point Sampling
  - Look up the nearest texel to the texture coordinate $(u,v)$
  - Only one texel affects the pixel at $(x,y)$
  - 1 Memory Access

- Bilinear Filtering
  - Use a 2x2 texel neighborhood around $(u,v)$ to compute the final texture value
  - 4 Memory Accesses
  - 8 Multiplications
  - 5 Additions

$$
\tau = T\left(\left\lfloor u \right\rfloor, \left\lfloor v \right\rfloor\right)
$$
Texture Mapping (7)

- **Linear Filtering**
  - Interpolated linearly between two mip-map levels
  - 2 Memory Accesses
  - 2 Multiplications
  - 1 Addition

- **Trilinear Filtering**
  - Bilinear interpolation within two adjacent mip-map levels
  - Linear interpolation between those two values
  - 8 Memory Accesses
  - 18 Multiplications
  - 16 Additions

\[
\tau = \alpha \cdot T_{\lambda}(\lfloor u \rfloor, \lfloor v \rfloor) + (1 - \alpha) \cdot T_{\lambda+1}(\lfloor u/2 \rfloor, \lfloor v/2 \rfloor)
\]

\[
s_{\lambda} = \lfloor u \rfloor; \quad t_{\lambda} = \lfloor v \rfloor;
\]

\[
s_{\lambda+1} = \lfloor u/2 \rfloor; \quad t_{\lambda+1} = \lfloor v/2 \rfloor
\]

Texture Mapping (8)

- **Different filtering methods have different complexity**
  - Trilinear interpolation is most expensive, point sampling cheapest
  - Quality of filtering increases with required computation
Texture Filtering (4)

- **Minification**
  - All texels contributing to the pixel must be taken into account
  - Integrate over the area a pixel covers in the texture map
  - This will take into account all texels contributing to the pixel
  - Fairly expensive to compute
  - There are several approximations to deal with this problem:

- **Mip-maps**
  - Down-sampled copies of the texture

- **Summed-area tables**
  - Precomputed integrals over axis-aligned rectangular areas in the texture

Texture Filtering (9): Summed Area Table

- **Precomputed table that stores the sum of all texels between the origin and a given texture coordinate**
  - Precomputation is costly
  - Supports rectangular regions instead of square regions in mip-mapping

- **For a given rectangular, axis-aligned area, the sum of all texels inside the rectangle is:**

\[
S(u_1, v_1, u_2, v_2) = 
S_{22} - S_{12} - S_{21} + S_{11}
\]
Assigning Texture Coordinates (1)

- **Basic problem**
  - Square texture images are applied to arbitrarily-shaped objects
  - Find a good way to map to (i.e. wrap around) texture to object

- **Application assigns texture coordinates at the vertices**
  - Complete freedom over how texture is applied to an object
  - Texture can be rotated, shifted and scaled
  - To avoid distortion of the texture, ensure that object's aspect ration matches the aspect ratio of the selected texture range

Assigning Texture Coordinates (2)

- **Typically, texture coordinates fall within the range [0,1].**
- **What happens if a texture coordinate falls outside of that range?**
  - Clamp texture coordinate: \( u' = \max (\min (u, 1.0), 0.0) \)
  - Repeat the texture: \( u' = u \mod 1.0 \)
Assigning Texture Coordinates (3)

- Textures can be applied "automatically"
  - Compute texture coordinates based on the distance of object point from a plane $ax+by+cz+d$
  - This allows to project the texture onto the object similar to a slide projector
  - For instance: $u = x$ and $v = z$

- For "cylindrical" objects, the texture can be wrapped around the object, by using the angle around an axis to address the texture
- For instance, for an object centered around the z-axis (use sign-aware atan function $!$):
  $$u = \frac{\arctan(x / y)}{2\pi}$$ and $v = z$

OpenGL: Texture Mapping (1)

- **glTexImage[D]()**
  - Specifies the texture image in various formats
  - Takes mipmap level, width, height, image data and various format parameters
  - Images must have width and height being powers of 2

- **glTexParameter[]()**
  - Specifies filtering methods for magnification and minification
    - Can choose from point sampling + linear, bilinear and trilinear filters
  - Repeat vs. Clamping of texture coordinates
    - Can be set differently for $u$ and $v$ coordinates

