Lecture 1 - Sep 5,2006

Questions to Start With

- Why are you here?
- What is Computer Graphics?

- What do you want to get out of it?
- What do you expect?
- What have you heard?

- Mechanics – 75 minute lecture too long
  - TRY for a break
- Do not want to blow a lecture on mechanics

Topics du Jour

- What is Computer Graphics – the topic
- What is Computer Graphics – the class
- Some basic things to get started

What is Computer Graphics?

- How computers create things we see

What kinds of "things we see"

- What?
- Computer Displays
- Movies / Video
- Print
- Interactive Media
  - Games
  - Virtual Reality
- Other devices (mobile)
- …

- Why?
- Computer Displays
- Entertainment
- Design
- Communication
- Simulation
- Medicine / Science

What is computer graphics?

(almost) Any picture we see!

and a lot more than "computer pictures."

Computers touch everything …

- All movies
- Photography (even film is printed digitally)
- Print
- …

What do we see?
What is an Image?

- Basics of Light
  - Electromagnetic radiation
    - Waves, frequencies (later)
  - Particle model
    - Travels from source to receiver
- Source to Viewer?
  - Not known until around 1000
    - Euclid and Ptolemy PROVED otherwise
  - Ibn Al-Haythan (Al-hazen) around 985
    - Triumph of the scientific method
      - Proof by observation – not authority
    - Experiment – stare at sun, burns eyes, …
    - Also figured out light travels in straight lines

Depth and Distance

- Light travels in straight lines
  - Except in weird cases that only occur in theoretical physics
- Doesn't matter how far away
  - Can't tell where photon comes from
  - Photons leaving source might not all make it to eye
  - Photons might bounce around on stuff
    - Longer distance, more chance of hitting something

Looking at things

- Light leaves source
- Light bounces off object
- Light goes to receiver
  - Eye, Camera
- Receiver is 2D, process is 3D
- Mathematics later

- Could be a picture (per eye)

What is Computer Graphics?

- Images - Visual Computing
- Geometry - Geometric Computing
  - Probably turned into an image at some point
- Not just pictures of world (text, painting, …)

Images

- Dictionary: a reproduction of the form of a person or object, especially a sculptured likeness
- Math: the range of a function
- A picture (2D)
- A sampled representation of a spatial thing

How to make images?

- Represent 3D World & Make a picture
  - Rendering (act of making a picture from a model)
  - Either simulate physics or other ways
- Capture measurements of the real world
- Make up 2D stuff (like painting text, …)

Kinds of Image Representations

- Old: Raster vs. Vector
- New: Sampled vs. Geometric
- Raster: regular measurements (independent of content)
- Geometric: mathematical description of content
- Display: vector vs. raster

Pixels

- A little square?
  - Bad model – but right idea
- A measurement (at a point)
  - In theory a point – in practice could be average over a region, …
  - Limited precision…
- Grid? (or any pattern)
  - Key point: independent of content

What is the field of Graphics?

(as far as we're concerned as a part of CS)

- Not content
- Not how to use graphics tools (***)

Related Fields / Courses

- Art
- Image Processing
- Computational Geometry
- Geometric Modeling
- Computer Vision
- Human Perception
- Human-Computer Interaction
- Advanced Graphics

What do you need to know?

- About images
- About geometry
- About 3D
- Importance of images in graphics classes
  - A new thing
  - Not well reflected in texts

What will we try to teach you?

- Eyes and Cameras – where images go
- Images (sampling, color, image processing)
- Drawing and representing things in 2D
  - Raster algorithms, transformations, curves, …
- Drawing and representing things in 3D
  - Viewing 3D in 2D, surfaces, lighting
  - Making realistic looking pictures
- Miscellaneous topics

How will we teach this to you?

- CS559 – Computer Graphics
- Basic course info – its all on the web
  www.cs.wisc.edu/~cs559-1
- Web for announcements – issues with mailing lists

Who

- Prof: Mike Gleicher
- 6385 CS
- Office Hours:
  - Tuesday: after class (11:00-11:45)
  - Wednesday 9:45-10:30
  - NOT Thursday
- gleicher@cs.wisc.edu

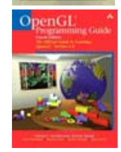
- TA: Yu-Chi Lai Mohamed Eldawy

- See the website

Books

- Fundamentals of Computer Graphics, 2nd ed
  - By Peter Shirley (and others)
  - NOT the 1st edition
  - Referred to as Shirley
  - or Tiger Book

- OpenGL Programming Guide
  - By Woo et al.
  - "red book" – common reference
  - Any version is OK for class
    - Old version is on the web



Collaboration

- Collaboration vs. Academic Misconduct

- We encourage collaboration (to a point)
  - Not on exams
  - You must do your own project work

Parts of the Course

- Exams

- Projects
- Assignments
  - Programming
  - Written

- **Something** due every Tuesday (start next week)

Software Infrastructure

- Visual Studio (C++ on Windows)
  - Your program must compile and run on machines in B240!

- FlTk
- OpenGL

- Class is not about tools, but we will help you with them

Other Administrative Questions?

- C++

- Workload

- Extra Credit

- Grading and Late Policies

CS559 – Lecture 2
Lights, Cameras, Eyes

Last time:
- what is an image
- idea of “image-based” (raster representation)

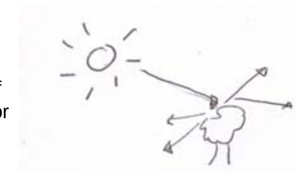
Today:
- image capture/acquisition, focus
- cameras and eyes
- displays and intensities

Corrected Notes

Not used as slides in class

Getting light to “imager”

- Light generally bounces off things in all directions
  - See from any direction
  - Not the same! (mirror)
  - Deal with this in detail later
- Generally doesn’t matter if emitter (source) or reflector
  - Same to receiver

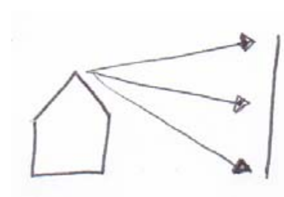


Depth and Distance

- Light travels in straight lines
  - Except in weird cases that only occur in theoretical physics
- Doesn’t matter how far away
  - Can’t tell where photon comes from
  - Photons leaving source might not all make it to eye
  - Photons might bounce around on stuff
    - Longer distance, more chance of hitting something

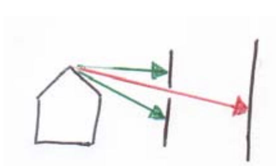
Capturing Images

- Measure light at all places on the “imaging plane”?
- Not quite…
- Potentially all paths between world and imager
  - Need to be picky about which rays we look at



“Ideal Imaging”

- Each point in world maps to a single point in image
  - Image appears sharp
  - Image is “in focus”
- Otherwise image is “blurry”
  - Image is out of focus
- How to do this?
- Pinhole Camera
  - Infinitessimal hole in blocking surface – just a point
  - Only 1 path from world point to image
  - Focal Point

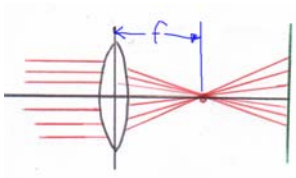


Why is pinhole imaging not so ideal in practice?

- Finite aperature
  - Always will be some blurryness
- Too selective about light
  - Lets very little light
- Smaller aperature
  - Less blurry
  - Less light
- Want bigger aperature, but keep sharpness

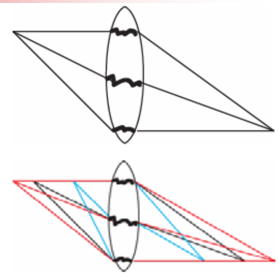
A “virtual pinhole” - Lenses

- Lens bends light
- convex lenses – take bundles of light and make them converge (pass through a point)
- Parallel rays converge
- A virtual pinhole!
- Light rays from “far away” are (effectively) parallel
- What about non-parallel rays?
- Infinitessimal aperature = infinite sharpness



“Thin” Lenses

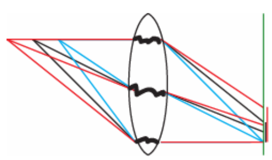
- All points at one distance get to another place
- Different distances map to different distances
- If we fix the distance to the image plane, then only objects at a single distance will be in focus
  - 1/D + 1/I = 1/F
  - Farther objects image closer



Picture is wrong – inverse relationship between I and D

Focusing with a lens

- Objects at “focussed distance” – sharp (in focus)
- Objects at other distances are not sharp
- Some blurryness is OK
  - Circle of Confusion
- Depth of Field
  - Range of distances that things are “close enough” to being in focus



Controlling the image

- Smaller aperature = less blurry = larger depth of field
  - But less light
- Lens determines
  - What gets to the imaging surface
  - What is in focus

Measuring on the image plane

- Want to measure / record the light that hits the image plane
- At every position on the image plane (in the image) we can measure the amount of light
  - Continuous phenomenon (move a little bit, and it can be different)
  - Can think of an image as a function that given a position (x,y) tells us the “amount” of light at that position
    i = f(x,y)
  - For now, simplify “amount” as just a quantity, ignoring that light can be different colors

Measuring on the image plane

- i = f(x,y)
- Continuous quantities
  - Continuous in space
  - Continuous in value
- Computers (and measuring in general) is difficult with continuous things
- Major issue
  - Limits to how much we can gather
  - Reconstruct continuous thing based on discrete set of observations
  - Manipulate discrete representations

Measuring on the image

- Water/rain analogy
- Put a set of buckets to catch water
- Wait over a duration of time
  - Use a shutter to control the amount of time
- Measurement depends on
  - Amount of light
  - Size of aperature (how much of the light we let through)
  - Duration

Types of “buckets”

- Film
  - silver halide crystals change when exposed to light
- Electronic
  - Old analog ways – vidicon tubes
    - Store the charge on a plate, scan the plate to read
    - http://www.answers.com/topic/video-camera-tube
  - New ways: use an MOS transistor as a bucket
- Biological
  - Chemicals (photo-pigments) store the photon and release it as electricity
  - Isn’t really a shutter

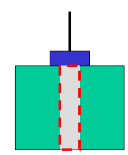
Similarities

- Low light levels are hard
  - Need to get enough photons to measure
  - Small counting errors (noise) – are big relative to small measurements
- Tradeoffs on bucket sizes
  - Big buckets are good (lower noise in low light)
  - Lots of buckets are good (sense more places)
  - For a fixed area, there is a tradeoff
    - Especially in digital cameras/videocameras

MOS Transistors

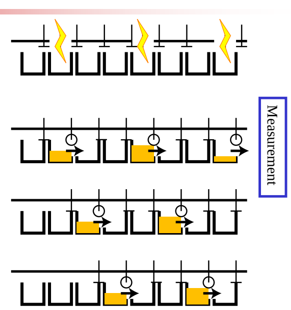
Not really discussed

- Metal Oxide Semiconductors
- Semiconductor acts as a “bucket” for electrons
- Metal at top is a “gate” – creates electric field that can connect/disconnect the two sides



CCD sensors

- CCD = Charge Coupled Device
  - “Bucket Brigade” of MOS transistors
  - Use gates to move charge along
  - Read out “at edge”
  - Shift register to transfer out images
- Advantage:
  - Cheap / easy to make large numbers of buckets
  - Uniform
- Blooming



CMOS sensors

Not really discussed

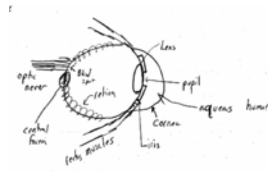
- Disadvantage of CCDs
  - Have to shift things out (slow, lose info)
  - Different than computer chips
- CMOS (complimentary Metal Oxide Semiconductor)
  - Just like computer chips
  - Put more circuitry around each sensor transistor
    - Amplify / switch signals as needed
  - Use normal “wires” to carry info to where it needs to go
- Downside: space for circuit means less space for sensors (smaller buckets = more noise), not uniform
- Upside: same “technology curve” as computers, so will get better, faster, cheaper, lower power, …

Digital Camera

- Megapixels = number of buckets
  - 7 or 8 million buckets in a consumer camera
- But…
  - How big are the sensors?
    - Same size / more megapixels = smaller buckets = more noise
    - (unless the sensor technology gets better)
  - How good is the lens?
    - Smaller buckets don't do you any good if the lens can't aim it into the right bucket

Eye

- Pupil – hole in the eye
- Lens
- Iris
  - round muscle – size of pupil
- Cornea
  - Clear protective coating
- Fluid filled spaces – acts as lens
  - Aqueous humor
  - Vitreous humor
- Rectus Muscles
  - Change shape of eyeball to focus
- Optic Nerve
  - Carries information away
- Blind Spot
  - Where the optic nerve is
- Central Fovea



Retina – the "image plane" of the eye

- Only place on body to see blood vessels directly
- Has photoreceptors
  - Cells sensitive to light
- Photopigments
  - chemicals that change when exposed to light
  - Different photoreceptors have different pigments
  - Different pigments behave differently
    - Sensitivity, color selectivity, regeneration speed
- Types of photoreceptors

Types of photoreceptors: Rods

- Photopigment: Rhodopsin
  - Breaks into retinene + protein
  - Must be reassembled before can work again
- Very sensitive
  - Bright light means that it breaks down faster than it is regenerated
  - Less useful in bright light
- Blinded by bright light at night

Cones

- Photopigments reform quickly
- Different types of cones sensitive to different kinds of light (color sensitivity)
  - Humans – 3 types of cones
    - Except for color blindness
  - Dogs – 1 type of cone
  - Many mammals (horses, cows, deer, …) – 2 types
  - Ducks, Pigeons - 5 types (?)
    - Birds range in number – European Starling 4
    - http://people.eku.edu/ritchisong/birdbrain2.html
- We'll talk about this more later

Persistence of Vision

- Photopigments take some time to regenerate
- If you see a flash, you sense it for a while afterwards
- This is NOT how you fuse movie frames together in order for it to seem continuous
  - This is actually hard psychological science that is not well understood
  - Integration happens as a higher level process in the brain
  - Many other effects

"Flicker-based Displays"

- If something flashes fast enough, it seems to be continuous
  - Flicker frequency – approx 40-45 hz in a dim/dark room
  - Sensitivity varies with age and ambient brightness
- Used to create different types of displays
  - CRT
  - Movies

How many megapixels is the eye?

- Density of photorecptors varies (see book)
- Dense area of cones = **fovea**
  - Eye moves the scene around, fovea looks at a little piece and over time gets the whole picture
  - Saccade – movement of the eye to see different piece
  - Fixation –
- Wide angle view means "resolution" hard to talk about – easiest to talk about in terms of angle
- Discriminate about ½ minute of arc (for 20/20 vision)
  - At .5 meters, this is .1mm

How sensitive is the eye?

- Amazing range!
  - Night vision – when eyes adjusted, camping
  - Bright daylight
    - `Sunlight 10000.`
    - `Twilight    10.`
    - `Starlight    0.001`
- Catch: at any given time, can't see this range
  - Adaptation – bright light, iris closes, lets in less light, …
- At any given time, about 100:1 contrast ratio
  - This is a lot more than most displays
  - Better displays = more constrast
    - Often by blacker blacks

High Dynamic Range Imagery

- Most sensors/displays have less range than eye
  - Certainly less range than scenes do
- What happens?
  - Bright areas – all white (no details)
  - Dark (shadow) areas – all black (no details)
- What to do?
  - Adjust exposure (time, aperature, sensitivity) to get the most important stuff
  - Acquire "High Dynamic Range" Imagery
    - Special sensors
    - Multiple exposures (at different settings) – cool thing to do
  - Tone Map -> display on device with less range
    - A chapter in the book we won't get to

Perception of intensity

- Eye senses relative differences
  - Equivalent differences 50:100 20:40
  - Hard to tell absolute differences directly
    - Adaptation to current setting
- Can sense 1% differences
- At any given time 100:1 contrast ratio
- How many levels can you see in an image?
  - 1.01 ^ 463 = 100.2 (e.g. 463 1% differences = 100:1)
  - This is about 8 bits of precision (less than 9)
  - But its VERY non linear 1, 1.01, …. , 99.2, 100.2

CS559 – Lecture 3 Part 1
Intensity, Quantization, Sampling

Non-linearity of intensity

- Non-linear mapping from "amount of light" to perceived brightness
- Want uniform mapping of intensities -> perception
  - Level 1, 2, 3, …. 255 -> 1, 1.01, 1.02, … 99, 100
- Worse: displays are non-linear too
  - Voltage -> amount of light is non-linear
  - Different displays are different
- Want to linearize the system
  - Intensity levels map nicely to perceived levels

Gamma correction

- Idea: put a non-linear function between intensity and output
  - Done as the last step (usually) – after all computations
- Could create arbitrary functions for mapping
  - Too cumbersome
- Exponential is a good approximate model
  - Exponential non-linearity of perception
  - Exponential power laws in CRTs

Modeling a display device

- 5/2 power law (five-halves)
  - Models physics of a CRT
  - Real CRTs are close, LCDs designed to be similar
- L = M (i+ε)^γ
  - i = input intensity value
  - L = amount of light
  - ε = since zero isn't really black
  - M = maximum intensity
  - γ = specific property of display

Linearizing the display

- Define a function **g** that corrects for non-linearity
- L = M (g(i))^γ (ignoring ε)
  - G = 1/ γ
- Where do we get γ from?
  - Pick it so things look right
- Note: 1st order approximation (very simple)
  - Only 1 parameter to specify (γ), many factors

Gamma correction

- Want value 0 = minimum intensity
- Want value max (1 or 255) = maximum intensity
  --- those 2 are easy to get
- Pick one more point
  - Midpoint should be 50%
  - Easy – show 50% black white + 50% gray
  - Adjust gamma until it looks the same
- All this happens "behind the scenes"
- Everything gets harder when we deal with color

What to store in the frame buffer?

- Frame Buffer = rectangular chunk of memory
- Intensity measurements
  - Deal with color later, basically store multiple monochrome
- Continuous range of intensities
  - 8-9 bits of precision ideally
    - More since can't get exactly right (10-12 bits)
    - More since want more dynamic range (12-14 bits)
    - More since want linear space to make math easy (16-32 bits)
- Discrete set of choices – **QUANTIZATION**
  - Inks, palettes, color tables, …
  - Less storage cost + Color table animation

Faking more "colors" than you have

- Eye tends to average stuff together
  - Trade spatial resolution for intensity resolution

Quantization

- What happens when we want smaller numbers of values?
  - Black and white for printing
  - Limited color pallete
- Old problem
  - Printing
  - Artists (pen and ink drawing)

Thresholding

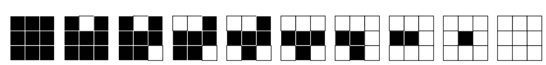
- Threshold – pick value / above or below
- Each pixel picks nearest value
  - 49% looks the same as 1%
  - 49% looks very different than 51%
- Better: trade spatial resolution for value resolution
  - Brain blurs stuff together anyway
  - Art example: hatching to show "gray"

Dithering

- Add some random noise
- 50% + noise -> half black, half white
- Values at extreme less likely to get changed
- Eye doesn't mind noise as much as it does blocky edges

Patterns

- Make display resolution greater than image resolution
- Each pixel gives a block w/appropriate number of pixels on
- For example: 3x3 blocks give 10 levels



Ordered Halftoning

- Do patterns, but apply for each pixel seperately (no scaling images)
- Divide image into nxn blocks (repeated pattern)
  - Each pixel decides if it would be turned on if its value was used to pick the pattern
- Easy implementation: Threshold Matrix or Mask
  - Used in traditional printing (a halftone screen)
  - Each pixel has a different threshold
  - Example: 4 values

0	2
1	3

More on Halftone screens

- Other factors can go into designs
- Cluster things together (since you know that ink tends to clump)
- Or make artistic effects
- Can be used with dithering (adding randomness)

Error Minimization

- For each pixel, compute error (how different from result)
  - Try to pick result to minimize error
- Global minimization: each pixel should equal average of destination image
  - Too hard to solve efficiently since it's a combinatorial problem
- Local minimization: each pixel should be as close as possible (thresholding) – but spread error around evenly

Error Diffusion

- Many ways to do this
- Old standby: Floyd-Steinberg
- Start at upper left
- Pick value for pixel
- Push error into neighbors (that haven't been visited yet)

d	3/8 e
3/8 e	1/8 e

	d	7/16 e
3/16 e	5/16 e	1/16 e

- Problem: directional artifacts (fix by alternating directions)
- Good news: generalizes (colors, multiple levels, …)

CS559 – Lecture 3 Part 2
Intensity, Quantization, **Sampling**

Continuous vs. Discrete

- Image is a continuous thing
- Can measure anywhere
  - F(x,y)
- Only really have a discrete set of points
  - Number of places to measure
  - Number of measurements to store
  - Number of “dots” to draw
- We are throwing away information
  - No choice – unless take infinite number of measurements, infinite storage, …

What is a pixel?

- Raster means regular, or uniform “grid”
- Two views of a pixel
  - A pixel is a POINT SAMPLE
    - Measurement at an infinitessimally small place
  - A pixel is finite region with constant value
    - Assumes image is collection of piecewise constant regions
- Point sample is better model
  - Constant regions are a special case of sampling filter
  - More correct, better mathematics, can model the other

Little squares lose differently

- Are squares better than point samples?
- Average over a little square
- But:
  - Don’t know what really happened
  - Was it really constant, or was it a spike?
- Good intuition for what is coming up

Point Sampling Has Problems

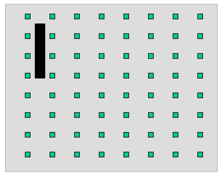
- Miss small things
- Problem: discretization throws away information
- Don’t know what happens between samples
- Sampling loses information – you cannot get back the information once its lost!

