Lecture 4:
Drawing a Triangle (and an Intro to Sampling)

Computer Graphics
CMU 15-462/15-662
Assignment 0.5 due, MiniHW 0 due, Assignment 1 out, MiniHW 1 out!

- **GOAL:** Implement a basic “rasterizer”
  - (Topic of today’s lecture)
  - We hand you a bunch of lines, triangles, etc.
  - You draw them by lighting up pixels on the screen!
- Code skeleton available from course webpage
- First checkpoint due February 9th.
- Completed assignment due February 16th.
Assignment 0.5 due, MiniHW 0 due, Assignment 1 out, MiniHW 1 out!

- GOAL: Implement a basic “rasterizer”
  - (Topic of today’s lecture)
  - We hand you a bunch of lines, triangles, etc.
  - You draw them by lighting up pixels on the screen!
- Code skeleton available from course webpage
- First checkpoint due February 9th.
- Completed assignment due February 16th.

See piazza for build tutorial!

Extra office hours today 3-5pm for build questions!
TODAY: Rasterization

- Two major techniques for “getting stuff on the screen”
  - Rasterization (TODAY)
    - for each primitive (e.g., triangle), which pixels light up?
    - extremely fast (BILLIONS of triangles per second on GPU)
    - harder (but not impossible) to achieve photorealism
    - perfect match for 2D vector art, fonts, quick 3D preview, ...
  - Ray tracing (LATER)
    - for each pixel, which primitives are seen?
    - easier to get photorealism
    - generally slower
    - much more later in the semester!
3D Image Generation Pipeline(s)

- Can talk about image generation in terms of a “pipeline”:
  - **INPUTS** — what image do we want to draw?
  - **STAGES** — sequence of transformations from input → output
  - **OUTPUTS** — the final image

E.g., our pipeline from the first lecture:

**VERTICES**

- A: (1, 1, 1)
- B: (-1, 1, 1)
- C: (1, -1, 1)
- D: (-1, -1, 1)
- E: (1, 1, -1)
- F: (-1, 1, -1)
- G: (1, -1, -1)
- H: (-1, -1, -1)

**EDGES**

- AB, CD, EF, GH
- AC, BD, EG, FH
- AE, CG, BF, DH

**INPUT**

**PERSPECTIVE PROJECTION STAGE**

**LINE DRAWING STAGE**

**OUTPUT**

\[ p = (x, y, z) \]

\[ q = (u, v) \]
Rasterization Pipeline

- Modern real time image generation based on rasterization
  - INPUT: 3D “primitives”—essentially all triangles!
  - possibly with additional attributes (e.g., color)
  - OUTPUT: bitmap image (possibly w/ depth, alpha, ...)

Our goal: understand the stages in between*

*In practice, usually executed by graphics processing unit (GPU)
Why triangles?

- Rasterization pipeline converts all primitives to triangles
  - even points and lines!
- Why?
  - can approximate any shape
  - always planar, well-defined normal
  - easy to interpolate data at corners
    - “barycentric coordinates”
- Key reason: once everything is reduced to triangles, can focus on making an extremely well-optimized pipeline for drawing them

“point” □

“line”
The Rasterization Pipeline

Rough sketch of rasterization pipeline:

- Reflects standard “real world” pipeline (OpenGL/Direct3D)
  — the rest is just details (e.g., API calls)
Let’s draw some triangles on the screen

Question 1: what pixels does the triangle overlap? (“coverage”)

Question 2: which triangle is closest to the camera in each pixel? (“occlusion”)
The visibility problem

Recall the pinhole camera...
The visibility problem

Recall the pinhole camera... which we can simplify with a “virtual sensor”:

Visibility problem in terms of rays:
- **COVERAGE**: What scene geometry is hit by a ray from a pixel through the pinhole?
- **OCCLUSION**: Which object is the first hit along that ray?
Computing triangle coverage

“Which pixels does the triangle overlap?”

