graphics pipeline

• sequence of operations to generate an image using object-order processing
  – primitives processed one-at-a-time
  – software pipeline: e.g. Renderman
    • high-quality and efficiency for large scenes
  – hardware pipeline: e.g. graphics accelerators
    • lower-quality solution for interactive applications

• will cover algorithms of modern hardware pipeline
  – but evolve drastically every few years
  – we will only look at triangles
graphics pipeline

- handles only simple primitives by design
  - points, lines, triangles, quads (as two triangles)
  - efficient algorithm
- complex primitives by tessellation
  - complex curves: tessellate into line strips
  - curves surfaces: tessellate into triangle meshes
- “pipeline” name derives from architecture design
  - sequences of stages with defined input/output
  - easy-to-optimize, modular design

graphics pipeline

- object-local algorithm
  - processes only one-surface-at-a-time
- various effects have to be approximated
  - shadows: shadow volume and shadow maps
  - reflections: environment mapping
  - hard to implement
- advanced effects cannot be implemented
  - soft shadows
  - blurry reflections and diffuse-indirect illumination
graphics pipeline stages

- vertex processing
  - input: vertex data (position, normal, color, etc.)
  - output: transformed vertices in homogeneous canonical view-volume, colors, etc.
  - applies transformation from object-space to clip-space
  - passes along material and shading data

- clipping and rasterization
  - turns sets of vertices into primitives and fills them in
  - output: set of fragments with interpolated data

- transformations
  - input: vertex data
  - output: transformed vertices

- convert to pixels
  - input: transformed vertices
  - output: fragments with interpolated data

- final colors
  - input: fragments with interpolated data
  - output: fragments with color and depth

- blending hidden-surface
  - input: fragments with color and depth
  - output: framebuffer

- framebuffer processing
  - input: framebuffer
  - output: frame buffer
graphics pipeline stages

• fragment processing
  – output: final color and depth
  – traditionally mostly for texture lookups
    • lighting was computed for each vertex
  – today, computes lighting per-pixel

• framebuffer processing
  – output: final picture
  – hidden surface elimination
  – compositing via alpha-blending

vertex processing

vertex data
vertex processing
transformed vertex data
clipping and rasterization
fragments w/interpolated data
fragment processing
fragments color and depth
framebuffer processing
framebuffer
vertex processing

- transform vertices from model to clip space

vertex processing

- other geometry tasks
  - deformation: skinning, mesh blending
  - low-quality lighting
  - pass other properties to next stages of pipeline
  - the only place to algorithmically alter shape

- programmable hardware unit
  - algorithm can be changed at run-time by application
  - only recent change
clipping and rasterization

- vertex data
  - vertex processing
    - transformed vertex data
      - clipping and rasterization
        - fragments w/interpolated data
          - fragment processing
            - fragments color and depth
              - framebuffer processing
                - framebuffer

clipping

- remove (partial) objects not in the view frustum
  - efficiency: cull later stages of the pipeline
  - correctness: perspective transform can cause trouble
  - often referred as culling when full objects removed
clipping to ensure correctness

in front of eye

behind eye

point clipping

- point-plane clipping
  - test if the point is on the right side of the plane
  - by taking dot-product with the plane normal
  - can be performed in homogeneous coordinates

- point-frustum clipping
  - point-plane clipping for each frustum plane
line clipping

• segment-plane clipping
  – test point-plane clipping for endpoints
  – if endpoints are clipped, clip whole segment
  – if endpoints are not clipped, accept whole segment
  – if one endpoint is clipped, clip segment
    • compute segment-plane intersection
    • create shorter segment

line clipping

• segment-frustum clipping
  – clip against each plane incrementally
  – guarantee to create the correct segment

• more efficient algorithms available
  – previous incremental approach might try too hard
  – provide early rejection for common cases
  – so, only clip when necessary
polygon clipping

- convex polygons similar to line clipping
  - clip each point in sequence
    - remove outside points
    - create new points on boundary
  - clipped triangles are not necessarily triangles

---
culling

- further optimize by rejecting “useless” triangles

- backface culling
  - if triangle face is oriented away from camera, cull it
  - only ok for closed surfaces

- early z-culling
  - if triangle is behind existing scene, cull it
  - uses z-buffer introduced later on
viewport transformation

- transform the canonical view volume to the pixel coordinates of the screen
- also rescale $z$ in the $[0..1]$ range
  - we will see later why
- perspective divide is often performed here

![Viewport Transformation Diagram](Marschner_2004)

rasterization

- approximate primitives into pixels
  - pixel centered at integer coordinates

- determine which pixels to turn on
  - no antialiasing (jaggies): pixel in the primitive
  - consider antialiasing for some primitives
  - input: vertex position in homogenous coordinates

- interpolate values across primitive
  - colors, normals, position at vertices
  - input: any vertex property
line rasterization

- approximate line with a collection of pixels
- desirable properties
  - uniform thickness and brightness
  - continuous appearance (no holes)
  - efficiency
  - simplicity (for hardware implementation)

