Chapter 4 - 3D Camera & Optimizations, Rasterization

- Classical Viewing Taxonomy
- 3D Camera Model
- Optimizations for the Camera
- How to Deal with Occlusion
- Rasterization
  - Clipping
  - Drawing lines
  - Filling areas

Partially based on material from:
6th ed, Addison-Wesley 2012
Classical Views of 3D Scenes

- As used in arts, architecture, and engineering
  - Traditional terminology has emerged
  - Varying support by 3D graphics SW and HW

- Assumptions:
  - Objects constructed from flat faces (polygons)
  - Projection surface is a flat plane
    - Nonplanar projections also exist in special cases

- General situation:
  - Scene consisting of 3D objects
  - Viewer with defined position and projection surface
    - Projectors (Projektionsstrahlen) are lines going from objects to the projection surface

- Main classification:
  - Parallel projectors or converging projectors
Taxonomy of Views

- planar geometric projections
  - parallel
    - multiview
    - orthographic
  - axonometric
  - oblique
    - isometric
    - dimetric
    - trimetric
  - perspective
    - 1 point
    - 2 point
    - 3 point

Angel 2012
Orthographic Projection

• Projectors are orthogonal to the projection plane

• In the “pure” case, projection plane is parallel to a coordinate plane
  – top/front/side view
  – Often used as a multi-view combination
  – together with overview (e.g. isometric view)

• Advantage:
  – No distortions
  – Can be used for measurements

http://www.frog.com/tutorials/panzer_studio/teapot_monster_part1.jpg
Axonometric Projections

- Using orthographic projection, but with arbitrary placement of projection plane
- Classification of special cases:
  - Look at a corner of a projected cube
  - How many angles are identical?
  - None: trimetric
  - Two: dimetric
  - Three: isometric
- Advantage:
  - Preserves lines
  - Somehow realistic
- Disadvantage:
  - Angles not preserved
Optical Illusions in Isometric Projections

Oblique Projection (*Schiefe Parallelprojektion*)

- Projectors are not orthogonal to projection plane
  - Usually projection plane parallel to one coordinate plane
- Traditional subclasses:
  - *Cavalier perspective*
    - Constant angle (30°/45°) between direction of projectors (*dop*) and projection plane
    - No foreshortening
  - *Cabinet perspective*
    - Constant angle (30°/45°/63.4°) between dop and projection plane
    - *Foreshortening* (*Verkürzung*) (of depth) by factor 0.5
Perspective Projection (*Perspektivische Projektion*)

- Projectors converge at *center of projection (cop)*
- Parallel lines (not parallel to projection plane) appear to converge in a *vanishing point (Fluchtpunkt)*

**Advantage:**
- very realistic

**Disadvantage:**
- non-uniform foreshortening
- only few angles preserved
Number of Vanishing Points in Perspective Projection

One point

Two points

Three points

http://mathworld.wolfram.com/Perspective.html
How to Realize Projection in Three.js?

• Parallel / Orthographic projections:

  − var camera = new THREE.OrthographicCamera(w/-2, w/2, h/2, h/-2, 1, 1000);
  − scene.add(camera);

• Perspective projections:

  − var camera = new THREE.PerspectiveCamera(45, w/h, 1, 1000);
  − scene.add(camera);
Chapter 4 - 3D Camera & Optimizations, Rasterization

- Classical Viewing Taxonomy
- 3D Camera Model
- Optimizations for the Camera
- How to Deal with Occlusion
- Rasterization
  - Clipping
  - Drawing lines
  - Filling areas
The 3D rendering pipeline (our version for this class)
Perspective Projection and Photography

• In photography, we usually have the *center of projection (cop)* between the object and the image plane
  – Image on film/sensor is upside down

• In CG perspective projection, the image plane is in front of the camera!
The mathematical camera model for perspective proj.