Input:
projected position of triangle vertices: $P_0, P_1, P_2$

Output:
set of pixels “covered” by the triangle
What does it mean for a pixel to be covered by a triangle?

Q: Which triangles “cover” this pixel?
One option: compute fraction of pixel area covered by triangle, then color pixel according to this fraction.

Intuition: if triangle covers 10% of pixel, then pixel should be 10% red.
Coverage gets tricky when considering occlusion

Pixel covered by triangle 1, other half covered by triangle 2

Two regions of triangle 1 contribute to pixel. One of these regions is not even convex.

Interpenetration of triangles: even trickier
Coverage via sampling

- Real scenes are complicated!
  - occlusion, transparency, ...
  - will talk about this more in a future lecture!

- Computing exact coverage is not practical

- Instead: view coverage as a sampling problem
  - don’t compute exact/analytical answer
  - instead, test a collection of sample points
  - with enough points & smart choice of sample locations, can start to get a good estimate

- First, let’s talk about sampling in general...
Sampling 101: Sampling a 1D signal

\[ f(x) \]
Sampling = taking measurements of a signal

Below: 5 measurements ("samples") of \( f(x) \)
Audio file: stores samples of a 1D signal

(amplitude)

(time)

(most consumer audio is sampled 44,100 times per second, i.e., at 44.1 KHz)
Reconstruction: given a set of samples, how might we attempt to reconstruct the original signal $f(x)$?
Piecewise constant approximation

\[ \hat{f}(x) = \text{value of sample closest to } x \]
Piecewise linear approximation

\( \hat{f}(x) = \text{linear interpolation between values of two closest samples to } x \)
How can we represent the signal more accurately?

Sample signal more densely
(increase sampling rate)
Reconstruction from denser sampling

- reconstruction via linear interpolation
- reconstruction via nearest
2D Sampling & Reconstruction

- Basic story doesn’t change much for images:
  - sample values measure image (i.e., signal) at sample points
  - apply interpolation/reconstruction filter to approximate image

![Original Image](image_url)
![Piecewise Constant](image_url)
![Piecewise Bi-linear](image_url)

original

piecewise constant ("nearest neighbor")

piecewise bi-linear
Sampling 101: Summary

- Sampling = measurement of a signal
  - Encode signal as discrete set of samples
  - In principle, represent values at specific points (though hard to measure in reality!)

- Reconstruction = generating signal from a discrete set of samples
  - Construct a function that interpolates or approximates function values
  - E.g., piecewise constant/“nearest neighbor”, or piecewise linear
  - Many more possibilities! For all kinds of signals (audio, images, geometry…)

[Image credit: Wikipedia]
For rasterization, what function are we sampling?

\[
\text{coverage}(x, y) := \begin{cases} 
1, & \text{triangle contains point } (x, y) \\
0, & \text{otherwise}
\end{cases}
\]
Simple rasterization: just **sample** the coverage function

Example: Here I chose the coverage sample point to be at a point corresponding to the pixel center.

\[ (x+0.5, y+0.5) \]

\[ \text{Pixel } (x,y) \]

\[ \text{Example: Here I chose the coverage sample point to be at a point corresponding to the pixel center.} \]

\[ \text{= triangle covers sample} \]

\[ \text{= triangle does not cover sample} \]
Edge cases (literally)

Is this sample point covered by triangle 1? or triangle 2? or both?
Breaking Ties*

- When edge falls directly on a screen sample point, the sample is classified as within triangle if the edge is a “top edge” or “left edge”
  - Top edge: horizontal edge that is above all other edges
  - Left edge: an edge that is not exactly horizontal and is on the left side of the triangle. (triangle can have one or two left edges)

*These are the rules used in OpenGL/Direct3D, i.e., in modern GPUs. Source: Direct3D Programming Guide, Microsoft
Results of sampling triangle coverage
I have a sampled signal, now I want to display it on a screen
Pixels on a screen

Each image sample sent to the display is converted into a little square of light of the appropriate color:
(a pixel = picture element)