Dealing with discretization

- Sampling
  - Understand what information we are throwing away
- Reconstruction
  - Recreate as well as possible from the samples
- Re-Sampling
  - Transform the image
- Signal Processing / Image Processing
- Consider the 1D case first since its easier

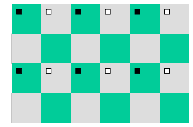
Bad sampling is bad

- Miss small things between samples

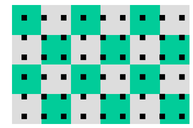


Get really weird results

- Sample a checkerboard



- Too few samples
  - Get all black
  - Get all white
  - Get weird patterns
    - Aliasing
    - Moire'

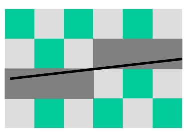


  - Arbitrary algorithm decision gives very different answers!
- Imagine resampling

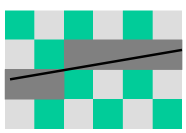
Demonstration ratios: 4/6 (here) = 2/3

Ugly

- Imagine line drawing



- Jaggies
- Crawlies
  - Small change causes jump
  - Smooth motion becomes jumpy

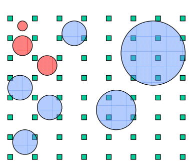


Dealing with discretization

- Sampling
  - Understand what information we are throwing away
- Reconstruction
  - Recreate as well as possible from the samples
- Re-Sampling
  - Transform the image
- Signal Processing / Image Processing
- Consider the 1D case first since its easier

Intuition

- Too few samples = BAD



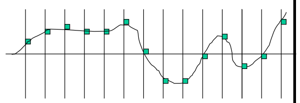
- Sampling rate depends on the thing you're sampling
- Need to sample close enough to get smallest object
- Need to limit small objects to be big enough that they aren't missed

A different intuition

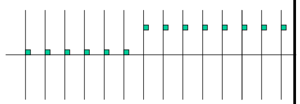
- Not really point sampling
  - Mesurements average over a finite range
  - Displays make finite dots
- Need to model these
  - Sampling filters, reconstruction filters
  - Averages over regions -> Convolution (generalized)
- Need to be realistic about what they mean
  - Can't see everything (too small, …)
- Sampling theory gives a nice mathematics for this!

Point sampling in 1D

- Only record samples
- Don't know what happens in between samples

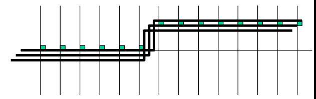


- Given the samples, don't know what really happened!

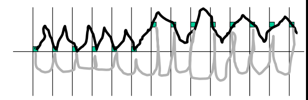


Reconstruction from Sampling

- Can't localize events
  - Bigger problems than that
- No idea! Signal could be anything



- Without additional information, we're guessing as to what the signal is



- But what additional info?

Sampling Intuitions

- Reconstruct the "smoothest" signal that makes sense from samples
- If signal is "smooth enough", sampling will give something we can reconstruct
- If signal is not "smooth", sampling will give something that will reconstruct to something else
  - Aliasing
- But how do we define "smooth"

Signal processing

- Need better "language" for talking about signals
- Idea: represent signals in a different way
- Up till now: time domain (graph against time)
  - Good for asking "what does signal do at time X"
- New idea: frequency domain
  - Good for talking about how smooth signals are
- Different view of the same thing

Frequency Domain

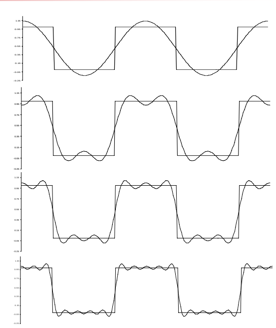
- Fourier Theorem:
  - Any periodic signal can be represented as a sum of sine and cosine waves with harmonic frequencies
  - If one function has frequency *f*, then its harmonics are function with frequency *nf* for integer *n*
  - Extensions to non-periodic signals later
  - Also works in any dimension (e.g. 2 for images, 3, …)
- Example: box

Example: Box (Square Wave)

- 1 cosine – bad
- More cosines, better approx

$$f(x) = \begin{cases} 1 & |x| \le \tfrac{1}{2} \\ 0 & |x| > \tfrac{1}{2} \end{cases}$$

$$S_{boxes}(x) = \frac{1}{2} + \frac{2}{\pi}\sum_{k=1}^{\infty}(-1)^{k-1}\frac{\cos(2k-1)\omega x}{2k-1}$$
$$= \frac{1}{2} + \frac{2}{\pi}\left(\cos\omega x - \frac{1}{3}\cos 3\omega x + \frac{1}{5}\cos 5\omega x - \cdots\right)$$



Intuitions

- Low frequencies are smooth
  - High frequencies change fast, are not smooth
- If a signal can be made of only low frequencies, it is smooth
- If a signal has sharp changes, it will require high frequencies to represent

General Functions

- A non-periodic function can be represented as a sum of sin's and cos's of (possibly) all frequencies:

$$f(x) = \frac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega)e^{i\omega x}d\omega$$
$$e^{i\omega x} = \cos\omega x + i\sin\omega x$$

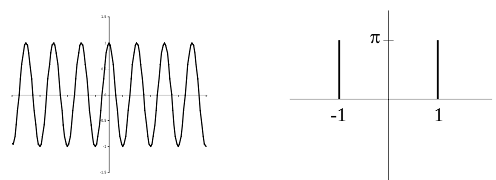
- *F(ω)* is the *spectrum* of the function *f(x)*
  - The spectrum is how much of each frequency is present in the function
  - We're talking about functions, not colors, but the idea is the same

Fourier Transform

- *F(ω)* is the Fourier Transform of f(t)
  - A different representation of the same signal
- To get f(t) back you use the Inverse Fourier Transform
- You don't need to know how to compute them

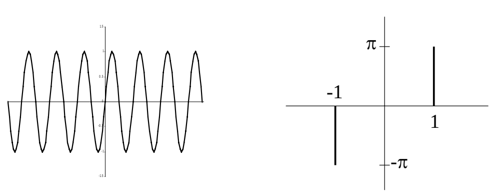
$$F(\omega) = \int_{-\infty}^{\infty} f(x)e^{-i\omega x}dx$$

Cosine and Its Transform



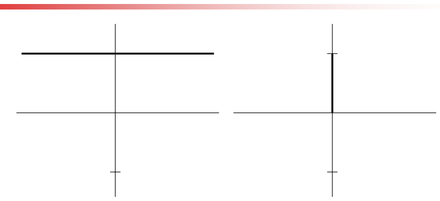
If *f(x)* is even, so is *F(ω)*

Sine and Its Transform



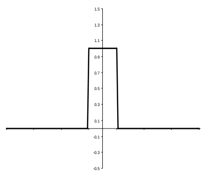
If *f(x)* is odd, so is *F(ω)*

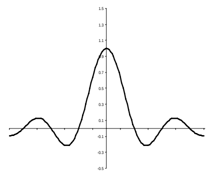
Constant Function and Its Transform



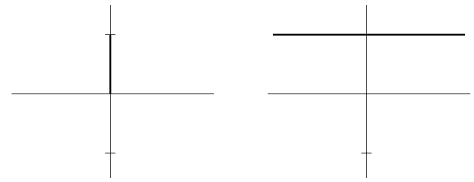
The constant function only contains the 0th frequency – it has no wiggles

Box Function and Its Transform



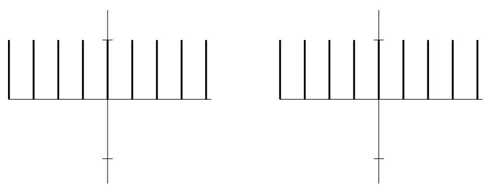


Delta Function and Its Transform

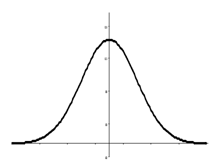


Fourier transform and inverse Fourier transform are qualitatively the same, so **knowing one direction gives you the other**

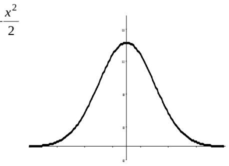
Shah (Spikes) and Its Transform



Gaussian and Its Transform



$$\frac{1}{2\pi} e^{-\frac{x^2}{2}}$$



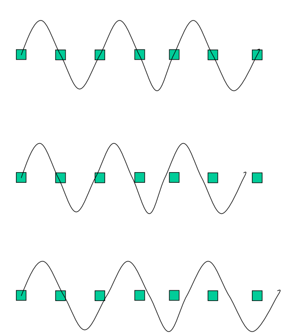
They are the same

Qualitative Properties

- The spectrum of a functions tells us the relative amounts of high and low frequencies
  - **Sharp edges give high frequencies**
  - **Smooth variations give low frequencies**
- A function is *bandlimited* if its spectrum has no frequencies above a maximum limit
  - sin, cos are band limited
  - Box, Gaussian, etc are not
- To band-limit a signal we *low-pass filter* it

Sampling Theorem (intuition)

- High frequencies get lost
  - Can only sample band limited signals
- Sampling rate must be 2 times higher than signal
- Signal must be half frequency of sample rate
  - Otherwise, signal can "turn around" between samples
- Nyquist rate
  - 2x highest frequency in signal



Sampling Theorem

- If your signal is bandlimited
- And you know what the band limit is
- And you sample at (at least) twice that frequency
  - Above the Nyquist rate
- Then – you can reconstruct your signal EXACTLY!

- Caveat
  - Ideal reconstruction requires perfect band limiting in both sampling and reconstruction

Sampling Theorem

- If your signal is bandlimited
- And you know what the band limit is
- And you sample at (at least) twice that frequency
  - Above the Nyquist rate
- Then – you can reconstruct your signal EXACTLY!

- Caveat
  - Ideal reconstruction requires perfect band limiting in both sampling and reconstruction

Need to know about convolutions

- We need to have band limited signals
  - Need low pass filters
  - Which are implemented as **convolutions**
- Reconstruction requires low-pass filtering
  - Which is implemented as **convolution**
- Need to see Sampling theory in Fourier domain
  - Need **convolution**

- **Convolution is the mathematical generalization of averaging**

Filtering: Convolutions

- A general filter is a function on an image that produces another image

- Many common filters are simpler in the Fourier domain
- Choice:
  - Transform image, filter, inverse transform image
  - Inverse transform operator, apply in spatial domain

  - Transform (or inverse) of multiplication is convolution

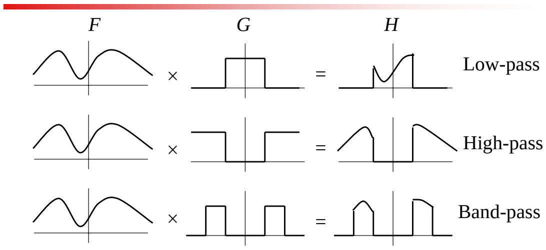
Filters

- A *filter* is something that attenuates or enhances particular frequencies
- Easiest to visualize in the frequency domain, where filtering is defined as multiplication:

$$H(\omega) = F(\omega) \times G(\omega)$$

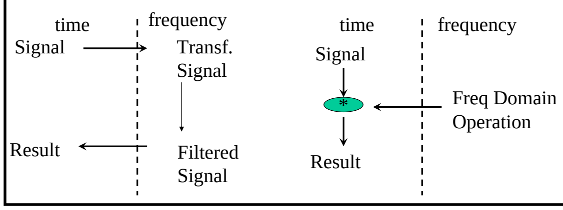
- Here, *F* is the spectrum of the function, *G* is the spectrum of the filter, and *H* is the filtered function. Multiplication is point-wise

Qualitative Filters



Can you transform an operator?

- Many filters are multiplication in frequency domain
- Fourier transform of multiplication is convolution
- Fourier transform of convolution is multiplication



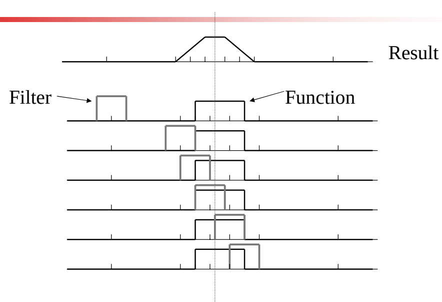
Filtering in the Spatial Domain

- Filtering the spatial domain is achieved by *convolution*

$$h(x) = f \otimes g = \int_{-\infty}^{\infty} f(u) g(x-u) du$$

- Qualitatively: Slide the filter to each position, x, then sum up the function multiplied by the filter at that position

Convolution Example



Convolution Theorem

- Convolution in the spatial domain is the same as multiplication in the frequency domain
  - Take a function, f, and compute its Fourier transform, F
  - Take a filter, g, and compute its Fourier transform, G
  - Compute $H=F\times G$
  - Take the inverse Fourier transform of H, to get h
  - Then $h=f\otimes g$
- Multiplication in the spatial domain is the same as convolution in the frequency domain

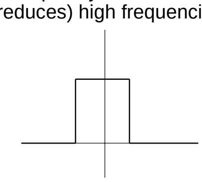
Filtering Images

- Work in the discrete spatial domain
- Convert the filter into a matrix, the *filter mask*
- Move the matrix over each point in the image, multiply the entries by the pixels below, then sum
  - eg 3x3 box filter
  - Effect is averaging

$$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

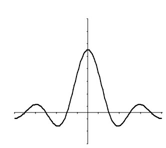
Box Filter

- Box filters smooth by averaging neighbors
- In frequency domain, keeps low frequencies and attenuates (reduces) high frequencies, so clearly a low-pass filter



Spatial: Box

$$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$



Frequency: sinc

Filter Widths

- Fourier Transform of a Time scaling:
  - f(k t) -> F(1/k omega)
  - As time gets scaled, frequency gets scaled by the inverse
- Box filter: wider box in frequency domain = narrower filter in time domain

- To filter higher frequencies use a narrow (in time/space) filter
- Lower Frequency cutoff (in a High-pass filter), you use a bigger (in time/space) filter

Handling Boundaries

$$I_{output}[x][y] = \sum_{i=-k/2}^{k/2} \sum_{j=-k/2}^{k/2} I_{input}[x+i][y+j] M[i+k/2][j+k/2]$$

- At (0,0) for instance, you might need pixel data for (-1,-1), which doesn't exist
- Option 1: Make the output image smaller – don't evaluate pixels you don't have all the input for
- Option 2: Replicate the edge pixels
  - Equivalent to: posn = x + i; if (posn < 0) posn = 0; and so on for other indices
- Option 3: Reflect image about edge
  - Equivalent to: posn = x + i; if (posn < 0) posn = -posn; and similar for others

Seperable Filters

- Some 2D filters can be implemented as 2 1D filters

- Each dimension at a time

- Much easier
  - Don't need to build 2D filter kernel
  - Much faster (O(mn) not O(m^2 n))

- Box filters are seperable
- Other 2D filters are designed by seperated pieces

Constructing Masks: 2D

- Multiply 2 1D masks together using *outer product*

$$M[i][j] = m[i]m[j]$$

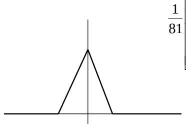
- *M* is 2D mask, *m* is 1D mask

	0.2	0.6	0.2
0.2	0.04	0.12	0.04
0.6	0.12	0.36	0.12
0.2	0.04	0.12	0.04

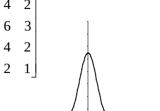
Bartlett Filter

- Triangle shaped filter in spatial domain
- In frequency domain, product of two box filters, so attenuates high frequencies more than a box

$$\frac{1}{81}\begin{bmatrix} 1 & 2 & 3 & 2 & 1 \\ 2 & 4 & 6 & 4 & 2 \\ 3 & 6 & 9 & 6 & 3 \\ 2 & 4 & 6 & 4 & 2 \\ 1 & 2 & 3 & 2 & 1 \end{bmatrix}$$



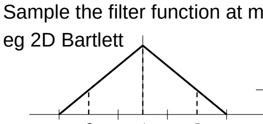
Spatial: Triangle (Box⊗Box)



Frequency: sinc^2

Constructing Masks: 1D

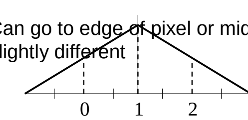
- Sample the filter function at matrix "pixels", then normalize
- eg 2D Bartlett



$\rightarrow \frac{1}{5}$

1	3	1

- Can go to edge of pixel or middle of next: results are slightly different



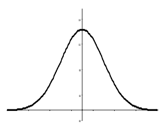
$\rightarrow \frac{1}{4}$

1	2	1

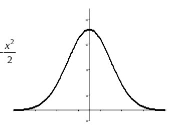
Gaussian Filter

$$\frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

- Attenuates high frequencies even further
- In 2d, rotationally symmetric, so fewer artifacts



$$\frac{1}{2\pi}e^{-\frac{x^2}{2}}$$



Constructing Gaussian Mask

- Use the binomial coefficients
  - Central Limit Theorem (probability) says that with more samples, binomial converges to Gaussian

$\frac{1}{4}$

1	2	1

$\frac{1}{16}$

1	4	6	4	1

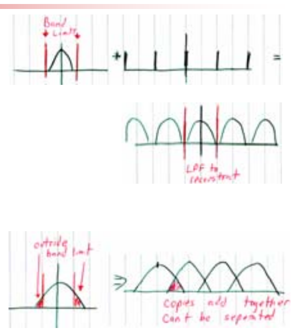
$\frac{1}{64}$

1	6	15	20	15	6	1

```
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
```

Sampling Theory

- Sampling is multiply by spike chain in time domain
  - Fourier transform of spike chain is spike chain
  - Fourier transform of multiply is convolution
- Sampling is convolution by spike chain in frequency
- Makes infinite copies of signal
- Reconstruction low-pass filters to remove all but one
- Non-band limited, things "spill"



Sampling / Reconstruction

- Both sampling and reconstruction require
  Low Pass Filtering
- Sampling:
  - Low pass filter signal to make sure is band-limited
- Reconstruction:
  - Low pass filter spike chain to figure out what happens between samples
- Resampling:
  - Reconstruction followed by sampling

Resizing = Resampling

- Same image – different number of samples
- Issues:
  - New samples are in between old samples
  - Too few new samples to capture all the frequency
- Basic idea (in theory)
  - Reconstruct original signal (LPF the samples)
  - Low-pass filter (so sampling works)
  - Sample at new sampling rate

Resampling – Little Square Model

- Region of source = Region of Dst
- Pixel is a region
  - Dest region might be bigger than pixel in source
  - Average over the region (convolution gives us the weights)
- In-between pixels is piecewise constant
  - Chunky look is what the model says is right

Pre-Filtering

- If SRC is bigger than DST it may have HF
  - If its close, might need it anyway because of imperfect reconstruction
- Need to LPF

- LPF before sampling?
  - Requires you to do a complete reconstruction
  - Only really need to do it at points you will sample
- Pre-Filtering
  - Do LPF before reconstruction / as part of reconstruction
  - Order is OK (convolutions commute)

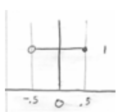
Reconstruction in Practice

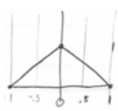
- Sample a sample – no problem!
- Issue is samples between samples

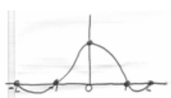
- Theory: LPF a spike chain
  - Convolve “resonstruction kernel” with samples
  - Only really need to evaluate at places where you’ll sample

- Another view: interpolation
  - Different interpolations are different filters

Some reconstruction kernels


Constant


Triangle (Bartlet)


Interpolating Cubic (Catmull-Rom)

Spacing (1 unit = sample distance)

Scaling issues

Interpolating (non-interpolating kernels exist as well)

Approx to Ideal LPF

Reconstruction Example

- Sample at sample
- Sample between samples

- Bartlett filter
  - Width correct for sample spacing
- See how we get linear interpolation

- Could do this as linear interpolation
  - Generalizes nicely this way
- Need to evaluate filter for various values

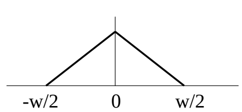
- Convolve reconstruction kernel with sampling kernel (LPF for frequency limit)

- Easier ways to implement nearest neighbor

Functional Form for Filters

- Consider the Bartlett in 1D:

$$H_w(s) = \frac{2}{w}\left(1 - \frac{2|s|}{w}\right)$$



- To apply it at a point x_{orig} and find the contribution from point x where the image has value $I(x)$

$$f(x) = \frac{2}{w}\left(1 - \frac{2|x - x_c|}{w}\right) I(x)$$

- Extends naturally to 2D:

$$f(x,y) = \frac{4}{w^2}\left(1 - \frac{2|x - x_c|}{w}\right)\left(1 - \frac{2|y - y_c|}{w}\right) I(x,y)$$

General Resampling

- Could be any transformation on x,y
- X’,y’ = f(x,y)

- Scale, translate, rotate, something weird

- Kernel should get warped too
  - Little square -> some weird shape
  - Little circle/square (of kernel) -> some weird shape
  - In practice, stick with squares

Reverse Warping

- Note we generally need the INVERSE:
  - X', y' = f(x,y) (x' = dst, x = src)
  - Know x', need to find x is inverse
- Reverse warping is easier (scan over each pixel in the dst, figure out where it comes from)
- Forward warping is tricker
  - Usually can invert function, but if you can't
  - Need to worry about holes
- Lots of fun warps to do!

①

NOTES FOR 9/19/2006

Reconstruction Example (from students)

 gives what?

assume that it was properly sampled
⇒ orig signal has sin waves only greater than 2
samples

 is impossible (this F was cut out)

only answer = 

Reconstruction

ideally low-pass filter
↳ can't really implement LPF

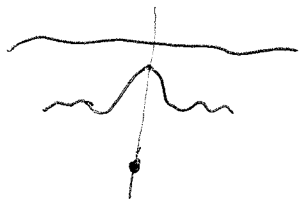
interpolation → filtering
- linear interpolation ⇒ tent filter
  show convolution gives same
- nearest neighbor interpolation ⇒ box filter
  show convolution gives the same
- interpolating cubic

non-interpolating kernels
scaling issues

9/19/2006

Sampling

Prefiltering - convolve w/ LPF



- not really an LPF
  hard to approximate
  ringing

Gaussian → Binomial (skip to p4)

Resampling

What we really do often
- change number of samples (resize)
- move the samples around (warp - later)

resample to half size:

steps: reconstruct $* r$ ← ~~sampling~~ reconstruct kernel

sample $* s$ ← sampling kernel

point sample

$f * r * s$ ← can put these together into 1

Since we only need to sample at certain places reconstruction might be easy (have samples)

since the signal might already be sufficiently band-passed, sampling might be easy

only 1 filter is really active
- apply 2 ideal LPF ⟹ applying lowest one

9/19/2006

3

Scale down by half - ~~no need~~

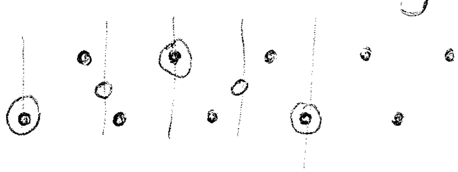
pick sampling kernel LPF w/
cutoff > 2 cycles
Binomial Approx: $\frac{1}{4}[1\ 2\ 1]$
$\frac{1}{16}[1\ 4\ 6\ 4\ 1]$

still just need some points
1. apply pre-filter everywhere ← fast because uniform
2. only pre-filter at samples ← fast because less work

Double

already bandpassed
(if < F, < 2F)
just need reconstruct

Scale Down by 1.5



- need some sampling filter
  $\frac{1}{4}$ 1 2 1 might be too much
  ~~k~~ 0 1 0 too little,
  take halfway ?
  $\frac{1}{8}\ \frac{3}{4}\ \frac{1}{8}$
- need to reconstruct ?
  not really - just evaluate in continuous case

(4)

Intuitions on Filter Kernels ....

