CS130 : Computer Graphics
Lecture 2: Graphics Pipeline

Tamar Shinar
Computer Science & Engineering
UC Riverside
Raster Devices and Images
Raster Devices

- Raster displays show images as a rectangular array of pixels
- Most printers are also raster devices
  - Image is made by depositing ink at points on a grid
- Digital cameras – have image sensors made of grid of light-sensitive pixels (2D array)
- Scanner – linear array of pixels swept across page to create grid of pixels (1D array)
virtually all graphics system are **raster based**, meaning the image we see is a **raster of pixels**

or a rectangular array of pixels

Here a raster scan device display an image as a set of discrete points across each scanline
Displays are either transmissive or emissive.

One pixel of an LCD display:
(LEFT) In the off state, the front polarizer blocks all the light that passes the back polarizer. In the on state, the liquid crystal rotates the polarization of the light so it can pass through the front polarizer. The degree of rotation can be adjusted by an applied voltage.

(RIGHT) LED display
Raster Display

red, green, blue subpixels

get different colors by mixing red, green, and blue
this is from an LCD monitor
printers are also raster-based. image is made out of points on a grid
What is an image?

**Continuous image**

$I : R \rightarrow V$

$R \subset \mathbb{R}^2$

$V = \mathbb{R}^+ \quad \text{(grayscale)}$

$V = (\mathbb{R}^+)^3 \quad \text{(color)}$

An (continuous) image is a function defined over some 2D area, that maps points to intensity level.
What is an image?

**Sampled image**

\[ I : R \rightarrow V \]

\[ R \subset \mathbb{Z}^2 \]

\[ V = [0, 1] \quad \text{(grayscale)} \]

\[ V = [0, 1]^3 \quad \text{(color)} \]

\[ n_x = \text{number of columns} \]

\[ n_y = \text{number of rows} \]

\[ R = [-0.5, n_x - 0.5] \times [-0.5, n_y - 0.5] \]

Each pixel value represents the **average color** of the image over that pixel’s area.
in practice, it is sufficient for pixels to have a bounded range e.g., \([0,1]\)
They are represented in integers
A **raster image** is 2D array storing pixel values at each pixel (picture element)
3 numbers for color
alternative: **vector image** -- essentially a set of instructions for rendering an image
Monitor Gamma

displayed intensity = (max intensity) $a^\gamma$

monitors convert pixel values, $a$, into displayed intensities
monitors are nonlinear with respect to input
Gamma Correction

displayed intensity = (max intensity) \left(\frac{1}{a} \right)^\gamma

find gamma using, e.g., checkboard

then gamma-correct the input

find gamma, so that you can give the monitor a^{1/\gamma}

- find a such that a^{\gamma} = .5 through checkboard test and solve for gamma
Color representation

additive color – Primary colors are red, green, blue. Form a color by adding these. CRTs, projectors, LCD displays, positive film

subtractive color – Form a color by filtering white light with cyan, magenta, and yellow filters. Printing, negative film
Alpha Channel

\[ c = \alpha c_f + (1 - \alpha)c_b \]

Compositing: two different interpretations: **pixel coverage** (fraction of pixel covered) and **blending**
Graphics Pipeline
Modern graphics system

the pixels are stored in a location in memory call the frame buffer
frame buffer resolution determines the details in the image
- e.g., 24 bit color “full color”
- high dynamic range or HDR use 12 or more bits for each color
frame buffer = color buffers + other buffer
Z-buffer Rendering

- Z-buffering is very common approach, also often accelerated with hardware
- OpenGL is based on this approach
Choice of primitives

• Which primitives should an API contain?
  • small set - supported by hardware, or
  • lots of primitives - convenient for user
Choice of primitives

- Which primitives should an API contain?
  - small set - supported by hardware
  - lots of primitives - convenient for user

Performance is in 10s millions polygons/sec -- portability, hardware support key
Choice of primitives

• Which primitives should an API contain?
  ➡ small set - supported by hardware
  • lots of primitives - convenient for user