* Thinking of each LCD pixel as emitting a square of uniform intensity light of a single color is a bit of an approximation to how real displays work, but it will do for now.
So if we send the display this:
We see this when we look at the screen
(assuming a screen pixel emits a square of perfectly uniform intensity of light)
But the real coverage signal looked like this!
Aliasing
Sampling & Reconstruction

Goal: reproduce original signal as accurately as possible.
1D signal can be expressed as a superposition of frequencies

\[ f_1(x) = \sin(\pi x) \]

\[ f_2(x) = \sin(2\pi x) \]

\[ f_4(x) = \sin(4\pi x) \]

\[ f(x) = f_1(x) + 0.75 f_2(x) + 0.5 f_4(x) \]
E.g., audio spectrum analyzer shows the amplitude of each frequency.
Aliasing in Audio

Get a constant tone by playing a sinusoid of frequency $\omega$:

Q: What happens if we increase $\omega$ over time?

$\omega(t) = 6000t$

Why did that happen?
Undersampling high-frequency signals results in **aliasing**

**Low-frequency signal**: sampled adequately for accurate reconstruction

**High-frequency signal**: insufficiently sampled; reconstruction appears to be from a low frequency signal

"Aliasing": high frequencies in the original signal masquerade as low frequencies after reconstruction (due to undersampling)
Images can also be decomposed into “frequencies”
Low frequencies only (smooth gradients)

Spatial domain result

Spectrum (after low-pass filter)
All frequencies above cutoff have 0 magnitude
Mid-range frequencies

Spatial domain result

Spectrum (after band-pass filter)
Mid-range frequencies

Spatial domain result

Spectrum (after band-pass filter)
High frequencies (edges)

Spatial domain result (strongest edges)

Spectrum (after high-pass filter)
All frequencies below threshold have 0 magnitude
An image as a sum of its frequency components
Spatial aliasing: the function $\sin(x^2 + y^2)$

Rings in center-left:
Actual signal (low frequency oscillation)

Right:
aliasing from undersampling high frequency oscillation makes it appear that rings are low-frequency (they’re not!)

Middle: ring frequency approaches limit of what we can represent w/ individual pixels

Figure credit: Pat Hanrahan and Bryce Summers
Temporal aliasing: wagon wheel effect

Camera’s frame rate (temporal sampling rate) is too low for rapidly spinning wheel.
Nyquist-Shannon theorem

- Consider a band-limited signal: has no frequencies above some threshold $\omega_0$
  - 1D example: low-pass filtered audio signal
  - 2D example: blurred image example from a few slides ago

- The signal can be perfectly reconstructed if sampled with period $T = 1 / 2\omega_0$
- ...and if interpolation is performed using a “sinc filter”
  - ideal filter with no frequencies above cutoff (infinite extent!)

$$\text{sinc}(x) = \frac{1}{\pi x} \sin(\pi x)$$
Challenges of sampling in computer graphics

- **Signals are often not band-limited in computer graphics. Why?**

**Hint:**

- Also, infinite extent of “ideal” reconstruction filter (sinc) is impractical for efficient implementations. Why?
Aliasing artifacts in images

- Imperfect sampling + imperfect reconstruction leads to image artifacts
  - “Jaggies” in a static image
  - “Roping” or “shimmering” of images when animated
  - Moiré patterns in high-frequency areas of images
How can we reduce aliasing?

- No matter what we do, aliasing is a fact of life: any sampled representation eventually fails to capture frequencies that are too high.

- But we can still do our best to try to match sampling and reconstruction so that the signal we reproduce looks as much as possible like the signal we acquire.

- For instance, if we think of a pixel as a “little square” of light, then we want the total light emitted to be the same as the total light in that pixel.
  - I.e., we want to integrate the signal over the pixel (“box filter”)

Let’s (approximately) integrate the signal coverage \((x,y)\) by sampling...
Initial coverage sampling rate (1 sample per pixel)
Increase frequency of sampling coverage signal
Resampling

Converting from one discrete sampled representation to another

Original signal (high frequency edge) → Dense sampling of reconstructed signal

Reconstructed signal (lacks high frequencies) → Coarsely sampled signal
Resample to display’s pixel resolution

(Because a screen displays one sample value per screen pixel...)