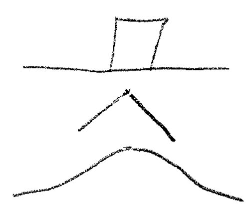
narrower box = wider sinc

extreme: spike = constant

problems w/ Sync

- infinite extent
- negative values → ringing
- hard to sample

approximators → Bunch of filters in the book § 4.3



box

tent (bartlett)

gaussian (still infinite)

binomial / B-Spline

convolutions of box

interpolating cubic

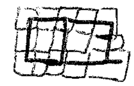
continuous	discrete
	$\frac{1}{2}$ 1 1 $\frac{1}{4}$ 1 2 1 $\frac{1}{8}$ 1 3 3 1 $\frac{1}{16}$ 1 4 6 4 1 ⋮

Wide enough to cover

5

Into 2D!

Little Square Model (for intuition, its wrong)

 old pixels
new pixels  or 

Everything from 1D still works
convolutions hard to draw on Board

Details of convolution
- for infinite extents
  - makes signal bigger ← or *just forget new stuff
- dealing w/ boundaries
  - 0 pad / edge replication
  - reflection
  - normalization
- 0 centering

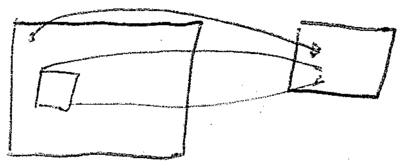
Seperable Filters

QUALITATIVE CONVOLUTIONS
- LPF
- HPF
- Shift
- Shadow

1

9/21/2006

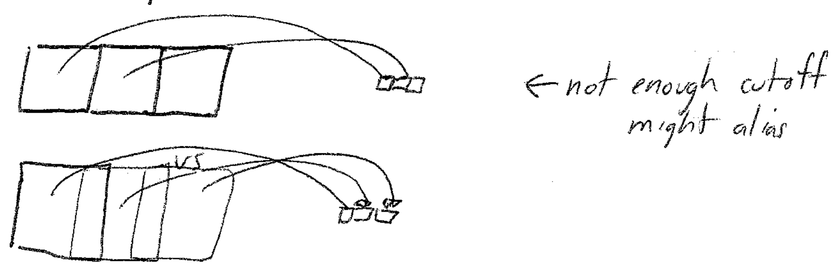
Resize as Warp



$x' y' = f(x,y)$

pixel → some region
point sample ← some region
inverse / forward splatting ← way to look at convolution

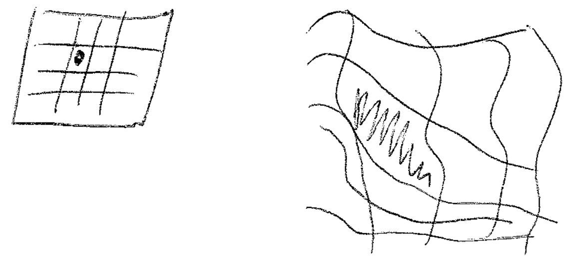
What does this say about kernel size?
need to be big enough to cover everything (downsample)
why not no overlap?



9/21/2006
2

Image Warping

$x', y' = f(x,y)$ ← resize is special case



pixel → some region

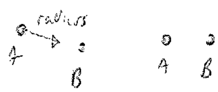
some region ← pixel

forward warp - splat

revese warp - sample (need f^{-1})

How to filter?

1. don't (point sample)
2. size of kernel = size of area mapped
   1. derivative
   2. differences



   3. capture shape
      (gaussian/circle ⇒ ellipse, or ...)
   4. super sample

      multiple samples per pixel
      uniform resampling is easy
5. map little squares

CS559 – Lecture 6 (part b)
Raster Algorithms

These are course notes (not used as slides)
Written by Mike Gleicher, Sept. 2005
With some slides adapted from the notes of Stephen Chenney

Geometric Graphics

- Mathematical descriptions of sets of points
  - Rather than sampled representations
- Ultimately, need sampled representations for display
- Rasterization
- Usually done by low-level
  - OS / Graphics Library / Hardware
  - Hardware implementations counter-intuitive
    - Modern hardware doesn't work anything like what you'd expect

Drawing Points

- What is a point?
  - Position – without any extent
  - Can't see it – since it has no extent, need to give it some
- Position requires co-ordinate system
  - Consider these in more depth later
- How does a point relate to a sampled world?
  - Points at samples?
  - Pick closest sample?
  - Give points finite extent and use little square model?
  - Use proper sampling

Sampling a point

- Point is a spike – need to LPF
  - Gives a circle w/roll-off
- Point sample this
- Or…
  - Samples look in circular (kernel shaped) regions around their position
- But, we can actually record a unique "splat" for any individual point

Anti-Aliasing

- Anti-Aliasing is about avoiding aliasing
  - once you've aliased, you've lost
- Draw in a way that is more precise
  - E.g. points spread out over regions
- Not always better
  - Lose contrast, might not look even if gamma is wrong, might need to go to binary display, …

Line drawing

- Was really important, now, not so important
- Let us replace expensive vector displays with cheap raster ones
- Modern hardware does it differently
  - Actually, doesn't draw lines, draws small, filled polygons
- Historically significant algorithms

Line Drawing (2)

- Consider the integer version
  - (x1,y1) -> (x2,y2) are integers
  - Not anti-aliased (binary decision on pixels)
- Naïve strawman version:
  - Y = mx + b

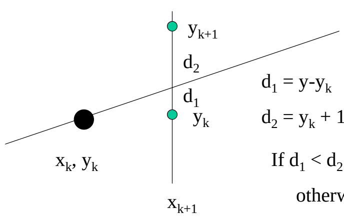
```
For x = x1 to x2
    y = mx + b
    set(x,y)
```

- Problems:
  - Too much math (floating point)
  - gaps

Brezenham's algorithm (and variants)

- Consider only 1 octant (get others by symmetry)
  - 0 >= m > = 1
- Loop over x pixels
  - Guaruntees 1 per column
- For each pixel, either move up 1 or not
  - If you plotted x,y then choose either x+1,y or x+1,y+1
  - Trick: how to decide which one easily
  - Same method works for circles (just need different test)
- Decision variable
  - Implicit equation for line (d=0 means on the line)

Midpoint method



$d_1 = y - y_k$

$d_2 = y_k + 1 - y$

If $d_1 < d_2$ pick y_k

otherwise pick y_k+1

If $d_1 - d_2 < 0$ pick y_k

^^^--- Δd

Derivation

$\Delta d = d_1 – d_2$

$\Delta d = (y-y_k) – (y_k+1-y)$

$y = m(x_k+1) + b$

$\Delta d = 2 (m (x_k+1) + b) – 2 y_k – 1$

$m = \Delta y / \Delta x$

Multiply both sides by Δx (since we know its positive)

$\Delta d\Delta x = 2\Delta y\ x_k + 2\Delta y + 2b\Delta x – 2\Delta x\ y_k – \Delta x$

$P_k = \Delta d\Delta x = 2\Delta\ y\ x_k + 2\Delta x\ y_k + c$

$c = 2\Delta y + \Delta x(2b – 1)$

(all the stuff that doesn't depend on k)

Incremental Algorithm

- Suppose we know p_k – what is p_{k+1} ?
- $p_{k+1} = p_k + 2\Delta y – 2\Delta x\ (y_{k+1} – y_k)$
  - Since $x_{k+1} = x_k+1$
- And $y_{k+1} – y_k$ is either 1 or 0, depending on p_k

Brezenham's Algorithm

- P_k = 2 \Delta y + x
- $Y = y_1$
- For $X = x_1$ to x_2
  - Set X,Y
  - If $P_k < 0$
  - Y += 1
  - P_k += 2 Δ y – 2 Δ x
  - Else: P_k += 2Δ y

Why is this cool?

- No division!
- No floating point!
- No gaps!

- Extends to circles

- But…
  - Jaggies
  - Lines get thinner as they approach 45 degrees
  - Can't do thick primitives

CS559 – Lecture 7
Color

These are course notes (not used as slides)
Written by Mike Gleicher, Sept. 2005
With some slides adapted from the notes of Stephen Chenney

Sensing Color

- Different sensors have different sensitivities
  - Spectrum of sensor
  - Convolution with spectrum gives response
- Ideal photo sensor / real photo sensor
- Cameras – wide range sensor
  - Put filters in front of each CCD element
  - Different parts of spectra (R,G,B)
  - Bayer Mosaic (need to interpolate)
  - Foveon

Shift Gears: Color

- Color
- Quality of Light
  - Has a wavelength – not just an amount
- Can measure the spectrum of light
  - Graph wavelength vs. amount at the measurement
- Different spectra give different "color impressions"

Color Vision in Animals

- Rods = all the same
  - No color vision
- Cones = have different kinds
  - 1-chromat (can't see color) -> Dogs
  - bi-chromat (2 different types) -> large mamals
  - Tri-chromat -> humans ***
    - Color blindness = lack of 1 type
    - Rare genetics condition gives a 4^{th} type
  - Some birds have 4 or 5 types of cones
    - Ducks&Pigeons have 5, European starlings have 4

Colors

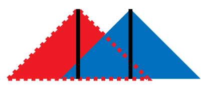
- One dominant wavelength = pure color
- No dominant wavelength = "white" (or black/gray)
- What do we perceive?
  - Luminence (amount of light)
  - Color (dominant)
  - Purity of Color
- Complications
  - Differences in perception
  - Artist notions vs. physics vs. psychology

Distinguishing colors

- 1 sensor
  - All colors look the same
  - Combination of colors looks like any color
- Metamers – perceptually indistinguishable
- 2 sensors
  - Non-overlap case (what differences?)
  - Overlap case
    - Middle vs. combination of sides

Faking Colors

- Metamers allow for faking
- Two different overlapping cones respond
  - Some of each color?
  - Some of the in-between color
- Can fake responses using N “point” colors
- 2 cones = 2 frequencies
- Get either cone, or anything in overlap
- Colors outside of overlap can’t be faked



Different Sensitivities

- Convert to gray requires scaling for sensitivities
- R = 0.212671 * Y
- G = 0.715160 * Y
- B = 0.072169 * Y.

Gamut

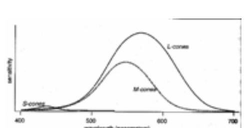
- The range of colors that a device can represent
  - Perceptual range

Perceptual Color Space

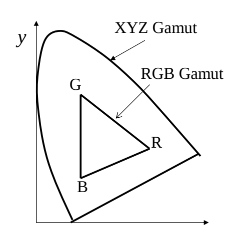
- Choose 3 primaries that do span human vision
  - Complete Gamut – can recreate any color
  - Not physically realizable (since has negative energies)
- CIE XYZ
  - Y is “lightness” – intensity w/o color
  - XZ are color directions

(normal) Human Vision

- 3 types of Cones
  - S (short wavelength) cones
  - M (mid wavelength) cones
  - L (long wavelength) cones
- Sortof RGB, but not quite
- Lots of overlap
- Far fewer S cones than L and M



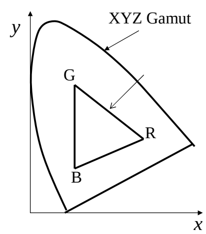
Determining Gamuts



- Gamut: The range of colors that can be represented or reproduced
- Plot the matching coordinates for each primary. eg R, G, B
- Region contained in triangle (3 primaries) is gamut
- Really, it’s a 3D thing, with the color cube distorted and embedded in the XYZ gamut

Gamut Analysis

- Space of colors a device can reproduce depends on primaries
- Device reproduce linear combinations of primaries = space inside of points
- Different devices have different ranges
  - Print with more inks
  - Films with different formulations



CS559 – Lecture 8
Color – Image Representation

These are course notes (not used as slides)

Written by Mike Gleicher, Sept. 2005

With some slides adapted from the notes of Stephen Chenney

Is RGB good enough?

- Sortof – gets close to all colors
  - Need better gamut
- No
  - Inconvenient for talking about color
  - Perceptually non-linear
  - Can't get really vivid colors
    - Purples are particularly bad
    - Can't be RGV – since violet sensitivity isn't good
    - Old film had different gamuts
      - Robin hood in technicolor

Other Color Systems: YCC

- Y = Luminence
  - Could be R+G+B
  - Better to be .3R + .6G + .1B
- Redundant – so send just 2 colors
  - Or send color differences: Y-R, Y-G
- Why?
  - Video: luminance is most important, subsample chroma
  - Perceptually more uniform since corrected for sensitivity
  - Start to separate color (direction in 2D)

Subtractive Color

- Printers combine inks that filter light
  - Remove colors
- So far additive
  - Black + red + green = yellow
- Ink is subtractive
  - White – red = cyan, White-green=magenta, white-blue=yellow
- Use "subtractive primaries"
  - Cyan, Magenta, Yellow

Artist – Centric Systems

- Hue = "name" of color
  - Red, orange, yellow, …
  - Color wheel
    - Complements add up to white
- Saturation = purity
- Value = luminence
- HSV (hexcone) vs. HLS (double hexcone)
  - RGB Color Cube viewed from the end
- Cone shape
  - Value is zero, hard to talk about color
- More convenient way to talk about color (for artists)

Where color gets messy…

- Color reproduction is hard
- When you see something on a monitor, does it look like the real thing? (shopping)
  - When you buy a real object?
  - When you print it?

Representing Color

- RGB
  - Store brightness for each channel
  - 8 bits argument (1% difference, 100:1 ~~ 400)
- Color Tables
  - A small table of integers->color
  - Store small integers for each pixel
  - Used a lot in old days (24 bits of frame buffer was a lot of memory!)
  - Still useful in some settings
    - Animate color tables, restrict pallete, …
  - Lots of algorithms for picking sets of colors
    - Median Cut is the most famous

Image File Formats

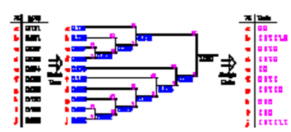
- Need to store all of the samples
- At whatever the necessary bits per pixel
- Lots of data
- Uncompressed = big
- Compress to take less space
  - Lossless (get same thing out)
  - Lossy (lose some information)

Lossless Coding 1

- Run-Length Encoding (RLE)
- Send pairs of values/run lengths
  - Only a win if (on average) your runs are long
- Look ahead:
  - Small change can mean big difference in coding
  - What if the changes were small enough that no one notices?

Lossless Coding 2

- Intelligent coding – give short codes to more common strings
  - Example: letters – rather than each getting 8 bits, let E=10, A=00, T=001, …
  - If you know the frequency distribution, you can distribute things optimally – Huffman encoding
  - Optimal Distribution may be uniform!
  - Entropy: the amount of distribution in the data
- Some things can't be made smaller by lossless encoding



Entropy Coding

- Fixed / Variable sized strings for codes
- Standard Codebook vs. per-corpus (file/image)
- Many algorithms for doing this
  - Huffman coding is just one classic one
- Lempel-Ziv (or Ziv-Lempel)
  - Variable length strings
  - Fixed code sizes (all the same)

Lossless Image Compression

- Use entropy coding (like LZ) on the actual pixels
- File formats
  - GIF – patented, only for small color paletes
  - PNG
- Uncompressed (or optionally compressed)
  - TGA (targa)
  - TIFF
  - BMP

Lossy Image Compression

- What if we limit our codebook?
  - Some data cannot be represented exactly
- Vector Quantization
  - Fixed length strings (and fixed codebook size)
  - Pick a set of codes that are as good as possible
  - Encode data by picking closest codes
  - Other than picking codes, encoding/decoding is really easy!

Lossy Coding 2

- Suppose we can only send a fraction of the image
  - Which part?
- Send half an image:
  - Send the top half (not too good)
  - Halve the image in size (send the low frequency half)
- Idea: re-order (transform) the image so the important stuff is first

Lossy Coding 2

- Suppose we can only send a fraction of the image
  - Which part?
- Send half an image:
  - Send the top half (not too good)
  - Halve the image in size (send the low frequency half)
- Idea: re-order (transform) the image so the important stuff is first

Perceptual Image Coding

- Idea: lose stuff in images that is least important perceptually
  - Stuff least likely to notice
  - Stuff most likely to convey image
- Who knows about this stuff: The experts!
  - Joint Picture Experts Group
  - Idea of perceptual image coding

559 Course Notes
Lossy Compression

Mike Gleicher

Fall 2006 (taken from Fall 2005)

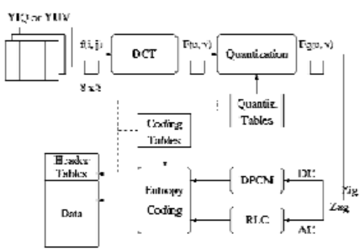
Notes for lecture – not shown in class

JPEG

- Key Ideas
  - Frequency Domain (small details are less important)
  - Block Transforms (works on 8x8 blocks)
    - Discrete Cosine Transform (DCT)
  - Control Quantization of frequency components
    - More quality = use more bits
    - Generally, use less bits for HF
    - [..\2005\2005-09.ppt]

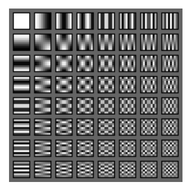
JPEG

- Multi-stage process intended to get very high compression with controllable quality degradation
- Start with YIQ color
  - Why? Recall, it's the color standard for TV



Discrete Cosine Transform

- A transformation to convert from the *spatial* to *frequency* domain – done on 8x8 blocks
- Why? Humans have varying sensitivity to different frequencies, so it is safe to throw some of them away
- Basis functions:



Quantization

- Reduce the number of bits used to store each coefficient by dividing by a given value
  - If you have an 8 bit number (0-255) and divide it by 8, you get a number between 0-31 (5 bits = 8 bits – 3 bits)
  - Different coefficients are divided by different amounts
  - Perceptual issues come in here
- Achieves the greatest compression, but also quality loss
- "Quality" knob controls how much quantization is done

Entropy Coding

- Standard lossless compression on quantized coefficients
  - Delta encode the DC components
  - Run length encode the AC components
    - Lots of zeros, so store number of zeros then next value
  - Huffman code the encodings

Lossless JPEG With Prediction

- Predict what the value of the pixel will be based on neighbors
- Record error from prediction
  - Mostly error will be near zero
- Huffman encode the error stream
- Variation works really well for fax messages

Video Compression

- Much bigger problem (many images per second)
- Could code each image seperately
  - Motion JPEG
  - DV (need to make each image a fixed size for tape)
- Need to take advantage that different images are similar
  - Encode the Changes ?

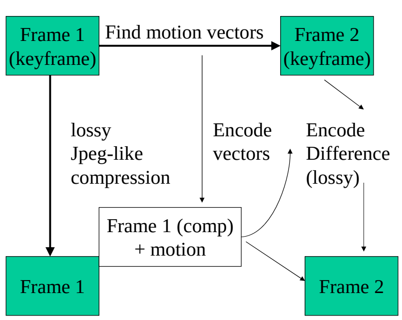
MPEG

- Motion Picture Experts Group
  - Standards organization
- MPEG-1 simple format for videos (fixed size)
- MPEG-2 general, scalable format for video
- MPEG-4 computer format (complicated, flexible)
- MPEG-7 future format
- What about MPEG-3? – it doesn't exist (?)
  - MPEG-1 Layer 3 = audio format

MPEG Concepts

- Keyframe
  - Need something to start from
  - "Reset" when differences get too far
- Difference encoding
  - Differences are smaller/easier to encode than images
- Motion
  - Some differences are groups of pixels moving around
  - Block motion
  - Object motion (models)

MPEG



559 General Polygons

Mike Gleicher
Fall 2006 (taken from Fall 2005)
Notes for lecture – not shown in class

Triangles?

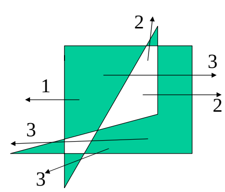
- Old way: Scan conversion
  - Start at top
  - Brezenham's algorithm gives left/right sides
  - Draw horizontal scans
- New Way: point in triangle tests
  - Generate sets of points that might be in triangle
  - Do half-plane tests to see if inside
- Tricky part: edges
  - Need to decide which triangle draws shared edges

General Polygons?

- Inside / Outside not obvious for general polygons
- Usually require simple polygons
  - Convex (easy to break into triangles)
- For general case, three common rules:
  - Non-exterior rule: A point is inside if every ray to infinity intersects the polygon
  - Non-zero winding number rule: trace around the polygon, count the number of times the point is circled (+1 for clockwise, -1 for counter clockwise). Odd winding counts = inside (note: I got this wrong in class)
  - Parity rule: Draw a ray to infinity and count the number or edges that cross it. If even, the point is outside, if odd, it's inside (ray can't go through a vertex)

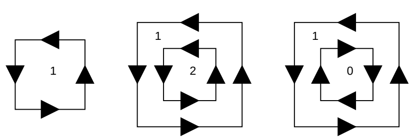
Parity

- Any point, take any ray (that doesn't go through a vertex)
- Odd number of crossings = inside
- Even number of crossings = outside



Power Point uses this rule!

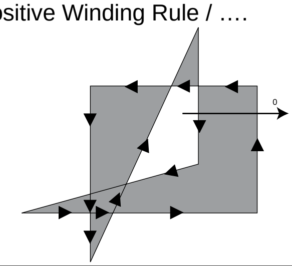
Winding Numbers



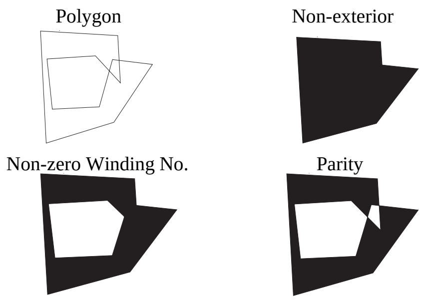
- Count the number of times a point is circled counter clockwise
  - Clockwise counts negative
- Can pick any ray from point and count left/right
  - Right (relative to away direction) = CCW = +1
  - Left = CW = -1

Non-Zero Winding Rule

- Any non-zero winding is "inside"
- What Adobe Illustrator does
- Odd Winding Rule / Positive Winding Rule / ….