- line equation: \( y = mx + b \)
  - in this lecture, for simplicity, assume \( m \) in \([0,1)\)

point-sampled line rasterization

- represent line as rectangle
- approximated by all pixel within the line
  - for each pixel center, test if inside the rectangle
- inefficient
  - many inside tests
- inaccurate
  - thickness not constant
midpoint line rasterization

- for each column
  - only turn on closest pixel
- simple algorithm
  - given line equation
  - eval. eq. for each column
  - between endpoints

```plaintext
for x = ceil(x0) to floor(x1) {
    y = m*x + b
    write(x, round(y))
}
```

optimizing midpoint line rasterization

- evaluating $y$ is slow
- use incremental difference, DDA
  \[ m = \Delta y / \Delta x \]
  \[ y(x + 1) = y(x) + m \]

```plaintext
x = ceil(x0)
y = m*x + b
while x < floor(x1) {
    write(x, round(y), 1)
    y += m
    x += 1
}
bresenham’s line rasterization

• at each pixel \((x_p, y_p)\),
  only two options:
  \(E(x_p+1, y_p)\) or \(NE(x_p+1, y_p+1)\)
• \(d = (x_p + 1)m + b - y_p\)
  – if \(d > 0.5\) then \(NE\)
  – else \(E\)
• can evaluate \(d\) using
  incremental differences
  – \(NE\): \(d += m - 1\)
  – \(E\): \(d += m\)
• can use integers only

\[
x = \text{ceil}(x_0) \\
y = \text{round}(m x + b) \\
d = m(x+1)+b-y \\
\text{while } x < \text{floor}(x_1) \\
\text{write}(x, y, 1) \\
x += 1 \\
d += m \\
\text{if } d > 0.5 \\
y += 1 \\
d -= 1
\]
**midpoint vs. point-sampled line**

- **point-sampled**
  - varying thickness
- **midpoint**
  - same thickness

**antialiased line rasterization**

- for each pixel, color is the ratio of the area covered by the line
- need to touch multiple pixels per column
- can be done efficiently by precomputation and lookup tables
  - area only depends on line to pixel distance
interpolating parameters along a line

- Often associate params $q_i$ at line vertices
  - Colors, alphas
- Linearly interpolate $q_i$
  \[ q_i(s) = q_{i0} \cdot (1 - s) + q_{i1} \cdot s \]
  - $s$ is fractional distance along the line
  - Can be done using incremental differences

triangle rasterization

- Most common operation in graphics pipelines
  - Can be the only one: turn everything into triangles
- Input: 2D triangle with vertex attributes
  - 2D vertex coordinates: $\{(x_0, y_0), (x_1, y_1), (x_2, y_2)\}$
  - Other attributes: $\{q_{i0}, q_{i1}, q_{i2}\}$
- Output: list of fragments with interpolated attributes
  - List of pixel coordinates that are to be drawn
  - Linearly interpolated vertex attributes
brute force triangle rasterization

- foreach pixel in image
  - determine if inside triangle
  - interpolate attributes
- use baricentric coordinates
- optimize by only checking triangle bounding box
triangle baricentric coordinates

\[ \mathbf{p} = \alpha \mathbf{a} + \beta \mathbf{b} + \gamma \mathbf{c} \quad \alpha + \beta + \gamma = 1 \]

- analytic interpretation
  - coordinate system on the triangle
    \[ \mathbf{p} = \mathbf{a} + \beta (\mathbf{b} - \mathbf{a}) + \gamma (\mathbf{c} - \mathbf{a}) \]

- geometric interpretation
  - relative areas
  - relative distances

- also useful for ray-triangle intersection

---

brute force triangle rasterization

for each pixel \((x,y)\) in triangle bounding box
compute \((\alpha,\beta,\gamma)\)
if \((\alpha,\beta,\gamma)\) in \([0,1]^3\)
\[ q_i = \alpha \cdot q_{i0} + \beta \cdot q_{i1} + \gamma \cdot q_{i2} \]
write\((x,y,\{q_i\})\)

- can be made incremental as in line drawing
- more efficient options exists, but …
triangle rasterization on hardware

• old hardware: optimized for large triangles
  – use smart algorithm
    • clip triangle to screen window
    • setup initial values
    • interpolate
  – hard to parallelize, high set up cost
• modern hardware: optimized for small triangles
  – use incremental brute force algorithm
    • only clip against near plane for correctness
    • work with clipped bounding box
  – easily parallelizable, little setup cost
    • use tiles in image plane

rasterization take home message

• complex but efficient set of algorithms
  – lots of small little details that matters for correctness
• no clear winner
  – architecture: parallel vs. serial
  – input: e.g. size of triangles
  – amortization: one-time vs. step-by-step cost
• complex algorithms have often hidden costs
  – verify if they can be amortized
• loops are expensive: optimize as you can
fragment processing

- compute final fragment colors, alphas and depth
  - depth is often untouched if no special effects
  - final lighting computations
  - lots of texture mapping: see later
- programmable hardware unit
  - algorithm can be changed at run-time by application
  - only recent change

vertex data

vertex processing

transformed vertex data

clipping and rasterization

fragments w/interpolated data

fragment processing

fragments color and depth

framebuffer processing

framebuffer
framebuffer processing

- vertex data
  - vertex processing
  - transformed vertex data
  - clipping and rasterization
  - fragments w/interpolated data