- **glTexCoord[]()**
  - Specifies a texture coordinate, similar to glVertex[]() or glColor[]()
OpenGL: Texture Mapping (2)

- Texture Objects allow to define and use multiple textures efficiently
  - A texture object stores the texture image and parameters defined for that texture, e.g. repeat, border and filtering modes

- glGenTextures()
  - Generates texture names (integer numbers)

- glBindTexture()
  - Creates texture object of specified type with given name (number)
  - Makes the bound texture the active texture, i.e. the one used for texture mapping, until a new texture is bound

OpenGL: Texture Mapping (3)

- Automatic texture generation
  - Uses a plane specified by four parameters
  - Plane equation is either evaluate for object coordinates (GL_OBJECT_LINEAR) or eye coordinates (GL_EYE_LINEAR)
  - Result of that evaluation determines texture coordinate

  - OpenGL also supports generation of texture coordinates for environment mapping, i.e. reflection on an ideal sphere.
Alpha-Blending

Alpha-Blending: Basics

- Basic extension of RGB color model
  - A fourth component is added
    - Commonly referred to as Alpha or A: RGBA
    - So far, the pixel color was fully replaced by the fragment color
    - Alpha is used to blend a fragment's color with the stored pixel color
    - This allows to create a mix of the pixel and the fragment color
    - A=1 means fully opaque, A=0 means fully transparent

- Alpha-blending is used for various purposes
  - Transparency
  - Anti-aliasing
  - Digital Compositing
Alpha-Blending: Transparency

- Both fragment and frame buffer pixel may have an associated alpha value
  - There are numerous possibilities to combine fragment and pixel color, taking into account the 2 alpha values (see e.g. OpenGL programming manual)

- One of the most useful applications of alpha blending is to model transparent objects:
  \[ C_P = \alpha \cdot C_F + (1 - \alpha) \cdot C_P \]

- When rendering scenes with transparent and opaque objects: Render all opaque objects first, then render transparent objects without writing the z-buffer

Double-Buffering
**Double-Buffering**

- If all rendering occurs into the same buffer that is used for screen refresh, the image construction process is apparent
  - No illusion of a standing image
  - Flickering as image is erased and updated
  - Memory contention between screen refresh and image generation

- **Double-buffering provides two buffers**
  - Front-buffer used for screen refresh, contains previous frame
  - Back-buffer used to construct the new image, i.e. the current frame
  - After the new image is finished, front and back buffers are swapped
  - To reduce flicker, buffer swap is synchronized with vertical retrace

**Hidden Surface Removal**
**Hidden Surface Removal**

- Determine which objects are visible from a given viewpoint, i.e. which objects are hiding other objects

- This is a complex problem of at least $O(n^2)$ complexity (test every object against every other object)
  - Complexity increases if there is no clear A-hides-B relationship between objects

- We will look at different hidden-surface removal algorithms (a.k.a. visibility algorithms)
  - Z-Buffer: Image space HSR algorithm
  - Scan-line Algorithms: Image space HSR algorithm
  - Depth-sorting Algorithm: Object space HSR algorithm

---

**Z-Buffering: Basic Algorithm**

- Simple algorithm, that trades computational simplicity for memory requirements
  - Allocate for every pixel a depth value
  - The depth value stores the z-value of the front-most (visible) object at that pixel
  - For new fragment, compare fragment's z-value with pixel's z-value
  - If fragment is closer to the viewer, replace pixel z and color

```plaintext
// Clear z-buffer
FOR (all pixels px)
    zb[px.x][px.y] = infinity;

// Scan conversion w/ z-buffer
FOR (each polygon p)
    FOR (each fragment f in p)
        { x = f.x ;   y = f.y ;
          IF (f.z < zbuffer[x][y])
            { zb[x][y] = f.z ;
              fb[x][y] = f.color ;
            }
        }
```
Z-Buffering: Properties

- Requires significant amounts of memory
  - $W \times H \times \text{nbytes}$, e.g. 1280 $\times$ 1024 $\times$ 32 bit = 5 MBytes
  - However, memory becomes cheaper rapidly