Inside/Outside Rules



559 Course Notes
Transforms (lecture 10+11)

Mike Gleicher
Fall 2006 (taken from Fall 2005)
Notes for lecture – not shown in class

Coordinate Systems

- Tells us how to interpret positions (coordinates)
- In graphics we deal with many coordinate systems and move between them
  - Use what is convenient for what we're doing
- Examples
  - Chalkboard as coordinate system
  - One panel of chalkboard as coordinate system
  - Monitor as coordinate system

What is a coordinate system

- Position of the zero point
- Directions for each axis
  - Represent points as a linear combination of vectors
  - Vectors (basis) are axes
  - Scale of vectors matter (what is "1 unit")
  - Directions matter (which way is up)
  - Doesn't need to be perpindicular (just can't be parallel)

Describing Coordinate systems

- Need to have some "reference"
  - Where we will measure from
- Give origin, vectors
- Once we have 1 system, can define others
- Can move points by changing their coordinate system
  - Piece of paper is a coordinate system
  - Move piece of paper around
  - If it were a rubber sheet could stretch it as well

Changing Coordinate Systems

- Changing coordinate systems allows us to change large numbers of points all at once
- Need to move points between coordinate systems
  - A coordinate system *transforms* points to a more canonical coordinate system
  - Can define coordinate systems by transformations between coordinate systems

Transformations

- Something that changes points
  - y',y' = f(x,y) $f \in R^2 \rightarrow R^2$
- Coordinate systems are a special case
- Other examples
  - F(x,y) = x+2, y+3
  - F(x,y) = -y, x
  - F(x,y) = x^2, y
- Easy way to effect large numbers of points

Interpreting Transformations

- Can be viewed as a change of coordinates
  - What happens to a piece of graph paper?
  - Just sometimes to a stretchy piece of paper
- View as a function applied to points
- Function composition
  - F(g(h(x))) (note order)

$$x \rightarrow \boxed{h} \rightarrow \boxed{g} \rightarrow \boxed{f} \rightarrow x'$$

Linear Transformations

- Important special case – linear functions
- Can be written as a matrix x' = M x (x is a vector)
- Good points
  - Many useful transformations are of this form
  - Composition by matrix multiply
  - Easy analysis
  - Straight lines stay straight lines
  - Inverses by inverting the matrix
- Note: linear operators preserve zero!

Example Linear Operators

- Uniform Scale $$scale(s) = \begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix}$$
- Non-Uniform Scale $$nuscale(s,t) = \begin{bmatrix} s & 0 \\ 0 & t \end{bmatrix}$$
- Reflect $$reflect(s,t) = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$
- Skew $$skew(a) = \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix}$$

More linear operators

- Rotate $$rotate(\theta) = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix}$$
- Note: all of this keeps zero
- All linear operations are around the origin (?)

Understanding linear operators

$$\mathrm{Mx} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- This is POST-Multiply (vector on the right)
  - Pre-multiply convention works too
  - All the matrices get transposed
- What does each element do?
  - Left column – where does X axis go (put in unit X vector)
  - Right column – where does Y axis go
- Can't do anything about origin!

Post-Multiply vs. Pre-Multiply

- Post multiply – column vector on the left

  F G H x
- Pre-multiply – row vector on the right
  - Older convention, not used as often

  $\mathbf{x}^T \mathbf{H}^T \mathbf{G}^T \mathbf{F}^T$
- I will (almost always) use the post-multiply convention

Affine Transformations

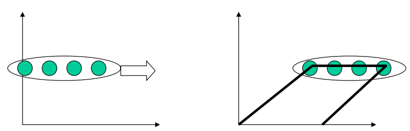
- Translation = move all points the same (vector +)
- Affine = Linear operations plus translation
- Cannot be encoded in a 2x2 matrix (for 2d)
  - Need six numbers for 2d
  - Could be a 3x2 matrix – but then no more multiplies
- Rather than treat as a special case, improve our coordinates a bit

Homogeneous Coordinates

- Big idea for graphics – really important
  - Will be used for several things – translation is just 1
- Basic idea: add an extra coordinate
  - 2D becomes 3D (3x3 matrices)
  - 3D becomes 4D (4x4 matrices)
- Convert “back” from homogeneous coordinates by division
  - (x,y) -> (x,y,1)
  - (x,y,w) -> (x/w, y/w)
- Projection
  - Many points in higher dim space = 1 point in lower dim space
- For now, just make w=1

Translation in Homogeneous Coords

- Translate in 2D = Skew in 3D
  - Deck of cards



$$trans(x, y) = \begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{bmatrix}$$

What about other linear ops

- Just add an extra coordinate
- Don’t change w (unless you know what you’re doing)

$$scale(s) = \begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$rotate(\theta) = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Matrices as Coordinate Systems

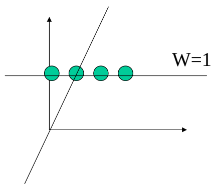
- Where does X axis go?
- Where does Y axis go?
- Where does origin go?
- Assumes that bottom row is [0 0 1]
- Can you scale by changing w?
  - Yes, but often we prefer to renormalize so bottom right number is 1

Homogeneous Coordinates

- Big idea for graphics – really important
  - Will be used for several things – translation is just 1
- Basic idea: add an extra coordinate
  - 2D becomes 3D (3x3 matrices)
  - 3D becomes 4D (4x4 matrices)
- Convert “back” from homogeneous coordinates by division
  - (x,y) -> (x,y,1)
  - (x,y,w) -> (x/w, y/w)
- Projection
  - Many points in higher dim space = 1 point in lower dim space
- For now, just make w=1

Homogeneous Coordinates

- “Normal” space is a subspace
  - W = 1
- Think about 1D case (so embed into 2D x,w)
- Many equivalent points (projection)

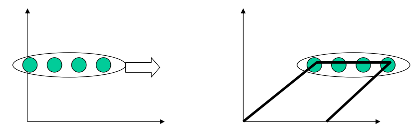


Only 1D Linear operation is scale

(about origin)

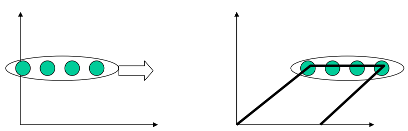
Translation in Homogeneous Coords

- 1D Translation = 2D Skew



Translation in Homogeneous Coords

- Translate in 2D = Skew in 3D
  - Deck of cards



$$trans(x,y) = \begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{bmatrix}$$

What about other linear ops

- Just add an extra coordinate
- Don’t change w (unless you know what you’re doing)

$$scale(s) = \begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$rotate(\theta) = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Homogeneous Coordinates

- Makes translation (affine transforms) linear
- Need to work in higher dimensional space
- Useful for lots of other things
  - Viewing (perspective)

Matrices as Coordinate Systems

- Where does X axis go?
- Where does Y axis go?
- Where does origin go?
- Assumes that bottom row is [0 0 1]
- Can you scale by changing w?
  - Yes, but often we prefer to renormalize so bottom right number is 1

Composing Transformations

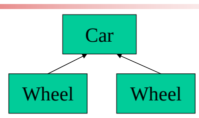
- Order matters!
  - Scale / rotate vs. rotate/scale
- Can implement by multiplying matrices
  - $T_1 T_2 T_3 \mathbf{x} = (T_1 T_2 T_3) \mathbf{x}$

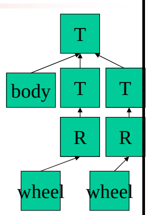
Why Compose?

- Rotate about a point
  - $T_c R T_{-c} x$
- Scale along an axis
  - Move point to origin
  - Align axis w/major axis
  - Scale
  - Put things back
  - $T_c R_\theta S R_{-\theta} T_{-c} x$

Hierarchical coordinate Systems

- Car
  - Wheel
  - Wheel
- Person
  - Head / Neck
  - Arm / forearm / hand





Matrix Stack

- Multiply things onto the top
- Top is “current” coordinate system
- Push (copy the top) if you’ll come back
- Pop to go back
- Think about it as moving the coordinate system
- Top of stack is “current coordinate system”
  - Where we will draw
- Transformations change current coord system
  - Or change the objects that we are going to draw

Matrix Stack Example

- Draw Car = …. Push trans wheel pop …
- Push trans – draw car – pop push trans – draw car

3D

- 3D coordinate system & handedness
- Prefer right-handed coordinate systems
- Right-hand rule

What happens in 3D?

- 4D Homogeneous Points
  - 4x4 matrices
- Basic transforms are the same
  - Translate
  - Scale
  - Skew
- Rotation is different
  - Rotation in 3D is more complicated?

What is a rotation?

- A transformation that:
  - Preserves distances between points
  - Preserves the zero
  - Preserves "handedness" (in 2D clockwiseness)
- A subset of linear transformations

- Some things that come out of these:
  - Axes remain perpindicular
  - Axes remain unit length
  - Cross product holds

Parameterizing Rotations

- Rotations are Linear Transformations
  - 2x2 matrix in 2D
  - 3x3 matrix in 3D
- The set of rotations = set of OrthoNormal Matrices
- Inconvenient way to deal with them
  - Can't work with them directly
  - Not stable (small change makes it not a rotation)

- Is there an easier way to parameterize the set?

Measuring rotation in 2D

- Pick 1 point (1,0)
- Any rotation must put this on a circle
- If you know where this point goes, can figure out any other point
  - Distances (w/point & origin) + handedness says where things go
- Parameterize rotations by distance around circle
  - Angle
- Issues with wrap around
  - Many different angles = same rotation

Much harder in 3D

- Any point can go to a sphere
- That one point doesn't uniquely determine things

- No vector in R^n can compactly represent rotations
  - Singularities
    - nearby rotations / far away numbers
    - Nearby numbers / far away rotations
- Hairy-Ball Theorem
  - Any parameterization of 3D rotations in R^n will have singularities

Representation of 3D Rotations

- Two Theorems of Euler
  - Any rotation can be represented by a single rotation about an arbitrary axis (axis-angle form)
  - Any rotation can be represented by 3 rotations about fixed axes (Euler Angle form)
    - XYZ, XZX, any non-repeating set works
    - Each set is different (gets different singularities)

- Building rotations
  - Pick a vector (for an axis)
  - Pick another perpindicular vector (or make one w/cross product)
  - Get third vector by cross product

Euler Angles

- Pick convention
  - Are axes local or global?
  - Local: roll, pitch, yaw
  - What order?
- Apply 3 rotations
- Good news: 3 numbers
- Bad news:
  - Can't add, can't compose
  - Many representations for any rotation
  - Singularities

3D ROTATION NOTES

10/10/06

ROTATIONS

- preserve O
- preserve handedness
- preserve distance

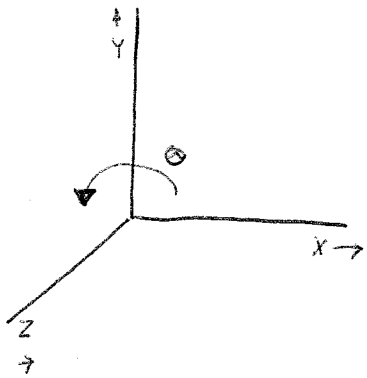
ARE LINEAR Transforms (N×N orthonormal matrices with positive determinant)

They compose
if A is a rotation and B is a rotation
$R = AB$ is a rotation
$R' = BA$ is a rotation
$R != R$ (they do not commute)

Rot 90 X, Rot 90 Z
vs.
Rot 90 Z, Rot 90 X

LOCAL VS. GLOBAL AXES
can do either - but they are different

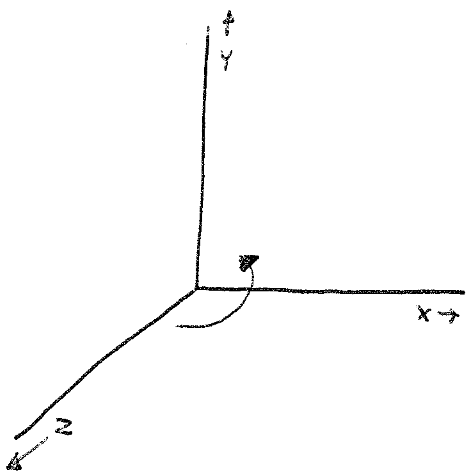
Z-Axis



$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

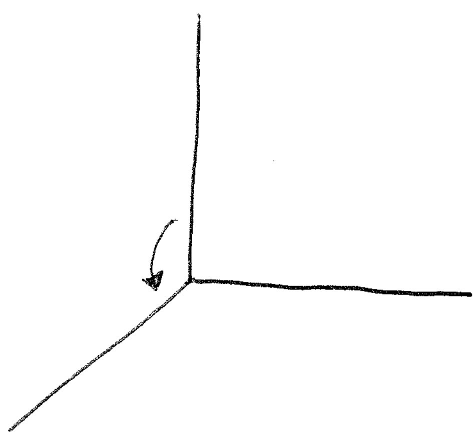
Y-Axis

- remember the right hand rule to get +/- right

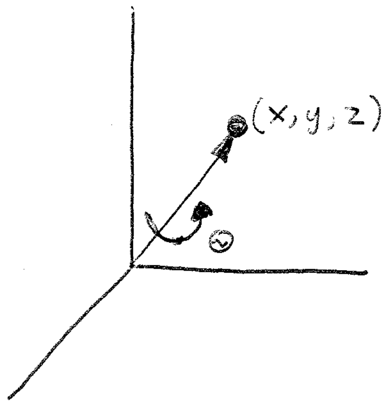


$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

X - Axis

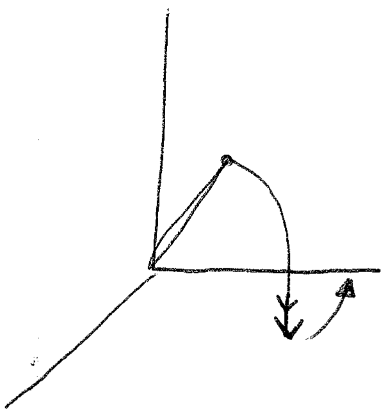


$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



rotate about an arbitrary axis
(x, y, z), Θ

1. rotate so axis is aligned w/ x-axis
   rotate around z-axis (point is on XZ plane)
   rotate around Y-axis (point is on X axis)



NOTE: YOU CAN PICK ANY 3 AXES!

NOTE 2:
We can make a rotation just using rotations around the fixed axes!

EULER'S Theorems

ANY ROTATION CAN BE REPRESENTED BY:

1. A single rotation about an arbitrary axis
2. A sequence of 3 rotations around fixed axes

So - you can represent a rotation
(a 3×3 orthonormal, positive matrix)
by:

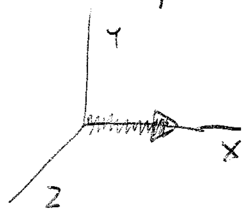
9 numbers - if you're very careful
4 numbers - axis + angle
3 numbers Rot X Rot Y Rot Z
or Z X Y or Z Y X
or Z X Z

Good things about Euler Angles
compact = 3 numbers
any 3 numbers → rotation
complete (any rotation can be represented)
~~efficient~~ (relatively easy - make 3 1d rotations)
easy

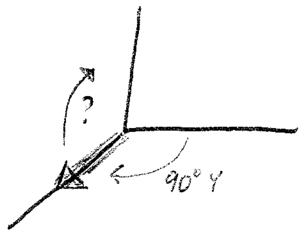
GIMBALL LOCK

Assume XYZ (any order has this problem)

if $Y = 90°$, X and Z are the same!
arrow points down X axis



if $Y = 90°$
how to rotate "up"?

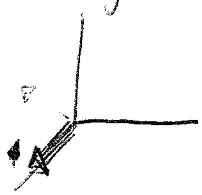


can't rotate X (does nothing; arrow is axis)

can't rotate Z (does nothing, arrow is axis - after Y rotation)

Why is this bad?

How to get from



a small change in rotation = a big change in numbers

0, 90, 0 → ? I can't even figure it out!

or if you go right to left, imagine an airplane starting down the Z axis - how to make it point towards X, but a little bit down

What do we do with rotation in 559

1. rotate around a single axis
   no problems

2. build a rotation matrix as needed
   use cross-product of first 2 vectors
   to get Z

3. use axis angle

4. Euler Angles are OK if:
   - you can figure out their values
   - you don't need to compose them
   - you don't need to interpolate them

5. you can always compose rotation matrices

CS559 – Lecture 12
Transformations, Viewing

These are course notes (not used as slides)
Written by Mike Gleicher, Oct. 2005
With some slides adapted from the notes of Stephen Chenney

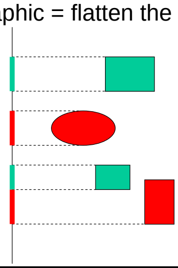
Final version

Viewing Transformations

- How do we transform the 3D world to the 2D window?
- Concepts:
  - World Coordinates
  - View (or window) VOLUME
    - Need 3rd dimension later to get occlusions right
  - Viewing Coordinates
    - 3D Viewing coordinates
- Separate Issues
  - Visibility (what's in front)
  - Clipping (what is outside of the view volume)

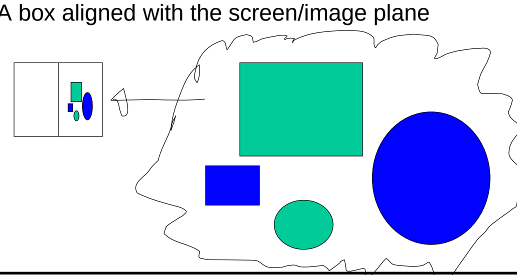
Orthographic Projection

- Projection = transformation that reduces dimension
- Orthographic = flatten the world onto the film plane



View Volumes / Transformations

- Viewing transformation puts the world into the viewing volume
- A box aligned with the screen/image plane



Canonical View Volume

- -1 to 1 (zero centered)
- XY is screen (y-up)
- Z is towards viewer (right handed coordinates)
  - Negative Z is into screen
- For this reason, some people like left-handed

2 Views of Viewing Transform

- Put world into viewing volume
- Position camera in world (view volume into world)
- Clip stuff that is outside of the volume
- Somehow get closer stuff to show up instead of farther things (if we want solid objects)

Orthographic Projection

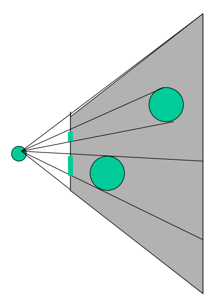
- Rotate / Translate / Scale View volume
  - Can map any volume to view volume
- Sometimes pick skews
- Things far away are just as big
  - No perspective
- Easy – and we can make measurements
  - Useful for technical drawings
  - Looks weird for real stuff
    - Far away objects too big

Perspective Projection

- Farther objects get smaller
- Eye (or focal) point
- Image plane
- View frustum (truncated pyramid)
- Two ways to look at it:
  - Project world onto image plane
  - Transform world into rectangular view volume (that is then orthographically projected)

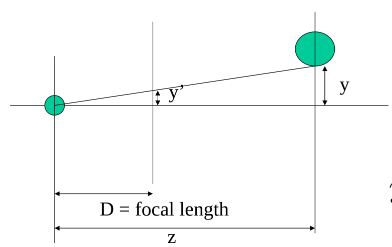
Perspective

- Eye point
- Film plane
- Frustum
- Simplification
  - Film plane centered with respect to eye
  - Site down negative Z axis
    - Can transform world to fit



Basic Perspective

- Similar Triangles
- Warning = using d for focal length (like book)
  - F will be "far plane"



$$\frac{y}{z} = \frac{y'}{d}$$

$$y' = \frac{d}{z}y$$

Use Homogeneous coordinates!

- Use divide by w to get perspective divide
- Issues with simple version:
  - Font / back of viewing volume
  - Need to keep some of Z in Z (not flatten)

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z \end{bmatrix} = \begin{bmatrix} x/z \\ y/z \\ z/z = 1 \\ 1 \end{bmatrix}$$

The real perspective matrix

- N = near distance, F = far distance
- Z = n put on front plane, z=f put on far plane

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{n+f}{n} & -f \\ 0 & 0 & -n & 0 \end{bmatrix}$$

Shirley's Perspective Matrix

- After we do the divide, we get an unusual thing for z – it does preserve the order and keeps n&f

$$\mathbf{Px} = \mathbf{P}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x\frac{n}{z} \\ y\frac{n}{z} \\ n + f - \frac{fn}{z} \\ 1 \end{bmatrix}$$

Camera Model

- The "window coordinate" system is all the we really know
- In a sense, it is the camera coordinate system
- Easiest to think about it as a camera taking a picture of the work
- Transform world coordinates into camera coordinates
  - Or, think about it the other way…

How to describe cameras?

- Rotate and translate (and scale) the world to be in view
- The camera is a physical object (that can be rotated and translated in the world)
- Easier ways to specify cameras
  - Lookfrom/at/vup

CS559 – Lecture 13
OpenGL Survival

These are course notes (not used as slides)

Written by Mike Gleicher, Oct. 2006

The Basics of doing 3D Graphics

- Stuff you need to know to write programs
- Toolkit details best done by looking at code
  - And trying it yourself!
- See online tutorials (e.g. Survival Guide)
- See the red book
- Try to refresh the concepts behind using library
- Goal: get you to know enough to do Project

List of stuff you need to know

- Basics of Toolkits
- Dealing with a window
- Double buffering
- Drawing context
- Transformations / Coordinate Systems / Cameras
- 3D Viewing / Visibility (Z-Buffer)
- Polygon drawing
- Lighting
- Picking and UI

Basics of a toolkit

- OpenGL is for drawing graphics in a window
- Doesn't care where the window comes from
  - Need something to deal with Operating system
- Less good for text and widgets
- Use some toolkit to do windowing and UI support
  - FlTk – supports OpenGL well
  - Glut – simple, designed for doing OpenGL demos
  - Native windows – um, I can't comment

The Drawing Context

- OpenGL is *stateful*
  - Draw in the current window, current color, …
  - Contrast with stateless systems
    draw(x1,y1,x2,y2)
    draw(window, coordsys, x1, y1, x2, y2, color, pattern, …)
- Where is all that state kept?
  - Drawing Context
- Each window has its own state
  - Need mechanisms for keeping track of it
  - Making it the current state
  - FlTk does this for you (in draw, or with make_current)
- Beware! You can only draw with a current context

When does drawing happen

- Two different types of graphics toolkits
  - Immediate mode – stuff goes right to frame buffer
  - Retained mode – keep 3D objects on list, *system* draws all at once
- OpenGL supports both (usually immediate mode)
- What happens with a triangle

Double Buffering

- Double Buffering – independent of immediate/retained!
- Prevent from seeing partially drawn results
- (potentially) keep synced with screen refresh

- Draw into back buffer
- Swap-buffers

- FlTk will take care of this for you

When do I draw?

- When the window is "damaged"
- Periodically (animation / interaction)

- With FlTk:
- It calls the draw function when needed
  - NEVER call it yourself
- If you want to force a redraw, damage your window
  - It will be redrawn when appropriate

Where do I draw

- Screen coordinates – the main place everyone can agree
- OpenGL uses unit coordinates
  - Depth is -1 to 1 as well

- The Viewport
  - GL lets you limit things to a rectangular area of the screen
  - This is the only thing measured in pixels!