  - fragment processing
  - fragments color and depth
  - framebuffer processing
  - framebuffer

framebuffer processing

- hidden surface elimination
  - decides which surfaces are visible
- framebuffer blending
  - composite transparent surfaces if necessary
hidden surface removal – painter alg.

- sort objects back to front
- draw in sorted order
- does not work in many cases
hidden surface removal – z buffer

- brute force algorithm
- for each pixel, keep distance to closest object
- foreach object, rasterize updating pixels if distance is closer
  - opaque objects: works in every case
  - transparent objects: cannot properly composite
hidden surface removal – z buffer

\[
\begin{array}{c|c|c}
\infty & \infty & \infty \\
\infty & \infty & 3 \\
\infty & 5 & 3 \\
7 & 5 & 3 \\
\hline
1 & \infty & \infty \\
1 & 3 & 3 \\
1 & 3 & 3 \\
1 & 3 & 3 \\
\end{array}
\]

adapted from Shirley

which z distance

- use $z$ value after homogenous xform
  - linear interpolation works
  - storage non-linear: more precision around near frame

projection plane

eye point

equally spaced $z$ (distance)

adapted from Shirley

[Marschner 2004]
which z distance

- use z value after homogenues xform
  - linear interpolation works
  - storage non-linear: more precision around near frame

hidden surface removal – raycasting

- foreach ray, find intersection to closest surface
  - works for opaque and transparent objects
- loops over pixels and then over surfaces
  - inefficient
  - would like to loop over surfaces only once
hidden surface removal - scanline

- for each scanline, sort primitives
  - incremental rasterization
  - sorting can be done in many ways
  - needs complex data structures
  - works for opaque and transparent objects

hidden surface removal - REYES

- foreach primitives, turn into small grids of quads
- hit-test quads by ray-casting
- keep list of sorted list hit-points per pixel
  - like z-buffer but uses a list
  - works for opaque and transparent objects
- hybrid between raycast and z-buffer
  - very efficient for high complexity
    - when using appropriate data-structures
  - solves many other problems we will encounter later
framebuffer processing

- hidden surface elimination using Z-buffer

- framebuffer blending using $\alpha$-compositing
  - but cannot sort fragments properly
  - incorrect transparency blending
  - need to presort transparent surfaces only
    - like painter's algorithm, so not correct in many cases

lighting computation

- where to evaluate lighting?
  - flat: at vertices but do not interpolate colors
  - Gourand: at vertices, with interpolated color
  - Phong: at fragments, with interpolated normals
lighting computation – flat shading

- compute using normals of the triangle
  - same as in raytracing
- flat and faceted look
- correct: no geometrical inconsistency

lighting computation – Gouraud shading

- compute light at vertex position
  - with vertex normals
- interpolate colors linearly over the triangle
lighting computation – Phong shading

- interpolate normals per-pixels: shading normals
- compute lighting for each pixel
  - lighting depends less on tessellation
lighting computation

• per-pixel lighting is becoming ubiquitous
  – much more robust
  – move lighting from vertex to fragment processing
    • new hardware architectures allows for this
    • we introduce Gouraud for historical reasons
  – raytracing can have this by using shading normals
• shading normals introduce inconsistencies
  – lights can come from “below” the surface

why graphics pipelines?

• simple algorithms can be mapped to hardware
• high performance using on-chip parallel execution
  – highly parallel algorithms
  – memory access tends to be coherent
  – one-object at a time
graphics pipeline architecture

- multiple arithmetic units
  - NVidia Geforce 7800: 8 vertex units, 24 pixel units
- very small caches
  - not needed since memory access are very coherent
- fast memory architecture
  - needed for color/z-buffer traffic
- restricted memory access patterns
  - no read-modify-write
  - bound to change hopefully
- easy to make fast: this is what Intel would love!
- research into using for scientific computing

graphics pipeline performance

![Graph showing graphics pipeline performance](image-url)
graphics pipeline performance

- SAXPY
- Segment
- SGEMV
- FFT
- Ray-tracer

compared against:
- Intel Math Library
- Atlas Math Library
- cached blocked segmentation
- FFTW
- Wald [’04] SSE Ray-Triangle

graphics pipelines vs. raytracing

raycasting

- foreach pixel
  - foreach object
- project pixels onto objects
- discretize first
- access objects many times
  - scene must fit in memory
- very general solution
- \(O(\log(n))\) w/accel. struct.
  - but constant very high

graphics pipeline

- foreach object
  - foreach pixel
- project objects onto pixels
- discretize last
- access objects once
  - image must fit in memory
- hard for complex effects
- \(O(n)\) or lower sometimes
  - but constant very small