- Simple to implement
  - Many hardware and software implementation
  - Fast execution

- Universal
  - Can be used with any primitive type
  - For instance, polygons, quadrics, splines, depthmaps

- Image space algorithm
  - No unnecessary computations
  - Subject to aliasing

Z-Buffering: Artifacts

- Z-Buffer errors
  - Colinear edges and coplanar faces may generate slightly different depth values if not supported by the same vertices
  - Frequent changes in visibility creates typical z-buffer errors

- Aliasing
  - Only 1 object can be visible in each pixel
  - No blending amongst several objects sharing a pixel

- Depth Compression
  - Perspective projection distributes depth values non-uniformly
  - Depth values are spaced more closely near the eye, i.e. better resolution in the near field
  - Two different, distant points may map to the same z-value
Other Visibility Algorithms

- Scanline Algorithm
- Depth-sorting Algorithm

- We will look at some more HSR algorithms when we talk about spatial data structures
  - BSP trees
  - Octrees

Scanline Algorithm

- Last week we discussed a scanline algorithm to scan and convert polygons

- We will extend this algorithm
  - Several polygons per scanline
  - Resolve visibility between polygons sharing a scanline
Scanline Algorithm for Scan Conversion of Polygons

- **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

Scanline Algorithm Extension to Multiple Polygons (1)

- In addition to the pixels covered by the individual polygons, the visible polygons must be determined
  - If polygons do not penetrate, visibility changes only at edges
  - For example, scanline (c)
Scanline Algorithm
Extension to Multiple Polygons (2)

- Edge table ET is refined:
  - Bucket sort edges by $Y_{\text{MIN}}$
  - Within each bucket, sort edges by slope
  - For each edge store $X(Y_{\text{MIN}})$, $Y_{\text{MAX}}$, $dX/dY$, polygon id

- Active Edge Table AET remains:
  - Edges are sorted by X of intersections with current scanline

Scanline Algorithm
Extension to Multiple Polygons (3)

- In addition to ET and AET, we also maintain a polygon table PT
  - Geometric information, e.g. the plane equation
  - Attribute information
  - In/Out flag, initialized at leftmost pixel

  - Geometric and attribute data is read-only during scan conversion
  - Only the In/Out flag changes during scan conversion
**Scanline Algorithm**

**Extension to Multiple Polygons (4)**

- **Basic Algorithm**
  - Once the scanline enters a polygon, the respective In/Out flag is set.
  - The algorithm keeps track of the number of set flags, e.g. by maintaining a list of active polygons (APT).
  - If at least one flag is set when the scanline enters a polygon, visibility of the new span is evaluated.
  - Otherwise, the new span is visible.

- **Visibility Determination**
  - Determine starting point of new span:
    - \( x \): edge-scanline intersection, \( y \): current scanline.
  - Evaluate plane equation for all active polygons (In/Out flag).
  - The polygon with the closest \( z \) value is visible in the current span.

**Scanline Algorithm**

**Extension to Multiple Polygons (5)**

- **Example**
  - Scanline a:
    - \( \text{AET} = \{AC, AB\} \)
  - Scanline b:
    - \( \text{AET} = \{AC, AB, DF, DE\} \)
  - Scanline c:
    - \( \text{AET} = \{AC, DF, AB, DE\} \)
    - Compute visibility when entering right triangle (both triangle active).
  - Scanline d:
    - \( \text{AET} = \{AC, FE, BC, DE\} \)
    - Compute visibility when entering right triangle (both triangles active).
  - Scanline e:
    - \( \text{AET} = \{AC, BC, FE, DE\} \)
Scanline Algorithm: Special Cases

- **Background color**
  - Pixels without any polygons need to be set, too
  - Initialize the frame buffer before scan conversion
  - Or place a screen-sized rectangle behind all objects

- **Penetrating polygons**
  - If objects penetrate, visibility changes not only at edges
  - Either split objects to avoid piercing
  - Or calculate a "false edge" where visibility may change