- Need to correct for aspect ration of screen

Getting my own coordinate system

- OpenGL only knows 1 coordinate system
  - The "Normalize Device Coordinates" - NDC
  - Viewport mapped to unit cube
  - There is actually 1 other coord system, but that's a detail for lighting

- If you're transformation is the identity, you get NDC

- All points transformed by the "current transformation"

OpenGL coordinate transforms

- OpenGL has 2 "current" transforms

  n = **P M x**

  n = point in NDC x = point in your coordinate system
  P = projection matrix M = Model View matrix
  P and M are both stacks (although P is a short stack
- Why 2 matrices?
  - Esoteric detail of lighting
- Only the perspective transform goes into **P**
  - Unless you're doing something wierd
- **M** gives "camera coordinates"
  - Only lighting happens there in GL

Is OpenGL Post-Multiply?

- An internal detail – unless you look at the matrices

- Think of it as Post-Multiply
  - And if everything is being transposed, no big deal

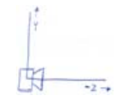
- Only "load" is to load the transpose
  - OpenGL used to be pre-multiply, but since everyone else is post-multiply

How do I set the transform?

- Need to pick which matrix "stack"
  - Projection, ModelView
- Can either load, or post-multiply
  - Almost everything does a post-multiply
  - Except for the load operations
  - BEWARE: make sure to do a load identity first!
- Most matrix operations build a matrix and post-multiply it onto the "current" stack

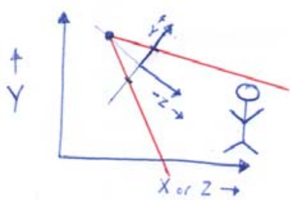
Getting your coordinate systems

- Need things in camera coordinates
- Rotate and translate the world coordinates (and possibly scale)
- Think of placing and pointing the camera



Getting the camera scale?

- Projection does some scaling (by Z)
- Projection puts eye at z?
- Projection puts near clipping plane at -1, far plane at 1
- Use OpenGL's projection matrix
- Field of view/aspect ratio

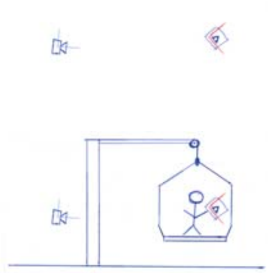


Moving coordinate systems

- Multiplying matrix means changing the coordinate system
- Or think about it as things closest to the object go first

Your own coordinate system

- Draw your triangle…
  - On a piece of paper
  - In your hand
  - When you're on a platform
  - On a crane
- Build transforms!
  - Camera->world
  - World->crane
  - Crane->top of crane
  - Crane->platform
  - Platform->person
  - Person->arm
  - Arm->paper . . .



Convenient ways to make transforms

- Projection
  - gluFrustum, glPerspective
- Matrix handling
  - Load, get, pushmatrix, popmatrix
  - Rarely load anything but the identity

Actually drawing

- Begin / end blocks of points
- Send each point by itself (or as an array)
- Uniformity in how you draw different things
  - Lines
  - Triangles
  - Strips of triangles
  - Quads
- Things are drawn in the “current” state
- Color, line style, …

Normal Vectors

- Assign per-vertex or per-triangle
- Unit vector towards the “outside”
- Not done automatically for you

- Will be very useful for lighting, so get in the habit

Got to this slide on day 13

What color are things?

- Turn off lighting – and say colors directly

- Turn on lighting – and let the games begin!

- Idea: color of object is affected by lights
  - Need some light to see things
  - Direction of light affects how things look
  - Say where the lights are, how strong they are
  - What the reflectance of the surfaces are
- A whole topic for days in this class

What happens to stuff off the screen?

- Clipping
  - Things get chopped by a plane
  - Each side of the viewing volume
  - Other planes as well – if you want

- Important to do correctly and efficiently
- A lot of work into the methods – but really boring

Visibility

- A topic for later in the class:
  How to get objects to occlude each other

- Give polygons in any order (even back ones last)
- Use a Z-Buffer to store depth at each pixel

- Things that can go wrong:
  - Near and far planes DO matter
  - Backface culling and other tricks can be problematic
  - You may need to turn the Z-buffer on
  - Don’t forget to clear the Z-Buffer!

So, I got a black screen…

- Celebrate – you’ve gotten a window, and that’s step 1!

- Are you drawing at the right time?
- Do you have a drawing context?
- Are you drawing objects?
- Is the camera pointing at them?
- Are they getting mapped to the screen?
- Is something occluding them?
- Are they in the view volume?
- Are they lit correctly?
- And a zillion other things that can go wrong…

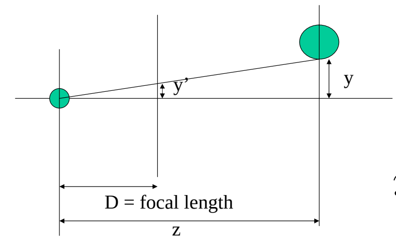
CS559 – Lecture 14
Visibility, Projection

These are course notes (not used as slides)

Written by Mike Gleicher, Oct. 2006

Basic Perspective

- Similar Triangles
- Warning = using d for focal length (like book)
  - F will be "far plane"



$$\frac{y}{z} = \frac{y'}{d}$$

$$y' = \frac{d}{z} y$$

Use Homogeneous coordinates!

- Use divide by w to get perspective divide
- Issues with simple version:
  - Font / back of viewing volume
  - Need to keep some of Z in Z (not flatten)

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z \end{bmatrix} = \begin{bmatrix} x/z \\ y/z \\ z/z = 1 \\ 1 \end{bmatrix}$$

The real perspective matrix

- N = near distance, F = far distance
- Z = n put on front plane, z=f put on far plane

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{n+f}{n} & -f \\ 0 & 0 & -n & 0 \end{bmatrix}$$

Shirley's Perspective Matrix

- After we do the divide, we get an unusual thing for z – it does preserve the order and keeps n&f

$$\mathbf{Px} = \mathbf{P} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x\frac{n}{z} \\ y\frac{n}{z} \\ n + f - \frac{fn}{z} \\ 1 \end{bmatrix}$$

Camera Model

- The "window coordinate" system is all the we really know
- In a sense, it is the camera coordinate system
- Easiest to think about it as a camera taking a picture of the work
- Transform world coordinates into camera coordinates
  - Or, think about it the other way…

How to describe cameras?

- Rotate and translate (and scale) the world to be in view
- The camera is a physical object (that can be rotated and translated in the world)
- Easier ways to specify cameras
  - Lookfrom/at/vup

A Hack: Painted Shadows

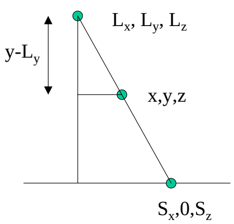
- Use projection to squash objects onto floor
- Paint a copy of them in black on the floor
- Useful for UI
- Drop Straight onto floor = set Y to zero

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Beware – might want to have things float above floor

Projective Shadows – point light

- Position of light L_x, L_y, L_z
- Position of point x,y,z
- Position of Shadow $S_x, 0, S_z$
  - Assume ground (y) = 0



$$\frac{x-l_x}{l_y-y} = \frac{S_x-l_x}{l_y-0}$$

$$S_x - l_x = l_y \frac{x-l_x}{l_y-y}$$

$$S_x = \frac{l_y(x-l_x)}{l_y-y} + l_x \frac{(l_y-y)}{(l_y-y)}$$

$$S_x = \frac{l_y x - l_x l_y}{l_y-y} + \frac{l_y l_x - l_x y}{l_y-y}$$

$$S_x = \frac{l_y x}{l_y-y} + \frac{-l_x y}{l_y-y} + \cancel{\frac{l_x l_y - l_x l_y}{l_y-y}} \rightarrow 0$$

$$\begin{array}{ll} \begin{bmatrix} l_y & -l_x & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -l_z & l_y & 0 \\ 0 & -1 & 0 & l_y \end{bmatrix} & \begin{array}{l} \\ \leftarrow \text{zero out } Y \\ \\ \\ \end{array} \end{array}$$

Visibility

- A topic for later in the class:
  How to get objects to occlude each other
- Give polygons in any order (even back ones last)
- Use a Z-Buffer to store depth at each pixel
- Things that can go wrong:
  - Near and far planes DO matter
  - Backface culling and other tricks can be problematic
  - You may need to turn the Z-buffer on
  - Don't forget to clear the Z-Buffer!

How to make objects solid

- So far, just curves (outlines of things)
- Can fill regions (polygons)
  - But how to get stuff in front to occlude stuff in back
- General categories
  - Re-think drawing
    - From eye (pixels) not objects
  - Analytically compute what can be seen
    - Hidden line drawing (hard)
  - Hidden Surface Removal

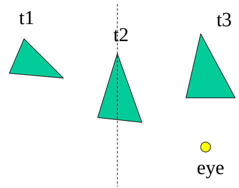
Painter's Algorithm

- Simplest hidden surface algorithm
- Draw farthest objects first
  - Nearer object cover further ones
- Problems
  - Cycles / intersections (no order possible)
    - Fix by splitting triangles
  - Need all triangles ahead of time
  - O(n log n) sort
  - Must resort for every view direction
- Depth Complexity (amount of time each pixels is drawn)

Binary Space Partitions

- Fancy data structure to help painters algorithm
- Stores order from any viewpoint

- A plane (one of the triangles) divides other triangles
- Things on same side as eye get drawn last



T2 divides into groups
T3 is on same side of eye

Using a BSP tree

- Recursively divide up triangles

- Traverse entire tree
  – Draw farther from eye subtree
  – Draw root
  – Draw closer to eye subtree
- Always O(n) to traverse
  – (since we explore all nodes)
  – No need to worry about it being balanced

Building a BSP tree

- Each triangle must divide other triangles
  – Cut triangles if need be (like painters alg)

- Goal in building tree: minimize cuts

Z-Buffer

- Throw memory at the problem
- A hardware visibility solution
  – Useful in software, but a real win for hardware

- For every pixel, store depth that pixel came from
- No object? Store ∞

- When you draw a pixel, only write the pixel if you pass the “z-test”

Things to notice about Z-Buffer

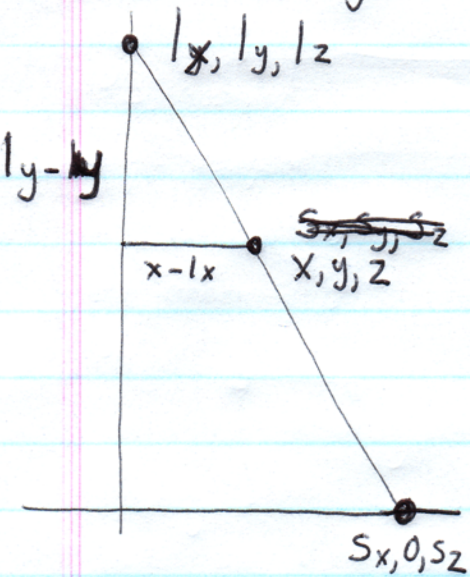
- Pretty much order independent
  – Same Z-values
  – Transparent objects
- Z-fighting
  – Objects have same Z-value, ordering is “random”
  – Bucketing (finite resolution) causes more things to be same
  – As things move, they may flip order
- Anti-Aliasing
  – Things done per-pixel, so sampling issues

Resolution of Z-Buffer

- Old days: big deal
  – Integer Z-buffers, limited resolution
- Future: floating point z-buffer
  – Still have resolution issues, not as bad

- Need to bucket things from near to far
  – Don’t set near too near or far to far
- Non-linear nature of post-divide Z
  – Remember that perspective divide gives fn/z

Similar Triangles



$$\frac{x - l_x}{l_y - y} = \frac{s_x - l_x}{l_y - 0}$$

$$s_x - l_x = l_y \frac{x - l_x}{l_y - y}$$

$$s_x = \frac{l_y (x - l_x)}{l_y - y} + l_x \frac{(l_y - y)}{(l_y - y)}$$

$$s_x = \frac{l_y x - l_x l_y}{l_y - y} + \frac{l_y l_x - l_x y}{l_y - y}$$

$$s_x = \frac{l_y x}{l_y - y} + \frac{-l_x y}{l_y - y} + \cancel{\frac{l_x l_y - l_x l_y}{l_y - y}} \rightarrow 0$$

$$\begin{bmatrix} l_y & -l_x & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -l_z & l_y & 0 \\ 0 & -1 & 0 & l_y \end{bmatrix} \begin{array}{l} \\ \leftarrow \text{zero out } Y \\ \\ \\ \end{array}$$

CS559 – Lecture 13-14
Curves

These are course notes (not used as slides)
Written by Mike Gleicher, Oct. 2005
Updates Oct 2006

Shape Modeling

- Creating Mathematical Descriptions of Shape
- Why?
  - Drawing, Sample, Analyze
- Why is this hard
  - Shapes can be arbitrary and complex – hard to describe
  - Conflicting goals
    - Concise
    - Intuitive
    - Expressive
    - Analyzable
    - …

What is a Shape

- Mathematical definition is elusive
- Set of Points
  - Potentially (usually) infinite
- "Lives" in some bigger space (e.g. 2D or 3D)
- Many ways to describe sets
  - Set inclusion test (implicit representation)
  - Procedural for generating elements of the set
  - Explicit mapping from a known set

Some kinds of Shapes

- Curves
  - 1D Objects, like what you draw with a pen
- Surfaces / Areas
  - 2D Objects – the insides of 2D things
  - Bounded by a Curve
- Solids / Volumes
  - 3D Objects – the insides of things that take up volume
  - Different definition: set with the same dimension as the embedded space (an area of 2D)

Curves

- Intuitively, something you can draw with a pen
  - Not filled areas
  - Mathematical oddity: space filling curves
    - Requires infinite lengths, …
- Almost every point has 2 "neighbors"
- Locally equivalent to a line

Defining Curves

- Two different mathematical definitions
  1. The continuous image of some interval
  2. A continuous map from a one-dimensional space to an n-dimensional space
- Both definitions imply a mapping
  - From a line segment (which is a curve)
- #1 is a set of points, #2 is the mapping

Describing Curves

- Some curves have names
  - Line, line segment, ellipse, parabola, circular arc
- Some set of parameters to specify
  - Radius of an arc, endpoints of a line, …
- Other curves do not have distinct names
  - Need a *Free Form* representation

Curve Represenations

- Implicit
  - Function to test set membership
  - F(x,y) = 0
- Explicit / Parametric
  - Y = f(x)
  - (x,y) = f(t) – where t is a free parameter
    - Need to define a range for the parameter
- Procedural
  - Some other process for generating points in the set
- By definition, a curve has at least 1 parametric representation

Parameterizations

- For any curve (set of points) there may be many mappings from a segment of the reals
- Consider: line from 0,0 -> 1,1
  - (x,y) = (t,t) $t \in [0,1]$
  - (x,y) = (.5t, .5t) $t \in [0,2]$
  - (x,y) = (t^2,t^2) $t \in [0,1]$
- Many ways to represent a curve

Free Parameters

- Not really a property of the curve
  - Many different parameterizations
- Think of it as time in the pen analogy
  - Parameterization says “where is pen at time T”
  - Many different ways to trace out the same curve have different timings
- Can “reparameterize” a curve
  - Same curve, different parameterization
  - Add a function f(t) -> f(g(t)) $g \in$ R->R

Some nice Parameterizations

- Unit Parameterization
  - Parameter goes from 0 to 1
  - No need to remember what the range is!
- Arc-Length Parameterization
  - Constant magnitude of 1st derivative
  - Constant rate of free parameter change = constant velocity
  - Arc-length parameterizations are tricky

How do we define functions?

- Simple shapes: easy
- Complex shapes, divide and conquer
  - Break into small pieces, each an easy piece
  - Approximate if needed
    - Add more pieces to get better approximations
  - Need to make sure pieces connect
- Typically, pick simple, uniform pieces
  - Line segments, polynomials, …

Parametric Values for Compound Curves

- Could reparameterize however we want
- One parameter space for all pieces
- Switching at various points
- KNOTS are the switching points
  - (0, .5, 1) in the case below

$$f(u) = \begin{matrix} f_1(2 * u) & \text{if } 0 \le u < \frac{1}{2} \\ f_2(2 * u - 1) & \text{if } \frac{1}{2} \le u < 1 \end{matrix}$$

Connecting Pieces

- Only concerned about the knots
  - Assume the pieces are smooth
- Connection & Smoothness
  - Connection is a type of smoothness
- Derivative continuity
  - 0^{th} derivative = position
  - 1^{st} derivative = direction
  - 2^{nd} derivative = curvature

Types of Continuity

- C(n) continuity
  - Derivatives up to (and including N) match
  - May have less meaning since parameterizations don't mean anything
- G(n) continuity
  - C(0)
  - Higher derivatives may differ by a scale factor
  - Technically – c(n) in arc-length parameters
- How smooth?
  - C(2) = smooth in graphics
  - Higher continuity in design (boat hulls, …)

What kinds of pieces?

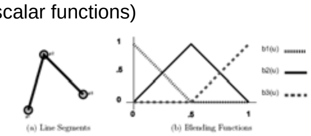
- Line segments
- Low-order polynomials
  - Quadrics (degree 2)
  - Cubics (degree 3)
  - Quartics, quintics, …
- Cubics are most popular in graphics
  - Best balance

What to control

- Control points
  - Where a curve goes (at a particular parameter value)
  - Derivaive (at a particular parameter value)
- Specify values at a site
- Specify line segment
  - End points
  - Center and one end
  - Center and offset to end
  - Center, length, orientation (non-linear change)

Line segments

- Endpoints $\mathbf{p_1}$ and $\mathbf{p_2}$
  - $\mathbf{p} = (1-u)\,\mathbf{p_1} + u\,\mathbf{p_2}$
- Blending functions
  - $\mathbf{p} = b_1(u)\ \mathbf{p_1} + b_2(u)\ \mathbf{p_2}$
  - Convenient way to describe functions (including polynomials)
  - Basis functions (scalar functions)



Line Segment Bases

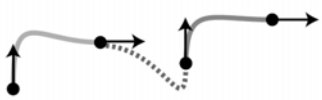
- Could choose different controls for line segment
  - Whatever was convenient
- Find conversions between different representations

Cubics

- Different than book: explain cubic forms first, derive them second
- Canonical form for polynomial
  - $f(u) = \sum a_i u^i$
  - Vector a of coefficients
- Polynomial coefficients not very convenient
  - $a_3 u^3 + a_2 u^2 + a_1 u + a_0$

Different ways to describe a cubic

- Positions of 4 points
  - u = 0, 1/3, 2/3, 1
  - Easy for 1 segment, hard to make connections
- Position & derivative at beginning and end
  - Hermite form

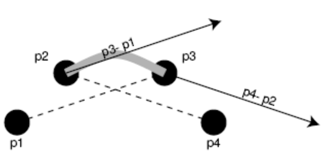


More ways to describe cubics

- Natural Cubics
  - "smoothest" curve
  - C(2)
- Each piece:
  - u=0: position, 1st derivative, 2nd derivative
  - u=1: position
- Piece 2 looks at values of previous piece (at end)
  - Propagation
- Non-local control (change at beginning changes everything)

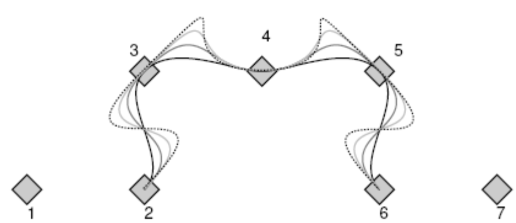
Cardinal Cubics
Catmull-Rom Splines

- Interpolate points
- Each segment interpolates p_1 and p_2
  - u=0, p_1 – derivative is k (p_2 – p_0)
  - u=1, p_2 – derivative is k (p_3 – p_1)
  - K = ½ for Catmull-Rom



Cardinal Interpolation

- k = ½ (1-t)   t = tension (0 for Catmull-Rom)



Polynomial Segments

- Canonical form: $\sum a_i u^i$
  - General – but not convenient for control
- Blending function form: $\sum b_i(u)\, p_i$
  - Canonical functions are a special case $u^0, u^1, \ldots$
  - Blending functions give easier to use points
- Example: line segment
  - Center, offset: $a_0 + a_1 u$
  - Endpoints: $b_0(u)\, p_0 + b_1(u)\, p_1$

Matrix form

- Can write canonical form as $\mathbf{a} \cdot \mathbf{u}$ $\mathbf{u}=[u^0\ u^1\ u^2 \ldots]$
- Given definitions of p, solve for **a** in terms of **p**
  - Line segment example

$f(0) = p_0$

$f(1) = p_1$

Plug in canonical equations

$p_0 = a_0\, 0^0 + a_1\, 0^1$

$p_1 = a_0\, 1^0 + a_1\, 1^1$

Matrix form

$p = C\, a$

Where

$$C = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

Basis Matrices

- The matrix C is called the constraint matrix
- The inverse of C is called B, the Basis Matrix
  - **a = B p**
- Since $\mathbf{f}(u) = \mathbf{u} \cdot \mathbf{a}$
  - $\mathbf{f}(u) = \mathbf{u\, B\, p}$ - $\mathbf{u\, B}$ are the blending functions
- In the example, **C**= [1 0 \ 0 1] **B**= $\mathbf{C}^{-1}$
  - B = $\begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$

$b_0(u) = 1-u$

$b_1(u) = u$

More complicated example: Catmull Rom Splines

- $f(0) = p_1$
- $f(1) = p_2$
- $f'(0) = ½ (p_2 – p_0)$
- $f'(1) = ½ (p_3 – p_1)$
- $p_0 = f(1) – 2 f'(0)$
- $p_3 = f(0) + 2 f'(1)$

- Remember…

$f(u) = a_0 + a_1 u^1 + a_2 u^2 + a_3 u^3$

- So

$f'(u) = a_1 + 2 a_2 u + 3 a_3 u^2$

1	1 – 2	1	1
1			
1	1	1	1
1	2 (1)	2(2)	2(3)

Catmull Rom Blending Function

- B = C^{-1}

0	1	0	0
- ½	0	½	0
1	-2 ½	2	- ½
- ½	1 ½	-1 ½	½

$b_0(u) = - ½u + u^2 - ½u^3$

$b_1(u) = 1 - 2 ½ u^2 + 1 ½ u^3$

$b_2(u) = ½ u – 2 u^2 - 1 ½ u^3$

$b_3(u) = - ½ u^2 + ½ u^3$

Natural Cubics

- Can get C(2) interpolating cubics – just not local
- Define each segment such that:
  - p_0 = f(0)
  - p_1 = f(1)
  - p_2 = f'(0)
  - p_3 = f''(0)
- Figure out beginning derivatives of piece n+1 from the end of piece n
- Changes propagate (change beginning, effects end)

CURVES 2

LAST TIME: Properties of Curves
This Time: MAKING CURVES FROM PIECES

What Kinds of pieces?
Polynomials - line segments
cubics ← why?

