[CS7031] Graphics and Console Hardware and Real-time Rendering

15th Lecture :: GPU - Rasterisation (one)

Dr. Michael Manzke
michael.manzke@cs.tcd.ie

Trinity College Dublin
Outline:

- Fundamentals
- System examples
- Special topics

Suggested reading:

- Triangle Scan Conversion Using 2D Homogeneous Coordinates [OG97]
- Optimal Depth Buffer for Low-cost Graphic Hardware [LJ99]
- Minimum Triangle Separation for Correct Z-buffer Occlusion [AS06]
Recall that...

- Straight lines project to straight lines:
  - When projection is to a plan (our assumption)
  - Only vertexes need to be transformed
  - That’s why we’re interested in lines and polygons

- Parameterizations (e.g., projected distance) are warped:
Recall that ...

- Ideal screen coordinates are continuous
- Implementations always use discrete math, but with substantial sub-pixel precision
- A pixel is a big thing
  - Addressable resolution equal to pixels on screen
  - Lots of data
- Points and lines have no geometric area ...
Terminology

- Rasterisation: conversion of *primitives* to *fragments*
  - Primitive: point, line, polygon, glyph, image, ...
  - Fragment: transient data structure, e.g.,
    - `short x, y;`
    - `long depth;`
    - `short r, g, b, a;`

- Pixels exist in an array (e.g., framebuffer)
  - Have implicit `⟨x, y⟩` cooordinates

- Fragment are routed to appropriate pixels
  - This is the "sort" we saw in the previous lecture
  - There will be more sorting ...
Two Fundamental Operation

- **Fragment selection**
  - Identify pixels for which fragments are to be generated
  - Must be conservative, efficiency matters
  - \( <x, y> \) parameters are special

- **Attribute assignment**
  - Assign attribute value to each fragment
  - E.g., color, depth, ...

- **Scheduling**: the third fundamental operation
  - Assign fragment to parallel fragment shaders
Fragment Selection

- Generate one fragment for each pixel that is intersected by the primitive
- Intersect could mean that the primitive’s area intersects the pixel’s:
  - Centre point, or
  - Square region, or
  - Filter-function
- Some examples on the following slides:
Point Sampled Fragment Selection

- Aliased rasterisation
Box Sampled Fragment Selection

- Tiled rasterisation (scaling the primitive doesn’t do this)
Filtered Fragment Selection

- Antialiased rasterisation
Fragment Selection (continued)

- What if the primitive doesn’t have a geometric area?
  - Points don’t
  - Lines don’t

- Two choices
  - Rule-based approach (e.g., Bresenham line)
    - allow desired properties to be maintained, but
    - may require additional hardware complexity
  - Assign an screen-space area (e.g. circle for point, rectangle for line)
    - can utilize polygon rasterisation algorithm, but
    - may result in wavy lines, flashing points, etc.
**Attribute Assignment**

- Identify a parameterisation for each attribute
  - e.g. planar
- Sample this parameterisation as required
- Which parameterisation?
  - Lots of possibilities (linear, cubic, ...)
  - Always define implicitly by vertex values
Properties of vertex-defined parameterisation:

- Zero-order continuity
  - If fully specified by the two vertexes that specify the edge
- Triangle allow planar parameterisation
- Polygons (4+ edges) are (almost) never planar
- Parameterisation may vary with screen orientation
Linear Interpolation

- Compute intermediate attribute value
  - Along a line: \( A = aA_1 + bA_2, \ a + b = 1 \)
  - On a plan: \( A = aA_1 + bA_2 + cA_3, \ a + b + c = 1 \)

- Only projected values interpolate linearly in screen space (straight lines project to straight lines)
  - \( a \) and \( y \) are projected (divided by \( w \))
  - Attribute values are not naturally projected
Linear Interpolation

- Choice for attribute interpolation in screen space
  - Interpolate unprojected values
    - Cheap and easy to do, but gives wrong values
    - Sometimes OK for color, but
    - Never acceptable for texture coordinates
  - Do it right
Incorrect Attribute Interpolation