- The Camera looks along the **negative Z axis**
- Image plane at  $z = -1$
- 2D image coordinates
  - $-1 < x < 1,$
  - $-1 < y < 1$
- Two steps
  - projection matrix
  - perspective division
Projection Matrix (one possibility)

- X and Y remain unchanged
- Z is preserved as well
- 4th (homogeneous) coordinate \( w \neq 1 \)

\[
\begin{pmatrix}
    x_{sicht} \\
    y_{sicht} \\
    z_{sicht} \\
    w_{sicht}
\end{pmatrix}
=
\begin{pmatrix}
    1 & 0 & 0 & 0 \\
    0 & 1 & 0 & 0 \\
    0 & 0 & 1 & 0 \\
    0 & 0 & -1 & 0
\end{pmatrix}
\begin{pmatrix}
    x \\
    y \\
    z \\
    1
\end{pmatrix}
=
\begin{pmatrix}
    x \\
    y \\
    z \\
    -z
\end{pmatrix}
\]

- Transformation from world coordinates into view coordinates
- This means that this is not a regular 3D point
  - otherwise the 4th component \( w \) would be \( = 1 \)
- View coordinates are helpful for culling (see later)
Perspective Division

- Divide each point by its 4th coordinate $w$

\[
\begin{pmatrix}
  x_{bild} \\
  y_{bild} \\
  z_{bild} \\
  w_{bild}
\end{pmatrix} = \frac{1}{w_{sicht}} \begin{pmatrix}
  x_{sicht} \\
  y_{sicht} \\
  z_{sicht} \\
  w_{sicht}
\end{pmatrix} = \begin{pmatrix}
  \frac{x_{sicht}}{w_{sicht}} \\
  \frac{y_{sicht}}{w_{sicht}} \\
  \frac{z_{sicht}}{w_{sicht}} \\
  \frac{w_{sicht}}{w_{sicht}}
\end{pmatrix} = \begin{pmatrix}
  x / -z \\
  y / -z \\
  -1 \\
  1
\end{pmatrix}
\]

- Transformation from view coordinates into image coordinates

- since $w = -z$ and we are looking along the negative $Z$ axis, we are dividing by a positive value

- hence the sign of $X$ and $Y$ remain unchanged

- points further away (larger absolute $Z$ value) will have smaller $x$ and $y$
  - this means that distant things are smaller
  - points on the optical axis will remain in the middle of the image
Controlling the Camera

• So far we can only look along negative Z
• Other camera positions and orientations:
  – Let C be the transformation matrix that describes the camera’s position and orientation in world coordinates
  – C is composed from a translation and a rotation, hence can be inverted
  – transform the entire world by $C^{-1}$ and apply the camera we know ;-) 

• Other camera view angles?
• If we adjust this coefficient
  – scaling factor will be different
  – larger abs value means _________ angle.
  – could also be done in the division step

$$\begin{pmatrix} x_{sicht} \\ y_{sicht} \\ z_{sicht} \\ w_{sicht} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ -z \end{pmatrix}$$
From image to screen coordinates

- Camera takes us from world via view to image coordinates
  - $-1 < x_{image} < 1$, $-1 < y_{image} < 1$

- In order to display an image we need to go to screen coordinates
  - assume we render an image of size $(w,h)$ at position $(x_{min}, y_{min})$
  - then $x_{screen} = x_{min} + w(1+x_{image})/2$, $y_{screen} = y_{min} + h(1-y_{image})/2$
Chapter 4 - 3D Camera & Optimizations, Rasterization

• Classical Viewing Taxonomy
• 3D Camera Model
• Optimizations for the Camera
• How to Deal with Occlusion
• Rasterization
  – Clipping
  – Drawing lines
  – Filling areas
Optimizations in the camera: Culling

- view frustum culling
- back face culling
- occlusion culling

View Frustum Culling

- Goal: Just render objects within the viewing volume (aka view frustum)
- Need an easy test for this...