Scanline Algorithm
Combining with Z-Buffer

- **Keeping track of visibility changes by monitoring active edges and polygons can be avoided**
  - Allocated a z-buffer for one scanline
  - For all active polygons generate pixel color and pixel depth using the standard scanline scan-conversion algorithm
  - Resolve visibility using z-buffer algorithm

- **Advantage**
  - Only small z-buffer must be allocated
  - Allows implementation for very high screen resolution

- **Drawback**
  - Still requires sorting of polygons into edge tables
**Depth-Sorting Algorithm:**

**Painter's Algorithm (1)**

- **Painter’s Algorithm**
  - Construct the image back-to-front
  - Objects closer to the viewer overwrite more distant objects
  - No depth comparison required during the scan-conversion stage

- Assumes that objects can be sorted (no overlaps or intersections)
- Special case: 2 1/2 D Rendering
  - Objects are thought of as belonging to layers with constant z (or priority)
  - Back-to-Front rendering is no simple

**Depth-Sorting Algorithm:**

**Painter's Algorithm (2a)**

- **Painter's Algorithm: Example**
Depth-Sorting Algorithm: Painter's Algorithm (2b)

- Painter's Algorithm: Example

Depth-Sorting Algorithm: Painter's Algorithm (2c)

- Painter's Algorithm: Example
Depth-Sorting Algorithm: Painter's Algorithm (2d)

- Painter's Algorithm: Example

Depth-Sorting Algorithm: Painter's Algorithm (2e)

- Painter's Algorithm: Example
Depth-Sorting Algorithm: Painter’s Algorithm (2f)

- Painter’s Algorithm: Example

Depth Sorting Algorithm (1)

- How can we ensure that objects are sorted in depth?
- What happens if there is no z-ordering?

Algorithm by Newell, Newell and Sancha

- 1. Sort polygons by farthest z coordinate
- 2. Resolve ambiguities in depth sorting
- 3. Render polygons back to front

- Without step 2, the algorithm defaults to the painter’s algorithm
- We will now look at various criteria to implement step 2
- The ambiguity is resolved as soon as one of the criteria is met
Depth Sorting Algorithm (2)

- **Criterion 1:** Overlapping extents
  - Do polygons overlap in X
  - Do polygons overlap in Y

Depth Sorting Algorithm (3)

- **Criterion 2:** Separating plane
  - Is one polygon entirely on one side of the other polygon's plane (here: B)
  - Draw that polygon first if the eye is on the same side, otherwise draw the separating polygon first (here: A then B)
Depth-Sorting Algorithm (4)

Criterion 3: Overlapping projections

- Do the projections of the polygons overlap?
- If no overlap, the order of drawing is not important.

Depth-Sorting Algorithm (5)

What if these criteria do not resolve the visibility?

- Objects must be split to break the cyclical occlusion relationship
- Split occurs by the clipping one polygon against the plane of the other polygon
- In order to avoid infinite loops (right example), polygons are marked as tested.
- If a marked polygon is encountered, it is split.
Visibility Algorithms: Sorting

- By now it should be clear that visibility determination is a sorting process
  - One of the seminal papers in Computer Graphics classifies different hidden surface algorithms by when they sort objects

- **Z-Buffer: Sorts in image space**
  - By Z in every pixel

- **Scanline algorithms: Sorts in image space**
  - First Y, then X, then Z

- **Depth-Sort algorithm: Sorts in object space**
  - First Z, then (if necessary) X and Y

Summary

- **Fragment Processing**
  - Texture Mapping
  - Alpha Blending
  - Z-Buffer

- **Visibility and Hidden-Surface Removal**
  - Z-Buffer
  - Scanline algorithm
  - Depth-sorting (painter’s algorithm)
Homework

- Read Foley et al. on Anti-aliasing
  - Chapter 3.17
  - Chapter 14.10 for more detailed discussion

- Familiarize yourself with VRML 2.0
  - Specification at www.vrml.org/Specifications/VRML97
  - Read chapters 4+5, skim over 6

Next Week ...

- Anti-aliasing
  - Gentle introduction to sampling theory
  - Area sampling
  - Oversampling
  - Use of alpha-channel for anti-aliasing

- VRML
  - Introduction to scene graph concepts
  - Attributes
  - Most important node types