Linear interpolation

\[ A' \neq A \]
Perspective-correct Linear Interpolation

- Only projected values interpolate correctly, so project $A$
  - Lineatly interpolate $\frac{A_1}{w_1}$ and $\frac{A_2}{w_2}$
  - Also interpolate $\frac{1}{w_1}$ and $\frac{1}{w_2}$
  - These also interpolate linearly in screen space
- Divide interpolants at each sample point to recover $A$
  - $\frac{A}{w_1} = A$
- Division is expensive (more than add or multiply, so)
  - Recover $w$ for the sample point (reciprocate), and
  - Multiply each projected attribute by $w$
Barycentric triangle parameterization:

\[ A = \frac{a A_1}{w_1} + \frac{b A_2}{w_2} + \frac{c A_3}{w_3} \]

\[ a + b + c = 1 \]
Example: Gouraud Shaded Quadrilateral

Fragment selection
- Walk edges
- Change edges at vertexes

Attribute assignment
- Loop in a loop algorithm:
  - Interpolate along edges
  - Interpolate edge-to-edge
- Outer loop is complex
  - E.g., either 2 or 3 regions
- Parameterization is a function of
  - Screen orientation
  - Choice of spans
“All” projected quadrilaterals are non-planar
  ■ Due to discrete coordinate precision

What if quadrilateral is concave?
  ■ Concave is complex (split spans -- see example)
  ■ Non-planar $\rightarrow$ concave for some view

What if quadrilateral intersects itself?
  ■ A real mess (no vertex to signal change -- see example)
  ■ Non-planar $\rightarrow$ “bowtie” for some view
All Polygons are Triangles

Three points define a plane
- All triangles are planar
- All parameterizations are planar

Triangle is always convex
- Regardless of arithmetic precision
- Simple rasterization, no special cases

Modern GPUs decompose $n$-gons to triangles
- SGI switched in 1990, VGX product
- OpenGL designed to support this
- Optimized quadrilateral decomposition developed
Normal-based Quad Decomposition

Compute \((A \text{ dot } C)\) and \((B \text{ dot } D)\)

Connect vertex pair with the greater dot product
  - Avoid connecting the stirrups

Avoid frame-to-frame jitter
  - If dot products are equal
  - E.g., symmetric curved surface
  - Transforming will add noise
Modern choice for aliased rendering

Fragment selection
- Generate fragment if pixel center is inside triangle
- Handle special-case edge/vertex intersections

Attribute assignment
- Sample function at pixel center
- Yields mean value for “surrounded” pixels
- Generates a consistent ray for depth buffering

Never sample outside the triangle
- Avoid color wrap, depth “punch-through”
- But how is antialiased filtering handled?
Point Sampled Points and Lines

Geometric points and lines have no area, so

- Pixel sample locations almost never “in” primitive
- Semantics are confused at best

Must assign attribute parameterizations for pseudo-area

- Point: attribute-constant disk
- Line: single-attribute-slope rectangle

Problem: how to outline filled, depth-buffered triangles?

- Depth values are “wrong”, lines disappear
- VGX introduced “hollow” polygons
- OpenGL 1.1 introduced glPolygonOffset()
Outline Problem
Goal: efficient interpolation

Direct evaluation is expensive

- Requires multiplications for each evaluation

Digital Differential Analyzer (DDA)

- Fixed point \textit{iiiiii.fffffff} representation, accumulator and slope

- Add slope repeatedly to accumulator to evaluate adjacent sample locations

- Planar DDA uses separate $X$ and $Y$ slopes
  - Can move around the plane arbitrarily

- Require $\log_2(n)$ fraction bits for $n$ accumulation steps!
Triangle Rasterisation Examples

Gouraud shaded (GTX)
Per-pixel evaluation (Pixel Planes 4)
Edge walk, planar parameterization (VGX)
Barycentric direct evaluation (InfiniteReality)
Small tiles (Bali - proposed)
Homogeneous recursive descent (NVIDIA)
Algorithm Properties

Setup and execution costs
- Setup: constant per triangle
- Execution: relative to triangle’s projected area

Ability to parallelize

Ability to cull to a rectangular screen region
- To support tiling
- To support “scissoring”
Gouraud Shaded (GTX)

Two-stage algorithm
- DDA edge walk
  - fragment selection
  - parameter assignment
- DDA scan-line walk
  - parameter assignment only

Requires expensive scan-line setup
- Location of first sample is non-unit distance from edge

Parallelizes in two stages (e.g., GTX)
Cannot scissor efficiently
Works on quadrilaterals
Engine per Pixel (Pixel Plane 4)

Pxpl4 Block Diagram

- 512 x 512 frame buffer
- 128 processors/chip
- 2048 EMC chips
- 32 boards
- Clock speed: ~ 10MHz
**Engine per Pixel (Pixel Plane 4)**

Individual engine at every pixel!

- Solves edge equations to determine inclusion
- Solves attribute equations to determine values

Setup involves computation of plane and edge slopes

Execution is in constant-time

- Clever evaluation tree makes this possible
- Extremely fast for large triangles, but
- Extremely inefficient for small triangles
  - Pixel depth complexity = # triangles in scene
  - Scissor culling is a non-issue
Pixel Planes 4 Fragment Selection

Linear Expressions

Ax + By + C