Polynomial

$$a_0 u^0 + a_1 u^1 + a_2 u^2 + a_3 u^3 \dots\dots$$

linear combination of blending functions

$$a_0 b_0(u) + a_1 b_1(u) \dots\dots$$

$\boxed{a_n = \text{controls}}$

simplest blending functions

$b_0(u) = 1$ $\quad b_2(u) = u^2 \quad \dots$

$b_1(u) = u$

Polynomials of degree n are a vector space
Set of functions are a basis ← just like vectors

Matrix Form $\quad \begin{bmatrix} 1 & u & u^2 & u^3 & \dots \end{bmatrix} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix} \begin{bmatrix} a_0 \\ \downarrow \\ a_n \end{bmatrix}$

different basis matrices = different sets of blending functions
= different controls

Example: LINE SEGMENTS

$f_x = a_{0x} + a_{1x}(u)$
$f_y = a_{0y} + a_{1y}(u)$ } just consider x, or use vector notation

$b_0 = 1 \quad b_1 = u$

a_0 and a_1 aren't convenient controls

want p_0, p_1 = end points

new blending functions:

$b_0 = (1 - u)$ $\qquad f(u) = p_0 \, b_0(u) + p_1 \, b_1(u)$

$b_1 = u$

overkill for line segments
better for harder things

Deriving the basis matrix / blending functions

write constraints

$p_0 = f(0)$ @ $u = 0$

$p_1 = f(1)$ @ $u = 1$

convert $p \to a$

$p_0 = a_0 + \cancel{u}^{0} a_1$

$p_1 = a_0 + \cancel{u}^{1} a_1$

$p = C a \qquad C = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$

$a = \underbrace{C^{-1}}_{B} p$

$f(u) = u B p$

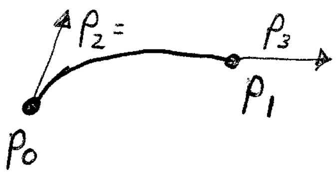
$b(u) = u B$

Cubics

$$a_0 + a_1 v + a_2 v^2 + a_3 v^3$$

Really inconvenient!

Hermite form:



easy for make $C(1)$

not same as book!

$$p_0 = f(0) = a_0 + a_1 \cancel{v_1}^{0} + a_2 \cancel{v^2}^{0} + a_3 \cancel{v^3}^{0}$$
$$p_1 = f(1) = a_0 + a_1 \cancel{v_1}^{1} + a_2 \cancel{v^2}^{1} + a_3 \cancel{v^3}^{1}$$
$$p_2 = f'(0) = a_1 + 2a_2 \cancel{v}^{0} + 3a_3 \cancel{v^2}^{0}$$
$$p_3 = f'(1) = a_1 + 2a_2 \cancel{v}^{1} + 3a_3 \cancel{v^2}^{1}$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 2 & 3 \end{bmatrix} \qquad B = C^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -3 & 3 & -2 & -1 \\ 2 & -2 & 1 & 1 \end{bmatrix}$$

$$b_1(v) = 1 - 3v^2 + 2v^3$$
$$b_2(v) = 3v^2 - 2v^3$$
$$b_3(v) = v - 2v^2 + v^3$$
$$b_4(v) = -v^2 + v^3$$

Check to see interpolation

CS559 – Lecture 17
Approximating Curves

These are course notes (not used as slides)
Written by Mike Gleicher, Oct. 2005
Updates Oct 2006

Approximating Curves

- Interpolation isn't the only way to describe a curve
- Give points that "influence" a curve

- Why?
  - Better control of what happens in between points

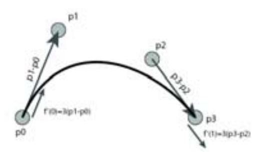
- 2 important cases for computer graphics
  - Bezier
  - B-Spline

Bezier Segments

- Curve is made of many segments
  - Nomenclature issue
- Each segment is a polynomial
  - Of any degree
  - 3 is most common in computer graphics

Bezier Segments

- A segment of degree d has (d+1) control points
- A segment interpolates its first and last controls
  - With u= 0, 1 respectively
- The first derivative at the beginning (end) is proportional to the vector between the first 2 (last 2) points – scaled by the degree of the curve



Can we do Beziers this way?

- Yes – set up constraints and solve
  - $f(0) = p_0$
  - $f(1) = p_3$
  - $f'(0) = 3(p_1 - p_0)$
  - $f'(1) = 3(p_3 - p_2)$
- Doesn't generalize well (OK for 2,3,4)
- General form for Bezier blending functions
  - Bernstein Basis Polynomials

$$b_{k,n}(u) \;=\; C(n,k)\; u^k\; (1-u)^{(n-k)}$$

$$C(n,k) = \frac{n!}{k!\;(n-k)!}.$$

Bezier Segments (2)

- The nth derivative depends on the first (or last) n points

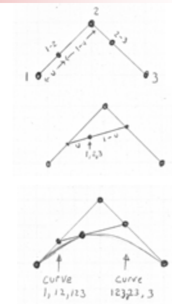
- Cubics are similar to Hermites
  - All points in space (not derivative amounts)
  - Scaling factors
- Pieces connected by placing points correctly
  - C(0) by matching endpoints
  - C(1) by aligning end vectors
  - G(1) by end-vectors being co-linear

Properties of Bezier Curves

- Simple mathematical form for basis functions
- Good algorithms for computation
  - Subdivision procedure
  - De Casteljau algorithm
  - Divide and conquer because…
- Convex Hull Properties
- Variation Diminishing
- Symmetric
- Affine invariant
  - NOT perspective invariant

De Casteljau Algorithm

- Evaluate curve at u
  - Divide line segments
  - U of the way
- Can use to subdivide curves
- Repeated linear interpolation for ANY degree!



Decastlejau to Bernstein

- Apply geometric construction to derive equations
- Different groups came at this differently
- Algebraic vs. Subdivision

$$pt\,12 = (1-u)\,P_1 + u\,P_2 \qquad pt\,23 = (1-u)\,P_2 + u\,P_3$$

$$pt\,123 = (1-u)\,P_{12} + u\,P_{23}$$

$$f(u) = (1-u)\big((1-u)P_1 + uP_2\big) + u\big((1-u)P_2 + uP_3\big)$$

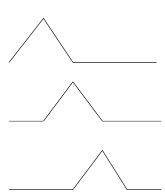
$$= \underbrace{(1-u)^2}_{b_1} P_1 + \underbrace{2u(1-u)}_{b_2} P_2 + \underbrace{u^2}_{b_3} P_3$$

How do we make Smooth Curves?

- Will be approximating
- Want flexibility
  - Any number of points
  - Any degree of continuity
- Want good properties
  - Locality
  - Convex Hull
  - Variation Diminishing
  - Shift Invariant (sum of blending functions = 1)

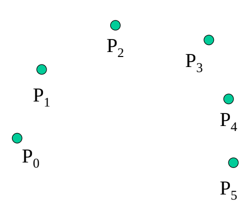
An example of blending functions

- Lines
- Blending functions are "bumps"
- Each piece is a spline
  - Two polynomials, degree 1
- Locality
  - Non-Zero over small range
  - Between 3 knots (d+2)
- Smoothness
  - D-1 continuity
- Shift invariant
- Convex Hull Property
- Shiftable (periodic)



Consider B-Splines

- N points
  - General – any N
  - $P_0 \ldots P_{n-1}$
- WARNING: not same notation as book
- Consider linear interpolation
  - $F(t)\ t \in 0 \ldots n-1$
- $F(t) = \sum b_i(t)\, P_i(t)$

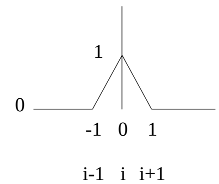


B-Splines

- General scheme for generating blending functions
  - Any number of points (need more points than degree)
  - Any degree of polynomial (higher degree = smoother)
  - Any knot vector
- Blending function of degree D are B-Splines
  - Made of D+1 segments (span D+2 knots)
  - Each segment is a degree D polynomial
  - Only D+1 of them are non-zero at any time
  - Sum to one
  - Zero outside of the range
  - D-1 continuous
- Note: usually talk about "order" (degree+1)

Linear B-Spline

- Each blending function is a bump
- All the same (different ones are shifts)
- Active from i-1 to i+1
  - Over 2 spans, 3 integers
- In between 2 pts are active
  - One in each "phase"
- "before" t=0 and "after" n-1
  - Not enough points



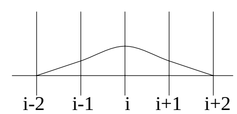
Warning:
This B-Spline is centered
Other notations start at 0

Creating B-Splines

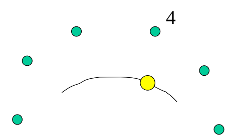
- Cox-de Boor recurrance
- Convolutions of the unit box
- When d>1, the functions do not interpolate
  - They never reach the value of 1

Cubic Blending Functions

- Active over 4 regions (d=3, k=d+1=4)
- At any time, one point in each phase

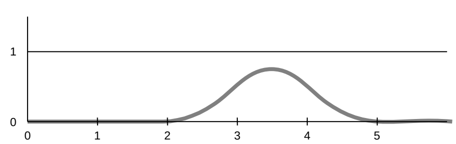


- Example t=4.5
  - Eval point 3 @ 1.5
  - Eval point 4 @ .5
  - Eval point 5 @ -.5
  - Eval point 6 @ -1.5
  - Each in a different part



Quadratic, Uniform B-Splines

$$b_{i,3}(t) = \begin{cases} \frac{1}{2}u^2 & \text{if } i \le t < i+1 \quad u = t - i \\ -u^2 + u + .5 & \text{if } i+1 \le t < i+2 \quad u = t - (i+1) \\ \frac{1}{2}(1-u)^2 & \text{if } i+2 \le t < i+3 \quad u = t - (i+2) \\ 0 & \text{otherwise.} \end{cases}$$



Even More General?

- B-Splines cannot represent conic sections
  - Can't make an exact circle
- Express curves / surfaces as the RATIO
- Non-Uniform Rational B-Spline Surfaces
  - (NURBS)
- Extensions to surfaces later in the class

Knot vectors

- Allow us to assign parameter values to points
- Makes it possible to alter the set of points but keep parameter values fixed

- Allows us to alter the spacing
- Allows us to create discontinuities

- (picture with lines)

- Uniform vs. Non-Uniform

Using B-Splines

- Figure out closed form basis functions
  - Rather than using Cox-de Boor
- Can encode into a Basis matrix
  - But cannot derive the same way

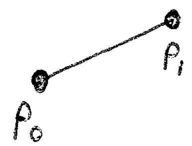
- Periodic basis functions are nice
  - Implement once

- Gives a nice way to get very smooth curves
  - Cubics (usually) in graphics to provide C(2)

MAKING A CURVE THROUGH LOTS OF POINTS

Blending Functions → B-Splines

Consider 1 piece

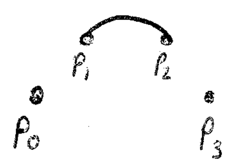


$f(u) = b_0(u) P_0 + b_1(P_1) \qquad u \in 0,1$

$b_0(u) = 1 - u$

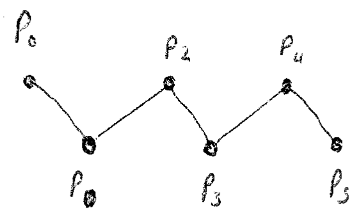
$b_1(u) = u$

or



$f(u) = \sum b_i(u) P_i$

Consider Multiple Segments ~~$t \in 0,n$~~



n points

$n-d$ segments

same blending functions - different points

one parameterization → shift to 0,1

$t \in 0,n$ in segment $\lfloor t \rfloor$

each point gets each blending function at some point

$\sum_i B_i P_i$

→ next page

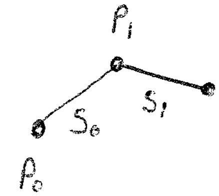
2

Write 1 blending function per point $\qquad \sum B_i P_i$

will have an if statement

1 clause for each

0 outside range



will have the form

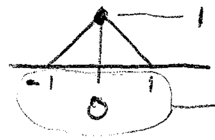
$$
\left.
\begin{array}{ll}
B_i = & \text{if segment } i \ (t \in i \to i+1) \ b_0 \\
 & \text{if segment } i-1 \ (t \in i-1 \to i) \ b_1 \\
 & \vdots
\end{array}
\right]
$$

d clauses

end cases miss some

What do these functions look like?

lines:

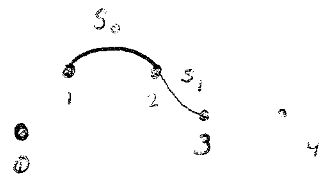


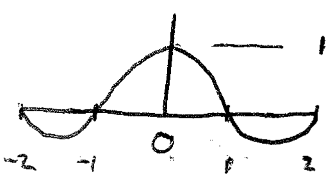
cardinal cubics

point i is interpolated:

beginning of segment $i-1$

end of segment $i-2$





the zero is $i-1$

notice 4 segments - so 5 knots

To make our notation easier (?) don't center the bumps. Besides, there might not always be a center

Properties of blending functions d points / degree

n blending functions

n + d + 1 knots (check)

any time d active] valid range 0.. n − d

sum to 1

1 in each phase

symmetric

locality

What if you want $C(d-1)$ continuity?

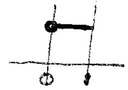
line segments do this

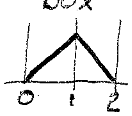
- it is sufficient if the blending functions are $C(d-1)$ since we're just taking linear combinations

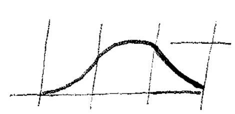
B-Splines (Uniform)

for convenience, "start" @ 0

Convolutions of the unit box ← cox de-boor recurrance

$B^0 =$ 

$B^1 =$ 

B^2  never gets to 1 so not interpolating

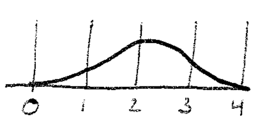
$B^3 =$ 

figure out B (from properties)

use it to get $b(v)$

So cubic B-Splines <u>can</u> be a basis matrix

$$[1 \quad v \quad v^2 \quad v^3] \; B$$

you just derive it differently

$B_{i,d}$

different blending functions are shifts of one another

Why B-Splines

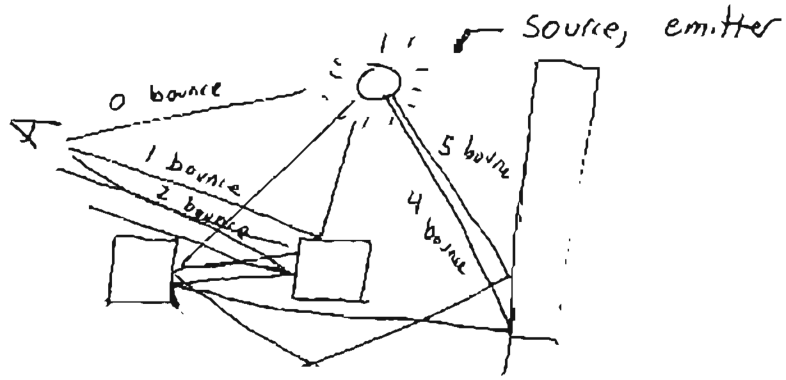
- smooth $C(d-1)$ $C(2)$ w/ cubics
- local control - a point only affects d segments
- splines (piecewise polynomials)
- easy to implement
- mathematically well behaved

Extensions:

Non-Uniform Knot Spacing
Rational (projective invariance)

What COLOR DO WE MAKE THESE SOLID OBJECTS?
DEPENDS ON OBJECT <u>AND</u> LIGHT!

HOW DOES LIGHTING WORK?



IN THE REAL WORLD .....
LIGHT BOUNCES OFF EVERYTHING
All objects influence all others

GLOBAL ILLUMINATION
hard to do - must consider all objects, interactions, ... interdependence (1 depends on 2 depends...)
good for getting complex lighting effects
an advanced topic

IN THE CG WORLD .....
LOCAL LIGHTING -
decision of how to light a point on ~~that~~ an object depends on:
- surface <u>AT that point</u>
- eye position
- lights

LOCAL LIGHTING :

Consider only 1 point on 1 object

No shadows
No self shadows
No color spill
No inter-reflection

if you want these, add with a hack

No area light sources - point sources only
might be at infinity

3 parts (per light)

specular (direct reflection)
diffuse (scattering)
ambient (hack for indirect)

Lighting is a hack

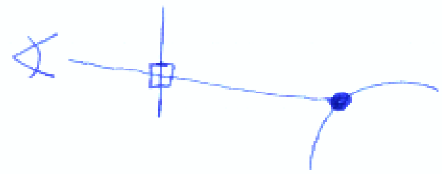
real lighting is complex
microstructure of materials

get "biggest" features of lighting correct

fancier models are still hacks + just get more features right

LIGHTING & SHADING

What color is a point?



Physics: depends on how light interacts with all objects in scene
- some of the object's reflected light goes off towards eye ⇒

CG: do some computation to determine color <u>Shader</u>

color = Shader (info)

what info do we give the shader?

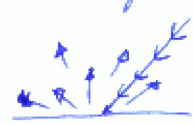
Simple Shading:
object properties (color) reflectance
light info (position, color, intensity)
eye position
local geometry (position, normal)

Diffuse Shading -

matte objects
rough surfaces
"micro surface texture" scatters light in all directions

chalk, paper, unpolished wood or stone, ...

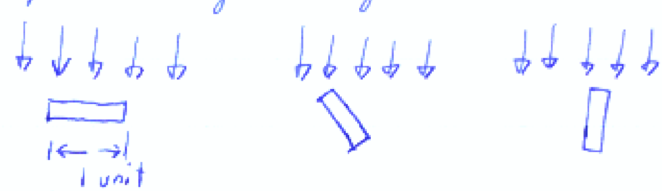
Lambertian reflector
scatters light in all directions equally



eye position doesn't matter

light position DOES matter
(relative to surface orientation)

consider fixed sized object:

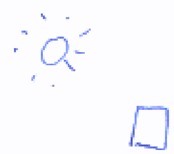


amount of light that hits is $\approx$
$\cos\theta$ where $\theta = \measuredangle$ between light and normal

$D_L \approx \hat{n} \cdot \hat{l}$

One last problem -

What about inter-reflected light -
room isn't totally black



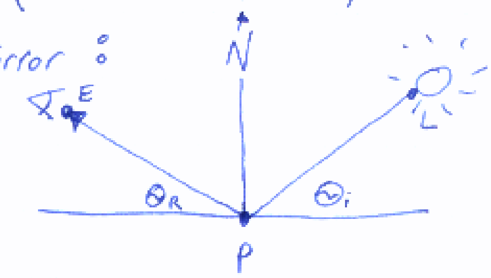
← this side of object should have
<u>some</u> light

"Ambient" light ≡ indirect light that is just
bouncing around

Hack : Add in a light source that effects all
objects equally - Ambient lighting

Specular (direct reflection)

Perfect mirror:



ping-pong ball model

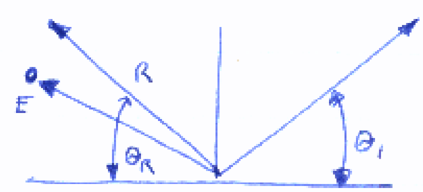
∡ incidence = ∡ reflection

light gets to eye only if things line up exactly

HACK → if it's close to the eye, that's good enough

falloff as it gets further away

~~define~~



$L \approx \hat{E} \cdot \hat{R} \cdot C_L$ ← color and brightness

need a falloff function

Phong Model $L \approx (\hat{E} \cdot \hat{R})^p C_L$ ← specular co-efficient

Eeasier Way H = half-way angle

$L = (\hat{N} \cdot \hat{H})^p C_L$

Hack Lighting Model (GL)

1. Eye Position
2. Object Local Geometry (NORMALS)

3. Each light source has a position (may be at infinity) and a brightness (color) I_i
4. Ambient light has a brightness (color) A

5. surface has a diffuse reflective color C_D
   a specular color C_S — white or $= C_D$
   a shinyness s
   an ambient color (reflection) C_A

these are usually the same

$$\text{color} = A * C_A + \sum_{i \in \text{lights}} \left(I_i * \left(C_D \cdot (\hat{n} \bullet \hat{l}) + C_S (\hat{N} \cdot \hat{h})^s \right) \right)$$

Some improvements :

1. falloff (brightness depends on distance)
2. more sophisticated ways of finding C_D, C_S based on position
3. more complex reflectance functions
   BRDF ≝ bi-directional
   reflectance
   distribution function

   given -
   input direction & output direction ⇒
   reflectance

How to use this?

Polygons are all the same color (one normal)

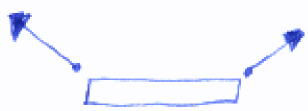
FLAT shading

$\Rightarrow$ approximation $\hat{L}$ and $\hat{E}$ do change, only a little

Problem:

polygons are an approximation to a smooth surface

normal per vertex



(1) compute color at vertices
linearly interpolate color

GOURAUD Shading

(2) linearly interpolate normals
compute lighting per-pixel

PHONG SHADING

(do not confuse with Phong LIGHTING)

CS559 – Lecture 20
Texture Mapping

These are course notes (not used as slides)

Written by Mike Gleicher, Nov 2006

Goal: Complexity

- How to make something complex?
- Given what we have: lots of small triangles
  – To now, Gouraud shading – color per vertex
- Why not?
  – Hard to model / author / design
  – Hard to draw fast
  – Hard to sample (triangles get smaller than a pixel)
  – Hard to maintain the models
  – Hard to store the models

Alternative Approach to Complexity: "Texture" Mapping (and its variants)

- Use simple geometry (big polygons)
- Vary color (and other things) over its surface
- Analogy: paint a picture on something
- Basic case: change color at each point
  – Advanced cases later

Why just paint objects?

- Why paint rather than model?
  – Easier (can use 2D tools, photographs)
  – Less to store
  – Less to model
  – Faster to draw (*)
  – Easier to sample
- Why not?
  – Things really aren't flat
  – Parallax / self-shadowing / illumination effects
  – More advanced "texturing" to get these later

Faster to draw requires special hardware!

Only recently has this become common!