[Diagram showing pixel resampling process]
Resample to display’s pixel rate (box filter)
Resample to display’s pixel rate (box filter)
Displayed result (note anti-aliased edges)
Recall: the real coverage signal was this
Single Sample vs. Supersampling

single sampling

2x2 supersampling
Single Sample vs. Supersampling

single sampling

4x4 supersampling
Single Sample vs. Supersampling

single sampling

32x32 supersampling
Checkerboard — Exact Solution

In very special cases we can compute the exact coverage:

Such cases are extremely rare—want solutions that will work in the general case!

See: Inigo Quilez, “Filtering the Checkerboard Pattern” & Apodaca et al, “Advanced Renderman” (p. 273)
How do we actually evaluate coverage(x,y) for a triangle?
Point-in-triangle test

Q: How do we check if a given point q is inside a triangle?

A: Check if it’s contained in three half planes associated with the edges.
Point-in-triangle test

Q: How do we check if a given point q is inside a triangle?

A: Check if it’s contained in three half planes associated with the edges.
Point-in-triangle test

Q: How do we check if a given point $q$ is inside a triangle?

A: Check if it’s contained in three half planes associated with the edges.
Point-in-triangle test

Q: How do we check if a given point $q$ is inside a triangle?

A: Check if it’s contained in three half planes associated with the edges.
Point-in-triangle test

Q: How do we check if a given point \( q \) is inside a triangle?

A: Check if it’s contained in three half planes associated with the edges.

Half plane test is then an exercise in linear algebra/vector calculus:

**GIVEN**: points \( P_i, P_j \) along an edge, and a query point \( q \)

**FIND**: whether \( q \) is to the “left” or “right” of the line from \( P_i \) to \( P_j \)

(Careful to consider triangle coverage edge rules...)
Traditional approach: incremental traversal

Since half-plane check looks very similar for different points, can save arithmetic by clever “incremental” schemes.

Incremental approach also visits pixels in an order that improves memory coherence: backtrack, zig-zag, Hilbert/Morton curves, …
Modern approach: parallel coverage tests

- Incremental traversal is very serial; modern hardware is highly parallel

- Alternative: test all samples in triangle “bounding box” in parallel

- Wide parallel execution overcomes cost of extra tests (most triangles cover many samples, especially when super-sampling)

- All tests share some “setup” calculations

- Modern graphics processing unit (GPU) has special-purpose hardware for efficiently performing point-in-triangle tests

Q: What’s a case where the naïve parallel approach is still very inefficient?
Naïve approach can be (very) wasteful...
Hybrid approach: tiled triangle traversal

Idea: work “coarse to fine”:
- First, check if large blocks intersect the triangle
- If not, skip this block entirely (“early out”)
- If the block is contained inside the triangle, know all samples are covered (“early in”)
- Otherwise, test individual sample points in the block, in parallel

This how real graphics hardware works!
Can we do even better for this example?
Hierarchical strategies in computer graphics

Q: Better way to find finest blocks?  
A: Maybe: incremental traversal!
Summary

- Can frame many graphics problems in terms of **sampling** and **reconstruction**
  - sampling: turn a **continuous** signal into **digital** information
  - reconstruction: turn **digital** information into a **continuous** signal
  - **aliasing** occurs when the reconstructed signal presents a false sense of what the original signal looked like
- Can frame rasterization as sampling problem
  - sample coverage function into pixel grid
  - reconstruct by emitting a “little square” of light for each pixel
  - aliasing manifests as jagged edges, shimmering artifacts, …
  - reduce aliasing via supersampling
- Triangle rasterization is basic building block for graphics pipeline
  - amounts to three half-plane tests
  - atomic operation—make it fast!
  - several strategies: incremental, parallel, blockwise, hierarchical…
Next time: 3D Transformations