Introduction

- Hi, I'm Kyle!
- Welcome to CS 6319.001.22S—Computational Geometry.
- Computational geometry emerged from the field of discrete algorithm design and analysis starting in the late 1970s.
- It involves designing algorithms and data structures for different geometric problems. Since it is a subfield of algorithms, I'll be emphasizing provably correct algorithms with low worst-case running times. Think 6363, but with more points and lines.
- So it is mostly a theory class, but I'll spend a lot of time focusing on material useful in relevant application areas.
- In fact, having this knowledge may help you differentiate yourself when looking for jobs!
- For example, when trying to draw scenes in computer graphics, you need to know which objects are visible from certain pixels and which are obscured by other objects. One method of solving this problem is to perform ray tracing, figuring out which object is hit first if I shoot a ray from a pixel of my screen into the scene.
- In geographic information systems, we need to know what information should be displayed based on how zoomed in we are. And if somebody points at a location in a map, you need to figure out which city they're pointing at. Both problems require fast geometric data structures.
- In robotics, we need to understand how to move robots around while avoiding obstacles. A simple version of this problem is asking how to move a robot from point a in the plane to point b, while avoiding a number of polygonal obstacles.
- So while the class will mostly focus on the algorithms themselves, I hope you'll learn about some useful techniques or algorithmic problems you hadn’t encountered before that can then be applied in your work.

Administrivia

- Before we get into any more detail, though, I'd like to talk about how the class is structured.
- Everything I'm about to say can be found on the course website: https://personal.utdallas.edu/~kyle.fox/courses/cs6319.001.22s/
- The course has one official prerequisite: CS 5343—Algorithm Analysis and Data Structures. This course should teach you about all the fundamental non-geometric data structures we'll use in this class and maybe some basic algorithms we'll use as subroutines.
- That said, you'll probably have a much easier time of it if you've taken CS 6363—Design
and Analysis of Computer Algorithms. I'll be using a few algorithms from that class like Dijkstra's shortest path algorithm as subroutines, and some algorithm design techniques like divide-and-conquer and dynamic programming will likely come up.

- I'll try to fill in any additional background you may have missed, but I can't always guess what you know. Please ask lots of questions!

- Your grade will be based on a weighted average of three things.
  1. 50% will come from homework assignments.
     - I'll release homework once every few weeks and make it due about two weeks later.
     - You may work in formal groups of up to two people if you'd like, but it is not a requirement to group up. Each group should turn in one copy of the homework on eLearning. I'll give everybody in the group the same grade.
     - I'll take late homework, but you'll give me a heads up. I'll automatically approve all requests for a one day extension, even if I don't get back to you. We'll have to talk about longer extensions, but I understand things are still hard.
  2. The rest of the grade will come from a course project. This can be anything suitably project-like such as a survey, implementation for some application or performance testing, or even original research in computational geometry.
  3. Finally, you'll get to form groups of up to two people again for the final project. You don't have to go through with your specific proposal if you like somebody else's better or the same group member. 40% of your grade will be a short paper of around 10 pages.
     - For research proposals, I expect almost all reports to be short surveys and descriptions of a few failed attempts. But if you partially or fully solve your problem, then all the better!

- I'll add up all of these things and then heavily curve the grades.
- Let me reemphasize, there are no set percentage targets you need to meet to get certain grades.

- That said, this curving scheme means I can ask tough questions. And so you're not surprised, I like to grade homework ruthlessly.
- I think fighting back against the homework is your best opportunity to learn, and me being ruthless with grading is your best way to get meaningful feedback on what you learned.
- So don't expect high percentages. Again, there's no set scale.
- Also, algorithms is a theory topic, so you need to justify your answers or your algorithms
with an argument, some might say a proof, that they are correct. Try to be at least as rigorous as I am in lecture.

- As a rule of thumb, if you can’t explain why your answer is correct after appealing to things in class and your own logic, it probably isn’t a correct algorithm.
- The website contains a whole page devoted to what you should and shouldn’t do when answering homework questions.

- I am tentatively planning to do office hours Tuesdays from 2:30pm to 3:30pm (but not today; sorry) and Wednesdays from 10am to 11am over MS Teams. I can also set additional meetings by appointment. If those regular times don’t work for too many of you, they can be changed.
- The required textbook is Computational Geometry—Algorithms and Applications by de Berg et al.
  - I find it a bit long-winded and loose myself, but it is the standard text for computational geometry classes.
  - I won’t be asking homework problems directly from the book, and I’ll post my lecture scripts online, so if you really don’t want to buy it, you don’t have to.
  - But, it is a good book to have on your shelf even after class is over. I refer to it occasionally for my own research.
- There’s also a link to some lecture notes by David Mount on the website. My lectures will largely follow his presentation. I’ll let you know what chapters from the books and lecture notes are relevant to whatever we’re covering. And by the end of the semester, I’ll probably be completely off script from both texts since we’ll be focusing on less classical topics.
- Oh, and a warning about my notes: They should be fairly thorough, but they aren’t pretty, and they may have bugs.

- And with all that, let’s actually talk some about computational geometry!

**Convex Hulls and Basic Definitions**

- If you’ve taken 6363, especially with me, you probably know I like to emphasize techniques over specific problems.
- In this class, you’ll see more of the opposite, with specific problems motivating techniques.
- In particular, I want to start with the standard first problem in any computational geometry course: computing convex hulls in 2D. This problem has a nice combination of being easy to describe, practical motivation, and having several algorithms, all of which demonstrate different techniques commonly used in computational geometry.
- We’ll begin with a few definitions suitable for any d-dimensional Euclidean space R^d.
Some of these definitions will prove useful later.