  GPUs are optimized for points, lines, and triangles

Other geometric shapes will be built out of these
Two classes of primitives

Geometric: points, lines, polygons
Image: arrays of pixels
Point and line segment types

GL_POINTS
GL_LINES
GL_LINE_STRIP
GL_LINE_LOOP
Polygons

- Multi-sided planar element composed of edges and vertices.
- Vertices (singular vertex) are represented by points.
- Edges connect vertices as line segments.
Valid polygons

- Simple
- Convex
- Flat
Valid polygons

- Simple
- Convex
- Flat
OpenGL polygons

- Only triangles are supported (in latest versions)

GL_POINTS

GL_TRIANGLES

GL_TRIANGLE_STRIP

GL_TRIANGLE_FAN
Other polygons

triangulation

as long as triangles are not collinear, they will be simple, flat, and convex -- easy to render
An arithmetic pipeline that computes \( c + (a \times b) \)

By pipelining the arithmetic operation, the **throughput**, or rate at which data flows through the system, has been **doubled**

If the pipeline had more boxes, the **latency**, or time it takes one datum to pass through the system, would be higher

**Throughput and latency must be balanced**
3D graphics pipeline

Vertices → Vertex processor → Clipper and primitive assembler → Rasterizer → Fragment processor → Pixels

**Geometry**: primitives – made of vertices
**Vertex processing**: coordinate transformations and color
**Clipping and primitive assembly**: output is a set of primitives
**Rasterization**: output is a set of fragments for each primitive
**Fragment processing**: update pixels in the frame buffer

The pipeline is best when we are doing the same operations on many data sets -- good for computer graphics!! where we process larges sets of vertices and pixels in the same manner.

1. **Geometry**: objects – made of primitives – made of vertices
2. **Vertex processing**: coordinate transformations and color
3. **Clipping and primitive assembly**: use clipping volume. must be primitive by primitive rather than vertex by vertex. therefore vertices must be assembled into primitives before clipping can take place. Output is a set of primitives.
4. **Rasterization**: primitives are still in terms of vertices -- must be converted to pixels. E.g., for a triangle specified by 3 vertices, the rasterizer must figure out which pixels in the frame buffer fill the triangle. Output is a set of **fragments for each primitive**. A fragment is like a **potential pixel**. Fragments can carry depth information used to figure out if they lie behind other fragments for a given pixel.
5. **Fragment processing**: update pixels in the frame buffer. some fragments may not be visible. texture mapping and bump mapping. blending.
Graphics Pipeline
(slides courtesy K. Fatahalian)
Vertex processing

Vertices are transformed into “screen space”
Vertex processing

Vertices are transformed into “screen space”

Vertices are transformed into “screen space”

EACH VERTEX IS TRANSFORMED INDEPENDENTLY
Primitive processing

Then organized into primitives that are clipped and culled…
Primitives are rasterized into “pixel fragments”
Primitives are rasterized into “pixel fragments”

Rasterization

EACH PRIMITIVE IS RASTERIZED INDEPENDENTLY
Fragment processing

Fragments are shaded to compute a color at each pixel

Vertices → Vertex processor → Clipper and primitive assembler → Rasterizer → Fragment processor → Pixels

Shaded fragments
Fragment processing

Fragments are shaded to compute a color at each pixel

EACH FRAGMENT IS PROCESSED INDEPENDENTLY
Pixel operations

Fragments are blended into the frame buffer at their pixel locations (z-buffer determines visibility)
Pipeline entities

Vertices

Primitives

Fragments

Fragments (shaded)

Pixels
Graphics pipeline

- Vertices: Vertex Generation → Vertex Processing → Primitive Generation → Primitive Processing
- Primitives: Primitive stream → Primitive Processing → Fragment Generation → Fragment Processing
- Fragments: Fragment stream → Fragment Processing → Pixel Operations
- Pixels: Output image (pixels)

Memory Buffers:
- Vertex Data Buffers: Fixed-function
- Textures: Programmable
- Textures: Programmable
- Textures: Programmable
- Output image (pixels): Programmable