Texture Mapping

- For every point on the object, have a "map" (function) to color
  – Later extend to other properties
- Big pieces here:
  – Need ways to "name" points on object
    Texture Coordinates
  – Need ways to describe the mappings
    - Procedural
    - Use images

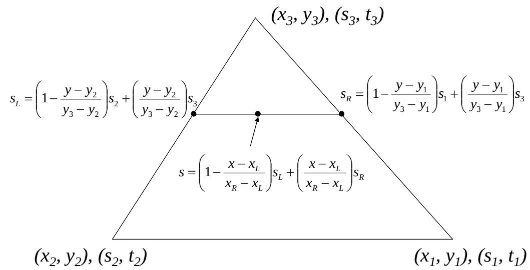
How to assign points to objects

- Use world space positions?
  – No – properties usually move with objects
  – Might be OK for things like lights that effect objects
- Use local 3D positions?
  – 3D Textures
  – Problem: harder to define functions that give colors for all points in a volume
  – Don't care about points off the surface anyway
  – Use 3D textures when its easy to make 3D functions
    - Procedural wood, stone, …

2D Texture Mapping

- So common, its almost synonymous with Texture
- For every point, give a 2D coordinate
  - Texture coordinate
  - U,V for every vertex
- Interpolate across triangles
  - (same as across quads)

Interpolating Coordinates



Barycentric Coordinates

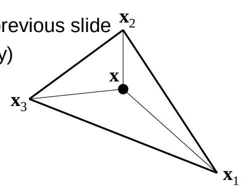
- An alternate way of describing points in triangles
- These can be used to interpolate texture coordinates
  - Gives the same result as previous slide
  - Method in textbook (Shirley)

$$\mathbf{x} = \alpha \mathbf{x}_1 + \beta \mathbf{x}_2 + \gamma \mathbf{x}_3$$

$$\alpha = \frac{\text{Area}(\mathbf{x}, \mathbf{x}_2, \mathbf{x}_3)}{\text{Area}(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)}$$

$$\beta = \frac{\text{Area}(\mathbf{x}_1, \mathbf{x}, \mathbf{x}_3)}{\text{Area}(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)}$$

$$\delta = 1 - \alpha - \beta$$

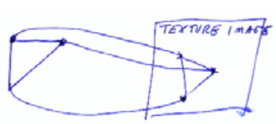


How to represent the function

- C(u,v)
  - Write code (needs programmable graphics system)
    - Programmable shaders (later in course)
  - Use an image and sample
- Sampling is an issue even for procedural texture
  - Its just harder!
- One pixel can be a large part of a triangle

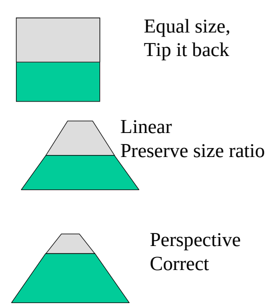
Image Based Texture Maps

- So common its synonymous
- U,V coords at vertices
- Specify where in texture to get colors



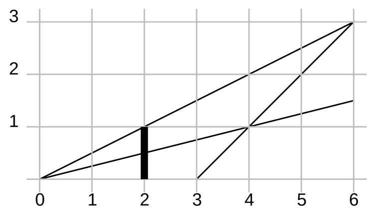
Perspective Correction

- Linear interpolation wrong if polygon isn't screen aligned
- Stuff farther away needs to be smaller
- Need to interpolate in world space, then do perspective
- Need to interpolate w, and divide (per-pixel)
- Divide per pixel used to be expensive



Perspective Correct Texture Mapping

- Don't worry – the graphics hardware does it
- 1/Z (or 1/W) is linear in screen space
  - This is a little tricky to prove



1/2 in screen line

Interpolate 1/3->1/6
halfway = 1.5 / 6

Z of halfway point=4

1/3 of line

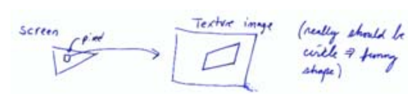
To do perspective correct

- Interpolate 1/Z (or 1/W)
- Compute Z (from 1/Z) – requires divide
- Compute fraction of way from begin to end in Z
- Use this fraction to get how far in U/V
- Can combine steps
- Big picture – need to do a divide for every conversion (pixel)
- See Shirley for details

Sampling

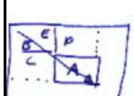
- Have U,V for the pixel – what color is it?
- Look it up in the texure map
- Point sample
- Bilinear interpolation (if between pixels)
  - Always will be between pixels
- Filtering – pixel maps to a region of texture

Fast Sampling



- Screen pixel is funny shape in Texture Space
- Perespective transform of circle (skewed ellipse)
- Use a simpler shape for sampling

Average over rectangular regions



sum over region in constant time w/ precomputed

table – area above and to the left

$A = B - C - D + E$

← 4 lookups, but need table – overflow issue

need to know rectangle

Square Region Centered at Point

- Pretend pixels are squares
- If region is 1 pixel big, this is easy!
  - Use bilinear interpolation to get position right
- If the region is bigger, halve both region and image
  - 2x2 region – halve the image (each pixel is average of a 2x2 block)
  - 4x4 region – halve the image twice

MIP Map

- Repeatedly halve the image to make a "pyramid"
  - Until there's 1 pixel (which is average of whole)
- Given a position and square size
  - Use square size to pick pyramid level
  - Use bilinear interpolation to get position
- But only have pyramid for 1,2,4,8… pixel squares
  - Linear interpolate between levels!
  - E.g. 5 = ¼ way between 4 and 8, so compute 4 and 8 and interpolate
  - Tri-Linear Interpolation! - looks at 8 texels (4 per level)

Making Textures Work

- Need to load textures into FAST memory
  - Multiple lookups per pixel
- Need to build MipMaps
- Need to give triangles UV values
- Need to decide how to filter
- How is texture color used
  - Replace existing color?
  - Blend with it?
  - Before or after specular highlight?
- Need to decide what happens to "out of bounds" texture coordinates
  - Clamp, repeat, border

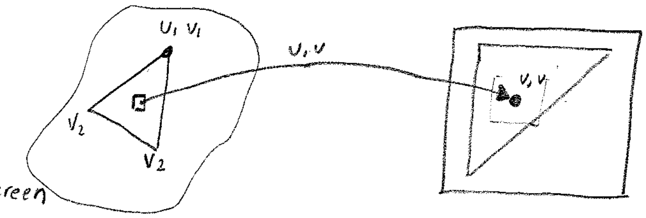
More stuff with textures

- Lots of extensions and uses!
- Multi-Texturing (combine several textures)
- Bump Mapping – lookup normal values
- Displacement Mapping
- Textures for lighting and shadows
- Can fake many complex effects by using texturing in interesting ways
  - Draw many times – each with another texture

TEXTURE MAPPING 2

RECAP:

Object / Triangle
Texture Coordinate
Interpolation Linear in Space / Perspective on Screen



For each pixel do lookup for color
Bilinear interpolation since u,v is continuous
TRI-LINEAR interpolation in MIPMAP to get areas

Small GOTCHA

Lighting computed at VERTEX
Color at PIXEL

1 do GOURAUD Shaing

Texture Modulates ← multiplies color

$$\text{color} = C_D (N \cdot L) + C_S (H \cdot N)^p + C_A L_a$$
$$\text{color} = C_T \left(\text{———} \right)$$

→ Make objects white / mult over color
→ Doesn't allow control of specular color sepeate
GL has a workaround
→ Modulate color
other choices (replace, subtract)
Opportunity to <u>combine</u> multiple color sources
Combine Multiple Textures = MULTI-TEXTURE
(more on that Later)

Additional Texture Map Tricks

Objects Look too flat

BUMP MAPPING ← use texture to change normal
need to compute lighting per-pixel (since normal changes)

Surface is flat, but reflects as if bumpy.
AKA Normal MAPS
NOTE: offsets to "real" normal are whats stored
for triangle have "u" and "v" vector

$$n = \underbrace{n + \alpha u + \beta v}_{\text{vectors from triangle}}$$

Only changes lighting
- no self-shadowing (hacks to fix)
- doesn't change sihouette
- per-pixel lighting required

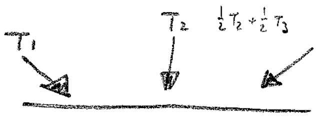
DISPLACEMENT MAPPING

- ACTUALLY MOVE POINT (which changes normal)
- Very hard to do (pixel might move to other pixel)
- Gets realistic effects

How to fake this without per-pixel computation

View Dependent Texture Mapping

- determine what object looks like under different view directions
- Blend between different textures



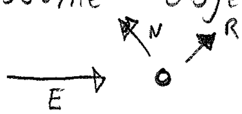
Use of Multi-Texture
1. Multipass w/ multiply or blend
2. Texture Combining

- Use texture over whole triangle
  per-pixel effects pre-computed (need some way to compute them)
- can vary based on light direction as well

More Mapping:

Environment mapping

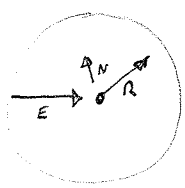
- make mirror reflections
- assume object is infinitessimal sphere



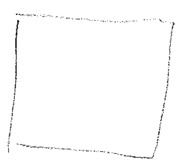
R depends on E, N

assume E is constant

Use R to Look up into Map of "Environment"



← spherical environment map



cubic environment map

Cylindrical environment map

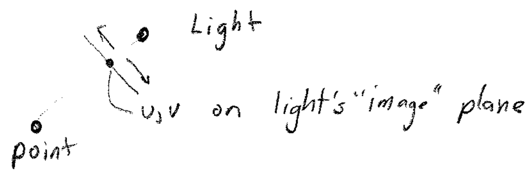
Lighting w/ Texture

Environment Map is mirror reflection (specularity)
put lights in it to get realistic lights
very bright spots in textre (use of HDR)

"Paint" Lighting Details on to objects

→ MULTI - TEXTURE TO ADD LAYERS OF LIGHTS

→ Slide Projector Mapping



Shadow MAPS - something different

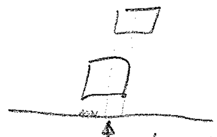
1. render scene from lights point of view
2. visible objects are lit, occluded are in shadow
   render and keep the z-buffer (the shadow map)
3. draw from the camera's viewpoint
   for any pixel, see its distance from light
   check in map to see it occluded

Hack Shadows / Hack Lights

draw black or light splotches

How to control where they go?

How to avoid overdraw



shouldn't be twice as dark, but if using blend w/ black it will be

Stencil Buffer

A buffer you can do anything with

Write values w/ drawing

Test values w/ drawing

Example

1. clear to zero
2. draw ground set stencil bit
3. draw shadows
   - only draw when stencil bit set
   - reset stencil bit when drawing

Lecture 22 –
3D Modeling & Polygons

Michael Gleicher
November, 2006
Used as notes, not projected as lecture

Modeling 3D Shapes

- Modeling = process of describing an object
  - Representation
- Can model shape, physical properties, behvaior, …
- Many uses of (geometric) models
  - Graphics – make a picture
  - CAD – represent for manufacture

Types of Shape Models in 3D

- Points
- Curves
- Surfaces and Solids
- Volumes

Surface vs. Volume

- Cube
  - Volume = space inside 0 <= x,y,z < 1
  - Surface = 6 squares (0,0,0)(0,0,1)(0,1,1)…
- Surface can be a boundary
  - But might not be
- Graphics (often) only need surfaces

When might we care about Volumes?

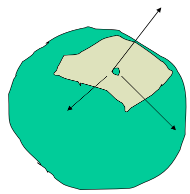
- Engineering / Manufacturing / Design
  - Can't be non-physical
- Some kinds of data has "insides"
  - Medical data (scanned)
- Some operations make sense
  - Constructive solid geometry
  - Cut / Join / Subtract / Union
    - Makes less sense on surfaces

How to do volumes?

- Hard: need to insure that you always have a volume!
- Operations on primitives
  - Make solid pieces (spheres, cylinders, polyhedra, …)
  - Combine with sensible operators (union, intersection, difference)
  - Construction Solid Geometry
- Boundary Representations
  - Store the surface
  - Represent what's inside
  - Be careful that there always is an inside – no holes!
- Implicit Representations
  - F(x) < 0 – for some fancy F
  - Distance fields, union of blobs, …
  - Tend to be special purpose
- Sampled Volumes (like medical data)

Surface Basics

- Locally flat
- At any point
  - Normal
  - Tangent Plane
  - Tangent vectors in plane

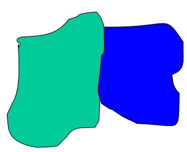


Surfaces

- Generally what we use in graphics
  - Hard enough!
- Similar issues to curves, but worse
- Named vs. Free-Form
- Build out of little pieces
- Linear pieces (polygons) – analogy to lines

Basic Strategy

- Break complicated surfaces into pieces
- Need to choose good pieces
- Need to make sure that the pieces connect
- Connections are more complicated



Polygons

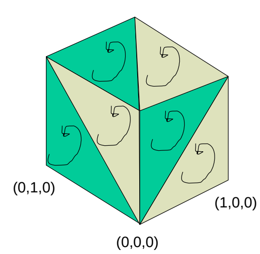
- Or triangles
- Need to have a front/back
- Outward facing normal
- Be consistent in orientation (e.g. CCW)

Polygon Soup

- Random Assortment
- Unstructured
  - At least get ordering right
- Tells little about how polygons connect
- Lots of redundancy

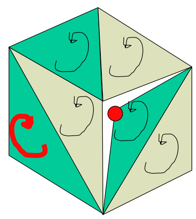
Cube Soup

```
struct Triangle Cube[12] =
    {{{1,1,1},{1,0,0},{1,1,0}},
     {{1,1,1},{1,0,1},{1,0,0}},
     {{0,1,1},{1,1,1},{0,1,0}},
     {{1,1,1},{1,1,0},{0,1,0}},
     …
    };
```



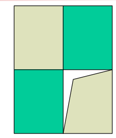
Polygon Soup

- Advantages
  - Easy
- Problems
  - Redundancy
  - No global info
  - No open/closed info
  - Hard to edit
  - Hard to prevent degeneracies
  - No non-local information
    Is it closed?
    Is it connected?
    Is this an edge or internal?

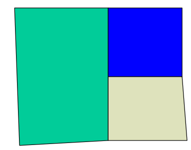


Cracks / Cracking

- Gaps in the surface
- Prevents from being solid
- Can be ugly
- Airtight / Watertight
  - No cracks



- Beware edge/vertex
  - Numerical errors cause cracks



Mesh

- Share vertices
  - Indirection to vertex table
  - Prevents cracking
  - More efficient (lots of info at vertex)

- Store Polygons as vertex lists
- Store Edges – Faces are lists of edges
  - Every edge borders 2 faces

- Simplicial Complex
  - Mathematically deep term
  - Fancy way to say “nice mesh” – all faces meet at an edge, …

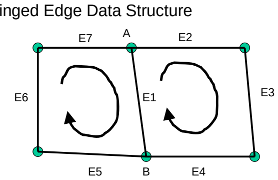
Vertex Indirection

- List of vertices
- Everything is an index into this table

- Good points:
  - Sharing prevents some cracking
  - Transform/Light each vertex once
  - Data reduction

More complex Mesh Structures

- Store Edges
  - Can be handy to have
- Each edge only 2 faces – one CW one CCW (pass through edge in opposite ways)
  - Store “next” edge for each direction
  - Winged Edge Data Structure



E1: A->B

Forw: next=E5, prev=E7

Back: next=E2, prev=E4

Getting Meshes to Hardware Fast

- Minimize number of vertices sent down pipe
  - Old days – definitely bottleneck
  - Now – maybe not, since lots of per-pixel computation

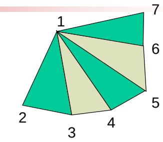
- Vertex Buffers
  - Send small number of vertices
  - Index into this small array (since memory << model size)
  - Group into small sets (like 8 or 16) of vertices, draw all triangles between them
- Vertex Cache
  - Automatically buffer, use LIFO

Vertex Arrays

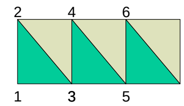
- Hardware caches vertices (after transform)
- Give vertex list and connectivity
- Do in an order to get cache performance
  - Groups of n vertices
- Hardware specific trick
- Best way to draw triangles in opengl
- Send blocks of data at once (avoid function call overhead)
  - Can be high since function call means call to low-level driver
- Possibly: store array in fast memory specific for graphics
  - On graphics card or in driver address space
- Issues with data formats

Regular Meshes

- Reduce number of vertices needed
- Reduce amount of connectivity info needed (which can be sizable!)



- Often have meshes with uniform patterns
- Grids, fans, strips



- Connectivity is implicit
- Very efficient
- Processing is easy
- Avoid redundant transforms

Lecture 23 – Surfaces

Michael Gleicher
November, 2006
Used as notes, not projected as lecture

Normals

- Per Face
  - Can be computed (assume polygon, order)
- Per Vertex
  - Assumes we're approximated smooth surface
- Per Face/Vertex
  - If you want discontinuous normals

Smooth Surfaces

- Approximate with polygons
- Consider Cylinder
  - Number of faces
  - Better looking with smooth shading
  - Even better with Phong Shading
- Tradeoff: more polygons = smoother
- Better: use pieces that are really curved
  - Will (almost always) draw them by tesselating
  - But – easier to model with fewer pieces,more accurate, adaptive, …

Surface Representations

- Very similar to curves
- Implicit f(x,y,z) = 0
- Parametricf(u,v) = 0
- Procedural f(?) -> points, polygons, …
- Subdivision
- Old days: parametric surfaces were kind
- Now: Subdivision!

Surface Patches

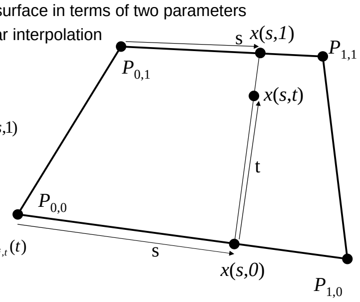
- A square (u,v) in (0->1, 0->1) that gets mapping into space
- Put squares together
  - Continuity Issues at edges
- Cut holes in patches
  - **Trim curves** defined in parameter space
- Stitch together at seams
  - Like sewing – cut pieces and sew them together
- Making things fit together requires dealing with the complicated math of the curve boundaries

Parametric Surfaces

- Define points on the surface in terms of two parameters
- Simplest case: bilinear interpolation

$$x(s,0) = (1-s)P_{0,0} + sP_{1,0}$$
$$x(s,1) = (1-s)P_{0,1} + sP_{1,1}$$
$$x(s,t) = (1-t)x(s,0) + tx(s,1)$$

$$F_{0,s} = 1-s, \quad F_{1,s} = s$$
$$F_{0,t} = 1-t, \quad F_{1,t} = t$$
$$x(s,t) = \sum_{i=0}^{1}\sum_{j=0}^{1} P_{i,j}F_{i,s}(s)F_{j,t}(t)$$



Bilinear Patches

- Edges are lines (so its easy)
- Patches are not flat (actually are curved)
- For a specific u, line in v
- For a diagonal line in u,v, a curve (quadratic actually)
- How do I cut a circular hole in the patch?
- (and bilinear is the easiest!)

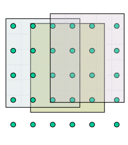
Tensor Product Surfaces

$$\sum_{i<d}\sum_{j<d} a_{ij}u^i v^j$$

- Polynomial in u and v
- Just like with curves, coefficients aren't the easiest – so switch bases
- Just a lot more control points
  - D^2 (16 for cubics!)
- A nightmare to derive…
- Note for fixed u or v, its just a polynomial in the other variable
  - Patch edges are polynomial curves

Tensor Product Cubics

- Each patch needs a 4x4 grid of control points
- Need to be very careful to make sure that there is continuity across edges
- B-Splines, Beziers, Cardinals, …



•Must be a regular grid

•Every point is in 16 patches

•Can't insert detail locally (need to add an entire row/column)

NURBS

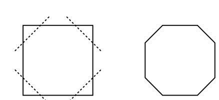
- Each patch is a B-Spline (often cubic)
- Need Rational B-Splines to make spheres and conics
  - And projective invariance
- If you thought B-Spline curves were hard…
- Issues in trimming
- Issues in stitching (without cracking)
- Issues in adding detail
- Issues in tesselating it well
- We won't bother teaching you about these anymore

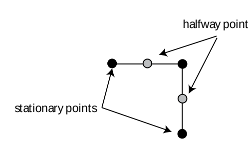
Subdivision Basics

- Works for curves (as well as surfaces and volumes)
- Becoming more popular (for reasons we will see)
- Idea:
  - More polygons (or linear elements) = smoother
  - Define rules to make more polygons "smoothing" a simple shape
  - Given a shape: smooth "enough" to get desired result
  - Define rules so that in the limit the surface is really smooth
  - Exact evaluation lets us determine limit surface directly

A simple subdivision curve scheme

- Corner cutting
  - Corners are too sharp?
  - Cut them off!
- Break edges into pieces
  - To make the corners
- Each corner is defined by 3 points
  - Two points are stationary
  - One point gets cut off

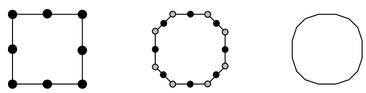




Gets smoother each time

- Repeat
- Keeps Getting Smoother
- Do this until its really smooth

- Infinitely many times?
- Notice: this is the DeCasteljau Algorithm! (quadratic, u=.25)



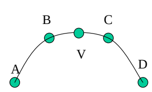
Limit Surface

- The surface we get if we subdivide infinitely many times
- MAY be smooth (someone has to prove this)
- MAY be a way to determine without doing infinite steps

- Corner cutting:
  - We know where stationary points go
  - We know what the tangent is (in the limit)
    - Use Bezier rules

Other Schemes

- Interpolating
  - All points are stationary
  - Insert new points in between existing points
- Approximating
  - All points are moved
  - In general: insert new points, then move old ones

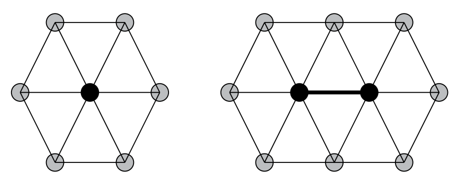


Catmull-Clark Edges
* insert midpoints of edges
* old points = 1/8 ¾ 1/8

V = -1/16a + 9/16 b + 9/16 c – 1/16 d

Schemes for Triangles

- Ordinary vs. Extra-Ordinary Points
  - Regular triangle grid
    - 6 neighbors per vertex
    - 8 vertex neighbors per edge



Lecture 24 – Subdivision

Michael Gleicher
November, 2006
Used as notes, not projected as lecture

Last time

- Representing Smooth Surfaces

- Piecewise parametric surfaces (tensor products)
- B-Splines, NURBS
- General, mathematically elegant
- Problematic

- Subdivision
  - Basic ideas
  - Schemes for curves

Subdivision Concepts

- Start with initial, discrete representation
  - Control points, line segments (for curves), polygons (for surfaces)

- Subdivision rules to make finer resolution
  - Still get a discrete approximation to smooth thing
- **Limit Surface** (or Curve)
  - The "mathematical" result is what happens after infinite steps

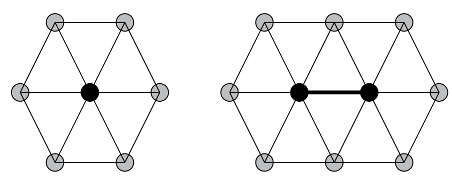
- **Exact Evaluation** – tells about points on Limit Surface

- Stationary Points / Schemes – stay put (interpolate)
- Non-Stationary Points – move (approximate original)

Regular Meshes

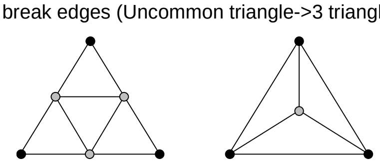
- Triangle "Grids" (regular hex patterns)
- Quad Grids (squares)

- Semi-Regular Meshes (few "extraordinary" points)



How to divide triangles

- Need to do all triangles the same way
  - Can't break edges on one side, not the other
- Break edges at midpoint
  - Common way each triangle -> 4 triangles
  - All new points are ordinary (or edges)
- Don't break edges (Uncommon triangle->3 triangle)

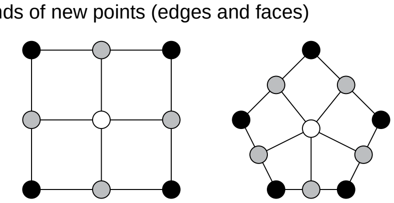


Midpoint (4) scheme
very common

Unconmmon 3-scheme
Does exist

How to divide other polygons

- Middle of Edges + Center of Face
  - Face point connects to edge points
- After 1 subdivision, everything is a quad
- All new points are ordinary points (or edges)
- 2 kinds of new points (edges and faces)



What does a Subdivision Scheme Need?