- Z-Axis: between 2 clipping planes
  - \( z_{\text{near}} > z_{\text{view}} > z_{\text{far}} \) (remember: negative z)

- X- and Y-Axis: inside the viewing cone
  - \(-w_{\text{view}} < x_{\text{view}} < w_{\text{view}}\)
  - \(-h_{\text{view}} < y_{\text{view}} < h_{\text{view}}\)

- Two simple comparisons for each axis!
Octrees Speed up View Frustum Culling

- Naive frustum culling needs $O(n)$ tests
  - where $n =$ number of objects

- Divide entire space into 8 cubes
  - see which objects are inside each

- Subdivide each cube again
  - Repeat recursively until cube contains less than $k$ objects

- Instead of culling objects, cull cubes

- Needs $O(\log n)$ tests
Back-face Culling

- Idea: polygons on the back side of objects don’t need to be drawn
- Polygons on the back side of objects face backwards
- Use the Polygon normal to check for orientation
  - normals are often stored in face mesh structure,
  - otherwise can be computed as cross product of 2 triangle edges
  - normal faces backwards if angle with optical axis is < 90° (i.e. scalar product is > 0)
Occlusion Culling

• Idea: objects that are hidden behind others don’t need to be drawn
• efficient algorithm using an occlusion buffer, similar to a Z-buffer
Chapter 4 - 3D Camera & Optimizations, Rasterization

• Classical Viewing Taxonomy
• 3D Camera Model
• Optimizations for the Camera

• How to Deal with Occlusion

• Rasterization
  – Clipping
  – Drawing lines
  – Filling areas
Occlusion: The problem space in general

• Need to determine which objects occlude which others
• want to draw only the frontmost (parts of) objects

• Culling worked at the object level, now look at the polygons

• More general: draw the frontmost polygons
  – ..or maybe parts of polygons?

• Occlusion is an important depth cue for humans
  – need to get this really correct!
Occlusion: simple solution: depth-sort

• Regularly used in 2D vector graphics

• Sort polygons according to their z position in view coordinates

• Draw all polygons from back to front

• Back polygons will be overdrawn

• Front polygons will remain visible

• Problem 1: self-occlusion
  – not a problem with triangles ;-) 

• Problem 2: circular occlusion
  – think of a pin wheel!
Occlusion: better solution: Z-Buffer

- Idea: compute depth not per polygon, but per pixel!
- Approach: for each pixel of the rendered image (frame buffer) keep also a depth value (Z-buffer)
- Initialize the Z-buffer with $z_{\text{far}}$ which is the far clipping plane and hence the furthest distance we need to care about
- loop over all polygons
  - Determine which pixels are filled by the polygon
  - for each pixel
    - compute the $z$ value (depth) at that position
    - if $z >$ value stored in Z-buffer (remember: negative $Z$!)
      - draw the pixel in the image
      - set Z-buffer value to $z$

Z-Buffer Example
Z-Buffer: Tips and Tricks

• Z-Buffer normally built into graphics hardware
• Limited precision (e.g., 16 bit)
  – potential problems with large models
  – set clipping planes wisely!
  – never have 2 polygons in the exact same place
  – otherwise typical errors (striped objects)

• Z-Buffer can be initialized partially to something else than $x_{\text{far}}$
  – at pixels initialized to $x_{\text{near}}$ no polygons will be drawn
  – use to cut out holes in objects
  – then rerender objects you want to see through these holes

http://www.youtube.com/watch?v=TogP1J9iUCg
Chapter 4 - 3D Camera & Optimizations, Rasterization

• Classical Viewing Taxonomy
• 3D Camera Model
• Optimizations for the Camera
• How to Deal with Occlusion

• Rasterization
  – Clipping
  – Drawing lines
  – Filling areas
The 3D rendering pipeline (our version for this class)

3D models in model coordinates → 3D models in world coordinates → 2D Polygons in camera coordinates → Pixels in image coordinates

- Scene graph
- Camera
- Rasterization
- Animation, Interaction
- Lights
Rasterization: The Problems

- **Clipping**: Before we draw a polygon, we need to make sure it is completely inside the image
  - if it already is: OK
  - if it is completely outside: even better ;-)
  - if it intersects the image border: need to do clipping!

- **Drawing lines**: How do we convert all those polygon edges into lines of pixels?

- **Filling areas**: How do we determine which screen pixels belong to the area of a polygon?