- We’ll use the terms points or vectors for elements of $\mathbb{R}^d$, depending on which makes more sense in the context we’re working in. I’ll sometimes use the term scalars for individual real numbers.

- Given points $p$ and $q$, we can express their common line $\langle -\rangle pq$ as the linear combination of the points / their coordinates where coefficients sum to 1, so all points $(1 - \alpha)p + \alpha q$. That’s also called an affine combination of $p$ and $q$. If we demand $0 \leq \alpha \leq 1$, we get a convex combination instead, and all points lie on the line segment $-pq$. We can extend those definitions to any number of points $\{p_1, \ldots, p_k\}$ so an affine combination for example would be any

$$\sum_{i=1}^{k} \alpha_i p_i, \text{ such that } \alpha_1 + \cdots + \alpha_k = 1.$$  

The set of all affine combinations form the affine span or affine closure of the points.

- Given a non-zero vector $v$ in $\mathbb{R}^d$ and a scalar $\alpha$ in $\mathbb{R}$, the set of points $\{p \mid p \cdot v = \alpha\}$ is a $(d-1)$-dimensional affine subspace called a hyperplane. In $\mathbb{R}^2$ these are lines. In $\mathbb{R}^3$, they are planes. A halfspace consists of points on one side of a hyperplane: $\{p \mid p \cdot v \leq \alpha\}$.

- The real formal definitions are a bit messy and belong more to a topology course, but we’ll say a set is open if it does not include its boundary and is closed otherwise.

- A subset $K$ of $\mathbb{R}^d$ is convex if for any pair of points $p$ and $q$ in $K$, the line segment $-pq$ is entirely in $K$. Equivalently, $K$ is closed under convex combinations. So points within a circular disk or polygon are convex. So are line (segments), rays, or halfspaces.

- We’ll rely on the follow fact: Let $K$ be a convex set in $\mathbb{R}^d$. For every point $p$ on the boundary of $K$, there is at least one hyperplane $h$ such that $K$ lies entirely in one of the two closed halfspaces of $h$. Hyperplane $h$ is called a support hyperplane for $K$.

- Finally, given a set of points $P$, the convex hull of $P$, denoted $\text{conv}(P)$ is the intersection of all convex sets containing $P$. Equivalently, it is the smallest convex body containing $P$ or the set of all convex combinations of $P$.

- For today and Thursday, we’re given a finite set $P$ of points in the plane, $\mathbb{R}^2$. Let $n := |P|$. Set $\text{conv}(P)$ is a polygon whose vertices come from a subset of $P$. The edges of the polygon follow support lines of $\text{conv}(P)$.

- Informally, the convex hull is what you get if you wrap a big rubber band around all the points of $P$ and let it snap down around them.
• It provides a simple summary of your point set.
  • For example, I can tell how spread out a point set is in any direction by looking at how spread out the hull is in that direction.
  • Convex hulls can also be used to approximate objects other than point sets. I could take the convex hull of a polygon’s vertices for example. If I have some polygons representing objects floating through space, I can use their convex hulls to simplify collision detections with only a small loss in accuracy.
• We want our algorithm to provide a good description of $\text{conv}(P)$, so we’ll have it compute a counterclockwise list of vertices lying along the convex hull.

Assumptions

• Computational geometry is an offshoot of discrete algorithms, so we make a couple assumptions to keep things clean while designing our algorithm.
• First, we’ll usually assume that our input is a collection of real numbers instead of floating point numbers. That way we can get to the algorithmic meat of the problem without worrying about computation details. We can usually modify algorithms designed for real number inputs to work nicely with floating point inputs as well. I won’t get into it today (ever?), but Chapter 1 of the book has some nice asides on the topic.
• The other big assumption we’ll make is that our inputs often lie in what’s called general position. Informally, this means we have no degeneracies in our input that might force us to deal with annoying special cases. When the input is a set of points like $P$, we’ll assume:
  • No two points share the same $x$ or $y$ coordinate.
  • No three points lie on the same line.
• Like real numbers vs. floating points, you can usually design an algorithm assuming general position and then do some simple modifications to make it work with all inputs, but only after we design the initial algorithm.
• After all, how can you handle the complicated case if you can’t even handle the “easy” case first?

Algorithm for Convex Hulls

• Today, I’ll present a modification by Andrew (’79) of the well known Graham’s scan (’72) algorithm that runs in $O(n \log n)$ time.
• This algorithm uses an idea called incremental construction that will come up many times over the course of the semester.
  • We’ll add points to a subcollection of the input one-by-one, maintaining a solution for exactly the points in our current subcollection.
• Usually, we like to work with the points in some convenient order, so we’ll start by sorting them from left to right in $O(n \log n)$ time. Let $p_1, \ldots, p_n$ be the points in this order.
• It's also convenient to find convex hull vertices in order from left to right.
• Now, you can't trace around the entire hull going left to right the whole time, but if you consider the hull boundary as separate upper and lower chains, you can.
• We'll focus on finding the upper chain vertices in order from left to right. The algorithm for the lower hull is symmetric. After computing both, we can concatenate the lower chain list with the reversal of the upper chain list to compute the whole convex hull.