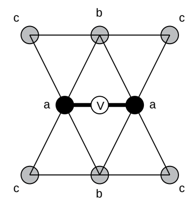
- Rules for ordinary points
- Rules for extra-ordinary points
- Rules for edges/corners
  - Treat them specially
  - Edges only depend on edges (so shared edges connect)

- Proof that the limit surface is continuous
- Exact evaluation methods

- Methods to introduce creases, provide texture coords, …

Butterfly Scheme

- Stationary (interpolating) scheme
- Only rule inserts new points between existing ones
- Regular mesh -> regular mesh
- C(1) at ordinary points



v = 1/2 a + 1/8 b - 1/16 c

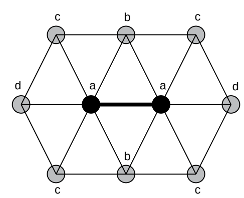
What about extra-ordinary points?

- They do happen!
  - Edges, corners
  - Holes
  - Places where things are stitched together
- Tensor product surfaces can't handle them well either

- Easy Method: do "nothing" – leave midpoint at midpoint
- Problem gets smaller on each iteration
  - Only edges adjacent to extraordinary point
  - And these get cut in half each time
- In limit: "problem" is very localized
  - Surface is C(1) "almost everywhere" (except extra-ordinary points)

Modified Butterfly

- Introduce tension parameter, use 10 points
- New rules for extraordinary points



v = (1/2-w) **a** + (1/8+2w) **b** - (1/16-w) **c** + w **d**

tension parameter w
sum over all 10 neighbors

Modified Butterfly

- Edge with 1 extra-ordinary point
  - Two extraordinary points? Do both as if 1, and average
  - Only happens on first pass
- For a K vertex – only use points around it (weight V=3/4)
  - S0 = point on edge we're dividing

  - K=3 S0=5/12, s1,s2=-1/12
  - K=4 S0=3/8, s1,s3 = 0, s2=-1/8
  - K>=5 (.25+cos(2pi j / K) + .5 * cos (4 pi j / K)) / K
    - J from 0->K-1
- Use 4 point curves around edges
  - -1/16, 9/16, 9/16, -1/16

Why not Butterfly?

- Is C(1) and Interpolating

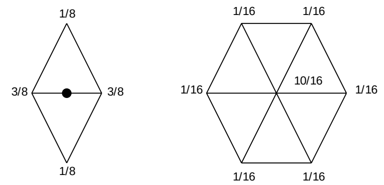
- Sensitive to noise in data (since it will interpolate)
- Not "Fair" (we get little wigglies)
- Not C(2)

- A lot like interpolating cubics

Loop Subdivision

- Named for Charles Loop (not because of loops in the rules)

- Approximating Scheme
  - New Points from close neighborhood of edge
  - Old points are then moved based on their neighbors (including new ones)



Loop Subdivision

- Extra Ordinary Points
  - Center = 1 – k B
  - Each connected point = B
  - B = 1/n (5/8 – ((3+2cos(2pi / n))^2) / 64)
  - Note: this gives the same answer as the ordinary case
  - B = 3 / (n (n+2)) (simpler version, really close, but not exact)
- Use special edge rules
  - Edge points at midpoints
  - Old points = 1/8 ¾ 1/8

Catmull-Clark Subdivision

- Regular Case is quads
- Same rules apply to non-quads
- Only have non-quads at first iteration

- Generalization of cubic B-Splines
  - On uniform mesh, gives same things
  - But works on non-uniform meshes

Catmull-Clark Rules

- Face point = center of polygon
- Edge points = average of 4 neighbors
  - (2 old points, 2 adjacent face points)
- Move old points
  - (n-2)/n times itself
  - 1/n^2 average of N adjacent edges
  - 1/n^2 average of N adjacent faces

Making Creases

- Hard edge subdivision
  - Don't displace points
  - Put edge points at midpoint
- Semi-hard edge subdivision
  - Use hard edge rules for 1^{st} few iterations
  - Then use the regular rules

Exact Evaluation

- For regular points on Catmull-Clark – its just a B-Spline!
- There are methods for extraordinary points (1998)

- For all types, "Masks" exist
  - Final answer depends on points in the neighborhood
  - Look them up in a book

Modeling with subdivision

- Any mesh can be subdivided
- Cut holes, create unusual topology, stitch pieces together
- No matter how complicated the mesh, it will lead to a smooth surface!

Why Subdivision

- B-Splines are Smooth
- B-Splines must be Tesselated
  - Sampling issues
  - How to decide triangle size
  - Need to worry about cracking
- B-Splines have uniform resolution
- Detail must be global

- Limit surfaces are smooth
- Subdivision gives meshes
  - Subdivide as needed
  - Always gives connected mesh
  - Get as many polys as you need
- Subdivision – put detail where you want it
- Detail is multi-resolution

Why Subdivision (2)

- B-Splines require regular grid
- Complex Topology is hard
  - No corners, holes, …
- Trimming is hard
- Stitching is hard
- Get a (u,v) parameterization
  - Not controllable
- Hard to make creases and sharp edges

- Subdivision of any mesh
- Any topology can be handled
  - Easy to make corners, holes, …
- Trimming is easy
- Stitching is easy
- (u,v) parameterization by subdivision of points
  - Controllable
- Easy to make creases and sharp edges

Lecture 25a – Subdivision

Michael Gleicher
November, 2006
Used as notes, not projected as lecture

Last Time

- Triangle Subdivision Schemes
- Today: Quad subdivision schemes
- And move on to next topic (rendering)

Catmull-Clark Subdivision

- Regular Case is quads
- Same rules apply to non-quads
- Only have non-quads at first iteration
- Generalization of cubic B-Splines
  - On uniform mesh, gives same things
  - But works on non-uniform meshes

Catmull-Clark Rules

- Face point = center of polygon (1/n times each)
- Edge points = average of 4 neighbors
  - (2 old points, 2 adjacent face points)
- Move old points
  - (n-2)/n times itself
  - 1/n^2 average of N adjacent edges
  - 1/n^2 average of N adjacent faces
- Edges
  - New point = midpoint
  - Old point = 1/8 ¾ 1/8
- Corners – stationary points

Making Creases

- Hard edge subdivision
  - Pretend that it is an edge of the surface
  - Put edge points at midpoint
- Semi-hard edge subdivision
  - Use hard edge rules for 1st few iterations
  - Then use the regular rules

Exact Evaluation

- For regular points on Catmull-Clark – its just a B-Spline!
- There are methods for extraordinary points (1998)
- For all types, “Masks” exist
  - Final answer depends on points in the neighborhood
  - Look them up in a book

Modeling with subdivision

- Any mesh can be subdivided
- Cut holes, create unusual topology, stitch pieces together
- No matter how complicated the mesh, it will lead to a smooth surface!

Why Subdivision

- B-Splines are Smooth
- B-Splines must be Tesselated
  - Sampling issues
  - How to decide triangle size
  - Need to worry about cracking
- B-Splines have uniform resolution
- Detail must be global

- Limit surfaces are smooth
- Subdivision gives meshes
  - Subdivide as needed
  - Always gives connected mesh
  - Get as many polys as you need
- Subdivision – put detail where you want it
- Detail is multi-resolution

Why Subdivision (2)

- B-Splines require regular grid
- Complex Topology is hard
  - No corners, holes, …
- Trimming is hard
- Stitching is hard
- Get a (u,v) parameterization
  - Not controllable
- Hard to make creases and sharp edges

- Subdivision of any mesh
- Any topology can be handled
  - Easy to make corners, holes, …
- Trimming is easy
- Stitching is easy
- (u,v) parameterization by subdivision of points
  - Controllable
- Easy to make creases and sharp edges

Lecture 25b – Rendering 1

Michael Gleicher
November, 2006
Used as notes, not projected as lecture

Rendering

- How to make an image (from a model)
- How we "draw" with computers
- Generally, term implies trying to make high-quality images
- Two main categories of approaches
  - Object-Based
  - Light-Based
- Distinction is a little fuzzier than that

Object-Based Rendering

- What we've been doing so far
- Draw each object independently
- Primitives and abstractions provided by hardware
  - Triangles, texture mapping, multi-pass, local shading, …
- Hacks to make better and better visual effects
- Pros: abstractions efficient on hardware
- Cons: it's a hack!
  - Can't achieve all effects (without more hacks)
  - Not accurate model of real world

Light-Based Rendering

- Model what happens with light in scene
- Assume that we have a model of the scene
- Figure out how light interacts with it
- Allows for global effects
  - Or at least non-local ones
- Simulate what really happens
  - To varying degrees of realism in the model

How the real world "renders"

- Photons (Rays) from source
- Bounce paths
- Some lucky photons make it to the eye (very few)
- Not a practical strategy – too inefficient

Ray Tracing

- Technically "Backward Ray Tracing"
  - From eye to light
  - There are cases where we actually do forward tracing
  - Terminology is confusing – I prefer "from the eye"
- Idea:
  - For each pixel (image space algorithm)
  - Figure out where the photon would have come from
  - Note: get projective transform from ray fan out
  - Note: could use real model of lens to determine ray directions
  - Note: Sampling Issue

Ray Tracing Pieces

- 1. Figure out what ray is
- 2. Figure out what ray hits (ray-object intersection)
- 3. Figure out where it could have come from
  - Recursive – since outgoing ray must have come from someplace

- Ray / Object Intersection
  - Straightforward mathematical calculation (root finding)
  - Tricky part: making it go fast
  - Accelleration structures:
    - Simplified models (bounding spheres/boxes)
    - Hierarchical models (check rough stuff first)
    - Spatial Data structures

Where did the ray come from?

- We know: outgoing direction, local surface geometry

- Specular bounce
  - Good for mirror reflection

- Real surfaces are diffuse – could come from any direction
  - Distribution of likelihoods
  - Different surfaces distribute light differently
  - Really requires an integral over incoming ray directions

  - **B**i-directional **R**eflectance **D**istribution **F**unction

  - Ideal case: sample all incoming directions

Hack ray-tracing

- Try to model the rays most likely to be important

- Mirror reflection bounce (or refraction bounce)
- Direction towards light sources
  - Probably important since they are bright
  - Check to see if path is clear (hit something = shadow)
  - Use local lighting model

- What does this give us?
  - Everything from local lighting
  - Shadows
  - Reflections and Refractions

Lecture 26 –
Rendering 2

Michael Gleicher
November, 2006
Used as notes, not projected as lecture

Hack ray-tracing

- Try to model the rays most likely to be important
- Mirror reflection bounce (or refraction bounce)
- Direction towards light sources
  - Probably important since they are bright
  - Check to see if path is clear (hit something = shadow)
  - Use local lighting model
- What does this give us?
  - Everything from local lighting
  - Shadows
  - Reflections and Refractions

Shadows

- Shadows of point lights give hard edges
  - Even in the real world!
  - Quite ugly
- Soft shadows are nicer
- Come from area light sources
  - Umbra / penumbra
- How to achieve?
  - More than one ray towards the light source
  - Sampling of directions

Distributed Ray Tracing

- Need to sample a **distribution** of ray directions
- Some uses:
  - Soft shadows (distribution of directions towards area light)
  - Anti-Aliasing (distribution of rays within the pixel)
  - Imperfect reflections (distribution of outgoing rays)
  - Motion Blur (distribution of times)
  - Depth of Field
  - All indirect light directions (for diffuse surfaces)
    - Get inter-object color transfer
  - Notice how quickly this becomes impractical

What can we do with Ray-Tracing?

- Given infinite rays, just about anything
- Realistically:
  - Can be clever about how to sample
  - But ultimately, limited in number of rays
- To understand limits, need to talk about light paths

Light Path Calculus

- Lights
- Diffuse Reflections
- Specular Reflections
- Eyes
- All paths L (D | S)* E
  - Regular expressions
- (Backward) Ray tracing can do:
  - L (D|S) S* E
- What ray tracing can't do
  - Anything else

Examples of other things

- Diffuse inter-reflections
  - L D+ E
  - Indirect lighting very important in the real world
  - Special case: all diffuse surfaces
  - Model energy transport
  - Radiosity methods for solution
- Caustics
  - Light bounces off mirror (or through lens) to light a diffuse object
  - L S* D E

Advanced "Physically-Based" Rendering

- Smart Sampling – of all possible paths
- Bi-Directional Ray Tracing
  - Do some "from the light" and store energy on surfaces
  - Photon Maps
- Complex reflection distribution functions
  - Require complex sampling mechanisms to express
  - Integration over incoming (or outgoing) ray directions

Graphics Hardware

Mike Gleicher
12/06/2006
Lecture Notes – not projected!

Graphics Hardware

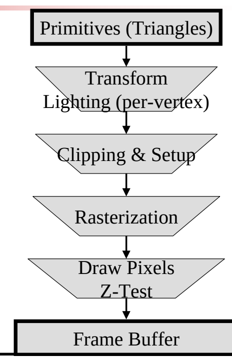
- Why?
  - Need lots of computation to do graphics
  - Lots of pixels, lots of polygons, lots of texels, …
- A few standard things done very often
  - Pipeline provides a standard set of abstractions
  - Break everything into triangles
- Regular computations + pipelineable
- Moving target – changing faster than processors!

History

- 1980s – first workstation 3D hardware (SGI)
- 1990s – extension of abstraction set
  - Texture mapping, compositing, multi-buffering
- 1990s – first PC graphics hardware
  - Low end (Apple's white magic project)
  - High end (3D solutions – expensive)
- 2000s – consumer graphics hardware
  - Driven by gaming market
  - Extensive use of the abstractions
- 2002++ - programmable graphics hardware
  - Better abstractions, generality, use as GP processor

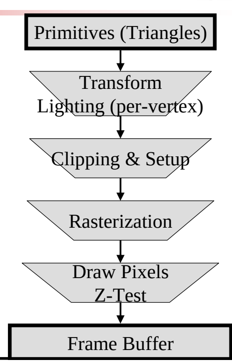
Graphics Pipeline

- Fixed set of abstractions
  - Doesn't really change
  - Can optimize
  - Fits a programming model
- Early Graphics Hardware
  - 4x4 transform engines
  - Fill Engines
  - Scanline hardware (Apple)



Working with the Pipeline

- Where is your bottleneck?
- Get your triangles fast
  - Vertex sharing schemes
  - Display lists / v-buffers
- Filling pixels
  - Lots of z tests (read/write)
  - Texture accesses per pixel
- Limitations
  - Set operations for each phase



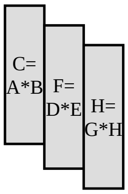
Early Extensions to Pipeline

- Texture Mapping
- Accumulation Buffer
  - More light sources
  - Compositing
  - Anti-Aliasing / Motion Blur
- Stencil Buffer

Pipelining in conventional processors

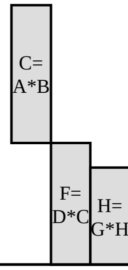
- Start step 2 before step 1 completes

C = A * B
F = D * E
J = G * H



- Unless step 2 depends on step 1
- Pipe Stall

C = A * B
F = D * **C**
J = G * H



Pipelines in graphics processors

- Conventional processors – stalls are bad
  - Need shorter pipelines

- Pixels and vertices are independent
- Pipes can be long
- Parallelism is easy
  - Start as many at a time as you want

Programming the Pipeline

- Vertex programs

- Given the info about a vertex
  - Local coords, transform matrices, colors
  - Lights
- Figure out the color and position
  - Typical: standard lighting model
  - Give a little program

- Parallel – all vertices happen at once
- Deeply pipelined (not intervention till end)

Fragment Shaders

- Fragment = Pixel (?)
  - Multiple fragments = 1 pixel if anti-aliasing
- Given “context” figure out color to write
  - Pixel (fragment) position already known
  - Gets control over z-test, …

- Highly parallel
- All pixels run the same program
  - SIMD – single instruction multiple data

Why is graphics hardware fast?

- Highly parallel
  - Simple parallel model
  - Lots of little processors
- Deeply pipelined
  - Results are independent

- Multiple processors on a chip is way of future
  - Speeds can’t get faster
  - Chips can’t get bigger (cross chip latencies)

Lecture 28 –
An hours worth of animation

Michael Gleicher
November, 2006
Used as notes, not projected as lecture

Computer Animation

- Worth its own course (at least)
  - We only get an hour (or less, since need to do evals)
- Go over some of the ideas / concepts
- See how some graphics concepts come into play
- A whole art form
- A wide range of technical challenges

What is Computer Animation?

- Making moving images with a computer
  - 3D movies and games
  - Web browser animations (annoying and useful)
  - Scientific Visualization and Design
  - Desktop animations – paper clip, windows moving
- Why is this hard/different
  - Need to consider how things move / what is their motion
  - Need things to be easier to control (so we can move them)
  - Lots of images, coherence issues

Principles of Animation

- Why animate the moving windows?
  - Looks cool
  - Easier to understand what's going on
- Ideas from animation help everywhere
  - Animation was (historically) hard to produce
  - Need to be economical about drawing
  - Needed to learn how to communicate in moving images
- Early animators
  - Anything can happen in animation
    - A talking and dancing mouse?
  - How to make it understandable
  - Control suprise

Principles of Animation

- Developed in the late 20s early 30s
- Disney was a key player
- Exageration
- Anticipation
- Follow Through and Overlap
- Secondary Action
- Squash and Stretch
- Staging
- Timing
- Slow in/slow out
- Straight ahead vs. pose to pose, Arcs, Appeal

How was this done historically

- Cel animation
  - Transparent sheets of celuloid
  - Allows characters drawn independent of background
  - Layers
- Character Animation
  - Needs a good artists
  - Lots of drawings to make
- Keyframing
  - Master artists draws a few poses
  - "Tweener" draws "in-betweens"

Keyframing by Computer

- Still how the best character animation is done
- Good artists can be extremely creative

- Set a small number of "key poses"
- Use interpolation to get in-betweens

- Big application of interpolating splines
  - Catmull-Rom Splines
  - TCB (tension continuity bias splines)
    - Cardinals with more control (still interpolate)
    - Change tension – per control point
    - Tension on each side of control point (bias)
    - Deviate from C(1) (continuity)

Parametric Models

- What do you interpolate?
  - Need to have a vector of numbers – point in pose space
- Controls or parameters
  - Need enough to be expressive
  - Few enough to be convenient

- Use position of every point on a mesh?
  - Lots of data to move around on every frame
- Use rigid transform?
  - Might not be enough to just move things around
  - OK for levels of detail
- Use deformations?

Articulated Figure Animation

- Humans and animals (vertibrate) modeled as rigid bones
  - Gets the main effect
- Bones are rigid (don't really change size)
- Connected by joints (rotation)

- Configuration = position of "root" + orientations

- Can pick any point to be root
  - Center or pelvis
  - Foot for convenience

Why is hierarchical good?

- Fewer parameters
- Enforce essential constraints
  - Keep from stretching
  - Keep limbs from falling off
- Keep constraints when posing or interpolating

- Why is it bad?
  - Hard to position end points
  - Hard to enforce constraints on end points (footskate)
  - Parameters are coupled

Controlling Hierarchical Models

- Forward Kinematics
  - Specify angles, see what happens
- Inverse Kinematics
  - Specify end-effector positions, figure out where joints angles must be

- Doing IK
  - Might be no solutions
  - Might be lots of solutions
  - Non-linear equations

  - Easy to solve for special cases (2 link arms)

Drawing Hierarchical Models

- Simple: Draw rigid pieces
- Complex: (arbitrarily)
  - Compute some "skin" over the bones
  - Use simulation, or anything

- One tradeoff:
  - Use a simple "skinning" model
  - Have a single mesh for the object
  - Associate each vertex with multiple coordinate systems
  - Weights determine how much

Skinning

- Smooth skinning, linear blend skinning, …
- Blend positions (interpolate) or matrices
  – Note that interpolating matrices doesn't preserve rigidity
- Good points
  – Easy
  – Efficient
  – Maps Nicely to hardware
- Bad points
  – Simple / hackish
  – Hard to find weights
  – Bad effects (collapsing, candy-wrappering)
- Improved methods keep coming…

Where does motion come from

- Keyframing
- Observation (motion capture)
- Procedural (compute it)

- Physics = a form of procedural
- Synthesis by example = combine procedure and observation

Animation by Observation

- Motion Capture
- Record the movements of a real performer

- Why?
  – Realistic motion
  – Get the actual person
  – Actors are directable
  – No need to have models of motion properties
    - Mathematical definition for "happy" or "skip"
- Why not?
  – Realistic motion
  – Get what actor does (at best)
  – Need special devices to record

How to do Motion Capture

- Video – not possible (yet)
  – Not enough information to make good measurements
  – Can't get correspondences for triangulation

- Optical Motion Capture
  – Engineer away what's hard about video
  – Dot markers – easy to find
    - Retro-reflective markers (lights on cameras)
    - LEDs (blinked to get correspondences)
  – Cameras only see bright spots
  – Many cameras (to do triangulation, disambiguation)

  – Some systems have dozens – if not hundreds – of cameras

Why not optical mocap

- Still get drop outs
- Real-time, online hard (drop outs, correspondence)

- Alternatives:
- Electromagnetic
- Mechanical

How to use motion capture

- Individual clips generally short

- String together into longer chains

- Transitions
  – Might be easy, might be hard
  – Look for easy cases

- Motion Graphs