- Part of this will be needed again towards the end of the semester in the shading/rendering chapter
Clipping (Cohen & Sutherland)

• Clip lines against a rectangle
  • For end points P and Q of a line
    – determine a 4 bit code each
    – 10xx = point is above rectangle
    – 01xx = point is below rectangle
    – xx01 = point is left of rectangle
    – xx10 = point is right of rectangle
    – easy to do with simple comparisons

• Now do a simple distinction of cases:
  – P OR Q = 0000: line is completely inside: draw as is (Example A)
  – P AND Q != 0000: line lies completely on one side of rectangle: skip (Example B)
  – P != 0000: intersect line with all reachable rectangle borders (Ex. C+D+E)
    • if intersection point exists, split line accordingly
  – Q != 0000: intersect line with all reachable rectangle borders (Ex. C+D+E)
    • if intersection point exists, split line accordingly
Drawing a Line: Naïve Approach

- Line from \((x_1,y_1)\) to \((x_2, y_2)\), Set \(dx := x_2 - x_1, dy := y_2 - y_1, m := dy/dx\)
- Assume \(x_2 > x_1\), otherwise switch endpoints
- Assume \(-1 < m < 1\), otherwise exchange \(x\) and \(y\)

For \(x\) from 0 to \(dx\) do:

- \(\text{setpixel}\ (x_1 + x, y_1 + m \times x)\)

In each step:

- 1 float multiplication
- 1 round to integer

Drawing a line: Bresenham‘s Algorithm

• Idea: go in incremental steps
• Accumulate error to ideal line
  – go one pixel up if error beyond a limit
• Uses only integer arithmetic
• In each step:
  – 2 comparisons
  – 3 or 4 additions

dx := x₂-x₁; dy := y₂-y₁;
d := 2*dy – dx; DO := 2*dy;
dNO := 2*(dy - dx);
x := x₁; y := y₁;
setpixel (x,y);
fehler := d;
WHILE x < x₂
  x := x + 1;
  IF fehler <= 0 THEN
    fehler := fehler + DO
  ELSE
    y := y + 1;
    fehler = fehler + dNO
  END IF;
  setpixel (x,y);
END WHILE

Antialiased Lines

• Problem: Bresenham‘s lines contain visible steps (aliasing effects)
• Opportunity: we can often display greyscale
• Idea: use different shades of grey as different visual weights
  – instead of filling half a pixel with black, fill entire pixel with 50% grey

• Different algorithms exist
  – Gupta-Sproull for 1 pixel wide lines
  – Wu for infinitely thin lines
Wu’s Antialiasing Approach

• Loop over all x values
• Determine 2 pixels closest to ideal line
  – slightly above and below
• Depending on distance, choose grey values
  – one is perfectly on line: 100% and 0%
  – equal distance: 50% and 50%
• Set these 2 pixels
Antialiasing in General

- Problem: hard edges in computer graphics
- Correspond to infinitely high spatial frequency
- Violate sampling theorem (Nyquist, Shannon)
  - reread 1st lecture „Digitale Medien“

- Most general technique: Supersampling
- Idea:
  - render an image at a higher resolution
    - this way, effectively sample at a higher resolution
  - scale it down to intended size
  - interpolate pixel values
    - this way, effectively use a low pass filter

Line Drawing: Summary

• With culling and clipping, we made sure all lines are inside the image
• With algorithms so far we can draw lines in the image
  – even antialiased lines directly
• This means we can draw arbitrary polygons now (in black and white)

• All algorithms extend to color
  – just modify the setpixel (x,y) implementation
  – choice of color not always obvious (think through!)
  – how about transparency?

• All these algorithms implemented in hardware
• Other algorithms exist for curved lines
  – mostly relevant for 2D graphics
Filling a Polygon: Scan Line Algorithm

• Define parity of a point in 2D:
  – send a ray from this point to infinity
  – direction irrelevant (!)
  – count number of lines it crosses
  – if 0 or even: even parity (outside)
  – if odd: odd parity (inside)

• Determine polygon area \((x_{\text{min}}, x_{\text{max}}, y_{\text{min}}, y_{\text{max}})\)
• Scan the polygon area line by line
• Within each line, scan pixels from left to right
  – start with parity = 0 (even)
  – switch parity each time we cross a line
  – set all pixels with odd parity
Rasterization Summary

- Now we can draw lines and fill polygons
- All algorithms also generalize to color
- How do we determine the shade of color?
  - this is called shading and will be discussed in the rendering section